axiom™



The 30 Year Horizon

Manuel Bronstein *William Burge* *Timothy Daly*
James Davenport *Michael Dewar* *Martin Dunstan*
Albrecht Fortenbacher *Patrizia Gianni* *Johannes Grabmeier*
Jocelyn Guidry *Richard Jenks* *Larry Lambe*
Michael Monagan *Scott Morrison* *William Sit*
Jonathan Steinbach *Robert Sutor* *Barry Trager*
Stephen Watt *Jim Wen* *Clifton Williamson*

Volume 10: Axiom Algebra: Packages

Portions Copyright (c) 2005 Timothy Daly

The Blue Bayou image Copyright (c) 2004 Jocelyn Guidry

Portions Copyright (c) 2004 Martin Dunstan

Portions Copyright (c) 1991-2002,
The Numerical ALgorithms Group Ltd.
All rights reserved.

This book and the Axiom software is licensed as follows:

Redistribution and use in source and binary forms, with or
without modification, are permitted provided that the following
conditions are
met:

- Redistributions of source code must retain the above
  copyright notice, this list of conditions and the
  following disclaimer.

- Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the
  following disclaimer in the documentation and/or other
  materials provided with the distribution.

- Neither the name of The Numerical ALgorithms Group Ltd.
  nor the names of its contributors may be used to endorse
  or promote products derived from this software without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

Cyril Alberga
George Andrews
Yurij Baransky
Gilbert Baumslag
Mark Botch
Martin Brock
William Burge
Bruce Char
Gregory V. Chudnovsky
Don Coppersmith
Gary Cornell
Timothy Daly Sr.
Jean Della Dora
Claire DiCrescendo
Martin Dunstan
Robert Edwards
Richard Fateman
Brian Ford
Constantine Frangos
Marc Gaetano
Patricia Gianni
Laureano Gonzalez-Vega
Matt Grayson
Oswald Gschnitzer
Vilya Harvey
Ralf Hemmecke
Pietro Iglio
Grant Keady
Klaus Kusche
Frederic Lehobey
Rudiger Loos
Camm Maguire
Ian Meikle
Gerard Milmeister
Michael Monagan
Mark Murray
John Nelder
Jinzhong Niu
Julian A. Padget
Susan Pelzel
Claude Quitte
Renaud Rioboo
Simon Robinson
Philip Santas
Gerhard Schneider
Fritz Schwarz
Elena Smirnova
Robert Sutor
James Thatcher
Dylan Thurston
Gregory Vanuxem
Juergen Weiss
James Wen
John M. Wiley
Stephen Wilson
Sandra Wityak

Roy Adler
Henry Baker
David R. Barton
Fred Blair
Alexandre Bouyer
Manuel Bronstein
Quentin Carpent
Cheekai Chin
Josh Cohen
George Corliss
Meino Cramer
Timothy Daly Jr.
Gabriel Dos Reis
Sam Dooley
Brian Dupee
Heow Eide-Goodman
Bertfried Fauser
Albrecht Fortenbacher
Timothy Freeman
Rudiger Gebauer
Holger Gollan
Stephen Gortler
James Griesmer
Jocelyn Guidry
Satoshi Hamaguchi
Henderson
Richard Jenks
Tony Kennedy
Bernhard Kutzler
Michel Levaud
Michael Lucks
Bob McElrath
David Mentre
Mohammed Mobarak
Marc Moreno-Maza
William Naylor
Godfrey Nolan
Michael O'Connor
Bill Page
Michel Petitot
Norman Ramsey
Jean Rivlin
Michael Rothstein
Alfred Scheerhorn
Martin Schoenert
Nick Simicich
Jonathan Steinbach
Moss E. Sweedler
Baldir Thomas
Barry Trager
Bernhard Wall
M. Weller
Thorsten Werther
Berhard Will
Shmuel Winograd
Waldemar Wiwianka

Richard Anderson
Stephen Balzac
Gerald Baumgartner
Vladimir Bondarenko
Peter A. Broadbery
Florian Bundschuh
Bob Caviness
David V. Chudnovsky
Christophe Conil
Robert Corless
Claire Di Crescenzo
James H. Davenport
Michael Dewar
Lionel Ducos
Dominique Duval
Lars Erickson
Stuart Feldman
George Frances
Korrinn Fu
Kathy Gerber
Teresa Gomez-Diaz
Johannes Grabmeier
Vladimir Grinberg
Steve Hague
Martin Hassner
Antoine Hersen
Kai Kaminski
Paul Kosinski
Larry Lambe
Howard Levy
Richard Luczak
Michael McGettrick
Victor S. Miller
H. Michael Moeller
Scott Morrison
C. Andrew Neff
Arthur Norman
Kostas Oikonomou
Jaap Weel
Didier Pinchon
Michael Richardson
Nicolas Robidoux
Martin Rubey
William Schelter
Marshall Schor
William Sit
Christine Sundaresan
Eugene Surowitz
Mike Thomas
Themos T. Tsikas
Stephen Watt
Mark Wegman
Michael Wester
Clifton J. Williamson
Robert Wisbauer
Knut Wolf

Contents

1 Chapter Overview **1**

2 Chapter A **3**

2.1 package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis 3

2.1.1 AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO) 4

2.2 package AFALGRES AffineAlgebraicSetComputeWithResultant 8

2.2.1 AffineAlgebraicSetComputeWithResultant (AFALGRES) 9

2.3 package AF AlgebraicFunction 13

2.3.1 AlgebraicFunction (AF) 13

2.4 package INTHERAL AlgebraicHermiteIntegration 19

2.4.1 AlgebraicHermiteIntegration (INTHERAL) 19

2.5 package INTALG AlgebraicIntegrate 21

2.5.1 AlgebraicIntegrate (INTALG) 21

2.6 package INTAF AlgebraicIntegration 28

2.6.1 AlgebraicIntegration (INTAF) 28

2.7 package ALGMANIP AlgebraicManipulations 30

2.7.1 AlgebraicManipulations (ALGMANIP) 30

2.8 package ALGMFACT AlgebraicMultFact 35

2.8.1 AlgebraicMultFact (ALGMFACT) 35

2.9 package ALGPKG AlgebraPackage 37

2.9.1 AlgebraPackage (ALGPKG) 37

2.10 package ALGFACT AlgFactor 47

2.10.1 AlgFactor (ALGFACT) 47

2.11 package INTPACK AnnaNumericalIntegrationPackage 50

2.11.1 AnnaNumericalIntegrationPackage (INTPACK) 50

2.12 package OPTPACK AnnaNumericalOptimizationPackage 61

2.12.1 AnnaNumericalOptimizationPackage (OPTPACK) 61

2.13 package ODEPACK AnnaOrdinaryDifferentialEquationPackage 69

2.13.1 AnnaOrdinaryDifferentialEquationPackage (ODEPACK) 69

2.14 package PDEPACK AnnaPartialDifferentialEquationPackage 78

2.14.1 AnnaPartialDifferentialEquationPackage (PDEPACK) 78

2.15 package ANY1 AnyFunctions1 84

2.15.1 AnyFunctions1 (ANY1) 84

2.16 package API ApplicationProgramInterface 86

2.16.1 ApplicationProgramInterface (API) . . . 90
2.17 package APPRULE ApplyRules . . . 92
2.17.1 ApplyRules (APPRULE) . . . 92
2.18 package APPLYORE ApplyUnivariateSkewPolynomial . . . 95
2.18.1 ApplyUnivariateSkewPolynomial (APPLYORE) . . . 95
2.19 package ASSOCEQ AssociatedEquations . . . 96
2.19.1 AssociatedEquations (ASSOCEQ) . . . 96
2.20 package PMPRED AttachPredicates . . . 99
2.20.1 AttachPredicates (PMPRED) . . . 99
2.21 package AXSERV AxiomServer . . . 100
2.21.1 AxiomServer (AXSERV) . . . 100

3 Chapter B **117**
3.1 package BALFACT BalancedFactorisation . . . 117
3.1.1 BalancedFactorisation (BALFACT) . . . 117
3.2 package BOP1 BasicOperatorFunctions1 . . . 119
3.2.1 BasicOperatorFunctions1 (BOP1) . . . 119
3.3 package BEZIER Bezier . . . 122
3.3.1 Bezier (BEZIER) . . . 139
3.4 package BEZOUT BezoutMatrix . . . 141
3.4.1 BezoutMatrix (BEZOUT) . . . 141
3.5 package BLUPPACK BlowUpPackage . . . 144
3.5.1 BlowUpPackage (BLUPPACK) . . . 146
3.6 package BOUNDZRO BoundIntegerRoots . . . 151
3.6.1 BoundIntegerRoots (BOUNDZRO) . . . 151
3.7 package BRILL BrillhartTests . . . 154
3.7.1 BrillhartTests (BRILL) . . . 154

4 Chapter C **157**
4.1 package CARTEN2 CartesianTensorFunctions2 . . . 157
4.1.1 CartesianTensorFunctions2 (CARTEN2) . . . 157
4.2 package CHVAR ChangeOfVariable . . . 159
4.2.1 ChangeOfVariable (CHVAR) . . . 159
4.3 package CPIMA CharacteristicPolynomialInMonogenicalAlgebra . . . 162
4.3.1 CharacteristicPolynomialInMonogenicalAlgebra (CPIMA) . . . 162
4.4 package CHARPOL CharacteristicPolynomialPackage . . . 164
4.4.1 CharacteristicPolynomialPackage (CHARPOL) . . . 164
4.5 package IBACHIN ChineseRemainderToolsForIntegralBases . . . 165
4.5.1 ChineseRemainderToolsForIntegralBases (IBACHIN) . . . 165
4.6 package CVMP CoerceVectorMatrixPackage . . . 170
4.6.1 CoerceVectorMatrixPackage (CVMP) . . . 170
4.7 package COMBF CombinatorialFunction . . . 171
4.7.1 CombinatorialFunction (COMBF) . . . 175
4.8 package CDEN CommonDenominator . . . 187
4.8.1 CommonDenominator (CDEN) . . . 187
4.9 package COMMONOP CommonOperators . . . 189

4.9.1 CommonOperators (COMMONOP) . . . 189
4.10 package COMMUPC CommuteUnivariatePolynomialCategory . . . 194
4.10.1 CommuteUnivariatePolynomialCategory (COMMUPC) . . . 194
4.11 package COMPFACT ComplexFactorization . . . 195
4.11.1 ComplexFactorization (COMPFACT) . . . 195
4.12 package COMPLEX2 ComplexFunctions2 . . . 198
4.12.1 ComplexFunctions2 (COMPLEX2) . . . 198
4.13 package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation . . . 199
4.13.1 ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE) . . . 199
4.14 package COMPLPAT ComplexPattern . . . 201
4.14.1 ComplexPattern (COMPLPAT) . . . 201
4.15 package CPMATCH ComplexPatternMatch . . . 202
4.15.1 ComplexPatternMatch (CPMATCH) . . . 202
4.16 package CRFP ComplexRootFindingPackage . . . 204
4.16.1 ComplexRootFindingPackage (CRFP) . . . 204
4.17 package CMPLXRT ComplexRootPackage . . . 216
4.17.1 ComplexRootPackage (CMPLXRT) . . . 216
4.18 package CTRIGMNP ComplexTrigonometricManipulations . . . 218
4.18.1 ComplexTrigonometricManipulations (CTRIGMNP) . . . 218
4.19 package ODECONST ConstantLODE . . . 221
4.19.1 ConstantLODE (ODECONST) . . . 221
4.20 package COORDSYS CoordinateSystems . . . 223
4.20.1 CoordinateSystems (COORDSYS) . . . 223
4.21 package CRAPACK CRApackage . . . 228
4.21.1 CRApackage (CRAPACK) . . . 228
4.22 package CYCLES CycleIndicators . . . 230
4.22.1 CycleIndicators (CYCLES) . . . 249
4.23 package CSTTOOLS CyclicStreamTools . . . 254
4.23.1 CyclicStreamTools (CSTTOOLS) . . . 254
4.24 package CYCLOTOM CyclotomicPolynomialPackage . . . 256
4.24.1 CyclotomicPolynomialPackage (CYCLOTOM) . . . 256

5 Chapter D 259
5.1 package DFINTTLS DefiniteIntegrationTools . . . 259
5.1.1 DefiniteIntegrationTools (DFINTTLS) . . . 259
5.2 package DEGRED DegreeReductionPackage . . . 265
5.2.1 DegreeReductionPackage (DEGRED) . . . 265
5.3 package DTP DesingTreePackage . . . 266
5.3.1 DesingTreePackage (DTP) . . . 268
5.4 package DIOSP DiophantineSolutionPackage . . . 277
5.4.1 DiophantineSolutionPackage (DIOSP) . . . 277
5.5 package DIRPROD2 DirectProductFunctions2 . . . 281
5.5.1 DirectProductFunctions2 (DIRPROD2) . . . 281
5.6 package DLP DiscreteLogarithmPackage . . . 283
5.6.1 DiscreteLogarithmPackage (DLP) . . . 283
5.7 package DISPLAY DisplayPackage . . . 285

5.7.1 DisplayPackage (DISPLAY) . . . 285
5.8 package DDFACT DistinctDegreeFactorize . . . 288
5.8.1 DistinctDegreeFactorize (DDFACT) . . . 288
5.9 package DFSFUN DoubleFloatSpecialFunctions . . . 294
5.9.1 DoubleFloatSpecialFunctions (DFSFUN) . . . 309
5.9.2 The Exponential Integral . . . 313
5.9.3 En:(PI,R)→OPR . . . 319
5.9.4 The Ei Function . . . 319
5.9.5 The Fresnel Integral[**?**, **?**] . . . 346
5.10 package DBLRESP DoubleResultantPackage . . . 350
5.10.1 DoubleResultantPackage (DBLRESP) . . . 350
5.11 package DRAWCX DrawComplex . . . 352
5.11.1 DrawComplex (DRAWCX) . . . 352
5.12 package DRAWHACK DrawNumericHack . . . 356
5.12.1 DrawNumericHack (DRAWHACK) . . . 356
5.13 package DROPT0 DrawOptionFunctions0 . . . 357
5.13.1 DrawOptionFunctions0 (DROPT0) . . . 357
5.14 package DROPT1 DrawOptionFunctions1 . . . 361
5.14.1 DrawOptionFunctions1 (DROPT1) . . . 361
5.15 package D01AGNT d01AgentsPackage . . . 363
5.15.1 d01AgentsPackage (D01AGNT) . . . 363
5.16 package D01WGTS d01WeightsPackage . . . 369
5.16.1 d01WeightsPackage (D01WGTS) . . . 369
5.17 package D02AGNT d02AgentsPackage . . . 375
5.17.1 d02AgentsPackage (D02AGNT) . . . 375
5.18 package D03AGNT d03AgentsPackage . . . 381
5.18.1 d03AgentsPackage (D03AGNT) . . . 381

6 Chapter E **385**
6.1 package EP EigenPackage . . . 385
6.1.1 EigenPackage (EP) . . . 385
6.2 package EF ElementaryFunction . . . 391
6.2.1 ElementaryFunction (EF) . . . 403
6.3 package DEFINTEF ElementaryFunctionDefiniteIntegration . . . 421
6.3.1 ElementaryFunctionDefiniteIntegration (DEFINTEF) . . . 421
6.4 package LODEEF ElementaryFunctionLODESolver . . . 426
6.4.1 ElementaryFunctionLODESolver (LODEEF) . . . 426
6.5 package ODEEF ElementaryFunctionODESolver . . . 432
6.5.1 ElementaryFunctionODESolver (ODEEF) . . . 432
6.6 package SIGNEF ElementaryFunctionSign . . . 438
6.6.1 ElementaryFunctionSign (SIGNEF) . . . 438
6.7 package EFSTRUC ElementaryFunctionStructurePackage . . . 442
6.7.1 ElementaryFunctionStructurePackage (EFSTRUC) . . . 442
6.8 package EFULS ElementaryFunctionsUnivariateLaurentSeries . . . 451
6.8.1 ElementaryFunctionsUnivariateLaurentSeries (EFULS) . . . 451
6.9 package EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries . . . 459

6.9.1 ElementaryFunctionsUnivariatePuiseuxSeries (EFUPXS) 459
6.10 package INTEF ElementaryIntegration 465
6.10.1 ElementaryIntegration (INTEF) 465
6.11 package RDEEF ElementaryRischDE 474
6.11.1 ElementaryRischDE (RDEEF) 474
6.12 package RDEEFS ElementaryRischDESystem 482
6.12.1 ElementaryRischDESystem (RDEEFS) 482
6.13 package ELFUTS EllipticFunctionsUnivariateTaylorSeries 485
6.13.1 EllipticFunctionsUnivariateTaylorSeries (ELFUTS) 485
6.14 package EQ2 EquationFunctions2 487
6.14.1 EquationFunctions2 (EQ2) 487
6.15 package ERROR ErrorFunctions 488
6.15.1 ErrorFunctions (ERROR) 488
6.16 package GBEUCLID EuclideanGroebnerBasisPackage 490
6.16.1 EuclideanGroebnerBasisPackage (GBEUCLID) 513
6.17 package EVALCYC EvaluateCycleIndicators 525
6.17.1 EvaluateCycleIndicators (EVALCYC) 525
6.18 package ESCONT ExpertSystemContinuityPackage 526
6.18.1 ExpertSystemContinuityPackage (ESCONT) 526
6.19 package ESCONT1 ExpertSystemContinuityPackage1 532
6.19.1 ExpertSystemContinuityPackage1 (ESCONT1) 532
6.20 package ESTOOLS ExpertSystemToolsPackage 534
6.20.1 ExpertSystemToolsPackage (ESTOOLS) 534
6.21 package ESTOOLS1 ExpertSystemToolsPackage1 542
6.21.1 ExpertSystemToolsPackage1 (ESTOOLS1) 542
6.22 package ESTOOLS2 ExpertSystemToolsPackage2 543
6.22.1 ExpertSystemToolsPackage2 (ESTOOLS2) 543
6.23 package EXPR2 ExpressionFunctions2 544
6.23.1 ExpressionFunctions2 (EXPR2) 544
6.24 package EXPRSOL ExpressionSolve 545
6.24.1 Bugs 545
6.24.2 ExpressionSolve (EXPRSOL) 546
6.25 package ES1 ExpressionSpaceFunctions1 548
6.25.1 ExpressionSpaceFunctions1 (ES1) 548
6.26 package ES2 ExpressionSpaceFunctions2 550
6.26.1 ExpressionSpaceFunctions2 (ES2) 550
6.27 package EXPRODE ExpressionSpaceODESolver 551
6.27.1 ExpressionSpaceODESolver (EXPRODE) 551
6.28 package OMEXPR ExpressionToOpenMath 556
6.28.1 ExpressionToOpenMath (OMEXPR) 556
6.29 package EXPR2UPS ExpressionToUnivariatePowerSeries 562
6.29.1 ExpressionToUnivariatePowerSeries (EXPR2UPS) 562
6.30 package EXPRTUBE ExpressionTubePlot 569
6.30.1 ExpressionTubePlot (EXPRTUBE) 569
6.31 package EXP3D Export3D 573
6.31.1 Export3D (EXP3D) 574

6.32 package E04AGNT e04AgentsPackage . . . 577
6.32.1 e04AgentsPackage (E04AGNT) . . . 577

7 Chapter F **585**
7.1 package FACTFUNC FactoredFunctions . . . 585
7.1.1 FactoredFunctions (FACTFUNC) . . . 585
7.2 package FR2 FactoredFunctions2 . . . 587
7.2.1 FactoredFunctions2 (FR2) . . . 589
7.3 package FRUTIL FactoredFunctionUtilities . . . 591
7.3.1 FactoredFunctionUtilities (FRUTIL) . . . 591
7.4 package FACUTIL FactoringUtilities . . . 593
7.4.1 FactoringUtilities (FACUTIL) . . . 593
7.5 package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRational-Number . . . 595
7.5.1 FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber (FACTEXT) . . . 596
7.6 package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber . . . 599
7.6.1 FactorisationOverPseudoAlgebraicClosureOfRationalNumber (FACTRN) 601
7.7 package FGLMICPK FGLMIfCanPackage . . . 604
7.7.1 FGLMIfCanPackage (FGLMICPK) . . . 604
7.8 package FORDER FindOrderFinite . . . 606
7.8.1 FindOrderFinite (FORDER) . . . 606
7.9 package FAMR2 FiniteAbelianMonoidRingFunctions2 . . . 608
7.9.1 FiniteAbelianMonoidRingFunctions2 (FAMR2) . . . 608
7.10 package FDIV2 FiniteDivisorFunctions2 . . . 609
7.10.1 FiniteDivisorFunctions2 (FDIV2) . . . 609
7.11 package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect . . . 610
7.11.1 FiniteFieldFactorizationWithSizeParseBySideEffect (FFFACTSE) . . . 611
7.12 package FFF FiniteFieldFunctions . . . 617
7.12.1 FiniteFieldFunctions (FFF) . . . 617
7.13 package FFHOM FiniteFieldHomomorphisms . . . 622
7.13.1 FiniteFieldHomomorphisms (FFHOM) . . . 622
7.14 package FFPOLY FiniteFieldPolynomialPackage . . . 630
7.14.1 FiniteFieldPolynomialPackage (FFPOLY) . . . 630
7.15 package FFPOLY2 FiniteFieldPolynomialPackage2 . . . 651
7.15.1 FiniteFieldPolynomialPackage2 (FFPOLY2) . . . 651
7.16 package FFSLPE FiniteFieldSolveLinearPolynomialEquation . . . 654
7.16.1 FiniteFieldSolveLinearPolynomialEquation (FFSLPE) . . . 654
7.17 package FFSQFR FiniteFieldSquareFreeDecomposition . . . 655
7.17.1 FiniteFieldSquareFreeDecomposition (FFSQFR) . . . 656
7.18 package FLAGG2 FiniteLinearAggregateFunctions2 . . . 659
7.18.1 FiniteLinearAggregateFunctions2 (FLAGG2) . . . 659
7.19 package FLASORT FiniteLinearAggregateSort . . . 662
7.19.1 FiniteLinearAggregateSort (FLASORT) . . . 662
7.20 package FSAGG2 FiniteSetAggregateFunctions2 . . . 665

7.20.1 FiniteSetAggregateFunctions2 (FSAGG2) . . . 665
7.21 package FLOATCP FloatingComplexPackage . . . 666
7.21.1 FloatingComplexPackage (FLOATCP) . . . 666
7.22 package FLOATRP FloatingRealPackage . . . 670
7.22.1 FloatingRealPackage (FLOATRP) . . . 670
7.23 package FCPAK1 FortranCodePackage1 . . . 673
7.23.1 FortranCodePackage1 (FCPAK1) . . . 673
7.24 package FOP FortranOutputStackPackage . . . 676
7.24.1 FortranOutputStackPackage (FOP) . . . 676
7.25 package FORT FortranPackage . . . 679
7.25.1 FortranPackage (FORT) . . . 679
7.26 package FRIDEAL2 FractionalIdealFunctions2 . . . 681
7.26.1 FractionalIdealFunctions2 (FRIDEAL2) . . . 681
7.27 package FFFG FractionFreeFastGaussian . . . 683
7.27.1 FractionFreeFastGaussian (FFFG) . . . 683
7.28 package FFFGF FractionFreeFastGaussianFractions . . . 693
7.28.1 FractionFreeFastGaussianFractions (FFFGF) . . . 693
7.29 package FRAC2 FractionFunctions2 . . . 696
7.29.1 FractionFunctions2 (FRAC2) . . . 696
7.30 package FRNAAF2 FramedNonAssociativeAlgebraFunctions2 . . . 697
7.30.1 FramedNonAssociativeAlgebraFunctions2 (FRNAAF2) . . . 697
7.31 package FSPECF FunctionalSpecialFunction . . . 699
7.31.1 FunctionalSpecialFunction (FSPECF) . . . 699
7.31.2 differentiation of special functions . . . 705
7.32 package FFCAT2 FunctionFieldCategoryFunctions2 . . . 708
7.32.1 FunctionFieldCategoryFunctions2 (FFCAT2) . . . 708
7.33 package FFINTBAS FunctionFieldIntegralBasis . . . 709
7.33.1 FunctionFieldIntegralBasis (FFINTBAS) . . . 709
7.34 package PMASSFS FunctionSpaceAssertions . . . 712
7.34.1 FunctionSpaceAssertions (PMASSFS) . . . 712
7.35 package PMPREDFS FunctionSpaceAttachPredicates . . . 714
7.35.1 FunctionSpaceAttachPredicates (PMPREDFS) . . . 714
7.36 package FSCINT FunctionSpaceComplexIntegration . . . 716
7.36.1 FunctionSpaceComplexIntegration (FSCINT) . . . 716
7.37 package FS2 FunctionSpaceFunctions2 . . . 718
7.37.1 FunctionSpaceFunctions2 (FS2) . . . 718
7.38 package FSINT FunctionSpaceIntegration . . . 720
7.38.1 FunctionSpaceIntegration (FSINT) . . . 720
7.39 package FSPRMELT FunctionSpacePrimitiveElement . . . 723
7.39.1 FunctionSpacePrimitiveElement (FSPRMELT) . . . 723
7.40 package FSRED FunctionSpaceReduce . . . 726
7.40.1 FunctionSpaceReduce (FSRED) . . . 726
7.41 package SUMFS FunctionSpaceSum . . . 728
7.41.1 FunctionSpaceSum (SUMFS) . . . 728
7.42 package FS2EXPXP FunctionSpaceToExponentialExpansion . . . 730
7.42.1 FunctionSpaceToExponentialExpansion (FS2EXPXP) . . . 730

7.43 package FS2UPS FunctionSpaceToUnivariatePowerSeries . . . 741
7.43.1 FunctionSpaceToUnivariatePowerSeries (FS2UPS) . . . 741
7.44 package FSUPFACT FunctionSpaceUnivariatePolynomialFactor . . . 757
7.44.1 FunctionSpaceUnivariatePolynomialFactor (FSUPFACT) . . . 757

8 Chapter G **763**
8.1 package GALFACTU GaloisGroupFactorizationUtilities . . . 763
8.1.1 GaloisGroupFactorizationUtilities (GALFACTU) . . . 763
8.2 package GALFACT GaloisGroupFactorizer . . . 767
8.2.1 GaloisGroupFactorizer (GALFACT) . . . 767
8.3 package GALPOLYU GaloisGroupPolynomialUtilities . . . 784
8.3.1 GaloisGroupPolynomialUtilities (GALPOLYU) . . . 784
8.4 package GALUTIL GaloisGroupUtilities . . . 787
8.4.1 GaloisGroupUtilities (GALUTIL) . . . 787
8.5 package GAUSSFAC GaussianFactorizationPackage . . . 790
8.5.1 GaussianFactorizationPackage (GAUSSFAC) . . . 790
8.6 package GHENSEL GeneralHenselPackage . . . 794
8.6.1 GeneralHenselPackage (GHENSEL) . . . 794
8.7 package GENMFACT GeneralizedMultivariateFactorize . . . 798
8.7.1 GeneralizedMultivariateFactorize (GENMFACT) . . . 798
8.8 package GPAFF GeneralPackageForAlgebraicFunctionField . . . 799
8.8.1 GeneralPackageForAlgebraicFunctionField (GPAFF) . . . 801
8.9 package GENPGCD GeneralPolynomialGcdPackage . . . 815
8.9.1 GeneralPolynomialGcdPackage (GENPGCD) . . . 815
8.10 package GENUPS GenerateUnivariatePowerSeries . . . 829
8.10.1 GenerateUnivariatePowerSeries (GENUPS) . . . 829
8.11 package GENEEZ GenExEuclid . . . 833
8.11.1 GenExEuclid (GENEEZ) . . . 833
8.12 package GENUFACT GenUFactorize . . . 838
8.12.1 GenUFactorize (GENUFACT) . . . 838
8.13 package INTG0 GenusZeroIntegration . . . 840
8.13.1 GenusZeroIntegration (INTG0) . . . 840
8.14 package GDRAW GnuDraw . . . 846
8.14.1 GnuDraw (GDRAW) . . . 847
8.15 package GOSPER GosperSummationMethod . . . 849
8.15.1 GosperSummationMethod (GOSPER) . . . 849
8.16 package GRDEF GraphicsDefaults . . . 854
8.16.1 GraphicsDefaults (GRDEF) . . . 854
8.17 package GRAY GrayCode . . . 857
8.17.1 GrayCode (GRAY) . . . 857
8.18 package GBF GroebnerFactorizationPackage . . . 859
8.18.1 GroebnerFactorizationPackage (GBF) . . . 863
8.19 package GBINTERN GroebnerInternalPackage . . . 871
8.19.1 GroebnerInternalPackage (GBINTERN) . . . 871
8.20 package GB GroebnerPackage . . . 881
8.20.1 GroebnerPackage (GB) . . . 907

8.21 package GROEBSOL GroebnerSolve . . . 911
8.21.1 GroebnerSolve (GROEBSOL) . . . 911
8.22 package GUESS Guess . . . 915
8.22.1 Guess (GUESS) . . . 916
8.23 package GUESSAN GuessAlgebraicNumber . . . 951
8.23.1 GuessAlgebraicNumber (GUESSAN) . . . 951
8.24 package GUESSF GuessFinite . . . 952
8.24.1 GuessFinite (GUESSF) . . . 952
8.25 package GUESSF1 GuessFiniteFunctions . . . 953
8.25.1 GuessFiniteFunctions (GUESSF1) . . . 953
8.26 package GUESSINT GuessInteger . . . 954
8.26.1 GuessInteger (GUESSINT) . . . 954
8.27 package GUESSP GuessPolynomial . . . 955
8.27.1 GuessPolynomial (GUESSP) . . . 955
8.28 package GUESSUP GuessUnivariatePolynomial . . . 956
8.28.1 GuessUnivariatePolynomial (GUESSUP) . . . 956

9 Chapter H 963
9.1 package HB HallBasis . . . 963
9.1.1 HallBasis (HB) . . . 963
9.2 package HEUGCD HeuGcd . . . 966
9.2.1 HeuGcd (HEUGCD) . . . 966

10 Chapter I 973
10.1 package IDECOMP IdealDecompositionPackage . . . 973
10.1.1 IdealDecompositionPackage (IDECOMP) . . . 973
10.2 package INCRMAPS IncrementingMaps . . . 982
10.2.1 IncrementingMaps (INCRMAPS) . . . 982
10.3 package INFPROD0 InfiniteProductCharacteristicZero . . . 983
10.3.1 InfiniteProductCharacteristicZero (INFPROD0) . . . 983
10.4 package INPRODFF InfiniteProductFiniteField . . . 985
10.4.1 InfiniteProductFiniteField (INPRODFF) . . . 985
10.5 package INPRODPF InfiniteProductPrimeField . . . 988
10.5.1 InfiniteProductPrimeField (INPRODPF) . . . 988
10.6 package ITFUN2 InfiniteTupleFunctions2 . . . 990
10.6.1 InfiniteTupleFunctions2 (ITFUN2) . . . 990
10.7 package ITFUN3 InfiniteTupleFunctions3 . . . 991
10.7.1 InfiniteTupleFunctions3 (ITFUN3) . . . 991
10.8 package INFINITY Infinity . . . 992
10.8.1 Infinity (INFINITY) . . . 992
10.9 package IALGFACT InnerAlgFactor . . . 993
10.9.1 InnerAlgFactor (IALGFACT) . . . 993
10.10 package ICDEN InnerCommonDenominator . . . 996
10.10.1 InnerCommonDenominator (ICDEN) . . . 996
10.11 package IMATLIN InnerMatrixLinearAlgebraFunctions . . . 998
10.11.1 InnerMatrixLinearAlgebraFunctions (IMATLIN) . . . 998

10.12package IMATQF InnerMatrixQuotientFieldFunctions . . . 1003
10.12.1 InnerMatrixQuotientFieldFunctions (IMATQF) . . . 1003
10.13package INMODGCD InnerModularGcd . . . 1005
10.13.1 InnerModularGcd (INMODGCD) . . . 1005
10.14package INNMFACT InnerMultFact . . . 1011
10.14.1 InnerMultFact (INNMFACT) . . . 1011
10.15package INBFF InnerNormalBasisFieldFunctions . . . 1020
10.15.1 InnerNormalBasisFieldFunctions (INBFF) . . . 1020
10.16package INEP InnerNumericEigenPackage . . . 1028
10.16.1 InnerNumericEigenPackage (INEP) . . . 1028
10.17package INFSP InnerNumericFloatSolvePackage . . . 1033
10.17.1 InnerNumericFloatSolvePackage (INFSP) . . . 1033
10.18package INPSIGN InnerPolySign . . . 1037
10.18.1 InnerPolySign (INPSIGN) . . . 1037
10.19package ISUMP InnerPolySum . . . 1039
10.19.1 InnerPolySum (ISUMP) . . . 1039
10.20package ITRIGMNP InnerTrigonometricManipulations . . . 1041
10.20.1 InnerTrigonometricManipulations (ITRIGMNP) . . . 1041
10.21package INFORM1 InputFormFunctions1 . . . 1045
10.21.1 InputFormFunctions1 (INFORM1) . . . 1045
10.22package INTERGB InterfaceGroebnerPackage . . . 1046
10.22.1 InterfaceGroebnerPackage (INTERGB) . . . 1047
10.23package INTBIT IntegerBits . . . 1049
10.23.1 IntegerBits (INTBIT) . . . 1049
10.24package COMBINAT IntegerCombinatoricFunctions . . . 1050
10.24.1 IntegerCombinatoricFunctions (COMBINAT) . . . 1053
10.25package INTFACT IntegerFactorizationPackage . . . 1056
10.25.1 IntegerFactorizationPackage (INTFACT) . . . 1056
10.25.2 squareFree . . . 1057
10.25.3 PollardSmallFactor . . . 1058
10.25.4 BasicSieve . . . 1060
10.25.5 BasicMethod . . . 1061
10.25.6 factor . . . 1062
10.26package ZLINDEP IntegerLinearDependence . . . 1063
10.26.1 IntegerLinearDependence (ZLINDEP) . . . 1067
10.27package INTHEORY IntegerNumberTheoryFunctions . . . 1068
10.27.1 IntegerNumberTheoryFunctions (INTHEORY) . . . 1082
10.28package PRIMES IntegerPrimesPackage . . . 1087
10.28.1 IntegerPrimesPackage (PRIMES) . . . 1088
10.28.2 smallPrimes . . . 1089
10.28.3 primes . . . 1094
10.28.4 rabinProvesCompositeSmall . . . 1094
10.28.5 rabinProvesComposite . . . 1095
10.28.6 prime? . . . 1095
10.28.7 nextPrime . . . 1097
10.28.8 prevPrime . . . 1097

10.29 package INTRET IntegerRetractions . . . 1098
10.29.1 IntegerRetractions (INTRET) . . . 1098
10.30 package IROOT IntegerRoots . . . 1099
10.30.1 IntegerRoots (IROOT) . . . 1099
10.30.2 perfectSquare? . . . 1100
10.30.3 perfectNthPower? . . . 1101
10.30.4 perfectNthRoot . . . 1101
10.30.5 approxNthRoot . . . 1101
10.30.6 perfectNthRoot . . . 1102
10.30.7 perfectSqrt . . . 1102
10.30.8 approxSqrt . . . 1102
10.31 package INTSLPE IntegerSolveLinearPolynomialEquation . . . 1103
10.31.1 IntegerSolveLinearPolynomialEquation (INTSLPE) . . . 1103
10.32 package IBATOOL IntegralBasisTools . . . 1105
10.32.1 IntegralBasisTools (IBATOOL) . . . 1105
10.33 package IBPTOOLS IntegralBasisPolynomialTools . . . 1108
10.33.1 IntegralBasisPolynomialTools (IBPTOOLS) . . . 1108
10.34 package IR2 IntegrationResultFunctions2 . . . 1111
10.34.1 IntegrationResultFunctions2 (IR2) . . . 1111
10.35 package IRRF2F IntegrationResultRFToFunction . . . 1113
10.35.1 IntegrationResultRFToFunction (IRRF2F) . . . 1113
10.36 package IR2F IntegrationResultToFunction . . . 1115
10.36.1 IntegrationResultToFunction (IR2F) . . . 1115
10.37 package INTTOOLS IntegrationTools . . . 1120
10.37.1 IntegrationTools (INTTOOLS) . . . 1120
10.38 package IPRNTPK InternalPrintPackage . . . 1124
10.38.1 InternalPrintPackage (IPRNTPK) . . . 1124
10.39 package IRURPK InternalRationalUnivariateRepresentationPackage . . . 1125
10.39.1 InternalRationalUnivariateRepresentationPackage (IRURPK) . . . 1125
10.40 package INTFRSP InterpolateFormsPackage . . . 1129
10.40.1 InterpolateFormsPackage (INTFRSP) . . . 1131
10.41 package INTDIVP IntersectionDivisorPackage . . . 1137
10.41.1 IntersectionDivisorPackage (INTDIVP) . . . 1138
10.42 package IRREDFFX IrredPolyOverFiniteField . . . 1141
10.42.1 IrredPolyOverFiniteField (IRREDFFX) . . . 1141
10.43 package IRSN IrrRepSymNatPackage . . . 1143
10.43.1 IrrRepSymNatPackage (IRSN) . . . 1143
10.44 package INVLAPLA InverseLaplaceTransform . . . 1150
10.44.1 InverseLaplaceTransform (INVLAPLA) . . . 1150

11 Chapter J **1153**

12 Chapter K **1155**
12.1 package KERNEL2 KernelFunctions2 1155
12.1.1 KernelFunctions2 (KERNEL2) 1155
12.2 package KOVACIC Kovacic 1156
12.2.1 Kovacic (KOVACIC) 1156

13 Chapter L **1159**
13.1 package LAPLACE LaplaceTransform 1159
13.1.1 LaplaceTransform (LAPLACE) 1159
13.2 package LAZM3PK LazardSetSolvingPackage 1164
13.2.1 LazardSetSolvingPackage (LAZM3PK) 1184
13.3 package LEADCDET LeadingCoefDetermination 1188
13.3.1 LeadingCoefDetermination (LEADCDET) 1188
13.4 package LEXTRIPK LexTriangularPackage 1191
13.4.1 LexTriangularPackage (LEXTRIPK) 1260
13.5 package LINDEP LinearDependence 1265
13.5.1 LinearDependence (LINDEP) 1265
13.6 package LODOF LinearOrdinaryDifferentialOperatorFactorizer 1267
13.6.1 LinearOrdinaryDifferentialOperatorFactorizer (LODOF) 1267
13.7 package LODOOPS LinearOrdinaryDifferentialOperatorsOps 1271
13.7.1 LinearOrdinaryDifferentialOperatorsOps (LODOOPS) 1271
13.8 package LPEFRAC LinearPolynomialEquationByFractions 1274
13.8.1 LinearPolynomialEquationByFractions (LPEFRAC) 1274
13.9 package LISYSER LinearSystemFromPowerSeriesPackage 1275
13.9.1 LinearSystemFromPowerSeriesPackage (LISYSER) 1277
13.10 package LSMP LinearSystemMatrixPackage 1279
13.10.1 LinearSystemMatrixPackage (LSMP) 1279
13.11 package LSMP1 LinearSystemMatrixPackage1 1281
13.11.1 LinearSystemMatrixPackage1 (LSMP1) 1281
13.12 package LSPP LinearSystemPolynomialPackage 1283
13.12.1 LinearSystemPolynomialPackage (LSPP) 1283
13.13 package LGROBP LinGroebnerPackage 1285
13.13.1 LinGroebnerPackage (LGROBP) 1285
13.14 package LOP LinesOpPack 1292
13.14.1 LinesOpPack (LOP) 1293
13.15 package LF LiouvillianFunction 1296
13.15.1 LiouvillianFunction (LF) 1296
13.16 package LIST2 ListFunctions2 1301
13.16.1 ListFunctions2 (LIST2) 1301
13.17 package LIST3 ListFunctions3 1302
13.17.1 ListFunctions3 (LIST3) 1302
13.18 package LIST2MAP ListToMap 1304
13.18.1 ListToMap (LIST2MAP) 1304
13.19 package LPARSPT LocalParametrizationOfSimplePointPackage 1306
13.19.1 LocalParametrizationOfSimplePointPackage (LPARSPT) 1307

14 Chapter M **1315**
14.1 package MKBCFUNC MakeBinaryCompiledFunction 1315
14.1.1 MakeBinaryCompiledFunction (MKBCFUNC) 1315
14.2 package MKFLCFN MakeFloatCompiledFunction 1317
14.2.1 MakeFloatCompiledFunction (MKFLCFN) 1317
14.3 package MKFUNC MakeFunction 1320
14.3.1 MakeFunction (MKFUNC) 1324
14.4 package MKRECORD MakeRecord 1326
14.4.1 MakeRecord (MKRECORD) 1326
14.5 package MKUCFUNC MakeUnaryCompiledFunction 1327
14.5.1 MakeUnaryCompiledFunction (MKUCFUNC) 1327
14.6 package MAPHACK1 MappingPackageInternalHacks1 1328
14.6.1 MappingPackageInternalHacks1 (MAPHACK1) 1328
14.7 package MAPHACK2 MappingPackageInternalHacks2 1330
14.7.1 MappingPackageInternalHacks2 (MAPHACK2) 1330
14.8 package MAPHACK3 MappingPackageInternalHacks3 1331
14.8.1 MappingPackageInternalHacks3 (MAPHACK3) 1331
14.9 package MAPPKG1 MappingPackage1 1332
14.9.1 MappingPackage1 (MAPPKG1) 1341
14.10package MAPPKG2 MappingPackage2 1343
14.10.1 MappingPackage2 (MAPPKG2) 1352
14.11package MAPPKG3 MappingPackage3 1353
14.11.1 MappingPackage3 (MAPPKG3) 1362
14.12package MAPPKG4 MappingPackage4 1364
14.12.1 MappingPackage4 (MAPPKG4) 1369
14.13package MATCAT2 MatrixCategoryFunctions2 1371
14.13.1 MatrixCategoryFunctions2 (MATCAT2) 1371
14.14package MCDEN MatrixCommonDenominator 1373
14.14.1 MatrixCommonDenominator (MCDEN) 1373
14.15package MATLIN MatrixLinearAlgebraFunctions 1375
14.15.1 MatrixLinearAlgebraFunctions (MATLIN) 1375
14.16package MTHING MergeThing 1382
14.16.1 MergeThing (MTHING) 1382
14.17package MESH MeshCreationRoutinesForThreeDimensions 1384
14.17.1 MeshCreationRoutinesForThreeDimensions (MESH) 1384
14.18package MDDFACT ModularDistinctDegreeFactorizer 1387
14.18.1 ModularDistinctDegreeFactorizer (MDDFACT) 1387
14.19package MHROWRED ModularHermitianRowReduction 1393
14.19.1 ModularHermitianRowReduction (MHROWRED) 1393
14.20package MRF2 MonoidRingFunctions2 1398
14.20.1 MonoidRingFunctions2 (MRF2) 1398
14.21package MONOTOOL MonomialExtensionTools 1400
14.21.1 MonomialExtensionTools (MONOTOOL) 1400
14.22package MSYSCMD MoreSystemCommands 1402
14.22.1 MoreSystemCommands (MSYSCMD) 1402
14.23package MPCPF MPolyCatPolyFactorizer 1403

14.23.1 MPolyCatPolyFactorizer (MPCPF) . . . 1403
14.24 package MPRFF MPolyCatRationalFunctionFactorizer . . . 1405
14.24.1 MPolyCatRationalFunctionFactorizer (MPRFF) . . . 1405
14.25 package MPC2 MPolyCatFunctions2 . . . 1409
14.25.1 MPolyCatFunctions2 (MPC2) . . . 1409
14.26 package MPC3 MPolyCatFunctions3 . . . 1410
14.26.1 MPolyCatFunctions3 (MPC3) . . . 1410
14.27 package MRATFAC MRationalFactorize . . . 1412
14.27.1 MRationalFactorize (MRATFAC) . . . 1412
14.28 package MFINFACT MultFiniteFactorize . . . 1414
14.28.1 MultFiniteFactorize (MFINFACT) . . . 1414
14.29 package MMAP MultipleMap . . . 1424
14.29.1 MultipleMap (MMAP) . . . 1424
14.30 package MCALCFN MultiVariableCalculusFunctions . . . 1426
14.30.1 MultiVariableCalculusFunctions (MCALCFN) . . . 1426
14.31 package MULTFACT MultivariateFactorize . . . 1430
14.31.1 MultivariateFactorize (MULTFACT) . . . 1430
14.32 package MLIFT MultivariateLifting . . . 1431
14.33 package MULTSQFR MultivariateSquareFree . . . 1436
14.33.1 MultivariateSquareFree (MULTSQFR) . . . 1436

15 Chapter N 1445
15.1 package NAGF02 NagEigenPackage . . . 1445
15.1.1 NagEigenPackage (NAGF02) . . . 1511
15.2 package NAGE02 NagFittingPackage . . . 1523
15.2.1 NagFittingPackage (NAGE02) . . . 1652
15.3 package NAGF04 NagLinearEquationSolvingPackage . . . 1665
15.3.1 NagLinearEquationSolvingPackage (NAGF04) . . . 1731
15.4 package NAGSP NAGLinkSupportPackage . . . 1739
15.4.1 NAGLinkSupportPackage (NAGSP) . . . 1739
15.5 package NAGD01 NagIntegrationPackage . . . 1741
15.5.1 NagIntegrationPackage (NAGD01) . . . 1818
15.6 package NAGE01 NagInterpolationPackage . . . 1827
15.6.1 NagInterpolationPackage (NAGE01) . . . 1866
15.7 package NAGF07 NagLapack . . . 1873
15.7.1 NagLapack (NAGF07) . . . 1887
15.8 package NAGF01 NagMatrixOperationsPackage . . . 1890
15.8.1 NagMatrixOperationsPackage (NAGF01) . . . 1946
15.9 package NAGE04 NagOptimisationPackage . . . 1953
15.9.1 NagOptimisationPackage (NAGE04) . . . 2104
15.10 package NAGD02 NagOrdinaryDifferentialEquationsPackage . . . 2113
15.10.1 NagOrdinaryDifferentialEquationsPackage (NAGD02) . . . 2203
15.11 package NAGD03 NagPartialDifferentialEquationsPackage . . . 2213
15.11.1 NagPartialDifferentialEquationsPackage (NAGD03) . . . 2249
15.12 package NAGC02 NagPolynomialRootsPackage . . . 2253
15.12.1 NagPolynomialRootsPackage (NAGC02) . . . 2267

15.13package NAGC05 NagRootFindingPackage 2269
15.13.1 NagRootFindingPackage (NAGC05) 2286
15.14package NAGC06 NagSeriesSummationPackage 2289
15.14.1 NagSeriesSummationPackage (NAGC06) 2333
15.15package NAGS NagSpecialFunctionsPackage 2340
15.15.1 NagSpecialFunctionsPackage (NAGS) 2486
15.16package NSUP2 NewSparseUnivariatePolynomialFunctions2 2502
15.16.1 NewSparseUnivariatePolynomialFunctions2 (NSUP2) 2502
15.17package NEWTON NewtonInterpolation 2504
15.17.1 NewtonInterpolation (NEWTON) 2504
15.18package NPOLYGON NewtonPolygon 2505
15.18.1 NewtonPolygon (NPOLYGON) 2506
15.19package NCODIV NonCommutativeOperatorDivision 2511
15.19.1 NonCommutativeOperatorDivision (NCODIV) 2511
15.20package NONE1 NoneFunctions1 2514
15.20.1 NoneFunctions1 (NONE1) 2514
15.21package NODE1 NonLinearFirstOrderODESolver 2515
15.21.1 NonLinearFirstOrderODESolver (NODE1) 2515
15.22package NLINSOL NonLinearSolvePackage 2519
15.22.1 NonLinearSolvePackage (NLINSOL) 2519
15.23package NORMPK NormalizationPackage 2521
15.23.1 NormalizationPackage (NORMPK) 2521
15.24package NORMMA NormInMonogenicAlgebra 2526
15.24.1 NormInMonogenicAlgebra (NORMMA) 2526
15.25package NORMRETR NormRetractPackage 2528
15.25.1 NormRetractPackage (NORMRETR) 2528
15.26package NPCOEF NPCoef 2530
15.26.1 NPCoef (NPCOEF) 2530
15.27package NFINTBAS NumberFieldIntegralBasis 2534
15.27.1 NumberFieldIntegralBasis (NFINTBAS) 2534
15.28package NUMFMT NumberFormats 2539
15.28.1 NumberFormats (NUMFMT) 2539
15.29package NTPOLFN NumberTheoreticPolynomialFunctions 2544
15.29.1 NumberTheoreticPolynomialFunctions (NTPOLFN) 2544
15.30package NUMERIC Numeric 2546
15.30.1 Numeric (NUMERIC) 2546
15.31package NUMODE NumericalOrdinaryDifferentialEquations 2555
15.31.1 NumericalOrdinaryDifferentialEquations (NUMODE) 2555
15.32package NUMQUAD NumericalQuadrature 2563
15.32.1 NumericalQuadrature (NUMQUAD) 2563
15.33package NCEP NumericComplexEigenPackage 2575
15.33.1 NumericComplexEigenPackage (NCEP) 2575
15.34package NCNTFRAC NumericContinuedFraction 2577
15.34.1 NumericContinuedFraction (NCNTFRAC) 2577
15.35package NREP NumericRealEigenPackage 2579
15.35.1 NumericRealEigenPackage (NREP) 2579

15.36 package NUMTUBE NumericTubePlot . . . 2581
15.36.1 NumericTubePlot (NUMTUBE) . . . 2581

16 Chapter O **2585**
16.1 package OCTCT2 OctonionCategoryFunctions2 . . . 2585
16.1.1 OctonionCategoryFunctions2 (OCTCT2) . . . 2585
16.2 package ODEINT ODEIntegration . . . 2587
16.2.1 ODEIntegration (ODEINT) . . . 2587
16.3 package ODETOOLS ODETools . . . 2589
16.3.1 ODETools (ODETOOLS) . . . 2589
16.4 package ARRAY12 OneDimensionalArrayFunctions2 . . . 2591
16.4.1 OneDimensionalArrayFunctions2 (ARRAY12) . . . 2591
16.5 package ONECOMP2 OnePointCompletionFunctions2 . . . 2593
16.5.1 OnePointCompletionFunctions2 (ONECOMP2) . . . 2593
16.6 package OMPKG OpenMathPackage . . . 2594
16.6.1 OpenMathPackage (OMPKG) . . . 2594
16.7 package OMSERVER OpenMathServerPackage . . . 2597
16.7.1 OpenMathServerPackage (OMSERVER) . . . 2597
16.8 package OPQUERY OperationsQuery . . . 2599
16.8.1 OperationsQuery (OPQUERY) . . . 2599
16.9 package ORDCOMP2 OrderedCompletionFunctions2 . . . 2600
16.9.1 OrderedCompletionFunctions2 (ORDCOMP2) . . . 2600
16.10 package ORDFUNS OrderingFunctions . . . 2601
16.10.1 OrderingFunctions (ORDFUNS) . . . 2601
16.11 package ORTHPOL OrthogonalPolynomialFunctions . . . 2603
16.11.1 OrthogonalPolynomialFunctions (ORTHPOL) . . . 2603
16.12 package OUT OutputPackage . . . 2606
16.12.1 OutputPackage (OUT) . . . 2606

17 Chapter P **2609**
17.1 package PAFF PackageForAlgebraicFunctionField . . . 2609
17.1.1 PackageForAlgebraicFunctionField (PAFF) . . . 2611
17.2 package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField . . . 2617
17.2.1 PackageForAlgebraicFunctionFieldOverFiniteField (PAFFFF) . . . 2619
17.3 package PFORP PackageForPoly . . . 2627
17.3.1 PackageForPoly (PFORP) . . . 2628
17.4 package PADEPAC PadeApproximantPackage . . . 2635
17.4.1 PadeApproximantPackage (PADEPAC) . . . 2635
17.5 package PADE PadeApproximants . . . 2636
17.5.1 PadeApproximants (PADE) . . . 2637
17.6 package PWFFINTB PAdicWildFunctionFieldIntegralBasis . . . 2641
17.6.1 PAdicWildFunctionFieldIntegralBasis (PWFFINTB) . . . 2641
17.7 package YSTREAM ParadoxicalCombinatorsForStreams . . . 2646
17.7.1 ParadoxicalCombinatorsForStreams (YSTREAM) . . . 2646
17.8 package PLEQN ParametricLinearEquations . . . 2648
17.8.1 ParametricLinearEquations (PLEQN) . . . 2648

17.9 package PARPC2 ParametricPlaneCurveFunctions2 2662
17.9.1 ParametricPlaneCurveFunctions2 (PARPC2) 2662
17.10package PARSC2 ParametricSpaceCurveFunctions2 2663
17.10.1 ParametricSpaceCurveFunctions2 (PARSC2) 2663
17.11package PARSU2 ParametricSurfaceFunctions2 2664
17.11.1 ParametricSurfaceFunctions2 (PARSU2) 2664
17.12package PARAMP ParametrizationPackage 2665
17.12.1 ParametrizationPackage (PARAMP) 2666
17.13package PFRPAC PartialFractionPackage 2668
17.13.1 PartialFractionPackage (PFRPAC) 2670
17.14package PARTPERM PartitionsAndPermutations 2672
17.14.1 PartitionsAndPermutations (PARTPERM) 2672
17.15package PATTERN1 PatternFunctions1 2675
17.15.1 PatternFunctions1 (PATTERN1) 2675
17.16package PATTERN2 PatternFunctions2 2677
17.16.1 PatternFunctions2 (PATTERN2) 2677
17.17package PATMATCH PatternMatch 2678
17.17.1 PatternMatch (PATMATCH) 2678
17.18package PMASS PatternMatchAssertions 2681
17.18.1 PatternMatchAssertions (PMASS) 2681
17.19package PMFS PatternMatchFunctionSpace 2682
17.19.1 PatternMatchFunctionSpace (PMFS) 2682
17.20package PMINS PatternMatchIntegerNumberSystem 2684
17.20.1 PatternMatchIntegerNumberSystem (PMINS) 2684
17.21package INTPM PatternMatchIntegration 2687
17.21.1 PatternMatchIntegration (INTPM) 2687
17.22package PMKERNEL PatternMatchKernel 2694
17.22.1 PatternMatchKernel (PMKERNEL) 2694
17.23package PMLSAGG PatternMatchListAggregate 2697
17.23.1 PatternMatchListAggregate (PMLSAGG) 2697
17.24package PMPLCAT PatternMatchPolynomialCategory 2698
17.24.1 PatternMatchPolynomialCategory (PMPLCAT) 2698
17.25package PMDOWN PatternMatchPushDown 2701
17.25.1 PatternMatchPushDown (PMDOWN) 2701
17.26package PMQFCAT PatternMatchQuotientFieldCategory 2704
17.26.1 PatternMatchQuotientFieldCategory (PMQFCAT) 2704
17.27package PATRES2 PatternMatchResultFunctions2 2705
17.27.1 PatternMatchResultFunctions2 (PATRES2) 2705
17.28package PMSYM PatternMatchSymbol 2707
17.28.1 PatternMatchSymbol (PMSYM) 2707
17.29package PMTOOLS PatternMatchTools 2708
17.29.1 PatternMatchTools (PMTOOLS) 2708
17.30package PERMAN Permanent 2712
17.30.1 Permanent (PERMAN) 2714
17.31package PGE PermutationGroupExamples 2718
17.31.1 PermutationGroupExamples (PGE) 2718

17.32package PICOERCE PiCoercions . . . 2726
17.32.1 PiCoercions (PICOERCE) . . . 2726
17.33package PLOT1 PlotFunctions1 . . . 2728
17.33.1 PlotFunctions1 (PLOT1) . . . 2728
17.34package PLOTTOOL PlotTools . . . 2729
17.34.1 PlotTools (PLOTTOOL) . . . 2729
17.35package PRJALGPK ProjectiveAlgebraicSetPackage . . . 2731
17.35.1 ProjectiveAlgebraicSetPackage (PRJALGPK) . . . 2733
17.36package PTFUNC2 PointFunctions2 . . . 2737
17.36.1 PointFunctions2 (PTFUNC2) . . . 2737
17.37package PTPACK PointPackage . . . 2738
17.37.1 PointPackage (PTPACK) . . . 2738
17.38package PFO PointsOfFiniteOrder . . . 2740
17.38.1 PointsOfFiniteOrder (PFO) . . . 2740
17.39package PFOQ PointsOfFiniteOrderRational . . . 2747
17.39.1 PointsOfFiniteOrderRational (PFOQ) . . . 2747
17.40package PFOTOOLS PointsOfFiniteOrderTools . . . 2749
17.40.1 PointsOfFiniteOrderTools (PFOTOOLS) . . . 2749
17.41package PLPKCRV PolynomialPackageForCurve . . . 2751
17.41.1 PolynomialPackageForCurve (PLPKCRV) . . . 2752
17.42package POLTOPOL PolToPol . . . 2754
17.42.1 PolToPol (POLTOPOL) . . . 2754
17.43package PGROEB PolyGroebner . . . 2757
17.43.1 PolyGroebner (PGROEB) . . . 2757
17.44package PAN2EXPR PolynomialAN2Expression . . . 2759
17.44.1 PolynomialAN2Expression (PAN2EXPR) . . . 2759
17.45package POLYLIFT PolynomialCategoryLifting . . . 2760
17.45.1 PolynomialCategoryLifting (POLYLIFT) . . . 2760
17.46package POLYCATQ PolynomialCategoryQuotientFunctions . . . 2762
17.46.1 PolynomialCategoryQuotientFunctions (POLYCATQ) . . . 2762
17.47package PCOMP PolynomialComposition . . . 2765
17.47.1 PolynomialComposition (PCOMP) . . . 2765
17.48package PDECOMP PolynomialDecomposition . . . 2766
17.48.1 PolynomialDecomposition (PDECOMP) . . . 2766
17.49package PFBR PolynomialFactorizationByRecursion . . . 2768
17.49.1 PolynomialFactorizationByRecursion (PFBR) . . . 2768
17.50package PFBRU PolynomialFactorizationByRecursionUnivariate . . . 2775
17.50.1 PolynomialFactorizationByRecursionUnivariate (PFBRU) . . . 2775
17.51package POLY2 PolynomialFunctions2 . . . 2780
17.51.1 PolynomialFunctions2 (POLY2) . . . 2780
17.52package PGCD PolynomialGcdPackage . . . 2782
17.52.1 PolynomialGcdPackage (PGCD) . . . 2782
17.53package PINTERP PolynomialInterpolation . . . 2790
17.53.1 PolynomialInterpolation (PINTERP) . . . 2790
17.54package PINTERPA PolynomialInterpolationAlgorithms . . . 2792
17.54.1 PolynomialInterpolationAlgorithms (PINTERPA) . . . 2792

17.55package PNTHEORY PolynomialNumberTheoryFunctions 2793
17.55.1 PolynomialNumberTheoryFunctions (PNTHEORY) 2793
17.56package POLYROOT PolynomialRoots 2798
17.56.1 PolynomialRoots (POLYROOT) 2798
17.57package PSETPK PolynomialSetUtilitiesPackage 2801
17.57.1 PolynomialSetUtilitiesPackage (PSETPK) 2801
17.58package SOLVEFOR PolynomialSolveByFormulas 2819
17.58.1 PolynomialSolveByFormulas (SOLVEFOR) 2819
17.59package PSQFR PolynomialSquareFree 2825
17.59.1 PolynomialSquareFree (PSQFR) 2825
17.60package POLY2UP PolynomialToUnivariatePolynomial 2828
17.60.1 PolynomialToUnivariatePolynomial (POLY2UP) 2828
17.61package LIMITPS PowerSeriesLimitPackage 2830
17.61.1 PowerSeriesLimitPackage (LIMITPS) 2830
17.62package PREASSOC PrecomputedAssociatedEquations 2841
17.62.1 PrecomputedAssociatedEquations (PREASSOC) 2841
17.63package PRIMARR2 PrimitiveArrayFunctions2 2844
17.63.1 PrimitiveArrayFunctions2 (PRIMARR2) 2844
17.64package PRIMELT PrimitiveElement 2846
17.64.1 PrimitiveElement (PRIMELT) 2846
17.65package ODEPRIM PrimitiveRatDE 2849
17.65.1 PrimitiveRatDE (ODEPRIM) 2849
17.66package ODEPRRIC PrimitiveRatRicDE 2853
17.66.1 PrimitiveRatRicDE (ODEPRRIC) 2853
17.67package PRINT PrintPackage 2859
17.67.1 PrintPackage (PRINT) 2859
17.68package PSEUDLIN PseudoLinearNormalForm 2860
17.68.1 PseudoLinearNormalForm (PSEUDLIN) 2860
17.69package PRS PseudoRemainderSequence 2864
17.69.1 PseudoRemainderSequence (PRS) 2864
17.70package INTPAF PureAlgebraicIntegration 2883
17.70.1 PureAlgebraicIntegration (INTPAF) 2883
17.71package ODEPAL PureAlgebraicLODE 2892
17.71.1 PureAlgebraicLODE (ODEPAL) 2892
17.72package PUSHVAR PushVariables 2893
17.72.1 PushVariables (PUSHVAR) 2893

18 Chapter Q **2897**
18.1 package QALGSET2 QuasiAlgebraicSet2 2897
18.1.1 QuasiAlgebraicSet2 (QALGSET2) 2897
18.2 package QCMPACK QuasiComponentPackage 2900
18.2.1 QuasiComponentPackage (QCMPACK) 2900
18.3 package QFCAT2 QuotientFieldCategoryFunctions2 2909
18.3.1 QuotientFieldCategoryFunctions2 (QFCAT2) 2909
18.4 package QUATCT2 QuaternionCategoryFunctions2 2910
18.4.1 QuaternionCategoryFunctions2 (QUATCT2) 2912

19 Chapter R **2915**
19.1 package REP RadicalEigenPackage . . . 2915
19.1.1 RadicalEigenPackage (REP) . . . 2915
19.2 package SOLVERAD RadicalSolvePackage . . . 2919
19.2.1 RadicalSolvePackage (SOLVERAD) . . . 2929
19.3 package RADUTIL RadixUtilities . . . 2936
19.3.1 RadixUtilities (RADUTIL) . . . 2936
19.4 package RDIST RandomDistributions . . . 2937
19.4.1 RandomDistributions (RDIST) . . . 2937
19.5 package RFDIST RandomFloatDistributions . . . 2939
19.5.1 RandomFloatDistributions (RFDIST) . . . 2939
19.6 package RIDIST RandomIntegerDistributions . . . 2941
19.6.1 RandomIntegerDistributions (RIDIST) . . . 2941
19.7 package RANDSRC RandomNumberSource . . . 2943
19.7.1 RandomNumberSource (RANDSRC) . . . 2943
19.8 package RATFACT RationalFactorize . . . 2945
19.8.1 RationalFactorize (RATFACT) . . . 2945
19.9 package RF RationalFunction . . . 2947
19.9.1 RationalFunction (RF) . . . 2947
19.10 package DEFINTRF RationalFunctionDefiniteIntegration . . . 2949
19.10.1 RationalFunctionDefiniteIntegration (DEFINTRF) . . . 2949
19.11 package RFFACT RationalFunctionFactor . . . 2952
19.11.1 RationalFunctionFactor (RFFACT) . . . 2952
19.12 package RFFACTOR RationalFunctionFactorizer . . . 2953
19.12.1 RationalFunctionFactorizer (RFFACTOR) . . . 2953
19.13 package INTRF RationalFunctionIntegration . . . 2955
19.13.1 RationalFunctionIntegration (INTRF) . . . 2955
19.14 package LIMITRF RationalFunctionLimitPackage . . . 2957
19.14.1 RationalFunctionLimitPackage (LIMITRF) . . . 2957
19.15 package SIGNRF RationalFunctionSign . . . 2960
19.15.1 RationalFunctionSign (SIGNRF) . . . 2960
19.16 package SUMRF RationalFunctionSum . . . 2963
19.16.1 RationalFunctionSum (SUMRF) . . . 2968
19.17 package INTRAT RationalIntegration . . . 2970
19.17.1 RationalIntegration (INTRAT) . . . 2970
19.18 package RINTERP RationalInterpolation . . . 2972
19.18.1 Introduction . . . 2972
19.18.2 Questions and Outlook . . . 2972
19.18.3 RationalInterpolation (RINTERP) . . . 2973
19.19 package ODERAT RationalLODE . . . 2976
19.19.1 RationalLODE (ODERAT) . . . 2976
19.20 package RATRET RationalRetractions . . . 2981
19.20.1 RationalRetractions (RATRET) . . . 2981
19.21 package ODERTRIC RationalRicDE . . . 2982
19.21.1 RationalRicDE (ODERTRIC) . . . 2982
19.22 package RURPK RationalUnivariateRepresentationPackage . . . 2989

19.22.1 RationalUnivariateRepresentationPackage (RURPK) . . . 2989
19.23 package POLUTIL RealPolynomialUtilitiesPackage . . . 2992
19.23.1 RealPolynomialUtilitiesPackage (POLUTIL) . . . 2993
19.24 package REALSOLV RealSolvePackage . . . 2996
19.24.1 RealSolvePackage (REALSOLV) . . . 2999
19.25 package REAL0 RealZeroPackage . . . 3001
19.25.1 RealZeroPackage (REAL0) . . . 3001
19.26 package REAL0Q RealZeroPackageQ . . . 3008
19.26.1 RealZeroPackageQ (REAL0Q) . . . 3008
19.27 package RMCAT2 RectangularMatrixCategoryFunctions2 . . . 3010
19.27.1 RectangularMatrixCategoryFunctions2 (RMCAT2) . . . 3010
19.28 package RECOP RecurrenceOperator . . . 3012
19.28.1 RecurrenceOperator (RECOP) . . . 3012
19.28.2 Defining new operators . . . 3013
19.28.3 Recurrences . . . 3015
19.28.4 Functional Equations . . . 3019
19.29 package RDIV ReducedDivisor . . . 3023
19.29.1 ReducedDivisor (RDIV) . . . 3023
19.30 package ODERED ReduceLODE . . . 3025
19.30.1 ReduceLODE (ODERED) . . . 3025
19.31 package REDORDER ReductionOfOrder . . . 3027
19.31.1 ReductionOfOrder (REDORDER) . . . 3027
19.32 package RSDCMPK RegularSetDecompositionPackage . . . 3029
19.32.1 RegularSetDecompositionPackage (RSDCMPK) . . . 3029
19.33 package RSETGCD RegularTriangularSetGcdPackage . . . 3035
19.33.1 RegularTriangularSetGcdPackage (RSETGCD) . . . 3035
19.34 package REPDB RepeatedDoubling . . . 3043
19.34.1 RepeatedDoubling (REPDB) . . . 3043
19.35 package REPSQ RepeatedSquaring . . . 3044
19.35.1 RepeatedSquaring (REPSQ) . . . 3045
19.36 package REP1 RepresentationPackage1 . . . 3046
19.36.1 RepresentationPackage1 (REP1) . . . 3046
19.37 package REP2 RepresentationPackage2 . . . 3053
19.37.1 RepresentationPackage2 (REP2) . . . 3053
19.38 package RESLATC ResolveLatticeCompletion . . . 3070
19.38.1 ResolveLatticeCompletion (RESLATC) . . . 3070
19.39 package RETSOL RetractSolvePackage . . . 3071
19.39.1 RetractSolvePackage (RETSOL) . . . 3071
19.40 package RFP RootsFindingPackage . . . 3073
19.40.1 RootsFindingPackage (RFP) . . . 3074

20 Chapter S **3079**
20.1 package SAERFFC SAERationalFunctionAlgFactor . . . 3079
20.1.1 SAERationalFunctionAlgFactor (SAERFFC) . . . 3079
20.2 package FORMULA1 ScriptFormulaFormat1 . . . 3080
20.2.1 ScriptFormulaFormat1 (FORMULA1) . . . 3080

20.3 package SEGBIND2 SegmentBindingFunctions2 . . . 3082
20.3.1 SegmentBindingFunctions2 (SEGBIND2) . . . 3082
20.4 package SEG2 SegmentFunctions2 . . . 3083
20.4.1 SegmentFunctions2 (SEG2) . . . 3083
20.5 package SAEFACT SimpleAlgebraicExtensionAlgFactor . . . 3085
20.5.1 SimpleAlgebraicExtensionAlgFactor (SAEFACT) . . . 3085
20.6 package SIMPAN SimplifyAlgebraicNumberConvertPackage . . . 3086
20.6.1 SimplifyAlgebraicNumberConvertPackage (SIMPAN) . . . 3086
20.7 package SMITH SmithNormalForm . . . 3087
20.7.1 SmithNormalForm (SMITH) . . . 3087
20.8 package SCACHE SortedCache . . . 3092
20.8.1 SortedCache (SCACHE) . . . 3092
20.9 package SORTPAK SortPackage . . . 3095
20.9.1 SortPackage (SORTPAK) . . . 3095
20.10 package SUP2 SparseUnivariatePolynomialFunctions2 . . . 3097
20.10.1 SparseUnivariatePolynomialFunctions2 (SUP2) . . . 3097
20.11 package SPECOUT SpecialOutputPackage . . . 3098
20.11.1 SpecialOutputPackage (SPECOUT) . . . 3098
20.12 package SFQCMPK SquareFreeQuasiComponentPackage . . . 3100
20.12.1 SquareFreeQuasiComponentPackage (SFQCMPK) . . . 3100
20.13 package SRDCMPK SquareFreeRegularSetDecompositionPackage . . . 3109
20.13.1 SquareFreeRegularSetDecompositionPackage (SRDCMPK) . . . 3109
20.14 package SFRGCD SquareFreeRegularTriangularSetGcdPackage . . . 3116
20.14.1 SquareFreeRegularTriangularSetGcdPackage (SFRGCD) . . . 3116
20.15 package MATSTOR StorageEfficientMatrixOperations . . . 3126
20.15.1 StorageEfficientMatrixOperations (MATSTOR) . . . 3126
20.16 package STREAM1 StreamFunctions1 . . . 3130
20.16.1 StreamFunctions1 (STREAM1) . . . 3130
20.17 package STREAM2 StreamFunctions2 . . . 3132
20.17.1 StreamFunctions2 (STREAM2) . . . 3132
20.18 package STREAM3 StreamFunctions3 . . . 3134
20.18.1 StreamFunctions3 (STREAM3) . . . 3134
20.19 package STINPROD StreamInfiniteProduct . . . 3136
20.19.1 StreamInfiniteProduct (STINPROD) . . . 3136
20.20 package STTAYLOR StreamTaylorSeriesOperations . . . 3138
20.20.1 StreamTaylorSeriesOperations (STTAYLOR) . . . 3138
20.21 package STNSR StreamTensor . . . 3148
20.21.1 StreamTensor (STNSR) . . . 3149
20.22 package STTF StreamTranscendentalFunctions . . . 3150
20.22.1 StreamTranscendentalFunctions (STTF) . . . 3150
20.23 package STTFNC StreamTranscendentalFunctionsNonCommutative . . . 3160
20.23.1 StreamTranscendentalFunctionsNonCommutative (STTFNC) . . . 3160
20.24 package SCPKG StructuralConstantsPackage . . . 3165
20.24.1 StructuralConstantsPackage (SCPKG) . . . 3165
20.25 package SHP SturmHabichtPackage . . . 3169
20.25.1 SturmHabichtPackage (SHP) . . . 3169

20.26package SUBRESP SubResultantPackage . . . 3177
20.26.1 SubResultantPackage (SUBRESP) . . . 3177
20.27package SUPFRACF SupFractionFactorizer . . . 3180
20.27.1 SupFractionFactorizer (SUPFRACF) . . . 3180
20.28package ODESYS SystemODESolver . . . 3182
20.28.1 SystemODESolver (ODESYS) . . . 3182
20.29package SYSSOLP SystemSolvePackage . . . 3188
20.29.1 SystemSolvePackage (SYSSOLP) . . . 3188
20.30package SGCF SymmetricGroupCombinatoricFunctions . . . 3193
20.30.1 SymmetricGroupCombinatoricFunctions (SGCF) . . . 3193
20.31package SYMFUNC SymmetricFunctions . . . 3204
20.31.1 SymmetricFunctions (SYMFUNC) . . . 3204

21 Chapter T **3207**
21.1 package TABLBUMP TableauxBumpers . . . 3207
21.1.1 TableauxBumpers (TABLBUMP) . . . 3207
21.2 package TBCMPPK TabulatedComputationPackage . . . 3210
21.2.1 TabulatedComputationPackage (TBCMPPK) . . . 3210
21.3 package TANEXP TangentExpansions . . . 3214
21.3.1 TangentExpansions (TANEXP) . . . 3214
21.4 package UTSSOL TaylorSolve . . . 3215
21.4.1 TaylorSolve (UTSSOL) . . . 3216
21.5 package TEMUTL TemplateUtilities . . . 3219
21.5.1 TemplateUtilities (TEMUTL) . . . 3219
21.6 package TEX1 TexFormat1 . . . 3221
21.6.1 TexFormat1 (TEX1) . . . 3221
21.7 package TOOLSIGN ToolsForSign . . . 3222
21.7.1 ToolsForSign (TOOLSIGN) . . . 3222
21.8 package DRAW TopLevelDrawFunctions . . . 3224
21.8.1 TopLevelDrawFunctions (DRAW) . . . 3224
21.9 package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves . . . 3231
21.9.1 TopLevelDrawFunctionsForAlgebraicCurves (DRAWCURV) . . . 3231
21.10package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions . . . 3235
21.10.1 TopLevelDrawFunctionsForCompiledFunctions (DRAWCFUN) . . . 3235
21.11package DRAWPT TopLevelDrawFunctionsForPoints . . . 3248
21.11.1 TopLevelDrawFunctionsForPoints (DRAWPT) . . . 3248
21.12package TOPSP TopLevelThreeSpace . . . 3251
21.12.1 TopLevelThreeSpace (TOPSP) . . . 3251
21.13package INTHERTR TranscendentalHermiteIntegration . . . 3252
21.13.1 TranscendentalHermiteIntegration (INTHERTR) . . . 3252
21.14package INTTR TranscendentalIntegration . . . 3254
21.14.1 TranscendentalIntegration (INTTR) . . . 3254
21.15package TRMANIP TranscendentalManipulations . . . 3264
21.15.1 TranscendentalManipulations (TRMANIP) . . . 3264
21.16package RDETR TranscendentalRischDE . . . 3273
21.16.1 TranscendentalRischDE (RDETR) . . . 3273

21.17 package RDETRS TranscendentalRischDESystem . . . 3277
21.17.1 TranscendentalRischDESystem (RDETRS) . . . 3277
21.18 package SOLVETRA TransSolvePackage . . . 3282
21.18.1 TransSolvePackage (SOLVETRA) . . . 3288
21.19 package SOLVESER TransSolvePackageService . . . 3299
21.19.1 TransSolvePackageService (SOLVESER) . . . 3299
21.20 package TRIMAT TriangularMatrixOperations . . . 3302
21.20.1 TriangularMatrixOperations (TRIMAT) . . . 3302
21.21 package TRIGMNIP TrigonometricManipulations . . . 3304
21.21.1 TrigonometricManipulations (TRIGMNIP) . . . 3304
21.22 package TUBETOOL TubePlotTools . . . 3308
21.22.1 TubePlotTools (TUBETOOL) . . . 3308
21.23 package CLIP TwoDimensionalPlotClipping . . . 3311
21.23.1 TwoDimensionalPlotClipping (CLIP) . . . 3311
21.24 package TWOFACT TwoFactorize . . . 3317
21.24.1 TwoFactorize (TWOFACT) . . . 3317

22 Chapter U **3323**
22.1 package UNIFACT UnivariateFactorize . . . 3323
22.1.1 UnivariateFactorize (UNIFACT) . . . 3323
22.2 package UFPS1 UnivariateFormalPowerSeriesFunctions . . . 3330
22.2.1 UnivariateFormalPowerSeriesFunctions (UFPS1) . . . 3330
22.3 package ULS2 UnivariateLaurentSeriesFunctions2 . . . 3331
22.3.1 UnivariateLaurentSeriesFunctions2 (ULS2) . . . 3331
22.4 package UPOLYC2 UnivariatePolynomialCategoryFunctions2 . . . 3333
22.4.1 UnivariatePolynomialCategoryFunctions2 (UPOLYC2) . . . 3333
22.5 package UPCDEN UnivariatePolynomialCommonDenominator . . . 3334
22.5.1 UnivariatePolynomialCommonDenominator (UPCDEN) . . . 3334
22.6 package UPDECOMP UnivariatePolynomialDecompositionPackage . . . 3336
22.6.1 UnivariatePolynomialDecompositionPackage (UPDECOMP) . . . 3336
22.7 package UPDIVP UnivariatePolynomialDivisionPackage . . . 3339
22.7.1 UnivariatePolynomialDivisionPackage (UPDIVP) . . . 3339
22.8 package UP2 UnivariatePolynomialFunctions2 . . . 3341
22.8.1 UnivariatePolynomialFunctions2 (UP2) . . . 3341
22.9 package UPMP UnivariatePolynomialMultiplicationPackage . . . 3342
22.9.1 UnivariatePolynomialMultiplicationPackage (UPMP) . . . 3342
22.10 package UPSQFREE UnivariatePolynomialSquareFree . . . 3345
22.10.1 UnivariatePolynomialSquareFree (UPSQFREE) . . . 3345
22.11 package UPXS2 UnivariatePuiseuxSeriesFunctions2 . . . 3348
22.11.1 UnivariatePuiseuxSeriesFunctions2 (UPXS2) . . . 3348
22.12 package OREPCTO UnivariateSkewPolynomialCategoryOps . . . 3349
22.12.1 UnivariateSkewPolynomialCategoryOps (OREPCTO) . . . 3349
22.13 package UTS2 UnivariateTaylorSeriesFunctions2 . . . 3353
22.13.1 UnivariateTaylorSeriesFunctions2 (UTS2) . . . 3353
22.14 package UTSODE UnivariateTaylorSeriesODESolver . . . 3354
22.14.1 UnivariateTaylorSeriesODESolver (UTSODE) . . . 3354

22.15package UNISEG2 UniversalSegmentFunctions2 . . . 3357
22.15.1 UniversalSegmentFunctions2 (UNISEG2) . . . 3357
22.16package UDPO UserDefinedPartialOrdering . . . 3359
22.16.1 UserDefinedPartialOrdering (UDPO) . . . 3359
22.17package UDVO UserDefinedVariableOrdering . . . 3361
22.17.1 UserDefinedVariableOrdering (UDVO) . . . 3361
22.18package UTSODETL UTSodetools . . . 3363
22.18.1 UTSodetools (UTSODETL) . . . 3363

23 Chapter V **3365**
23.1 package VECTOR2 VectorFunctions2 . . . 3365
23.1.1 VectorFunctions2 (VECTOR2) . . . 3365
23.2 package VIEWDEF ViewDefaultsPackage . . . 3367
23.2.1 ViewDefaultsPackage (VIEWDEF) . . . 3367
23.3 package VIEW ViewportPackage . . . 3372
23.3.1 ViewportPackage (VIEW) . . . 3372

24 Chapter W **3375**
24.1 package WEIER WeierstrassPreparation . . . 3375
24.1.1 WeierstrassPreparation (WEIER) . . . 3375
24.2 package WFFINTBS WildFunctionFieldIntegralBasis . . . 3379
24.2.1 WildFunctionFieldIntegralBasis (WFFINTBS) . . . 3379

25 Chapter X **3385**
25.1 package XEXPPKG XExponentialPackage . . . 3385
25.1.1 XExponentialPackage (XEXPPKG) . . . 3385

26 Chapter Y **3389**

27 Chapter Z **3391**
27.1 package ZDSOLVE ZeroDimensionalSolvePackage . . . 3391
27.1.1 ZeroDimensionalSolvePackage (ZDSOLVE) . . . 3454

28 Chunk collections **3465**

29 Index **3477**

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Chapter 1

Chapter Overview

This book contains the domains in Axiom, in alphabetical order.

Each domain has an associated 'dotpic' chunk which only lists the domains, categories, and packages that are in the layer immediately below in the build order. For the full list see the algebra Makefile where this information is maintained.

Each domain is preceded by a picture. The picture indicates several things. The colors indicate whether the name refers to a category, domain, or package. An ellipse means that the name refers to something in the bootstrap set. Thus,



Chapter 2

Chapter A

2.1 package AFALGGRO AffineAlgebraicSetCompute-WithGroebnerBasis

— AffineAlgebraicSetComputeWithGroebnerBasis.input —

```
)set break resume
)sys rm -f AffineAlgebraicSetComputeWithGroebnerBasis.output
)spool AffineAlgebraicSetComputeWithGroebnerBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineAlgebraicSetComputeWithGroebnerBasis
--R
--R AffineAlgebraicSetComputeWithGroebnerBasis(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E
--R Abbreviation for AffineAlgebraicSetComputeWithGroebnerBasis is AFALGGRO
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGGRO
--R
--R------------------------------ Operations --------------------------------
--R affineAlgSet : List(PolyRing) -> Union(List(ProjPt),"failed",Infinite,Integer)
--R affineRationalPoints : (PolyRing,PositiveInteger) -> List(ProjPt)
--R affineSingularPoints : PolyRing -> Union(List(ProjPt),"failed",Infinite,Integer)
--R
--E 1

)spool
)lisp (bye)
```

———-

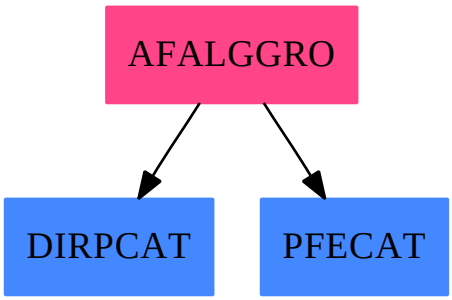
— **AffineAlgebraicSetComputeWithGroebnerBasis.help** —

```
==========================================================================
AffineAlgebraicSetComputeWithGroebnerBasis examples
==========================================================================

See Also:
o )show AffineAlgebraicSetComputeWithGroebnerBasis
```

———-

2.1.1 AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO)



Exports:
affineAlgSet affineRationalPoints affineSingularPoints

— **package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis** —

```
)abbreviation package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt):Exports_
 == Implementation where
  K   : Field
  symb: List(Symbol)
  OV  ==> OrderedVariableList(symb)
```

```
  E : DirectProductCategory(#symb,NonNegativeInteger)
  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt   : ProjectiveSpaceCategory(K)
  PCS      : LocalPowerSeriesCategory(K)

  OF    ==> OutputForm
  PI    ==> PositiveInteger
  NNI   ==> NonNegativeInteger
  RFP   ==> RootsFindingPackage
  SUP   ==> SparseUnivariatePolynomial
  PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)

  Exports ==>  with

    affineAlgSet: List PolyRing -> _
                  Union(List(ProjPt),"failed","Infinite",Integer)

    affineSingularPoints :  PolyRing -> _
                  Union(List(ProjPt),"failed","Infinite",Integer)

    affineRationalPoints: (PolyRing,PI) -> List ProjPt
        ++ \axiom{rationalPoints(f,d)} returns all points on the curve
++ \axiom{f} in the extension of the ground field of degree \axiom{d}.
++ For \axiom{d > 1} this only works if \axiom{K} is a
++ \axiomType{LocallyAlgebraicallyClosedField}

  Implementation ==>  add

    ss2:List Symbol:= [X1,X2]

    DD   ==> DistributedMultivariatePolynomial(ss2,K)
    LexE ==> DirectProduct(#ss2,NonNegativeInteger)
    OV2  ==> OrderedVariableList(ss2)
    InGB ==> InterfaceGroebnerPackage(K,ss2,LexE,OV2,DD)

    affineAlgSetLocal : List DD  -> _
                        Union(List(ProjPt),"failed","Infinite",Integer)

    import PPFC1
    import PolyRing
    import ProjPt

    listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]

    polyToYX1 : PolyRing ->  DD
    -- NOTE : polyToYX1 set the last variable to 1 and swap the 1st and 2nd var
    -- so that a call to grobner will eliminate the second var before the
    -- first one
    -- 23/10/98 : Ce n'est plus vrai. La fonction a ete "repare'".
    -- A priori ce la ne creait pas de bug, car on tenait compte de
```

```
    -- cette particulariite dans la fonction affineAlgSetLocal.
    -- cette derniere fct a aussi ete "ajuste'"
    -- 27/10/98
    -- Ce n'est pas vraie !!! Il fauit trouve X d'abord et ensuite Y !!
    -- sinon tout sr la notion de places distinguee fout le camp !!!

    polyToX10 : PolyRing -> SUP(K)

--fonctions de resolution de sys. alg. de dim 0

    if K has FiniteFieldCategory then

      affineRationalPoints(crv:PolyRing,extdegree:PI):List(ProjPt) ==
--The code of this is almost the same as for algebraicSet
--We could just construct the ideal and call algebraicSet
--Should we do that? This might be a bit faster.

listPtsIdl:List(ProjPt):= empty()

        x:= monomial(1,directProduct(vector([1,0])$Vector(NNI)))$DD
        y:= monomial(1,directProduct(vector([0,1])$Vector(NNI)))$DD

if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
  setTower!(1$K)$K
        q:= size()$K
        px:= x**(q**extdegree) - x
py:= y**(q**extdegree) - y

crvXY1 := polyToYX1 crv
        rpts:= affineAlgSetLocal([crvXY1,px,py])

        -- si les  3  tests qui suivent ne sont pas la,
        -- alors ca ne compile pas !!! ???
        rpts case "failed" =>_
          error "failed: From affineRationalPoints in AFALGGRO,"
        rpts case "Infinite" =>_
          error "Infinite: From affineRationalPoints in AFALGGRO,"
        rpts case Integer =>_
          error "Integer: From affineRationalPoints in AFALGGRO,"
        rpts case List(ProjPt) => rpts
        error "Unknown: From affineRationalPoints in AFALGGRO,"

    affineSingularPoints(crb)==
      F:= polyToYX1 crb
      Fx:=differentiate(F,index(1)$OV2)
      Fy:=differentiate(F,index(2)$OV2)
      affineAlgSetLocal([F,Fx,Fy])

    affineAlgSet(ideal : List PolyRing )==
      idealXY1 := [polyToYX1 pol for pol in ideal]
```

```
      affineAlgSetLocal idealXY1

    --fonctions de resolution de sys. alg. de dim 0
    affineAlgSetLocal(idealToXY1:List DD ) ==
      listPtsIdl:List(ProjPt)
      idealGroXY1:=groebner(idealToXY1)$InGB
      listZeroY:List(K):=empty()
      listZeroX:List(K):=empty()
      listOfExtDeg:List(Integer):=empty()
      polyZeroX:DD:=last(idealGroXY1)
      member?(index(1)$OV2, variables(polyZeroX)$DD) =>
        print(("The number of point in the algebraic set is not finite")::OF)
        print(("or the curve is not absolubtly irreducible.")::OF)
        error "Have a nice day"
        --now we find all of the projective points where z ^= 0
      recOfZerosX:=distinguishedRootsOf(univariate(polyZeroX),1$K)$RFP(K)
      -- HERE CHANGE
      degExtX:=recOfZerosX.extDegree
      listZeroX:List K := recOfZerosX.zeros
      listOfExtDeg:=cons(degExtX,listOfExtDeg)
      for a in listZeroX repeat
        tjeker := [(eval(f,index(2)$OV2,a)$DD) for f in idealGroXY1]
        idealGroaXb1 := [univariate(f)$DD for f in tjeker]
recOfZerosOfIdeal:=distinguishedCommonRootsOf(idealGroaXb1,a)$RFP(K)
        listZeroY:= recOfZerosOfIdeal.zeros
        listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree,listOfExtDeg)
        listPtsIdl:=
          concat( [projectivePoint([a,b,1]) for b in listZeroY] ,listPtsIdl)
      degExt:=lcm listOfExtDeg
      zero?(degExt) =>
        print(("------- Infinite number of points ------")::OF)
        "Infinite"
      ^one?(degExt) =>
        print(("You need an extension of degree")::OF)
        print(degExt::OF)
        degExt
      listPtsIdl

    polyToYX1(pol)==
      zero?(pol) => 0
      dd:= degree pol
      lc:= leadingCoefficient pol
      pp:= parts dd
      ppr:=  rest reverse pp
      ppv:Vector(NNI):= vector ppr
      eppr:=directProduct(ppv)$LexE
      monomial(lc,eppr)$DD + polyToYX1 reductum pol

    polyToX10(pol)==
      zero?(pol) => 0
```

```
      dd:= degree pol
      lc:= leadingCoefficient pol
      pp:= parts dd
      lp:= last pp
      ^zero?(lp) => polyToX10 reductum pol
      e1:= pp.1
      monomial(lc,e1)$SUP(K) + polyToX10 reductum pol
```

———-

— **AFALGGRO.dotabb** —

```
"AFALGGRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGGRO"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"AFALGGRO" -> "DIRPCAT"
"AFALGGRO" -> "PFECAT"
```

———-

2.2 package AFALGRES AffineAlgebraicSetComputeWithResultant

— **AffineAlgebraicSetComputeWithResultant.input** —

```
)set break resume
)sys rm -f AffineAlgebraicSetComputeWithResultant.output
)spool AffineAlgebraicSetComputeWithResultant.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineAlgebraicSetComputeWithResultant
--R
--R AffineAlgebraicSetComputeWithResultant(K: Field,symb: List(Symbol),PolyRing: PolynomialCateg
--R Abbreviation for AffineAlgebraicSetComputeWithResultant is AFALGRES
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGRES
--R
--R------------------------------ Operations ------------------------------
--R affineAlgSet : List(PolyRing) -> Union(List(ProjPt),"failed",Infinite,Integer)
--R affineAlgSetLocal : List(SparseUnivariatePolynomial(SparseUnivariatePolynomial(K))) -> Union
```

```
--R affineRationalPoints : (PolyRing,PositiveInteger) -> Union(List(ProjPt),"failed",Infinite,Integer)
--R affineSingularPoints : PolyRing -> Union(List(ProjPt),"failed",Infinite,Integer)
--R affineSingularPoints : SparseUnivariatePolynomial(SparseUnivariatePolynomial(K)) -> Union(List(ProjPt),"
--R allPairsAmong : List(SparseUnivariatePolynomial(SparseUnivariatePolynomial(K))) -> List(List(SparseUniva
--R polyRing2UPUP : PolyRing -> SparseUnivariatePolynomial(SparseUnivariatePolynomial(K))
--R
--E 1

)spool
)lisp (bye)
```

———-

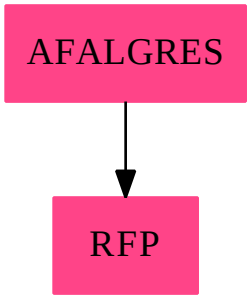
— **AffineAlgebraicSetComputeWithResultant.help** —

```
=====================================================================
AffineAlgebraicSetComputeWithResultant examples
=====================================================================

See Also:
o )show AffineAlgebraicSetComputeWithResultant
```

———-

2.2.1 AffineAlgebraicSetComputeWithResultant (AFALGRES)



Exports:

affineAlgSet	affineAlgSetLocal	affineRationalPoints
affineSingularPoints	allPairsAmong	polyRing2UPUP

— **package AFALGRES AffineAlgebraicSetComputeWithResultant** —

```
)abbrev package AFALGRES AffineAlgebraicSetComputeWithResultant
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt):Ex==Impl where
  K   : Field
  symb: List(Symbol)
  OV  ==> OrderedVariableList(symb)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt   : ProjectiveSpaceCategory(K)
  PCS      : LocalPowerSeriesCategory(K)

  SUP  ==> SparseUnivariatePolynomial
  UPUP ==> SUP(SUP(K))
  NNI  ==> NonNegativeInteger
  RFP  ==> RootsFindingPackage

  Ex ==>  with

    affineSingularPoints: PolyRing -> _
                          Union(List(ProjPt),"failed","Infinite",Integer)

    affineSingularPoints: UPUP -> _
                          Union(List(ProjPt),"failed","Infinite",Integer)

    affineAlgSetLocal: List UPUP -> _
                       Union(List(ProjPt),"failed","Infinite",Integer)

    affineAlgSet: List PolyRing  -> _
                  Union(List ProjPt ,"failed","Infinite",Integer)

    polyRing2UPUP: PolyRing -> UPUP

    allPairsAmong: List UPUP -> List List UPUP

    affineRationalPoints: (PolyRing, PositiveInteger) -> _
                          Union(List(ProjPt),"failed","Infinite",Integer)

  Impl ==>  add

    import ProjPt

    evAtcoef: (UPUP,K) -> SUP(K)

    evAtcoef(pol,a)==
      zero?(pol) => 0
      dd:= degree pol
```

```
      lc:= leadingCoefficient pol
      monomial( lc(a), dd )$SUP(K)  + evAtcoef( reductum(pol), a )

    polyRing2UPUP(pol)==
      zero?(pol) => 0
      dd:= degree pol
      lc:= leadingCoefficient pol
      pp:= parts dd
      monomial(monomial(lc,pp.1)$SUP(K),pp.2)$UPUP+polyRing2UPUP(reductum(pol))

    if K has FiniteFieldCategory then

      affineRationalPoints(crv:PolyRing,extdegree:PositiveInteger) ==
listPtsIdl:List(ProjPt):= empty()
        x:= monomial(1,directProduct(vector([1,0,0])$Vector(NNI)))$PolyRing
        y:= monomial(1,directProduct(vector([0,1,0])$Vector(NNI)))$PolyRing
if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
  setTower!(1$K)$K
        q:= size()$K
        px:= x**(q**extdegree) - x
py:= y**(q**extdegree) - y
        rpts:= affineAlgSet([crv,px,py])
        -- si les  3  tests qui suivent ne sont pas la,
        -- alors ca ne compile pas !!! ???
        rpts case "failed" => _
          error "case failed: From affineRationalPoints in AFALGRES"
        rpts case "Infinite" => _
          error "case infinite: From affineRationalPoints in AFALGRES"
        rpts case Integer => _
          error "case Integer: From affineRationalPoints in AFALGRES"
        rpts case List(ProjPt) => rpts
        error "case unknown: From affineRationalPoints in AFALGRES"

    allPairsAmong(lp)==
      #lp = 2 => [lp]
      rlp:=rest lp
      subL:= allPairsAmong rlp
      pol:=first lp
      frontL:= [[pol,p] for p in rlp]
      concat( frontL , subL )

    affineSingularPoints(pol:PolyRing)==
      affineSingularPoints( polyRing2UPUP pol )

    affineSingularPoints(pol:UPUP)==
      ground? pol => "failed"
      lc := coefficients pol
      lcb := [ ground?( c )$SUP(K) for c in lc ]
      reduce("and" , lcb) => "failed"
      dy:=differentiate(pol)
```

```
      dx:=map(differentiate$SUP(K),pol)
      affineAlgSetLocal( [ pol, dy, dx ] )

    resultantL: List UPUP -> SUP(K)
    resultantL(lp)==
      g:=first lp
      h:= last lp
      resultant(g,h)

    affineAlgSet(lpol:List PolyRing)==
      affineAlgSetLocal( [ polyRing2UPUP pol for pol in lpol ] )

    affineAlgSetLocal(lpol:List UPUP)==
      listPtsIdl:List(ProjPt)
      allP:= allPairsAmong lpol
      beforGcd:List SUP(K) := [resultantL(lp) for lp in allP]
      polyZeroX:SUP(K):=gcd beforGcd
      zero? polyZeroX => "failed"
      listZeroY:List(K):=empty()
      listZeroX:List(K):=empty()
      recOfZerosX:=distinguishedRootsOf(polyZeroX,1$K)$RFP(K)
      degExtX:=recOfZerosX.extDegree
      listZeroX:List K := recOfZerosX.zeros
      listOfExtDeg:List(Integer):=empty()
      listOfExtDeg:=cons(degExtX,listOfExtDeg)
      lpolEval:List SUP(K)
      for a in listZeroX repeat
        lpolEval := [ evAtcoef(p,a) for p in lpol ]
recOfZerosOfIdeal:=distinguishedCommonRootsOf( lpolEval ,a)$RFP(K)
        listZeroY:= recOfZerosOfIdeal.zeros
        listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree,listOfExtDeg)
        listPtsIdl:=
          concat( [projectivePoint([a,b,1]) for b in listZeroY] ,listPtsIdl)
      degExt:=lcm listOfExtDeg
      zero?(degExt) =>
        print(("AFALGRES:Infinite number of points")::OutputForm)
        "Infinite"
      ^one?(degExt) =>
        print(("AFALGRES:You need an extension of degree")::OutputForm)
        print(degExt::OutputForm)
        degExt
      listPtsIdl
```

———-

— AFALGRES.dotabb —

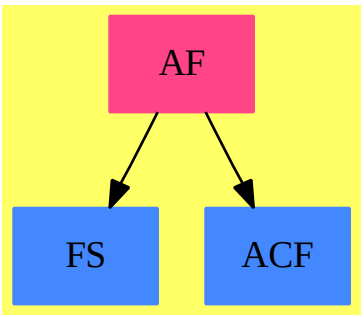
```
"AFALGRES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGRES"]
```

```
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"AFALGRES" -> "RFP"
```

———-

2.3 package AF AlgebraicFunction

2.3.1 AlgebraicFunction (AF)



Exports:

belong?	droot	definingPolynomial	inrootof	iroot
minPoly	operator	rootOf	?**?	

— **package AF AlgebraicFunction** —

```
)abbrev package AF AlgebraicFunction
++ Author: Manuel Bronstein
++ Date Created: 21 March 1988
++ Date Last Updated: 11 November 1993
++ Keywords: algebraic, function.
++ Description:
++ This package provides algebraic functions over an integral domain.

AlgebraicFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R

  SE  ==> Symbol
  Z   ==> Integer
  Q   ==> Fraction Z
  OP  ==> BasicOperator
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
```

```
UPR ==> SparseUnivariatePolynomial R
ALGOP       ==> "%alg"
SPECIALDISP ==> "%specialDisp"
SPECIALDIFF ==> "%specialDiff"

Exports ==> with
  rootOf  : (UP, SE) -> F
    ++ rootOf(p, y) returns y such that \spad{p(y) = 0}.
    ++ The object returned displays as \spad{'y}.
  operator: OP -> OP
    ++ operator(op) returns a copy of \spad{op} with the domain-dependent
    ++ properties appropriate for \spad{F}.
    ++ Error: if op is not an algebraic operator, that is,
    ++ an nth root or implicit algebraic operator.
  belong? : OP -> Boolean
    ++ belong?(op) is true if \spad{op} is an algebraic operator, that is,
    ++ an nth root or implicit algebraic operator.
  inrootof: (UP, F) -> F
    ++ inrootof(p, x) should be a non-exported function.
    -- un-export when the compiler accepts conditional local functions!
  droot : List F -> OutputForm
    ++ droot(l) should be a non-exported function.
    -- un-export when the compiler accepts conditional local functions!
  if R has RetractableTo Integer then
    "**"   : (F, Q) -> F
      ++ x ** q is \spad{x} raised to the rational power \spad{q}.
    minPoly: K  -> UP
      ++ minPoly(k) returns the defining polynomial of \spad{k}.
    definingPolynomial: F -> F
      ++ definingPolynomial(f) returns the defining polynomial of \spad{f}
      ++ as an element of \spad{F}.
      ++ Error: if f is not a kernel.
    iroot : (R, Z) -> F
      ++ iroot(p, n) should be a non-exported function.
      -- un-export when the compiler accepts conditional local functions!

Implementation ==> add
  ialg : List F -> F
  dvalg: (List F, SE) -> F
  dalg : List F -> OutputForm

  opalg  := operator("rootOf"::Symbol)$CommonOperators
  oproot := operator("nthRoot"::Symbol)$CommonOperators

  belong? op == has?(op, ALGOP)
  dalg l     == second(l)::OutputForm

  rootOf(p, x) ==
    k := kernel(x)$K
    (r := retractIfCan(p)@Union(F, "failed")) case "failed" =>
```

```
        inrootof(p, k::F)
      n := numer(f := univariate(r::F, k))
      degree denom f > 0 => error "roofOf: variable appears in denom"
      inrootof(n, k::F)

    dvalg(l, x) ==
      p := numer univariate(first l, retract(second l)@K)
      alpha := kernel(opalg, l)
      - (map((s:F):F +-> differentiate(s, x), p) alpha)_
          / ((differentiate p) alpha)

    ialg l ==
      f := univariate(p := first l, retract(x := second l)@K)
      degree denom f > 0 => error "roofOf: variable appears in denom"
      inrootof(numer f, x)

    operator op ==
      is?(op,  "rootOf"::Symbol) => opalg
      is?(op, "nthRoot"::Symbol) => oproot
      error "Unknown operator"

    if R has AlgebraicallyClosedField then
      UP2R: UP -> Union(UPR, "failed")

      inrootof(q, x) ==
        monomial? q => 0

        (d := degree q) <= 0 => error "rootOf: constant polynomial"
--        one? d=> - leadingCoefficient(reductum q) / leadingCoefficient q
        (d = 1) => - leadingCoefficient(reductum q) / leadingCoefficient q
        ((rx := retractIfCan(x)@Union(SE, "failed")) case SE) and
          ((r := UP2R q) case UPR) => rootOf(r::UPR, rx::SE)::F
        kernel(opalg, [q x, x])

      UP2R p ==
        ans:UPR := 0
        while p ^= 0 repeat
          (r := retractIfCan(leadingCoefficient p)@Union(R, "failed"))
            case "failed" => return "failed"
          ans := ans + monomial(r::R, degree p)
          p   := reductum p
        ans

    else
      inrootof(q, x) ==
        monomial? q => 0
        (d := degree q) <= 0 => error "rootOf: constant polynomial"
--        one? d => - leadingCoefficient(reductum q) /leadingCoefficient q
        (d = 1) => - leadingCoefficient(reductum q) /leadingCoefficient q
        kernel(opalg, [q x, x])
```

```
    evaluate(opalg, ialg)$BasicOperatorFunctions1(F)
    setProperty(opalg, SPECIALDIFF,
                               dvalg@((List F, SE) -> F) pretend None)
    setProperty(opalg, SPECIALDISP,
                               dalg@(List F -> OutputForm) pretend None)

    if R has RetractableTo Integer then
      import PolynomialRoots(IndexedExponents K, K, R, P, F)

      dumvar := "%%var"::Symbol::F

      lzero   : List F -> F
      dvroot  : List F -> F
      inroot  : List F -> F
      hackroot: (F, Z) -> F
      inroot0 : (F, Z, Boolean, Boolean) -> F

      lzero l == 0

      droot l ==
        x := first(l)::OutputForm
        (n := retract(second l)@Z) = 2 => root x
        root(x, n::OutputForm)

      dvroot l ==
        n := retract(second l)@Z
        (first(l) ** ((1 - n) / n)) / (n::F)

      x ** q ==
        qr := divide(numer q, denom q)
        x ** qr.quotient * inroot([x, (denom q)::F]) ** qr.remainder

      hackroot(x, n) ==
        (n = 1) or (x = 1) => x
        (((dx := denom x) ^= 1) and
           ((rx := retractIfCan(dx)@Union(Integer,"failed")) case Integer) and
           positive?(rx))
           => hackroot((numer x)::F, n)/hackroot(rx::Integer::F, n)
        (x = -1) and n = 4 =>
          ((-1::F) ** (1::Q / 2::Q) + 1) / ((2::F) ** (1::Q / 2::Q))
        kernel(oproot, [x, n::F])

      inroot l ==
        zero?(n := retract(second l)@Z) => error "root: exponent = 0"
--        one?(x := first l) or one? n => x
        ((x := first l) = 1) or (n = 1) => x
        (r := retractIfCan(x)@Union(R,"failed")) case R => iroot(r::R,n)
        (u := isExpt(x, oproot)) case Record(var:K, exponent:Z) =>
          pr := u::Record(var:K, exponent:Z)
```

```
          (first argument(pr.var)) **
              (pr.exponent /$Fraction(Z)
                   (n * retract(second argument(pr.var))@Z))
        inroot0(x, n, false, false)

-- removes powers of positive integers from numer and denom
-- num? or den? is true if numer or denom already processed
      inroot0(x, n, num?, den?) ==
        rn:Union(Z, "failed") := (num? => "failed"; retractIfCan numer x)
        rd:Union(Z, "failed") := (den? => "failed"; retractIfCan denom x)
        (rn case Z) and (rd case Z) =>
          rec := qroot(rn::Z / rd::Z, n::NonNegativeInteger)
          rec.coef * hackroot(rec.radicand, rec.exponent)
        rn case Z =>
          rec := qroot(rn::Z::Fraction(Z), n::NonNegativeInteger)
          rec.coef * inroot0((rec.radicand**(n exquo rec.exponent)::Z)
                                / (denom(x)::F), n, true, den?)
        rd case Z =>
          rec := qroot(rd::Z::Fraction(Z), n::NonNegativeInteger)
          inroot0((numer(x)::F) /
                  (rec.radicand ** (n exquo rec.exponent)::Z),
                   n, num?, true) / rec.coef
        hackroot(x, n)

      if R has AlgebraicallyClosedField then iroot(r, n) == nthRoot(r, n)::F
      else
        iroot0: (R, Z) -> F

        if R has RadicalCategory then
          if R has imaginary:() -> R then iroot(r, n) == nthRoot(r, n)::F
          else
            iroot(r, n) ==
              odd? n or r >= 0 => nthRoot(r, n)::F
              iroot0(r, n)

        else iroot(r, n) == iroot0(r, n)

        iroot0(r, n) ==
          rec := rroot(r, n::NonNegativeInteger)
          rec.coef * hackroot(rec.radicand, rec.exponent)

      definingPolynomial x ==
        (r := retractIfCan(x)@Union(K, "failed")) case K =>
          is?(k := r::K, opalg) => first argument k
          is?(k, oproot) =>
            dumvar ** retract(second argument k)@Z - first argument k
          dumvar - x
        dumvar - x

      minPoly k ==
```

```
        is?(k, opalg)  =>
           numer univariate(first argument k,
                                          retract(second argument k)@K)
        is?(k, oproot) =>
           monomial(1,retract(second argument k)@Z :: NonNegativeInteger)
             - first(argument k)::UP
        monomial(1, 1) - k::F::UP

      evaluate(oproot, inroot)$BasicOperatorFunctions1(F)
      derivative(oproot, [dvroot, lzero])

    else   -- R is not retractable to Integer
      droot l ==
        x := first(l)::OutputForm
        (n := second l) = 2::F => root x
        root(x, n::OutputForm)

      minPoly k ==
        is?(k, opalg)  =>
           numer univariate(first argument k,
                                          retract(second argument k)@K)
        monomial(1, 1) - k::F::UP

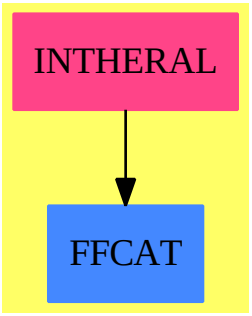
    setProperty(oproot, SPECIALDISP,
                            droot@(List F -> OutputForm) pretend None)
```

— AF.dotabb —

```
"AF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"AF" -> "FS"
"AF" -> "ACF"
```

2.4 package INTHERAL AlgebraicHermiteIntegration

2.4.1 AlgebraicHermiteIntegration (INTHERAL)



Exports:
HermiteIntegrate

— package INTHERAL AlgebraicHermiteIntegration —

```
)abbrev package INTHERAL AlgebraicHermiteIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 25 July 1990
++ Description:
++ Algebraic Hermite reduction.

AlgebraicHermiteIntegration(F,UP,UPUP,R):Exports == Implementation where
  F   : Field
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R   : FunctionFieldCategory(F, UP, UPUP)

  N   ==> NonNegativeInteger
  RF  ==> Fraction UP

  Exports ==> with
    HermiteIntegrate: (R, UP -> UP) -> Record(answer:R, logpart:R)
      ++ HermiteIntegrate(f, ') returns \spad{[g,h]} such that
      ++ \spad{f = g' + h} and h has a only simple finite normal poles.

  Implementation ==> add
    localsolve: (Matrix UP, Vector UP, UP) -> Vector UP

-- the denominator of f should have no prime factor P s.t. P | P'
-- (which happens only for P = t in the exponential case)
    HermiteIntegrate(f, derivation) ==
```

```
  ratform:R := 0
  n    := rank()
  m    := transpose((mat:= integralDerivationMatrix derivation).num)
  inum := (cform := integralCoordinates f).num
  if ((iden := cform.den) exquo (e := mat.den)) case "failed" then
    iden := (coef := (e exquo gcd(e, iden))::UP) * iden
    inum := coef * inum
  for trm in factors squareFree iden | (j:= trm.exponent) > 1 repeat
    u':=(u:=(iden exquo (v:=trm.factor)**(j::N))::UP) * derivation v
    sys := ((u * v) exquo e)::UP * m
    nn := minRowIndex sys - minIndex inum
    while j > 1 repeat
      j := j - 1
      p := - j * u'
      sol := localsolve(sys + scalarMatrix(n, p), inum, v)
      ratform := ratform + integralRepresents(sol, v ** (j::N))
      inum    := [((qelt(inum, i) - p * qelt(sol, i) -
                    dot(row(sys, i - nn), sol))
                      exquo v)::UP - u * derivation qelt(sol, i)
                        for i in minIndex inum .. maxIndex inum]
    iden := u * v
  [ratform, integralRepresents(inum, iden)]

localsolve(mat, vec, modulus) ==
  ans:Vector(UP) := new(nrows mat, 0)
  diagonal? mat =>
    for i in minIndex ans .. maxIndex ans
      for j in minRowIndex mat .. maxRowIndex mat
        for k in minColIndex mat .. maxColIndex mat repeat
          (bc := extendedEuclidean(qelt(mat, j, k), modulus,
            qelt(vec, i))) case "failed" => return new(0, 0)
          qsetelt_!(ans, i, bc.coef1)
    ans
  sol := particularSolution(
                  map(x+->x::RF, mat)$MatrixCategoryFunctions2(UP,
                    Vector UP, Vector UP, Matrix UP, RF,
                      Vector RF, Vector RF, Matrix RF),
                        map(x+->x::RF, vec)$VectorFunctions2(UP,
                          RF))$LinearSystemMatrixPackage(RF,
                                  Vector RF, Vector RF, Matrix RF)
  sol case "failed" => new(0, 0)
  for i in minIndex ans .. maxIndex ans repeat
    (bc := extendedEuclidean(denom qelt(sol, i), modulus, 1))
      case "failed" => return new(0, 0)
    qsetelt_!(ans, i, (numer qelt(sol, i) * bc.coef1) rem modulus)
  ans
```

———-

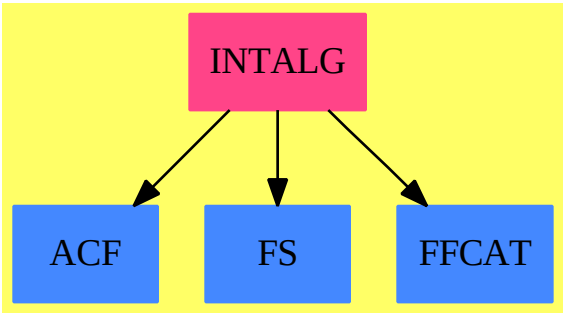
— INTHERAL.dotabb —

```
"INTHERAL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTHERAL"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"INTHERAL" -> "FFCAT"
```

———-

2.5 package INTALG AlgebraicIntegrate

2.5.1 AlgebraicIntegrate (INTALG)



Exports:
algintegrate palginfieldint palgintegrate

— package INTALG AlgebraicIntegrate —

```
)abbrev package INTALG AlgebraicIntegrate
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 19 May 1993
++ Description:
++ This package provides functions for integrating a function
++ on an algebraic curve.

AlgebraicIntegrate(R0, F, UP, UPUP, R): Exports == Implementation where
  R0   : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F    : Join(AlgebraicallyClosedField, FunctionSpace R0)
  UP   : UnivariatePolynomialCategory F
  UPUP : UnivariatePolynomialCategory Fraction UP
  R    : FunctionFieldCategory(F, UP, UPUP)
```

```
SE  ==> Symbol
Z   ==> Integer
Q   ==> Fraction Z
SUP ==> SparseUnivariatePolynomial F
QF  ==> Fraction UP
GP  ==> LaurentPolynomial(F, UP)
K   ==> Kernel F
IR  ==> IntegrationResult R
UPQ ==> SparseUnivariatePolynomial Q
UPR ==> SparseUnivariatePolynomial R
FRQ ==> Factored UPQ
FD  ==> FiniteDivisor(F, UP, UPUP, R)
FAC ==> Record(factor:UPQ, exponent:Z)
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
DIV ==> Record(num:R, den:UP, derivden:UP, gd:UP)
FAIL0 ==> error "integrate: implementation incomplete (constant residues)"
FAIL1==> error "integrate: implementation incomplete (non-algebraic residues)"
FAIL2 ==> error "integrate: implementation incomplete (residue poly has multiple non-linear fa
FAIL3 ==> error "integrate: implementation incomplete (has polynomial part)"
NOTI  ==> error "Not integrable (provided residues have no relations)"

Exports ==> with
  algintegrate  : (R, UP -> UP) -> IR
    ++ algintegrate(f, d) integrates f with respect to the derivation d.
  palgintegrate : (R, UP -> UP) -> IR
    ++ palgintegrate(f, d) integrates f with respect to the derivation d.
    ++ Argument f must be a pure algebraic function.
  palginfieldint: (R, UP -> UP) -> Union(R, "failed")
    ++ palginfieldint(f, d) returns an algebraic function g
    ++ such that \spad{dg = f} if such a g exists, "failed" otherwise.
    ++ Argument f must be a pure algebraic function.

Implementation ==> add
  import FD
  import DoubleResultantPackage(F, UP, UPUP, R)
  import PointsOfFiniteOrder(R0, F, UP, UPUP, R)
  import AlgebraicHermiteIntegration(F, UP, UPUP, R)
  import InnerCommonDenominator(Z, Q, List Z, List Q)
  import FunctionSpaceUnivariatePolynomialFactor(R0, F, UP)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                       K, R0, SparseMultivariatePolynomial(R0, K), F)

  F2R        : F  -> R
  F2UPR      : F  -> UPR
  UP2SUP     : UP -> SUP
  SUP2UP     : SUP -> UP
  UPQ2F      : UPQ -> UP
  univ       : (F, K) -> QF
  pLogDeriv  : (LOG, R -> R) -> R
  nonLinear  : List FAC -> Union(FAC, "failed")
```

```
mkLog      : (UP, Q, R, F) -> List LOG
R2UP       : (R, K) -> UPR
alglogint  : (R, UP -> UP) -> Union(List LOG, "failed")
palglogint : (R, UP -> UP) -> Union(List LOG, "failed")
trace00    : (DIV, UP, List LOG) -> Union(List LOG,"failed")
trace0     : (DIV, UP, Q, FD)    -> Union(List LOG, "failed")
trace1     : (DIV, UP, List Q, List FD, Q) -> Union(List LOG, "failed")
nonQ       : (DIV, UP)           -> Union(List LOG, "failed")
rlift      : (F, K, K) -> R
varRoot?   : (UP, F -> F) -> Boolean
algintexp  : (R, UP -> UP) -> IR
algintprim : (R, UP -> UP) -> IR

dummy:R := 0

dumx  := kernel(new()$SE)$K
dumy  := kernel(new()$SE)$K

F2UPR f == F2R(f)::UPR
F2R f   == f::UP::QF::R

algintexp(f, derivation) ==
  d := (c := integralCoordinates f).den
  v := c.num
  vp:Vector(GP) := new(n := #v, 0)
  vf:Vector(QF) := new(n, 0)
  for i in minIndex v .. maxIndex v repeat
    r := separate(qelt(v, i) / d)$GP
    qsetelt_!(vf, i, r.fracPart)
    qsetelt_!(vp, i, r.polyPart)
  ff := represents(vf, w := integralBasis())
  h := HermiteIntegrate(ff, derivation)
  p := represents(
         map((x1:GP):QF+->convert(x1)@QF, vp)$VectorFunctions2(GP, QF), w)
  zero?(h.logpart) and zero? p => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
                  mkAnswer(h.answer, empty(), [[p + h.logpart, dummy]])
  zero? p => mkAnswer(h.answer, u::List(LOG), empty())
  FAIL3

algintprim(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
                  mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
  mkAnswer(h.answer, u::List(LOG), empty())

-- checks whether f = +/[ci (ui)'/(ui)]
-- f dx must have no pole at infinity
palglogint(f, derivation) ==
```

```
      rec := algSplitSimple(f, derivation)
      ground?(r := doubleResultant(f, derivation)) => "failed"
-- r(z) has roots which are the residues of f at all its poles
      (u  := qfactor r) case "failed" => nonQ(rec, r)
      (fc := nonLinear(lf := factors(u::FRQ))) case "failed" => FAIL2
-- at this point r(z) = fc(z) (z - b1)^e1 .. (z - bk)^ek
-- where the ri's are rational numbers, and fc(z) is arbitrary
-- (fc can be linear too)
-- la = [b1....,bk]  (all rational residues)
      la := [- coefficient(q.factor, 0) for q in remove_!(fc::FAC, lf)]
-- ld = [D1,...,Dk] where Di is the sum of places where f has residue bi
      ld  := [divisor(rec.num, rec.den, rec.derivden, rec.gd, b::F) for b in la]
      pp  := UPQ2F(fc.factor)
-- bb = - sum of all the roots of fc (i.e. the other residues)
      zero?(bb := coefficient(fc.factor,
           (degree(fc.factor) - 1)::NonNegativeInteger)) =>
              -- cd = [[a1,...,ak], d]  such that bi = ai/d
              cd  := splitDenominator la
              -- g = gcd(a1,...,ak), so bi = (g/d) ci  with ci = bi / g
              -- so [g/d] is a basis for [a1,...,ak] over the integers
              g   := gcd(cd.num)
              -- dv0 is the divisor +/[ci Di] corresponding to all the residues
              -- of f except the ones which are root of fc(z)
              dv0 := +/[(a quo g) * dv for a in cd.num for dv in ld]
              trace0(rec, pp, g / cd.den, dv0)
      trace1(rec, pp, la, ld, bb)


    UPQ2F p ==
      map((x:Q):F+->x::F,p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

    UP2SUP p ==
      map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, UP, F, SUP)

    SUP2UP p ==
      map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, SUP, F, UP)

    varRoot?(p, derivation) ==
      for c in coefficients primitivePart p repeat
        derivation(c) ^= 0 => return true
      false

    pLogDeriv(log, derivation) ==
      map(derivation, log.coeff) ^= 0 =>
                 error "can only handle logs with constant coefficients"
--      one?(n := degree(log.coeff)) =>
      ((n := degree(log.coeff)) = 1) =>
        c := - (leadingCoefficient reductum log.coeff)
             / (leadingCoefficient log.coeff)
        ans := (log.logand) c
```

```
        (log.scalar)::R * c * derivation(ans) / ans
      numlog := map(derivation, log.logand)
      (diflog := extendedEuclidean(log.logand, log.coeff, numlog)) case
          "failed" => error "this shouldn't happen"
      algans := diflog.coef1
      ans:R := 0
      for i in 0..n-1 repeat
        algans := (algans * monomial(1, 1)) rem log.coeff
        ans    := ans + coefficient(algans, i)
      (log.scalar)::R * ans

    R2UP(f, k) ==
      x := dumx :: F
      g :=
       (map((f1:QF):F+->f1(x), lift f)_
         $UnivariatePolynomialCategoryFunctions2(QF,UPUP,F,UP))
           (y := dumy::F)
      map((x1:F):R+->rlift(x1, dumx, dumy), univariate(g, k, minPoly k))_
        $UnivariatePolynomialCategoryFunctions2(F,SUP,R,UPR)

    univ(f, k) ==
      g := univariate(f, k)
      (SUP2UP numer g) / (SUP2UP denom g)

    rlift(f, kx, ky) ==
      reduce map(x1+->univ(x1, kx), retract(univariate(f, ky))@SUP)_
        $UnivariatePolynomialCategoryFunctions2(F,SUP,QF,UPUP)

    nonQ(rec, p) ==
      empty? rest(lf := factors ffactor primitivePart p) =>
                      trace00(rec, first(lf).factor, empty()$List(LOG))
      FAIL1

-- case when the irreducible factor p has roots which sum to 0
-- p is assumed doubly transitive for now
    trace0(rec, q, r, dv0) ==
      lg:List(LOG) :=
        zero? dv0 => empty()
        (rc0 := torsionIfCan dv0) case "failed" => NOTI
        mkLog(1, r / (rc0.order::Q), rc0.function, 1)
      trace00(rec, q, lg)

    trace00(rec, pp, lg) ==
      p0 := divisor(rec.num, rec.den, rec.derivden, rec.gd,
                    alpha0 := zeroOf UP2SUP pp)
      q  := (pp exquo (monomial(1, 1)$UP - alpha0::UP))::UP
      alpha := rootOf UP2SUP q
      dvr := divisor(rec.num, rec.den, rec.derivden, rec.gd, alpha) - p0
      (rc := torsionIfCan dvr) case "failed" =>
        degree(pp) <= 2 => "failed"
```

```
        NOTI
      concat(lg, mkLog(q, inv(rc.order::Q), rc.function, alpha))

-- case when the irreducible factor p has roots which sum <> 0
-- the residues of f are of the form [a1,...,ak] rational numbers
-- plus all the roots of q(z), which is squarefree
-- la is the list of residues la := [a1,...,ak]
-- ld is the list of divisors [D1,...Dk] where Di is the sum of all the
-- places where f has residue ai
-- q(z) is assumed doubly transitive for now.
-- let [alpha_1,...,alpha_m] be the roots of q(z)
-- in this function, b = - alpha_1 - ... - alpha_m is <> 0
-- which implies only one generic log term
    trace1(rec, q, la, ld, b) ==
-- cd = [[b1,...,bk], d]  such that ai / b = bi / d
      cd  := splitDenominator [a / b for a in la]
-- then, a basis for all the residues of f over the integers is
-- [beta_1 = - alpha_1 / d,..., beta_m = - alpha_m / d], since:
--      alpha_i = - d beta_i
--      ai = (ai / b) * b = (bi / d) * b = b1 * beta_1 + ... + bm * beta_m
-- linear independence is a consequence of the doubly transitive assumption
-- v0 is the divisor +/[bi Di] corresponding to the residues [a1,...,ak]
      v0 := +/[a * dv for a in cd.num for dv in ld]
-- alpha is a generic root of q(z)
      alpha := rootOf UP2SUP q
-- v is the divisor corresponding to all the residues
      v := v0 - cd.den * divisor(rec.num, rec.den, rec.derivden, rec.gd, alpha)
      (rc := torsionIfCan v) case "failed" =>   -- non-torsion case
        degree(q) <= 2 => "failed"       -- guaranteed doubly-transitive
        NOTI                             -- maybe doubly-transitive
      mkLog(q, inv((- rc.order * cd.den)::Q), rc.function, alpha)

    mkLog(q, scalr, lgd, alpha) ==
      degree(q) <= 1 =>
        [[scalr, monomial(1, 1)$UPR - F2UPR alpha, lgd::UPR]]
      [[scalr,
         map(F2R, q)$UnivariatePolynomialCategoryFunctions2(F,UP,R,UPR),
                                        R2UP(lgd, retract(alpha)@K)]]

-- return the non-linear factor, if unique
-- or any linear factor if they are all linear
    nonLinear l ==
      found:Boolean := false
      ans := first l
      for q in l repeat
        if degree(q.factor) > 1 then
          found => return "failed"
          found := true
          ans   := q
      ans
```

```
-- f dx must be locally integral at infinity
    palginfieldint(f, derivation) ==
      h := HermiteIntegrate(f, derivation)
      zero?(h.logpart) => h.answer
      "failed"

-- f dx must be locally integral at infinity
    palgintegrate(f, derivation) ==
      h := HermiteIntegrate(f, derivation)
      zero?(h.logpart) => h.answer::IR
      (not integralAtInfinity?(h.logpart)) or
        ((u := palglogint(h.logpart, derivation)) case "failed") =>
                      mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
      zero?(difFirstKind := h.logpart - +/[pLogDeriv(lg,
            x1+->differentiate(x1, derivation)) for lg in u::List(LOG)]) =>
                mkAnswer(h.answer, u::List(LOG), empty())
      mkAnswer(h.answer, u::List(LOG), [[difFirstKind, dummy]])

-- for mixed functions. f dx not assumed locally integral at infinity
    algintegrate(f, derivation) ==
      zero? degree(x' := derivation(x := monomial(1, 1)$UP)) =>
         algintprim(f, derivation)
      ((xx := x' exquo x) case UP) and
        (retractIfCan(xx::UP)@Union(F, "failed") case F) =>
          algintexp(f, derivation)
      error "should not happen"

    alglogint(f, derivation) ==
      varRoot?(doubleResultant(f, derivation),
                         x1+->retract(derivation(x1::UP))@F) => "failed"
      FAIL0
```

———-

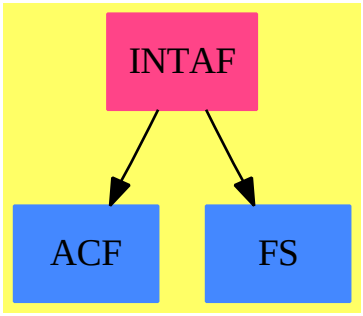
— **INTALG.dotabb** —

```
"INTALG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTALG"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"INTALG" -> "ACF"
"INTALG" -> "FS"
"INTALG" -> "FFCAT"
```

———-

2.6 package INTAF AlgebraicIntegration

2.6.1 AlgebraicIntegration (INTAF)



Exports:
algint

— package INTAF AlgebraicIntegration —

```
)abbrev package INTAF AlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 12 October 1988
++ Date Last Updated: 4 June 1988
++ Description:
++ This package provides functions for the integration of
++ algebraic integrands over transcendental functions;

AlgebraicIntegration(R, F): Exports == Implementation where
  R : Join(OrderedSet, IntegralDomain)
  F : Join(AlgebraicallyClosedField, FunctionSpace R)

  SY  ==> Symbol
  N   ==> NonNegativeInteger
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
  UPUP==> SparseUnivariatePolynomial RF
  IR  ==> IntegrationResult F
  IR2 ==> IntegrationResultFunctions2(curve, F)
  ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
  FAIL==> error "failed - cannot handle that integrand"

  Exports ==> with
    algint: (F, K, K, UP -> UP) -> IR
      ++ algint(f, x, y, d) returns the integral of \spad{f(x,y)dx}
```

```
    ++ where y is an algebraic function of x;
    ++ d is the derivation to use on \spad{k[x]}.

  Implementation ==> add
    import ChangeOfVariable(F, UP, UPUP)
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                        K, R, P, F)

    rootintegrate: (F, K, K, UP -> UP) -> IR
    algintegrate : (F, K, K, UP -> UP) -> IR
    UPUP2F       : (UPUP, RF, K, K) -> F
    F2UPUP       : (F, K, K, UP) -> UPUP
    UP2UPUP      : (UP, K) -> UPUP

    F2UPUP(f, kx, k, p) == UP2UPUP(univariate(f, k, p), kx)

    rootintegrate(f, t, k, derivation) ==
      r1     := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
      f1     := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
      r      := radPoly(r1.poly)::Record(radicand:RF, deg:N)
      q      := retract(r.radicand)
      curve  := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
      map(x1+->UPUP2F(lift x1, r1.coef, t, k),
                           algintegrate(reduce f1, derivation)$ALG)$IR2

    algintegrate(f, t, k, derivation) ==
      r1     := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
      f1     := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
      modulus:= UP2UPUP(p := minPoly k, t)
      curve  := AlgebraicFunctionField(F, UP, UPUP, r1.poly)
      map(x1+->UPUP2F(lift x1, r1.coef, t, k),
                           algintegrate(reduce f1, derivation)$ALG)$IR2

    UP2UPUP(p, k) ==
      map(x1+->univariate(x1,k),p)$SparseUnivariatePolynomialFunctions2(F,RF)

    UPUP2F(p, cf, t, k) ==
      map((x1:RF):F+->multivariate(x1, t),
         p)$SparseUnivariatePolynomialFunctions2(RF, F)
                                          (multivariate(cf, t) * k::F)

    algint(f, t, y, derivation) ==
      is?(y, "nthRoot"::SY) => rootintegrate(f, t, y, derivation)
      is?(y, "rootOf"::SY)  => algintegrate(f, t, y, derivation)
      FAIL
```

———-

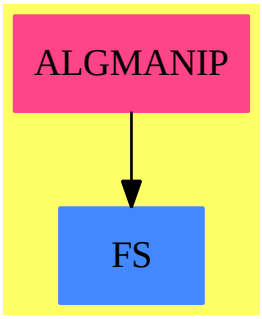
— **INTAF.dotabb** —

```
"INTAF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTAF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTAF" -> "ACF"
"INTAF" -> "FS"
```

———-

2.7 package ALGMANIP AlgebraicManipulations

2.7.1 AlgebraicManipulations (ALGMANIP)



Exports:

ratDenom	ratPoly	rootKerSimp	rootPower
rootProduct	rootSimp	rootSplit	

— **package ALGMANIP AlgebraicManipulations** —

```
)abbrev package ALGMANIP AlgebraicManipulations
++ Author: Manuel Bronstein
++ Date Created: 28 Mar 1988
++ Date Last Updated: 5 August 1993
++ Keywords: algebraic, manipulation.
++ Description:
++ AlgebraicManipulations provides functions to simplify and expand
++ expressions involving algebraic operators.

AlgebraicManipulations(R, F): Exports == Implementation where
  R : IntegralDomain
  F : Join(Field, ExpressionSpace) with
    numer  : $ -> SparseMultivariatePolynomial(R, Kernel $)
      ++ numer(x) \undocumented
```

```
    denom  : $ -> SparseMultivariatePolynomial(R, Kernel $)
      ++ denom(x) \undocumented
    coerce : SparseMultivariatePolynomial(R, Kernel $) -> $
      ++ coerce(x) \undocumented

  N  ==> NonNegativeInteger
  Z  ==> Integer
  OP ==> BasicOperator
  SY ==> Symbol
  K  ==> Kernel F
  P  ==> SparseMultivariatePolynomial(R, K)
  RF ==> Fraction P
  REC ==> Record(ker:List K, exponent: List Z)
  ALGOP ==> "%alg"
  NTHR  ==> "nthRoot"

  Exports ==> with
    rootSplit: F -> F
      ++ rootSplit(f) transforms every radical of the form
      ++ \spad{(a/b)**(1/n)} appearing in f into \spad{a**(1/n) / b**(1/n)}.
      ++ This transformation is not in general valid for all
      ++ complex numbers \spad{a} and b.
    ratDenom  : F -> F
      ++ ratDenom(f) rationalizes the denominators appearing in f
      ++ by moving all the algebraic quantities into the numerators.
    ratDenom  : (F, F) -> F
      ++ ratDenom(f, a) removes \spad{a} from the denominators in f
      ++ if \spad{a} is an algebraic kernel.
    ratDenom  : (F, List F) -> F
      ++ ratDenom(f, [a1,...,an]) removes the ai's which are
      ++ algebraic kernels from the denominators in f.
    ratDenom  : (F, List K) -> F
      ++ ratDenom(f, [a1,...,an]) removes the ai's which are
      ++ algebraic from the denominators in f.
    ratPoly  : F -> SparseUnivariatePolynomial F
      ++ ratPoly(f) returns a polynomial p such that p has no
      ++ algebraic coefficients, and \spad{p(f) = 0}.
    if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
      and F has FunctionSpace(R) then
        rootPower  : F -> F
          ++ rootPower(f) transforms every radical power of the form
          ++ \spad{(a**(1/n))**m} into a simpler form if \spad{m} and
          ++ \spad{n} have a common factor.
        rootProduct: F -> F
          ++ rootProduct(f) combines every product of the form
          ++ \spad{(a**(1/n))**m * (a**(1/s))**t} into a single power
          ++ of a root of \spad{a}, and transforms every radical power
          ++ of the form \spad{(a**(1/n))**m} into a simpler form.
        rootSimp   : F -> F
          ++ rootSimp(f) transforms every radical of the form
```

```
          ++ \spad{(a * b**(q*n+r))**(1/n)} appearing in f into
          ++ \spad{b**q * (a * b**r)**(1/n)}.
          ++ This transformation is not in general valid for all
          ++ complex numbers b.
        rootKerSimp: (OP, F, N) -> F
          ++ rootKerSimp(op,f,n) should be local but conditional.

  Implementation ==> add
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,P,F)

    innerRF    : (F, List K) -> F
    rootExpand : K -> F
    algkernels : List K -> List K
    rootkernels: List K -> List K

    dummy := kernel(new()$SY)$K

    ratDenom x                == innerRF(x, algkernels tower x)
    ratDenom(x:F, l:List K):F == innerRF(x, algkernels l)
    ratDenom(x:F, y:F)        == ratDenom(x, [y])
    ratDenom(x:F, l:List F)   == ratDenom(x, [retract(y)@K for y in l]$List(K))
    algkernels l  == select_!((z1:K):Boolean +-> has?(operator z1, ALGOP), l)
    rootkernels l == select_!((z1:K):Boolean +-> is?(operator z1, NTHR::SY), l)

    ratPoly x ==
      numer univariate(denom(ratDenom inv(dummy::P::F - x))::F, dummy)

    rootSplit x ==
      lk := rootkernels tower x
      eval(x, lk, [rootExpand k for k in lk])

    rootExpand k ==
      x  := first argument k
      n  := second argument k
      op := operator k
      op(numer(x)::F, n) / op(denom(x)::F, n)

-- all the kernels in ll must be algebraic
    innerRF(x, ll) ==
      empty?(l := sort_!((z1:K,z2:K):Boolean +-> z1 > z2,kernels x)$List(K)) or
        empty? setIntersection(ll, tower x) => x
      lk := empty()$List(K)
      while not member?(k := first l, ll) repeat
        lk := concat(k, lk)
        empty?(l := rest l) =>
          return eval(x, lk, [map((z3:F):F+->innerRF(z3,ll), kk) for kk in lk])
      q := univariate(eval(x, lk,
             [map((z4:F):F+->innerRF(z4,ll),kk) for kk in lk]),k,minPoly k)
      map((z5:F):F+->innerRF(z5, ll), q) (map((z6:F):F+->innerRF(z6, ll), k))
```

```
    if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
     and F has FunctionSpace(R) then
      import PolynomialRoots(IndexedExponents K, K, R, P, F)

      sroot  : K -> F
      inroot : (OP, F, N) -> F
      radeval: (P, K) -> F
      breakup: List K -> List REC

      if R has RadicalCategory then
        rootKerSimp(op, x, n) ==
          (r := retractIfCan(x)@Union(R, "failed")) case R =>
             nthRoot(r::R, n)::F
          inroot(op, x, n)
      else
        rootKerSimp(op, x, n) == inroot(op, x, n)

-- l is a list of nth-roots, returns a list of records of the form
-- [a**(1/n1),a**(1/n2),...], [n1,n2,...]]
-- such that the whole list covers l exactly
      breakup l ==
        empty? l => empty()
        k := first l
        a := first(arg := argument(k := first l))
        n := retract(second arg)@Z
        expo := empty()$List(Z)
        others := same := empty()$List(K)
        for kk in rest l repeat
          if (a = first(arg := argument kk)) then
            same := concat(kk, same)
            expo := concat(retract(second arg)@Z, expo)
          else others := concat(kk, others)
        ll := breakup others
        concat([concat(k, same), concat(n, expo)], ll)

      rootProduct x ==
        for rec in breakup rootkernels tower x repeat
          k0 := first(l := rec.ker)
          nx := numer x; dx := denom x
          if empty? rest l then x := radeval(nx, k0) / radeval(dx, k0)
          else
            n  := lcm(rec.exponent)
            k  := kernel(operator k0, [first argument k0, n::F], height k0)$K
            lv := [monomial(1, k, (n quo m)::N) for m in rec.exponent]$List(P)
            x  := radeval(eval(nx, l, lv), k) / radeval(eval(dx, l, lv), k)
        x

      rootPower x ==
        for k in rootkernels tower x repeat
          x := radeval(numer x, k) / radeval(denom x, k)
```

```
        x

-- replaces (a**(1/n))**m in p by a power of a simpler radical of a if
-- n and m have a common factor
      radeval(p, k) ==
        a := first(arg := argument k)
        n := (retract(second arg)@Integer)::NonNegativeInteger
        ans:F := 0
        q := univariate(p, k)
        while (d := degree q) > 0 repeat
          term :=
--            one?(g := gcd(d, n)) => monomial(1, k, d)
            ((g := gcd(d, n)) = 1) => monomial(1, k, d)
            monomial(1, kernel(operator k, [a,(n quo g)::F], height k), d quo g)
          ans := ans + leadingCoefficient(q)::F * term::F
          q := reductum q
        leadingCoefficient(q)::F + ans

      inroot(op, x, n) ==
--        one? x => x
        (x = 1) => x
--        (x ^= -1) and (one?(num := numer x) or (num = -1)) =>
        (x ^= -1) and (((num := numer x) = 1) or (num = -1)) =>
          inv inroot(op, (num * denom x)::F, n)
        (u := isExpt(x, op)) case "failed" => kernel(op, [x, n::F])
        pr := u::Record(var:K, exponent:Integer)
        q := pr.exponent /$Fraction(Z)
                                (n * retract(second argument(pr.var))@Z)
        qr := divide(numer q, denom q)
        x  := first argument(pr.var)
        x ** qr.quotient * rootKerSimp(op,x,denom(q)::N) ** qr.remainder

      sroot k ==
        pr := froot(first(arg := argument k),(retract(second arg)@Z)::N)
        pr.coef * rootKerSimp(operator k, pr.radicand, pr.exponent)

      rootSimp x ==
        lk := rootkernels tower x
        eval(x, lk, [sroot k for k in lk])
```

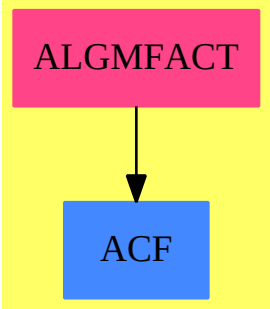
— ALGMANIP.dotabb —

```
"ALGMANIP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ALGMANIP"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ALGMANIP" -> "FS"
```

———-

2.8 package ALGMFACT AlgebraicMultFact

2.8.1 AlgebraicMultFact (ALGMFACT)



Exports:
factor

— package ALGMFACT AlgebraicMultFact —

```
)abbrev package ALGMFACT AlgebraicMultFact
++ Author: P. Gianni
++ Date Created: 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package factors multivariate polynomials over the
++ domain of \spadtype{AlgebraicNumber} by allowing the user
++ to specify a list of algebraic numbers generating the particular
++ extension to factor over.

AlgebraicMultFact(OV,E,P) : C == T
 where
  AN          ==> AlgebraicNumber
  OV          :   OrderedSet
  E           :   OrderedAbelianMonoidSup
  P           :   PolynomialCategory(AN,E,OV)
  BP          ==> SparseUnivariatePolynomial AN
  Z             ==> Integer
```

```
  MParFact     ==> Record(irr:P,pow:Z)
  USP          ==> SparseUnivariatePolynomial P
  SUParFact    ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact   ==> Record(contp:R,factors:List MParFact)

                 --  contp   =  content,
                 --  factors =  List of irreducible factors with exponent
  L          ==> List

  C == with
    factor      :   (P,L AN)  ->  Factored P
      ++ factor(p,lan) factors the polynomial p over the extension
      ++ generated by the algebraic numbers given by the list lan.
    factor      :   (USP,L AN)  ->  Factored USP
      ++ factor(p,lan) factors the polynomial p over the extension
      ++ generated by the algebraic numbers given by the list lan.
      ++ p is presented as a univariate polynomial with multivariate
      ++ coefficients.
  T == add

    AF := AlgFactor(BP)

    INNER ==> InnerMultFact(OV,E,AN,P)

    factor(p:P,lalg:L AN) : Factored P ==
      factor(p,(z1:BP):Factored(BP) +-> factor(z1,lalg)$AF)$INNER

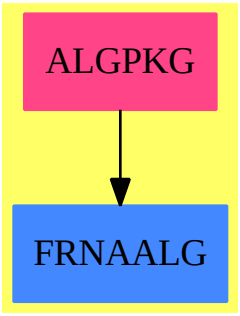
    factor(up:USP,lalg:L AN) : Factored USP ==
      factor(up,(z1:BP):Factored(BP) +-> factor(z1,lalg)$AF)$INNER
```

— ALGMFACT.dotabb —

```
"ALGMFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ALGMFACT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"ALGMFACT" -> "ACF"
```

2.9 package ALGPKG AlgebraPackage

2.9.1 AlgebraPackage (ALGPKG)



Exports:

basis	basisOfCenter	basisOfCentroid	basisOfCommutingElements	basisOfL
basisOfLeftNucleus	basisOfLeftNucloid	basisOfMiddleNucleus	basisOfNucleus	basisOfR
basisOfRightNucleus	basisOfRightNucloid	biRank	doubleRank	leftRank
radicalOfLeftTraceForm	rightRank	weakBiRank		

— package ALGPKG AlgebraPackage —

```
)abbrev package ALGPKG AlgebraPackage
++ Authors: J. Grabmeier, R. Wisbauer
++ Date Created: 04 March 1991
++ Date Last Updated: 04 April 1992
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: rank, nucleus, nucloid, structural constants
++ Reference:
++  R.S. Pierce: Associative Algebras
++  Graduate Texts in Mathematics 88
++  Springer-Verlag,  Heidelberg, 1982, ISBN 0-387-90693-2
++
++  R.D. Schafer: An Introduction to Nonassociative Algebras
++  Academic Press, New York, 1966
++
++  A. Woerz-Busekros: Algebra in Genetics
++  Lectures Notes in Biomathematics 36,
++  Springer-Verlag,  Heidelberg, 1980
++ Description:
++ AlgebraPackage assembles a variety of useful functions for
++ general algebras.

AlgebraPackage(R:IntegralDomain, A: FramedNonAssociativeAlgebra(R)): _
```

```
 public == private where

V  ==> Vector
M  ==> Matrix
I  ==> Integer
NNI  ==> NonNegativeInteger
REC  ==> Record(particular: Union(V R,"failed"),basis: List V R)
LSMP ==> LinearSystemMatrixPackage(R,V R,V R, M R)

public ==>  with

    leftRank: A -> NonNegativeInteger
      ++ leftRank(x) determines the number of linearly independent elements
      ++ in \spad{x*b1},...,\spad{x*bn},
      ++ where \spad{b=[b1,...,bn]} is a basis.
    rightRank: A -> NonNegativeInteger
      ++ rightRank(x) determines the number of linearly independent elements
      ++ in \spad{b1*x},...,\spad{bn*x},
      ++ where \spad{b=[b1,...,bn]} is a basis.
    doubleRank: A -> NonNegativeInteger
      ++ doubleRank(x) determines the number of linearly
      ++ independent elements
      ++ in \spad{b1*x},...,\spad{x*bn},
      ++ where \spad{b=[b1,...,bn]} is a basis.
    weakBiRank: A -> NonNegativeInteger
      ++ weakBiRank(x) determines the number of
      ++ linearly independent elements
      ++ in the \spad{bi*x*bj}, \spad{i,j=1,...,n},
      ++ where \spad{b=[b1,...,bn]} is a basis.
    biRank: A -> NonNegativeInteger
      ++ biRank(x) determines the number of linearly independent elements
      ++ in \spad{x}, \spad{x*bi}, \spad{bi*x}, \spad{bi*x*bj},
      ++ \spad{i,j=1,...,n},
      ++ where \spad{b=[b1,...,bn]} is a basis.
      ++ Note that if \spad{A} has a unit,
      ++ then doubleRank, weakBiRank, and biRank coincide.
    basisOfCommutingElements: () -> List A
      ++ basisOfCommutingElements() returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = commutator(x,a)} for all
      ++ \spad{a} in \spad{A}.
    basisOfLeftAnnihilator: A -> List A
      ++ basisOfLeftAnnihilator(a) returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = x*a}.
    basisOfRightAnnihilator: A -> List A
      ++ basisOfRightAnnihilator(a) returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = a*x}.
    basisOfLeftNucleus: () -> List A
      ++ basisOfLeftNucleus() returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = associator(x,a,b)}
      ++ for all \spad{a},b in \spad{A}.
```

```
    basisOfRightNucleus: () -> List A
      ++ basisOfRightNucleus() returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = associator(a,b,x)}
      ++ for all \spad{a},b in \spad{A}.
    basisOfMiddleNucleus: () -> List A
      ++ basisOfMiddleNucleus() returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{0 = associator(a,x,b)}
      ++ for all \spad{a},b in \spad{A}.
    basisOfNucleus: () -> List A
      ++ basisOfNucleus() returns a basis of the space of
      ++ all x of \spad{A} satisfying
      ++ \spad{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
      ++ for all \spad{a},b in \spad{A}.
    basisOfCenter: () -> List A
      ++ basisOfCenter() returns a basis of the space of
      ++ all x of \spad{A} satisfying \spad{commutator(x,a) = 0} and
      ++ \spad{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
      ++ for all \spad{a},b in \spad{A}.
    basisOfLeftNucloid:()-> List Matrix R
      ++ basisOfLeftNucloid() returns a basis of the space of
      ++ endomorphisms of \spad{A} as right module.
      ++ Note that left nucloid coincides with left nucleus
      ++ if \spad{A} has a unit.
    basisOfRightNucloid:()-> List Matrix R
      ++ basisOfRightNucloid() returns a basis of the space of
      ++ endomorphisms of \spad{A} as left module.
      ++ Note that right nucloid coincides with right nucleus
      ++ if \spad{A} has a unit.
    basisOfCentroid:()-> List Matrix R
      ++ basisOfCentroid() returns a basis of the centroid, i.e. the
      ++ endomorphism ring of \spad{A} considered as \spad{(A,A)}-bimodule.
    radicalOfLeftTraceForm: () -> List A
      ++ radicalOfLeftTraceForm() returns basis for null space of
      ++ \spad{leftTraceMatrix()}, if the algebra is
      ++ associative, alternative or a Jordan algebra, then this
      ++ space equals the radical (maximal nil ideal) of the algebra.
    if R has EuclideanDomain then
      basis : V A ->  V A
        ++ basis(va) selects a basis from the elements of va.


private ==>  add

    -- constants

    n  : PositiveInteger := rank()$A
    n2 : PositiveInteger := n*n
    n3 : PositiveInteger := n*n2
    gamma : Vector Matrix R  := structuralConstants()$A
```

```
-- local functions

convVM : Vector R -> Matrix R
  -- converts n2-vector to (n,n)-matrix row by row
convMV : Matrix R -> Vector R
  -- converts n-square matrix to  n2-vector row by row
convVM v  ==
  cond : Matrix(R) := new(n,n,0$R)$M(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n  repeat
      z := z+1
      setelt(cond,i,j,v.z)
  cond


-- convMV m ==
--     vec : Vector(R) := new(n*n,0$R)
--     z : Integer := 0
--     for i in 1..n repeat
--       for j in 1..n  repeat
--         z := z+1
--         setelt(vec,z,elt(m,i,j))
--     vec


radicalOfLeftTraceForm() ==
  ma : M R := leftTraceMatrix()$A
  map(represents, nullSpace ma)$ListFunctions2(Vector R, A)


basisOfLeftAnnihilator a ==
  ca : M R := transpose (coordinates(a) :: M R)
  cond : M R := reduce(vertConcat$(M R),
    [ca*transpose(gamma.i) for i in 1..#gamma])
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfRightAnnihilator a ==
  ca : M R := transpose (coordinates(a) :: M R)
  cond : M R := reduce(vertConcat$(M R),
    [ca*(gamma.i) for i in 1..#gamma])
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfLeftNucloid() ==
  cond : Matrix(R) := new(n3,n2,0$R)$M(R)
  condo: Matrix(R) := new(n3,n2,0$R)$M(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n repeat
```

```
      r1  : Integer := 0
      for k in 1..n repeat
        z := z + 1
        -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
        r2 : Integer := i
        for r in 1..n repeat
          r1 := r1 + 1
          -- here r1 equals (k-1)*n+r (loop-invariant)
          setelt(cond,z,r1,elt(gamma.r,i,j))
          -- here r2 equals (r-1)*n+i (loop-invariant)
          setelt(condo,z,r2,-elt(gamma.k,r,j))
          r2 := r2 + n
  [convVM(sol) for sol in nullSpace(cond+condo)]

basisOfCommutingElements() ==
  --gamma1 := first gamma
  --gamma1 := gamma1 - transpose gamma1
  --cond : Matrix(R) := gamma1 :: Matrix(R)
  --for  i in  2..n repeat
  --  gammak := gamma.i
  --  gammak := gammak - transpose gammak
  --  cond :=  vertConcat(cond, gammak :: Matrix(R))$Matrix(R)
  --map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

  cond : M R := reduce(vertConcat$(M R),
    [(gam := gamma.i) - transpose gam for i in 1..#gamma])
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfLeftNucleus() ==
  condi: Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
   for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
      for i in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
          entry :=  entry+elt(gamma.l,j,k)*elt(gamma.s,i,l)_
                         -elt(gamma.l,i,j)*elt(gamma.s,l,k)
        setelt(condi,z,i,entry)$Matrix(R)
  map(represents, nullSpace condi)$ListFunctions2(Vector R,A)

basisOfRightNucleus() ==
  condo : Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
   for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
```

```
      for i in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
          entry :=  entry+elt(gamma.l,k,i)*elt(gamma.s,j,l) _
                         -elt(gamma.l,j,k)*elt(gamma.s,l,i)
        setelt(condo,z,i,entry)$Matrix(R)
  map(represents, nullSpace condo)$ListFunctions2(Vector R,A)

basisOfMiddleNucleus() ==
  conda : Matrix(R) := new(n3,n,0$R)$Matrix(R)
  z : Integer := 0
  for k in 1..n repeat
   for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
      for i in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
          entry :=  entry+elt(gamma.l,j,i)*elt(gamma.s,l,k)
                         -elt(gamma.l,i,k)*elt(gamma.s,j,l)
        setelt(conda,z,i,entry)$Matrix(R)
  map(represents, nullSpace conda)$ListFunctions2(Vector R,A)


basisOfNucleus() ==
  condi: Matrix(R) := new(3*n3,n,0$R)$Matrix(R)
  z : Integer := 0
  u : Integer := n3
  w : Integer := 2*n3
  for k in 1..n repeat
   for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
      u := u+1
      w := w+1
      for i in 1..n repeat
        entry : R := 0
        enter : R := 0
        ent   : R := 0
        for l in 1..n repeat
          entry :=  entry + elt(gamma.l,j,k)*elt(gamma.s,i,l) _
                          - elt(gamma.l,i,j)*elt(gamma.s,l,k)
          enter :=  enter + elt(gamma.l,k,i)*elt(gamma.s,j,l) _
                          - elt(gamma.l,j,k)*elt(gamma.s,l,i)
          ent :=  ent  +  elt(gamma.l,j,k)*elt(gamma.s,i,l) _
                       -  elt(gamma.l,j,i)*elt(gamma.s,l,k)
        setelt(condi,z,i,entry)$Matrix(R)
        setelt(condi,u,i,enter)$Matrix(R)
        setelt(condi,w,i,ent)$Matrix(R)
  map(represents, nullSpace condi)$ListFunctions2(Vector R,A)
```

```
basisOfCenter() ==
  gamma1 := first gamma
  gamma1 := gamma1 - transpose gamma1
  cond : Matrix(R) := gamma1 :: Matrix(R)
  for  i in  2..n repeat
    gammak := gamma.i
    gammak := gammak - transpose gammak
    cond :=  vertConcat(cond, gammak :: Matrix(R))$Matrix(R)
  B := cond :: Matrix(R)
  condi: Matrix(R) := new(2*n3,n,0$R)$Matrix(R)
  z : Integer := 0
  u : Integer := n3
  for k in 1..n repeat
   for j in 1..n repeat
    for s in 1..n repeat
      z := z+1
      u := u+1
      for i in 1..n repeat
        entry : R := 0
        enter : R := 0
        for l in 1..n repeat
          entry :=  entry + elt(gamma.l,j,k)*elt(gamma.s,i,l) _
                          - elt(gamma.l,i,j)*elt(gamma.s,l,k)
          enter :=  enter + elt(gamma.l,k,i)*elt(gamma.s,j,l) _
                          - elt(gamma.l,j,k)*elt(gamma.s,l,i)
        setelt(condi,z,i,entry)$Matrix(R)
        setelt(condi,u,i,enter)$Matrix(R)
  D := vertConcat(condi,B)$Matrix(R)
  map(represents, nullSpace D)$ListFunctions2(Vector R, A)

basisOfRightNucloid() ==
  cond : Matrix(R) := new(n3,n2,0$R)$M(R)
  condo: Matrix(R) := new(n3,n2,0$R)$M(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n repeat
      r1  : Integer := 0
      for k in 1..n repeat
        z := z + 1
        -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
        r2 : Integer := i
        for r in 1..n repeat
          r1 := r1 + 1
          -- here r1 equals (k-1)*n+r (loop-invariant)
          setelt(cond,z,r1,elt(gamma.r,j,i))
          -- here r2 equals (r-1)*n+i (loop-invariant)
          setelt(condo,z,r2,-elt(gamma.k,j,r))
          r2 := r2 + n
  [convVM(sol) for sol in nullSpace(cond+condo)]
```

```
basisOfCentroid() ==
  cond : Matrix(R) := new(2*n3,n2,0$R)$M(R)
  condo: Matrix(R) := new(2*n3,n2,0$R)$M(R)
  z : Integer := 0
  u : Integer := n3
  for i in 1..n repeat
    for j in 1..n repeat
      r1  : Integer := 0
      for k in 1..n repeat
        z := z + 1
        u := u + 1
        -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
        -- u equals n**3 + (i-1)*n*n+(j-1)*n+k (loop-invariant)
        r2 : Integer := i
        for r in 1..n repeat
          r1 := r1 + 1
          -- here r1 equals (k-1)*n+r (loop-invariant)
          setelt(cond,z,r1,elt(gamma.r,i,j))
          setelt(cond,u,r1,elt(gamma.r,j,i))
          -- here r2 equals (r-1)*n+i (loop-invariant)
          setelt(condo,z,r2,-elt(gamma.k,r,j))
          setelt(condo,u,r2,-elt(gamma.k,j,r))
          r2 := r2 + n
  [convVM(sol) for sol in nullSpace(cond+condo)]


doubleRank x ==
  cond : Matrix(R) := new(2*n,n,0$R)
  for k in 1..n repeat
   z : Integer := 0
   u : Integer := n
   for j in 1..n repeat
     z := z+1
     u := u+1
     entry : R := 0
     enter : R := 0
     for i in 1..n repeat
       entry := entry + elt(x,i)*elt(gamma.k,j,i)
       enter := enter + elt(x,i)*elt(gamma.k,i,j)
     setelt(cond,z,k,entry)$Matrix(R)
     setelt(cond,u,k,enter)$Matrix(R)
  rank(cond)$(M R)

weakBiRank(x) ==
  cond : Matrix(R) := new(n2,n,0$R)$Matrix(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n repeat
      z := z+1
```

```
      for k in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
         for s in 1..n repeat
          entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
        setelt(cond,z,k,entry)$Matrix(R)
  rank(cond)$(M R)

biRank(x) ==
  cond : Matrix(R) := new(n2+2*n+1,n,0$R)$Matrix(R)
  z : Integer := 0
  for j in 1..n repeat
    for i in 1..n repeat
      z := z+1
      for k in 1..n repeat
        entry : R := 0
        for l in 1..n repeat
         for s in 1..n repeat
          entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
        setelt(cond,z,k,entry)$Matrix(R)
  u : Integer := n*n
  w : Integer := n*(n+1)
  c := n2 + 2*n + 1
  for j in 1..n repeat
     u := u+1
     w := w+1
     for k in 1..n repeat
       entry : R := 0
       enter : R := 0
       for i in 1..n repeat
         entry := entry + elt(x,i)*elt(gamma.k,j,i)
         enter := enter + elt(x,i)*elt(gamma.k,i,j)
       setelt(cond,u,k,entry)$Matrix(R)
       setelt(cond,w,k,enter)$Matrix(R)
     setelt(cond,c,j, elt(x,j))
  rank(cond)$(M R)

leftRank x ==
  cond : Matrix(R) := new(n,n,0$R)
  for k in 1..n repeat
   for j in 1..n repeat
     entry : R := 0
     for i in 1..n repeat
       entry := entry + elt(x,i)*elt(gamma.k,i,j)
     setelt(cond,j,k,entry)$Matrix(R)
  rank(cond)$(M R)

rightRank x ==
  cond : Matrix(R) := new(n,n,0$R)
  for k in 1..n repeat
```

```
    for j in 1..n repeat
      entry : R := 0
      for i in 1..n repeat
        entry := entry + elt(x,i)*elt(gamma.k,j,i)
      setelt(cond,j,k,entry)$Matrix(R)
    rank(cond)$(M R)


  if R has EuclideanDomain then
    basis va ==
      v : V A := remove(zero?, va)$(V A)
      v : V A := removeDuplicates v
      empty? v =>  [0$A]
      m : Matrix R := coerce(coordinates(v.1))$(Matrix R)
      for i in 2..maxIndex v repeat
        m := horizConcat(m,coerce(coordinates(v.i))$(Matrix R) )
      m := rowEchelon m
      lj : List Integer := []
      h : Integer := 1
      mRI : Integer := maxRowIndex m
      mCI : Integer := maxColIndex m
      finished? : Boolean := false
      j : Integer := 1
      while not finished? repeat
        not zero? m(h,j) =>  -- corner found
          lj := cons(j,lj)
          h := mRI
          while zero? m(h,j) repeat h := h-1
          finished? := (h = mRI)
          if not finished? then h := h+1
        if j < mCI then
          j := j + 1
        else
          finished? := true
      [v.j for j in reverse lj]
```

———-

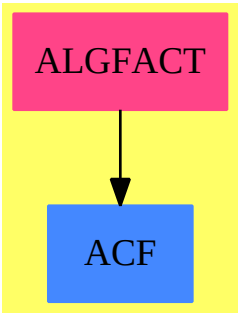
— ALGPKG.dotabb —

```
"ALGPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ALGPKG"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"ALGPKG" -> "FRNAALG"
```

———-

2.10 package ALGFACT AlgFactor

2.10.1 AlgFactor (ALGFACT)



Exports:
doublyTransitive? factor split

— package ALGFACT AlgFactor —

```
)abbrev package ALGFACT AlgFactor
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: ???
++ Description:
++ Factorization of univariate polynomials with coefficients in
++ \spadtype{AlgebraicNumber}.

AlgFactor(UP): Exports == Implementation where
  UP: UnivariatePolynomialCategory AlgebraicNumber

  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Integer
  AN  ==> AlgebraicNumber
  K   ==> Kernel AN
  UPQ ==> SparseUnivariatePolynomial Q
  SUP ==> SparseUnivariatePolynomial AN
  FR  ==> Factored UP

  Exports ==> with
    factor: (UP, List AN) -> FR
      ++ factor(p, [a1,...,an]) returns a prime factorisation of p
      ++ over the field generated by its coefficients and a1,...,an.
    factor: UP            -> FR
      ++ factor(p) returns a prime factorisation of p
      ++ over the field generated by its coefficients.
```

```
    split : UP            -> FR
      ++ split(p) returns a prime factorisation of p
      ++ over its splitting field.
    doublyTransitive?: UP -> Boolean
      ++ doublyTransitive?(p) is true if p is irreducible over
      ++ over the field K generated by its coefficients, and
      ++ if \spad{p(X) / (X - a)} is irreducible over
      ++ \spad{K(a)} where \spad{p(a) = 0}.

  Implementation ==> add
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                          K, Z, SparseMultivariatePolynomial(Z, K), AN)

    UPCF2 ==> UnivariatePolynomialCategoryFunctions2

    fact    : (UP,  List K) -> FR
    ifactor : (SUP, List K) -> Factored SUP
    extend  : (UP, Z) -> FR
    allk    : List AN -> List K
    downpoly: UP  -> UPQ
    liftpoly: UPQ -> UP
    irred?  : UP  -> Boolean

    allk l       == removeDuplicates concat [kernels x for x in l]
    liftpoly p   == map(x +-> x::AN,  p)$UPCF2(Q, UPQ, AN, UP)
    downpoly p   == map(x +-> retract(x)@Q, p)$UPCF2(AN, UP ,Q, UPQ)
    ifactor(p,l) == (fact(p pretend UP, l)) pretend Factored(SUP)
    factor p     == fact(p, allk coefficients p)

    factor(p, l) ==
      fact(p, allk removeDuplicates concat(l, coefficients p))

    split p ==
      fp := factor p
      unit(fp) *
            _*/[extend(fc.factor, fc.exponent) for fc in factors fp]

    extend(p, n) ==
--      one? degree p => primeFactor(p, n)
      (degree p = 1) => primeFactor(p, n)
      q := monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP
      primeFactor(q, n) * split((p exquo q)::UP) ** (n::N)

    doublyTransitive? p ==
      irred? p and irred?((p exquo
        (monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP))::UP)

    irred? p ==
      fp := factor p
--      one? numberOfFactors fp and one? nthExponent(fp, 1)
```

```
      (numberOfFactors fp = 1) and  (nthExponent(fp, 1) = 1)

    fact(p, l) ==
--      one? degree p => primeFactor(p, 1)
      (degree p = 1) => primeFactor(p, 1)
      empty? l =>
        dr := factor(downpoly p)$RationalFactorize(UPQ)
        (liftpoly unit dr) *
          _*/[primeFactor(liftpoly dc.factor,dc.exponent)
            for dc in factors dr]
      q   := minPoly(alpha := "max"/l)$AN
      newl  := remove((x:K):Boolean +-> alpha = x, l)
      sae := SimpleAlgebraicExtension(AN, SUP, q)
      ups := SparseUnivariatePolynomial sae
      fr  := factor(map(x +-> reduce univariate(x, alpha, q),p)_
               $UPCF2(AN, UP, sae, ups),_
                 x +-> ifactor(x, newl))$InnerAlgFactor(AN, SUP, sae, ups)
      newalpha := alpha::AN
      map((x:sae):AN +-> (lift(x)$sae) newalpha, unit fr)_
        $UPCF2(sae, ups, AN, UP) *
         _*/[primeFactor(map((y:sae):AN +-> (lift(y)$sae) newalpha,fc.factor)_
            $UPCF2(sae, ups, AN, UP),
              fc.exponent) for fc in factors fr]
```

———-

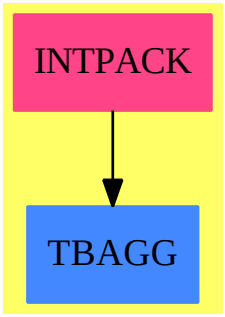
— **ALGFACT.dotabb** —

```
"ALGFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ALGFACT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"ALGFACT" -> "ACF"
```

———-

2.11 package INTPACK AnnaNumericalIntegrationPackage

2.11.1 AnnaNumericalIntegrationPackage (INTPACK)



Exports:
integrate measure

— package INTPACK AnnaNumericalIntegrationPackage —

```
)abbrev package INTPACK AnnaNumericalIntegrationPackage
++ Author: Brian Dupee
++ Date Created: August 1994
++ Date Last Updated: December 1997
++ Basic Operations: integrate, measure
++ Related Constructors: Result, RoutinesTable
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \axiomType{AnnaNumericalIntegrationPackage} is a \axiom{package}
++ of functions for the \axiom{category}
++ \axiomType{NumericalIntegrationCategory}
++ with \axiom{measure}, and \axiom{integrate}.

AnnaNumericalIntegrationPackage(): EE == II where

 EDF  ==> Expression DoubleFloat
 DF  ==> DoubleFloat
 EF  ==> Expression Float
 F  ==> Float
 INT  ==> Integer
 SOCDF  ==> Segment OrderedCompletion DoubleFloat
 OCDF  ==> OrderedCompletion DoubleFloat
```

```
SBOCF  ==> SegmentBinding OrderedCompletion Float
LSOCF  ==> List Segment OrderedCompletion Float
SOCF  ==> Segment OrderedCompletion Float
OCF  ==> OrderedCompletion Float
LS  ==> List Symbol
S  ==> Symbol
LST  ==> List String
ST  ==> String
RT  ==> RoutinesTable
NIA  ==> Record(var:S, fn:EDF, range:SOCDF, abserr:DF, relerr:DF)
MDNIA  ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
IFL  ==> List(Record(ifail:Integer,instruction:String))
Entry  ==> Record(chapter:String, type:String, domainName: String,
                  defaultMin:F, measure:F, failList:IFL, explList:List String)
Measure  ==> Record(measure:F, name:ST, explanations:LST, extra:Result)
EE ==> with
 integrate: (EF,SOCF,F,F,RT) -> Result
   ++ integrate(exp, a..b, epsrel, routines) is a top level ANNA function
   ++ to integrate an expression, {\tt exp}, over a given range {\tt a}
   ++ to {\tt b} to the required absolute and relative accuracy using
   ++ the routines available in the RoutinesTable provided.
   ++
   ++ It iterates over the \axiom{domains} of
   ++ \axiomType{NumericalIntegrationCategory}
   ++ to get the name and other
   ++ relevant information of the the (domain of the) numerical
   ++ routine likely to be the most appropriate,
   ++ i.e. have the best \axiom{measure}.
   ++
   ++ It then performs the integration of the given expression
   ++ on that \axiom{domain}.
 integrate: NumericalIntegrationProblem -> Result
   ++ integrate(IntegrationProblem) is a top level ANNA function
   ++ to integrate an expression over a given range or ranges
   ++ to the required absolute and relative accuracy.
   ++
   ++ It iterates over the \axiom{domains} of
   ++ \axiomType{NumericalIntegrationCategory} to get the name and other
   ++ relevant information of the the (domain of the) numerical
   ++ routine likely to be the most appropriate,
   ++ i.e. have the best \axiom{measure}.
   ++
   ++ It then performs the integration of the given expression
   ++ on that \axiom{domain}.

 integrate: (EF,SOCF,F,F) -> Result
   ++ integrate(exp, a..b, epsabs, epsrel) is a top level ANNA function
   ++ to integrate an expression, {\tt exp}, over a given range {\tt a}
   ++ to {\tt b} to the required absolute and relative accuracy.
   ++
```

```
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.

integrate: (EF,SOCF,F) -> Result
  ++ integrate(exp, a..b, epsrel) is a top level ANNA
  ++ function to integrate an expression, {\tt exp}, over a given
  ++ range {\tt a} to {\tt b} to the required relative accuracy.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.
  ++
  ++ If epsrel = 0, a default absolute accuracy is used.

integrate: (EF,SOCF) -> Result
  ++ integrate(exp, a..b) is a top
  ++ level ANNA function to integrate an expression, {\tt exp},
  ++ over a given range {\tt a} to {\tt b}.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.
  ++
  ++ Default values for the absolute and relative error are used.

integrate:(EF,LSOCF) -> Result
  ++ integrate(exp, [a..b,c..d,...]) is a top
  ++ level ANNA function to integrate a multivariate expression, {\tt exp},
  ++ over a given set of ranges.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
```

```
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.
    ++
    ++ Default values for the absolute and relative error are used.

  integrate:(EF,LSOCF,F) -> Result
    ++ integrate(exp, [a..b,c..d,...], epsrel) is a top
    ++ level ANNA function to integrate a multivariate expression, {\tt exp},
    ++ over a given set of ranges to the required relative
    ++ accuracy.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.
    ++
    ++ If epsrel = 0, a default absolute accuracy is used.

  integrate:(EF,LSOCF,F,F) -> Result
    ++ integrate(exp, [a..b,c..d,...], epsabs, epsrel) is a top
    ++ level ANNA function to integrate a multivariate expression, {\tt exp},
    ++ over a given set of ranges to the required absolute and relative
    ++ accuracy.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.

  integrate:(EF,LSOCF,F,F,RT) -> Result
    ++ integrate(exp, [a..b,c..d,...], epsabs, epsrel, routines) is a top
    ++ level ANNA function to integrate a multivariate expression, {\tt exp},
    ++ over a given set of ranges to the required absolute and relative
    ++ accuracy, using the routines available in the RoutinesTable provided.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
```

```
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.

  measure:NumericalIntegrationProblem -> Measure
    ++ measure(prob) is a top level ANNA function for identifying the most
    ++ appropriate numerical routine for solving the numerical integration
    ++ problem defined by \axiom{prob}.
    ++
    ++ It calls each \axiom{domain} of \axiom{category}
    ++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
    ++ and returns the best
    ++ i.e. the name of the most appropriate domain and any other relevant
    ++ information.
  measure:(NumericalIntegrationProblem,RT) -> Measure
    ++ measure(prob,R) is a top level ANNA function for identifying the most
    ++ appropriate numerical routine from those in the routines table
    ++ provided for solving the numerical integration
    ++ problem defined by \axiom{prob}.
    ++
    ++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
    ++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
    ++ and returns the best
    ++ i.e. the name of the most appropriate domain and any other relevant
    ++ information.
  integrate:(EF,SBOCF,ST) -> Union(Result,"failed")
    ++ integrate(exp, x = a..b, "numerical") is a top level ANNA function to
    ++ integrate an expression, {\tt exp}, over a given range, {\tt a}
    ++ to {\tt b}.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.
    ++
    ++ Default values for the absolute and relative error are used.
    ++
    ++ It is an error of the last argument is not {\tt "numerical"}.
  integrate:(EF,SBOCF,S) -> Union(Result,"failed")
    ++ integrate(exp, x = a..b, numerical) is a top level ANNA function to
    ++ integrate an expression, {\tt exp}, over a given range, {\tt a}
    ++ to {\tt b}.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory} to get the name and other
    ++ relevant information of the the (domain of the) numerical
```

```
   ++ routine likely to be the most appropriate,
   ++ i.e. have the best \axiom{measure}.
   ++
   ++ It then performs the integration of the given expression
   ++ on that \axiom{domain}.
   ++
   ++ Default values for the absolute and relative error are used.
   ++
   ++ It is an error if the last argument is not {\tt numerical}.

II ==> add

 zeroMeasure: Measure -> Result
 scriptedVariables?: MDNIA -> Boolean
 preAnalysis:(Union(nia:NIA,mdnia:MDNIA),RT) -> RT
 measureSpecific:(ST,RT,Union(nia:NIA,mdnia:MDNIA)) -> Record(measure:F,explanations:LST,extra:Result)
 changeName:(Result,ST) -> Result
 recoverAfterFail:(Union(nia:NIA,mdnia:MDNIA),RT,Measure,INT,Result) -> Record(a:Result,b:Measure)
 better?:(Result,Result) -> Boolean
 integrateConstant:(EF,SOCF) -> Result
 integrateConstantList: (EF,LSOCF) -> Result
 integrateArgs:(NumericalIntegrationProblem,RT) -> Result
 integrateSpecific:(Union(nia:NIA,mdnia:MDNIA),ST,Result) -> Result

 import ExpertSystemToolsPackage

 integrateConstantList(exp:EF,ras:LSOCF):Result ==
   c:OCF := ((retract(exp)@F)$EF)::OCF
   b := [hi(j)-lo(j) for j in ras]
   c := c*reduce((x,y) +-> x*y,b)
   a := coerce(c)$AnyFunctions1(OCF)
   text := coerce("Constant Function")$AnyFunctions1(ST)
   construct([[result@S,a],[method@S,text]])$Result

 integrateConstant(exp:EF,ra:SOCF):Result ==
   c := (retract(exp)@F)$EF
   r:OCF := (c::OCF)*(hi(ra)-lo(ra))
   a := coerce(r)$AnyFunctions1(OCF)
   text := coerce("Constant Function")$AnyFunctions1(ST)
   construct([[result@S,a],[method@S,text]])$Result

 zeroMeasure(m:Measure):Result ==
   a := coerce(0$DF)$AnyFunctions1(DF)
   text := coerce("Constant Function")$AnyFunctions1(String)
   r := construct([[result@Symbol,a],[method@Symbol,text]])$Result
   concat(measure2Result m,r)$ExpertSystemToolsPackage

 scriptedVariables?(mdnia:MDNIA):Boolean ==
   vars:List Symbol := variables(mdnia.fn)$EDF
   var1 := first(vars)$(List Symbol)
```

```
    not scripted?(var1) => false
    name1 := name(var1)$Symbol
    for i in 2..# vars repeat
      not ((scripted?(vars.i)$Symbol) and (name1 = name(vars.i)$Symbol)) =>
         return false
    true

  preAnalysis(args:Union(nia:NIA,mdnia:MDNIA),t:RT):RT ==
    import RT
    r:RT := selectIntegrationRoutines t
    args case nia =>
      arg:NIA := args.nia
      rangeIsFinite(arg)$d01AgentsPackage case finite =>
        selectFiniteRoutines r
      selectNonFiniteRoutines r
    selectMultiDimensionalRoutines r

  changeName(ans:Result,name:ST):Result ==
    sy:S := coerce(name "Answer")$S
    anyAns:Any := coerce(ans)$AnyFunctions1(Result)
    construct([[sy,anyAns]])$Result

  measureSpecific(name:ST,R:RT,args:Union(nia:NIA,mdnia:MDNIA)):
      Record(measure:F,explanations:ST,extra:Result) ==
    args case nia =>
      arg:NIA := args.nia
      name = "d01ajfAnnaType" => measure(R,arg)$d01ajfAnnaType
      name = "d01akfAnnaType" => measure(R,arg)$d01akfAnnaType
      name = "d01alfAnnaType" => measure(R,arg)$d01alfAnnaType
      name = "d01amfAnnaType" => measure(R,arg)$d01amfAnnaType
      name = "d01anfAnnaType" => measure(R,arg)$d01anfAnnaType
      name = "d01apfAnnaType" => measure(R,arg)$d01apfAnnaType
      name = "d01aqfAnnaType" => measure(R,arg)$d01aqfAnnaType
      name = "d01asfAnnaType" => measure(R,arg)$d01asfAnnaType
      name = "d01TransformFunctionType" =>
                    measure(R,arg)$d01TransformFunctionType
      error("measureSpecific","invalid type name: " name)$ErrorFunctions
    args case mdnia =>
      arg2:MDNIA := args.mdnia
      name = "d01gbfAnnaType" => measure(R,arg2)$d01gbfAnnaType
      name = "d01fcfAnnaType" => measure(R,arg2)$d01fcfAnnaType
      error("measureSpecific","invalid type name: " name)$ErrorFunctions
    error("measureSpecific","invalid type name")$ErrorFunctions

  measure(a:NumericalIntegrationProblem,R:RT):Measure ==
    args:Union(nia:NIA,mdnia:MDNIA) := retract(a)$NumericalIntegrationProblem
    sofar := 0$F
    best := "none" :: ST
    routs := copy R
    routs := preAnalysis(args,routs)
```

```
    empty?(routs)$RT =>
      error("measure", "no routines found")$ErrorFunctions
    rout := inspect(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    meth:LST := ["Trying " e.type " integration routines"]
    ext := empty()$Result
    for i in 1..# routs repeat
      rout := extract!(routs)$RT
      e := retract(rout.entry)$AnyFunctions1(Entry)
      n := e.domainName
      if e.defaultMin > sofar then
        m := measureSpecific(n,R,args)
        if m.measure > sofar then
          sofar := m.measure
          best := n
        ext := concat(m.extra,ext)$ExpertSystemToolsPackage
        str:LST := [string(rout.key)$S "measure: " outputMeasure(m.measure)
                    " - " m.explanations]
      else
        str:LST :=  [string(rout.key)$S " is no better than other routines"]
      meth := append(meth,str)$LST
    [sofar,best,meth,ext]

  measure(a:NumericalIntegrationProblem):Measure ==
    measure(a,routines()$RT)

  integrateSpecific(args:Union(nia:NIA,mdnia:MDNIA),n:ST,ex:Result):Result ==
    args case nia =>
      arg:NIA := args.nia
      n = "d01ajfAnnaType" => numericalIntegration(arg,ex)$d01ajfAnnaType
      n = "d01TransformFunctionType" =>
        numericalIntegration(arg,ex)$d01TransformFunctionType
      n = "d01amfAnnaType" => numericalIntegration(arg,ex)$d01amfAnnaType
      n = "d01apfAnnaType" => numericalIntegration(arg,ex)$d01apfAnnaType
      n = "d01aqfAnnaType" => numericalIntegration(arg,ex)$d01aqfAnnaType
      n = "d01alfAnnaType" => numericalIntegration(arg,ex)$d01alfAnnaType
      n = "d01akfAnnaType" => numericalIntegration(arg,ex)$d01akfAnnaType
      n = "d01anfAnnaType" => numericalIntegration(arg,ex)$d01anfAnnaType
      n = "d01asfAnnaType" => numericalIntegration(arg,ex)$d01asfAnnaType
      error("integrateSpecific","invalid type name: " n)$ErrorFunctions
    args case mdnia =>
      arg2:MDNIA := args.mdnia
      n = "d01gbfAnnaType" => numericalIntegration(arg2,ex)$d01gbfAnnaType
      n = "d01fcfAnnaType" => numericalIntegration(arg2,ex)$d01fcfAnnaType
      error("integrateSpecific","invalid type name: " n)$ErrorFunctions
    error("integrateSpecific","invalid type name: " n)$ErrorFunctions

  better?(r:Result,s:Result):Boolean ==
    a1 := search("abserr"::S,r)$Result
    a1 case "failed" => false
```

```
    abserr1 := retract(a1)$AnyFunctions1(DF)
    negative?(abserr1) => false
    a2 := search("abserr"::S,s)$Result
    a2 case "failed" => true
    abserr2 := retract(a2)$AnyFunctions1(DF)
    negative?(abserr2) => true
    (abserr1 < abserr2) -- true if r.abserr better than s.abserr

  recoverAfterFail(n:Union(nia:NIA,mdnia:MDNIA),routs:RT,m:Measure,iint:INT,
                        r:Result):Record(a:Result,b:Measure) ==
    bestName := m.name
    while positive?(iint) repeat
      routineName := m.name
      s := recoverAfterFail(routs,routineName(1..6),iint)$RoutinesTable
      s case "failed" => iint := 0
      if s = "changeEps" then
        nn := n.nia
        zero?(nn.abserr) =>
          nn.abserr := 1.0e-8 :: DF
          m := measure(n::NumericalIntegrationProblem,routs)
          zero?(m.measure) => iint := 0
          r := integrateSpecific(n,m.name,m.extra)
          iint := 0
      rn := routineName(1..6)
      buttVal := getButtonValue(rn,"functionEvaluations")$AttributeButtons
      if (s = "incrFunEvals") and (buttVal < 0.8) then
        increase(rn,"functionEvaluations")$AttributeButtons
      if s = "increase tolerance" then
        (n.nia).relerr := (n.nia).relerr*(10.0::DF)
      if s = "decrease tolerance" then
        (n.nia).relerr := (n.nia).relerr/(10.0::DF)
      fl := coerce(s)$AnyFunctions1(ST)
      flrec:Record(key:S,entry:Any):=[failure@S,fl]
      m2 := measure(n::NumericalIntegrationProblem,routs)
      zero?(m2.measure) => iint := 0
      r2:Result := integrateSpecific(n,m2.name,m2.extra)
      better?(r,r2) =>
        m.name := m2.name
        insert!(flrec,r)$Result
      bestName := m2.name
      m := m2
      insert!(flrec,r2)$Result
      r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
      iany := search(ifail@S,r2)$Result
      iany case "failed" => iint := 0
      iint := retract(iany)$AnyFunctions1(INT)
    m.name := bestName
    [r,m]

  integrateArgs(prob:NumericalIntegrationProblem,t:RT):Result ==
```

```
    args:Union(nia:NIA,mdnia:MDNIA) := retract(prob)$NumericalIntegrationProblem
    routs := copy(t)$RT
    if args case mdnia then
      arg := args.mdnia
      v := (# variables(arg.fn))
      not scriptedVariables?(arg) =>
        error("MultiDimensionalNumericalIntegrationPackage",
                "invalid variable names")$ErrorFunctions
      (v ~= # arg.range)@Boolean =>
        error("MultiDimensionalNumericalIntegrationPackage",
          "number of variables do not match number of ranges")$ErrorFunctions
    m := measure(prob,routs)
    zero?(m.measure) => zeroMeasure m
    r := integrateSpecific(args,m.name,m.extra)
    iany := search(ifail@S,r)$Result
    iint := 0$INT
    if (iany case Any) then
      iint := retract(iany)$AnyFunctions1(INT)
    if positive?(iint) then
      tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
      r := tu.a
      m := tu.b
    r := concat(measure2Result m,r)$ExpertSystemToolsPackage
    n := m.name
    nn:ST :=
      (# n > 14) => "d01transform"
      n(1..6)
    expl := getExplanations(routs,nn)$RoutinesTable
    expla := coerce(expl)$AnyFunctions1(LST)
    explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
    r := concat(construct([explaa]),r)
    args case nia =>
      att := showAttributes(args.nia)$IntegrationFunctionsTable
      att case "failed" => r
      concat(att2Result att,r)$ExpertSystemToolsPackage
    r

  integrate(args:NumericalIntegrationProblem):Result ==
    integrateArgs(args,routines()$RT)

  integrate(exp:EF,ra:SOCF,epsabs:F,epsrel:F,r:RT):Result ==
    Var:LS := variables(exp)$EF
    empty?(Var)$LS => integrateConstant(exp,ra)
    args:NIA := [first(Var)$LS,ef2edf exp,socf2socdf ra,f2df epsabs,f2df epsrel]
    integrateArgs(args::NumericalIntegrationProblem,r)

  integrate(exp:EF,ra:SOCF,epsabs:F,epsrel:F):Result ==
    integrate(exp,ra,epsabs,epsrel,routines()$RT)

  integrate(exp:EF,ra:SOCF,err:F):Result ==
```

```
    positive?(err)$F => integrate(exp,ra,0$F,err)
    integrate(exp,ra,1.0E-5,err)

  integrate(exp:EF,ra:SOCF):Result == integrate(exp,ra,0$F,1.0E-5)

  integrate(exp:EF,sb:SBOCF, st:ST) ==
    st = "numerical" => integrate(exp,segment sb)
    "failed"

  integrate(exp:EF,sb:SBOCF, s:S) ==
    s = (numerical::Symbol) => integrate(exp,segment sb)
    "failed"

  integrate(exp:EF,ra:LSOCF,epsabs:F,epsrel:F,r:RT):Result ==
    vars := variables(exp)$EF
    empty?(vars)$LS => integrateConstantList(exp,ra)
    args:MDNIA := [ef2edf exp,convert ra,f2df epsabs,f2df epsrel]
    integrateArgs(args::NumericalIntegrationProblem,r)

  integrate(exp:EF,ra:LSOCF,epsabs:F,epsrel:F):Result ==
    integrate(exp,ra,epsabs,epsrel,routines()$RT)

  integrate(exp:EF,ra:LSOCF,epsrel:F):Result ==
    zero? epsrel => integrate(exp,ra,1.0e-6,epsrel)
    integrate(exp,ra,0$F,epsrel)

  integrate(exp:EF,ra:LSOCF):Result == integrate(exp,ra,1.0e-4)
```

———-

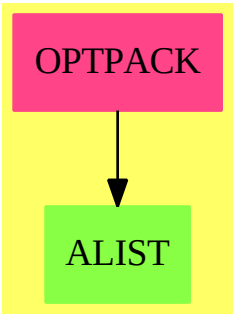
— INTPACK.dotabb —

```
"INTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPACK"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"INTPACK" -> "TBAGG"
```

———-

2.12 package OPTPACK AnnaNumericalOptimization-Package

2.12.1 AnnaNumericalOptimizationPackage (OPTPACK)



Exports:
goodnessOfFit measure optimize

— package OPTPACK AnnaNumericalOptimizationPackage —

```
)abbrev package OPTPACK AnnaNumericalOptimizationPackage
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: December 1997
++ Basic Operations: measure, optimize, goodnessOfFit.
++ Description:
++ \axiomType{AnnaNumericalOptimizationPackage} is a \axiom{package} of
++ functions for the \axiomType{NumericalOptimizationCategory}
++ with \axiom{measure} and \axiom{optimize}.

AnnaNumericalOptimizationPackage(): EE == II where

 EDF  ==> Expression DoubleFloat
 LEDF  ==> List Expression DoubleFloat
 LDF  ==> List DoubleFloat
 MDF  ==> Matrix DoubleFloat
 DF  ==> DoubleFloat
 LOCDF  ==> List OrderedCompletion DoubleFloat
 OCDF  ==> OrderedCompletion DoubleFloat
 LOCF  ==> List OrderedCompletion Float
 OCF  ==> OrderedCompletion Float
 LEF  ==> List Expression Float
 EF  ==> Expression Float
 LF  ==> List Float
 F  ==> Float
```

```
LS  ==> List Symbol
LST  ==> List String
INT  ==> Integer
NOA  ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA  ==> Record(lfn:LEDF, init:LDF)
IFL  ==> List(Record(ifail:Integer,instruction:String))
Entry  ==> Record(chapter:String, type:String, domainName: String,
                  defaultMin:F, measure:F, failList:IFL, explList:LST)
Measure  ==> Record(measure:F,name:String, explanations:List String)
Measure2  ==> Record(measure:F,explanations:String)
RT  ==> RoutinesTable
UNOALSA  ==> Union(noa:NOA,lsa:LSA)

EE ==> with
 measure:NumericalOptimizationProblem -> Measure
   ++ measure(prob) is a top level ANNA function for identifying the most
   ++ appropriate numerical routine from those in the routines table
   ++ provided for solving the numerical optimization problem defined by
   ++ \axiom{prob} by checking various attributes of the functions and
   ++ calculating a measure of compatibility of each routine to these
   ++ attributes.
   ++
   ++ It calls each \axiom{domain} of \axiom{category}
   ++ \axiomType{NumericalOptimizationCategory} in turn to calculate all
   ++ measures and returns the best i.e. the name of the most
   ++ appropriate domain and any other relevant information.

 measure:(NumericalOptimizationProblem,RT) -> Measure
   ++ measure(prob,R) is a top level ANNA function for identifying the most
   ++ appropriate numerical routine from those in the routines table
   ++ provided for solving the numerical optimization problem defined by
   ++ \axiom{prob} by checking various attributes of the functions and
   ++ calculating a measure of compatibility of each routine to these
   ++ attributes.
   ++
   ++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
   ++ \axiomType{NumericalOptimizationCategory} in turn to calculate all
   ++ measures and returns the best i.e. the name of the most
   ++ appropriate domain and any other relevant information.

 optimize:(NumericalOptimizationProblem,RT) -> Result
   ++ optimize(prob,routines) is a top level ANNA function to
   ++ minimize a function or a set of functions with any constraints
   ++ as defined within \axiom{prob}.
   ++
   ++ It iterates over the \axiom{domains} listed in \axiom{routines} of
   ++ \axiomType{NumericalOptimizationCategory}
   ++ to get the name and other relevant information of the best
   ++ \axiom{measure} and then optimize the function on that \axiom{domain}.
```

```
optimize:NumericalOptimizationProblem -> Result
  ++ optimize(prob) is a top level ANNA function to
  ++ minimize a function or a set of functions with any constraints
  ++ as defined within \axiom{prob}.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.

goodnessOfFit:NumericalOptimizationProblem -> Result
  ++ goodnessOfFit(prob) is a top level ANNA function to
  ++ check to goodness of fit of a least squares model
  ++ as defined within \axiom{prob}.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.
  ++ It then calls the numerical routine \axiomType{E04YCF} to get estimates
  ++ of the variance-covariance matrix of the regression coefficients of
  ++ the least-squares problem.
  ++
  ++ It thus returns both the results of the optimization and the
  ++ variance-covariance calculation.

optimize:(EF,LF,LOCF,LEF,LOCF) -> Result
  ++ optimize(f,start,lower,cons,upper) is a top level ANNA function to
  ++ minimize a function, \axiom{f}, of one or more variables with the
  ++ given constraints.
  ++
  ++ These constraints may be simple constraints on the variables
  ++ in which case \axiom{cons} would be an empty list and the bounds on
  ++ those variables defined in \axiom{lower} and \axiom{upper}, or a
  ++ mixture of simple, linear and non-linear constraints, where
  ++ \axiom{cons} contains the linear and non-linear constraints and
  ++ the bounds on these are added to \axiom{upper} and \axiom{lower}.
  ++
  ++ The parameter \axiom{start} is a list of the initial guesses of the
  ++ values of the variables.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.

optimize:(EF,LF,LOCF,LOCF) -> Result
  ++ optimize(f,start,lower,upper) is a top level ANNA function to
  ++ minimize a function, \axiom{f}, of one or more variables with
  ++ simple constraints.  The bounds on
```

```
  ++ the variables are defined in \axiom{lower} and \axiom{upper}.
  ++
  ++ The parameter \axiom{start} is a list of the initial guesses of the
  ++ values of the variables.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.

optimize:(EF,LF) -> Result
  ++ optimize(f,start) is a top level ANNA function to
  ++ minimize a function, \axiom{f}, of one or more variables without
  ++ constraints.
  ++
  ++ The parameter \axiom{start} is a list of the initial guesses of the
  ++ values of the variables.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.

optimize:(LEF,LF) -> Result
  ++ optimize(lf,start) is a top level ANNA function to
  ++ minimize a set of functions, \axiom{lf}, of one or more variables
  ++ without constraints i.e. a least-squares problem.
  ++
  ++ The parameter \axiom{start} is a list of the initial guesses of the
  ++ values of the variables.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.

goodnessOfFit:(LEF,LF) -> Result
  ++ goodnessOfFit(lf,start) is a top level ANNA function to
  ++ check to goodness of fit of a least squares model i.e. the minimization
  ++ of a set of functions, \axiom{lf}, of one or more variables without
  ++ constraints.
  ++
  ++ The parameter \axiom{start} is a list of the initial guesses of the
  ++ values of the variables.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalOptimizationCategory}
  ++ to get the name and other relevant information of the best
  ++ \axiom{measure} and then optimize the function on that \axiom{domain}.
  ++ It then calls the numerical routine \axiomType{E04YCF} to get estimates
```

```
   ++ of the variance-covariance matrix of the regression coefficients of
   ++ the least-squares problem.
   ++
   ++ It thus returns both the results of the optimization and the
   ++ variance-covariance calculation.

   ++ goodnessOfFit(lf,start) is a top level function to iterate over
   ++ the \axiom{domains} of \axiomType{NumericalOptimizationCategory}
   ++ to get the name and other relevant information of the best
   ++ \axiom{measure} and then optimize the function on that \axiom{domain}.
   ++ It then checks the goodness of fit of the least squares model.

II ==> add

 preAnalysis:RT -> RT
 zeroMeasure:Measure -> Result
 optimizeSpecific:(UNOALSA,String) -> Result
 measureSpecific:(String,RT,UNOALSA) -> Measure2
 changeName:(Result,String) -> Result
 recoverAfterFail:(UNOALSA,RT,Measure,INT,Result) -> Record(a:Result,b:Measure)
 constant:UNOALSA -> Union(DF, "failed")
 optimizeConstant:DF -> Result

 import ExpertSystemToolsPackage,e04AgentsPackage,NumericalOptimizationProblem

 constant(args:UNOALSA):Union(DF,"failed") ==
   args case noa =>
     Args := args.noa
     f := Args.fn
     retractIfCan(f)@Union(DoubleFloat,"failed")
   "failed"

 optimizeConstant(c:DF): Result ==
   a := coerce(c)$AnyFunctions1(DF)
   text := coerce("Constant Function")$AnyFunctions1(String)
   construct([[objf@Symbol,a],[method@Symbol,text]])$Result

 preAnalysis(args:UNOALSA,t:RT):RT ==
   r := selectOptimizationRoutines(t)$RT
   args case lsa =>
     selectSumOfSquaresRoutines(r)$RT
   r

 zeroMeasure(m:Measure):Result ==
   a := coerce(0$F)$AnyFunctions1(F)
   text := coerce("Zero Measure")$AnyFunctions1(String)
   r := construct([[objf@Symbol,a],[method@Symbol,text]])$Result
   concat(measure2Result m,r)

 measureSpecific(name:String,R:RT,args:UNOALSA): Measure2 ==
```

```
    args case noa =>
      arg:NOA := args.noa
      name = "e04dgfAnnaType" => measure(R,arg)$e04dgfAnnaType
      name = "e04fdfAnnaType" => measure(R,arg)$e04fdfAnnaType
      name = "e04gcfAnnaType" => measure(R,arg)$e04gcfAnnaType
      name = "e04jafAnnaType" => measure(R,arg)$e04jafAnnaType
      name = "e04mbfAnnaType" => measure(R,arg)$e04mbfAnnaType
      name = "e04nafAnnaType" => measure(R,arg)$e04nafAnnaType
      name = "e04ucfAnnaType" => measure(R,arg)$e04ucfAnnaType
      error("measureSpecific","invalid type name: " name)$ErrorFunctions
    args case lsa =>
      arg2:LSA := args.lsa
      name = "e04fdfAnnaType" => measure(R,arg2)$e04fdfAnnaType
      name = "e04gcfAnnaType" => measure(R,arg2)$e04gcfAnnaType
      error("measureSpecific","invalid type name: " name)$ErrorFunctions
    error("measureSpecific","invalid argument type")$ErrorFunctions

  measure(Args:NumericalOptimizationProblem,R:RT):Measure ==
    args:UNOALSA := retract(Args)$NumericalOptimizationProblem
    sofar := 0$F
    best := "none" :: String
    routs := copy R
    routs := preAnalysis(args,routs)
    empty?(routs)$RT =>
      error("measure", "no routines found")$ErrorFunctions
    rout := inspect(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    meth := empty()$(List String)
    for i in 1..# routs repeat
      rout := extract!(routs)$RT
      e := retract(rout.entry)$AnyFunctions1(Entry)
      n := e.domainName
      if e.defaultMin > sofar then
        m := measureSpecific(n,R,args)
        if m.measure > sofar then
          sofar := m.measure
          best := n
        str := [concat(concat([string(rout.key)$Symbol,"measure: ",
                 outputMeasure(m.measure)," - "],
                   m.explanations)$(List String))$String]
      else
        str := [concat([string(rout.key)$Symbol
                        ," is no better than other routines"])$String]
      meth := append(meth,str)$(List String)
    [sofar,best,meth]

  measure(args:NumericalOptimizationProblem):Measure == measure(args,routines()$RT)

  optimizeSpecific(args:UNOALSA,name:String):Result ==
    args case noa =>
```

```
    arg:NOA := args.noa
    name = "e04dgfAnnaType" => numericalOptimization(arg)$e04dgfAnnaType
    name = "e04fdfAnnaType" => numericalOptimization(arg)$e04fdfAnnaType
    name = "e04gcfAnnaType" => numericalOptimization(arg)$e04gcfAnnaType
    name = "e04jafAnnaType" => numericalOptimization(arg)$e04jafAnnaType
    name = "e04mbfAnnaType" => numericalOptimization(arg)$e04mbfAnnaType
    name = "e04nafAnnaType" => numericalOptimization(arg)$e04nafAnnaType
    name = "e04ucfAnnaType" => numericalOptimization(arg)$e04ucfAnnaType
    error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
  args case lsa =>
    arg2:LSA := args.lsa
    name = "e04fdfAnnaType" => numericalOptimization(arg2)$e04fdfAnnaType
    name = "e04gcfAnnaType" => numericalOptimization(arg2)$e04gcfAnnaType
    error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
  error("optimizeSpecific","invalid type name: " name)$ErrorFunctions

changeName(ans:Result,name:String):Result ==
  st:String := concat([name,"Answer"])$String
  sy:Symbol := coerce(st)$Symbol
  anyAns:Any := coerce(ans)$AnyFunctions1(Result)
  construct([[sy,anyAns]])$Result

recoverAfterFail(args:UNOALSA,routs:RT,m:Measure,
                  iint:INT,r:Result):Record(a:Result,b:Measure) ==
  while positive?(iint) repeat
    routineName := m.name
    s := recoverAfterFail(routs,routineName(1..6),iint)$RT
    s case "failed" => iint := 0
    (s = "no action")@Boolean => iint := 0
    fl := coerce(s)$AnyFunctions1(String)
    flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
    m2 := measure(args::NumericalOptimizationProblem,routs)
    zero?(m2.measure) => iint := 0
    r2:Result := optimizeSpecific(args,m2.name)
    m := m2
    insert!(flrec,r2)$Result
    r := concat(r2,changeName(r,routineName))
    iany := search(ifail@Symbol,r2)$Result
    iany case "failed" => iint := 0
    iint := retract(iany)$AnyFunctions1(INT)
  [r,m]

optimize(Args:NumericalOptimizationProblem,t:RT):Result ==
  args:UNOALSA := retract(Args)$NumericalOptimizationProblem
  routs := copy(t)$RT
  c:Union(DF,"failed") := constant(args)
  c case DF => optimizeConstant(c)
  m := measure(Args,routs)
  zero?(m.measure) => zeroMeasure m
  r := optimizeSpecific(args,n := m.name)
```

```
    iany := search(ifail@Symbol,r)$Result
    iint := 0$INT
    if (iany case Any) then
      iint := retract(iany)$AnyFunctions1(INT)
    if positive?(iint) then
      tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
      r := tu.a
      m := tu.b
    r := concat(measure2Result m,r)
    expl := getExplanations(routs,n(1..6))$RoutinesTable
    expla := coerce(expl)$AnyFunctions1(LST)
    explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
    r := concat(construct([explaa]),r)
    att:List String := optAttributes(args)
    atta := coerce(att)$AnyFunctions1(List String)
    attr:Record(key:Symbol,entry:Any) := [attributes@Symbol,atta]
    insert!(attr,r)$Result

  optimize(args:NumericalOptimizationProblem):Result == optimize(args,routines()$RT)

  goodnessOfFit(Args:NumericalOptimizationProblem):Result ==
    r := optimize(Args)
    args1:UNOALSA := retract(Args)$NumericalOptimizationProblem
    args1 case noa => error("goodnessOfFit","Not an appropriate problem")
    args:LSA := args1.lsa
    lf := args.lfn
    n:INT := #(variables(args))
    m:INT := # lf
    me := search(method,r)$Result
    me case "failed" => r
    meth := retract(me)$AnyFunctions1(Result)
    na := search(nameOfRoutine,meth)$Result
    na case "failed" => r
    name := retract(na)$AnyFunctions1(String)
    temp:INT := (n*(n-1)) quo 2
    ns:INT :=
      name = "e04fdfAnnaType" => 6*n+(2+n)*m+1+max(1,temp)
      8*n+(n+2)*m+temp+1+max(1,temp)
    nv:INT := ns+n
    ww := search(w,r)$Result
    ww case "failed" => r
    ws:MDF := retract(ww)$AnyFunctions1(MDF)
    fr := search(objf,r)$Result
    fr case "failed" => r
    f := retract(fr)$AnyFunctions1(DF)
    s := subMatrix(ws,1,1,ns,nv-1)$MDF
    v := subMatrix(ws,1,1,nv,nv+n*n-1)$MDF
    r2 := e04ycf(0,m,n,f,s,n,v,-1)$NagOptimisationPackage
    concat(r,r2)
```

```
  optimize(f:EF,start:LF,lower:LOCF,cons:LEF,upper:LOCF):Result ==
    args:NOA := [ef2edf(f),[f2df i for i in start],[ocf2ocdf j for j in lower],
                 [ef2edf k for k in cons], [ocf2ocdf l for l in upper]]
    optimize(args::NumericalOptimizationProblem)

  optimize(f:EF,start:LF,lower:LOCF,upper:LOCF):Result ==
    optimize(f,start,lower,empty()$LEF,upper)

  optimize(f:EF,start:LF):Result ==
    optimize(f,start,empty()$LOCF,empty()$LOCF)

  optimize(lf:LEF,start:LF):Result ==
    args:LSA := [[ef2edf i for i in lf],[f2df j for j in start]]
    optimize(args::NumericalOptimizationProblem)

  goodnessOfFit(lf:LEF,start:LF):Result ==
    args:LSA := [[ef2edf i for i in lf],[f2df j for j in start]]
    goodnessOfFit(args::NumericalOptimizationProblem)
```

———-

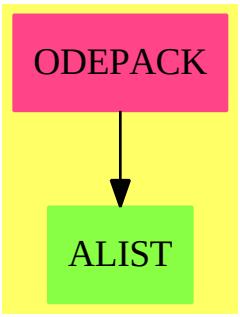
— **OPTPACK.dotabb** —

```
"OPTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OPTPACK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"OPTPACK" -> "ALIST"
```

———-

2.13 package ODEPACK AnnaOrdinaryDifferentialEquationPackage

2.13.1 AnnaOrdinaryDifferentialEquationPackage (ODEPACK)



Exports:
measure solve

— package ODEPACK AnnaOrdinaryDifferentialEquationPackage —

```
)abbrev package ODEPACK AnnaOrdinaryDifferentialEquationPackage
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: December 1997
++ Basic Operations: solve, measure
++ Description:
++ \axiomType{AnnaOrdinaryDifferentialEquationPackage} is a \axiom{package}
++ of functions for the \axiom{category}
++ \axiomType{OrdinaryDifferentialEquationsSolverCategory}
++ with \axiom{measure}, and \axiom{solve}.

AnnaOrdinaryDifferentialEquationPackage(): EE == II where

 EDF  ==> Expression DoubleFloat
 LDF  ==> List DoubleFloat
 MDF  ==> Matrix DoubleFloat
 DF  ==> DoubleFloat
 FI  ==> Fraction Integer
 EFI  ==> Expression Fraction Integer
 SOCDF  ==> Segment OrderedCompletion DoubleFloat
 VEDF  ==> Vector Expression DoubleFloat
 VEF  ==> Vector Expression Float
 EF  ==> Expression Float
 LF  ==> List Float
 F  ==> Float
 VDF  ==> Vector DoubleFloat
 VMF  ==> Vector MachineFloat
 MF  ==> MachineFloat
 LS  ==> List Symbol
 ST  ==> String
 LST  ==> List String
 INT  ==> Integer
 RT  ==> RoutinesTable
 ODEA  ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,_
                  g:EDF,abserr:DF,relerr:DF)
 IFL  ==> List(Record(ifail:Integer,instruction:String))
 Entry  ==> Record(chapter:String, type:String, domainName: String,
                  defaultMin:F, measure:F, failList:IFL, explList:LST)
 Measure  ==> Record(measure:F,name:String, explanations:List String)

 EE ==> with
  solve:(NumericalODEProblem) -> Result
```

```
    ++ solve(odeProblem) is a top level ANNA function to solve numerically a
    ++ system of ordinary differential equations i.e. equations for the
    ++ derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n], together
    ++ with starting values for x and y[1]..y[n] (called the initial
    ++ conditions), a final value of x, an accuracy requirement and any
    ++ intermediate points at which the result is required.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory}
    ++ to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ The method used to perform the numerical
    ++ process will be one of the routines contained in the NAG numerical
    ++ Library.  The function predicts the likely most effective routine
    ++ by checking various attributes of the system of ODE's and calculating
    ++ a measure of compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
  solve:(NumericalODEProblem,RT) -> Result
    ++ solve(odeProblem,R) is a top level ANNA function to solve numerically a
    ++ system of ordinary differential equations i.e. equations for the
    ++ derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n], together
    ++ with starting values for x and y[1]..y[n] (called the initial
    ++ conditions), a final value of x, an accuracy requirement and any
    ++ intermediate points at which the result is required.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
    ++ the table of routines \axiom{R} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ The method used to perform the numerical
    ++ process will be one of the routines contained in the NAG numerical
    ++ Library.  The function predicts the likely most effective routine
    ++ by checking various attributes of the system of ODE's and calculating
    ++ a measure of compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
  solve:(VEF,F,F,LF) -> Result
    ++ solve(f,xStart,xEnd,yInitial) is a top level ANNA function to solve
    ++ numerically a system of ordinary differential equations i.e. equations
    ++ for the derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n],
    ++ together with a starting value for x and y[1]..y[n] (called the initial
    ++ conditions) and a final value of x.  A default value
    ++ is used for the accuracy requirement.
```

```
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
  ++ the table of routines \axiom{R} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library.  The function predicts the likely most effective routine
  ++ by checking various attributes of the system of ODE's and calculating
  ++ a measure of compatibility of each routine to these attributes.
  ++
  ++ It then calls the resulting 'best' routine.
solve:(VEF,F,F,LF,F) -> Result
  ++ solve(f,xStart,xEnd,yInitial,tol) is a top level ANNA function to solve
  ++ numerically a system of ordinary differential equations, \axiom{f},
  ++ i.e. equations for the derivatives y[1]'..y[n]' defined in terms
  ++ of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd} with the initial
  ++ values for y[1]..y[n] (\axiom{yInitial}) to a tolerance \axiom{tol}.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
  ++ the table of routines \axiom{R} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library.  The function predicts the likely most effective routine
  ++ by checking various attributes of the system of ODE's and calculating
  ++ a measure of compatibility of each routine to these attributes.
  ++
  ++ It then calls the resulting 'best' routine.
solve:(VEF,F,F,LF,EF,F) -> Result
  ++ solve(f,xStart,xEnd,yInitial,G,tol) is a top level ANNA function to
  ++ solve numerically a system of ordinary differential equations,
  ++ \axiom{f}, i.e. equations for the derivatives y[1]'..y[n]' defined in
  ++ terms of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd} with the
  ++ initial values for y[1]..y[n] (\axiom{yInitial}) to a tolerance
  ++ \axiom{tol}. The calculation will stop if the function
  ++ G(x,y[1],..,y[n]) evaluates to zero before x = xEnd.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
  ++ the table of routines \axiom{R} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
```

```
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ The method used to perform the numerical process will be one of the
    ++ routines contained in the NAG numerical Library.  The function
    ++ predicts the likely most effective routine by checking various
    ++ attributes of the system of ODE's and calculating a measure of
    ++ compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
  solve:(VEF,F,F,LF,LF,F) -> Result
    ++ solve(f,xStart,xEnd,yInitial,intVals,tol) is a top level ANNA function
    ++ to solve numerically a system of ordinary differential equations,
    ++ \axiom{f}, i.e. equations for the derivatives y[1]'..y[n]' defined in
    ++ terms of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd} with the
    ++ initial values for y[1]..y[n] (\axiom{yInitial}) to a tolerance
    ++ \axiom{tol}. The values of y[1]..y[n] will be output for the values
    ++ of x in \axiom{intVals}.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
    ++ the table of routines \axiom{R} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ The method used to perform the numerical
    ++ process will be one of the routines contained in the NAG numerical
    ++ Library.  The function predicts the likely most effective routine
    ++ by checking various attributes of the system of ODE's and calculating
    ++ a measure of compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
  solve:(VEF,F,F,LF,EF,LF,F) -> Result
    ++ solve(f,xStart,xEnd,yInitial,G,intVals,tol) is a top level ANNA
    ++ function to solve numerically a system of ordinary differential
    ++ equations, \axiom{f}, i.e. equations for the derivatives y[1]'..y[n]'
    ++ defined in terms of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd}
    ++ with the initial values for y[1]..y[n] (\axiom{yInitial}) to a
    ++ tolerance \axiom{tol}. The values of y[1]..y[n] will be output for
    ++ the values of x in \axiom{intVals}.  The calculation will stop if the
    ++ function G(x,y[1],..,y[n]) evaluates to zero before x = xEnd.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
    ++ the table of routines \axiom{R} to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ The method used to perform the numerical
```

```
   ++ process will be one of the routines contained in the NAG numerical
   ++ Library.  The function predicts the likely most effective routine
   ++ by checking various attributes of the system of ODE's and calculating
   ++ a measure of compatibility of each routine to these attributes.
   ++
   ++ It then calls the resulting 'best' routine.
  solve:(VEF,F,F,LF,EF,LF,F,F) -> Result
   ++ solve(f,xStart,xEnd,yInitial,G,intVals,epsabs,epsrel) is a top level
   ++ ANNA function to solve numerically a system of ordinary differential
   ++ equations, \axiom{f}, i.e.
   ++ equations for the derivatives y[1]'..y[n]' defined in terms
   ++ of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd} with the initial
   ++ values for y[1]..y[n] (\axiom{yInitial}) to an absolute error
   ++ requirement \axiom{epsabs} and relative error \axiom{epsrel}.
   ++ The values of y[1]..y[n] will be output for the values of x in
   ++ \axiom{intVals}.  The calculation will stop if the function
   ++ G(x,y[1],..,y[n]) evaluates to zero before x = xEnd.
   ++
   ++ It iterates over the \axiom{domains} of
   ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
   ++ the table of routines \axiom{R} to get the name and other
   ++ relevant information of the the (domain of the) numerical
   ++ routine likely to be the most appropriate,
   ++ i.e. have the best \axiom{measure}.
   ++
   ++ The method used to perform the numerical
   ++ process will be one of the routines contained in the NAG numerical
   ++ Library.  The function predicts the likely most effective routine
   ++ by checking various attributes of the system of ODE's and calculating
   ++ a measure of compatibility of each routine to these attributes.
   ++
   ++ It then calls the resulting 'best' routine.
  measure:(NumericalODEProblem) -> Measure
   ++ measure(prob) is a top level ANNA function for identifying the most
   ++ appropriate numerical routine from those in the routines table
   ++ provided for solving the numerical ODE
   ++ problem defined by \axiom{prob}.
   ++
   ++ It calls each \axiom{domain} of \axiom{category}
   ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} in turn to
   ++ calculate all measures and returns the best i.e. the name of
   ++ the most appropriate domain and any other relevant information.
   ++ It predicts the likely most effective NAG numerical
   ++ Library routine to solve the input set of ODEs
   ++ by checking various attributes of the system of ODEs and calculating
   ++ a measure of compatibility of each routine to these attributes.
  measure:(NumericalODEProblem,RT) -> Measure
   ++ measure(prob,R) is a top level ANNA function for identifying the most
   ++ appropriate numerical routine from those in the routines table
   ++ provided for solving the numerical ODE
```

```
   ++ problem defined by \axiom{prob}.
   ++
   ++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
   ++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} in turn to
   ++ calculate all measures and returns the best i.e. the name of
   ++ the most appropriate domain and any other relevant information.
   ++ It predicts the likely most effective NAG numerical
   ++ Library routine to solve the input set of ODEs
   ++ by checking various attributes of the system of ODEs and calculating
   ++ a measure of compatibility of each routine to these attributes.

II ==> add

 import ODEA,NumericalODEProblem

 f2df:F -> DF
 ef2edf:EF -> EDF
 preAnalysis:(ODEA,RT) -> RT
 zeroMeasure:Measure -> Result
 measureSpecific:(ST,RT,ODEA) -> Record(measure:F,explanations:ST)
 solveSpecific:(ODEA,ST) -> Result
 changeName:(Result,ST) -> Result
 recoverAfterFail:(ODEA,RT,Measure,Integer,Result) -> Record(a:Result,b:Measure)

 f2df(f:F):DF == (convert(f)@DF)$F

 ef2edf(f:EF):EDF == map(f2df,f)$ExpressionFunctions2(F,DF)

 preAnalysis(args:ODEA,t:RT):RT ==
   rt := selectODEIVPRoutines(t)$RT
   if positive?(# variables(args.g)) then
     changeMeasure(rt,d02bbf@Symbol,getMeasure(rt,d02bbf@Symbol)*0.8)
   if positive?(# args.intvals) then
     changeMeasure(rt,d02bhf@Symbol,getMeasure(rt,d02bhf@Symbol)*0.8)
   rt

 zeroMeasure(m:Measure):Result ==
   a := coerce(0$F)$AnyFunctions1(F)
   text := coerce("Zero Measure")$AnyFunctions1(ST)
   r := construct([[result@Symbol,a],[method@Symbol,text]])$Result
   concat(measure2Result m,r)$ExpertSystemToolsPackage

 measureSpecific(name:ST,R:RT,ode:ODEA):Record(measure:F,explanations:ST) ==
   name = "d02bbfAnnaType" => measure(R,ode)$d02bbfAnnaType
   name = "d02bhfAnnaType" => measure(R,ode)$d02bhfAnnaType
   name = "d02cjfAnnaType" => measure(R,ode)$d02cjfAnnaType
   name = "d02ejfAnnaType" => measure(R,ode)$d02ejfAnnaType
   error("measureSpecific","invalid type name: " name)$ErrorFunctions

 measure(Ode:NumericalODEProblem,R:RT):Measure ==
```

```
    ode:ODEA := retract(Ode)$NumericalODEProblem
    sofar := 0$F
    best := "none" :: ST
    routs := copy R
    routs := preAnalysis(ode,routs)
    empty?(routs)$RT =>
      error("measure", "no routines found")$ErrorFunctions
    rout := inspect(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    meth := empty()$LST
    for i in 1..# routs repeat
      rout := extract!(routs)$RT
      e := retract(rout.entry)$AnyFunctions1(Entry)
      n := e.domainName
      if e.defaultMin > sofar then
        m := measureSpecific(n,R,ode)
        if m.measure > sofar then
          sofar := m.measure
          best := n
        str:LST := [string(rout.key)$Symbol "measure: "
                    outputMeasure(m.measure)$ExpertSystemToolsPackage " - "
                     m.explanations]
      else
        str := [string(rout.key)$Symbol " is no better than other routines"]
      meth := append(meth,str)$LST
    [sofar,best,meth]

  measure(ode:NumericalODEProblem):Measure == measure(ode,routines()$RT)

  solveSpecific(ode:ODEA,n:ST):Result ==
    n = "d02bbfAnnaType" => ODESolve(ode)$d02bbfAnnaType
    n = "d02bhfAnnaType" => ODESolve(ode)$d02bhfAnnaType
    n = "d02cjfAnnaType" => ODESolve(ode)$d02cjfAnnaType
    n = "d02ejfAnnaType" => ODESolve(ode)$d02ejfAnnaType
    error("solveSpecific","invalid type name: " n)$ErrorFunctions

  changeName(ans:Result,name:ST):Result ==
    sy:Symbol := coerce(name "Answer")$Symbol
    anyAns:Any := coerce(ans)$AnyFunctions1(Result)
    construct([[sy,anyAns]])$Result

  recoverAfterFail(ode:ODEA,routs:RT,m:Measure,iint:Integer,r:Result):
                                            Record(a:Result,b:Measure) ==
    while positive?(iint) repeat
      routineName := m.name
      s := recoverAfterFail(routs,routineName(1..6),iint)$RT
      s case "failed" => iint := 0
      if s = "increase tolerance" then
        ode.relerr := ode.relerr*(10.0::DF)
        ode.abserr := ode.abserr*(10.0::DF)
```

```
    if s = "decrease tolerance" then
      ode.relerr := ode.relerr/(10.0::DF)
      ode.abserr := ode.abserr/(10.0::DF)
    (s = "no action")@Boolean => iint := 0
    fl := coerce(s)$AnyFunctions1(ST)
    flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
    m2 := measure(ode::NumericalODEProblem,routs)
    zero?(m2.measure) => iint := 0
    r2:Result := solveSpecific(ode,m2.name)
    m := m2
    insert!(flrec,r2)$Result
    r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
    iany := search(ifail@Symbol,r2)$Result
    iany case "failed" => iint := 0
    iint := retract(iany)$AnyFunctions1(Integer)
  [r,m]

solve(Ode:NumericalODEProblem,t:RT):Result ==
  ode:ODEA := retract(Ode)$NumericalODEProblem
  routs := copy(t)$RT
  m := measure(Ode,routs)
  zero?(m.measure) => zeroMeasure m
  r := solveSpecific(ode,n := m.name)
  iany := search(ifail@Symbol,r)$Result
  iint := 0$Integer
  if (iany case Any) then
    iint := retract(iany)$AnyFunctions1(Integer)
  if positive?(iint) then
    tu:Record(a:Result,b:Measure) := recoverAfterFail(ode,routs,m,iint,r)
    r := tu.a
    m := tu.b
  r := concat(measure2Result m,r)$ExpertSystemToolsPackage
  expl := getExplanations(routs,n(1..6))$RoutinesTable
  expla := coerce(expl)$AnyFunctions1(LST)
  explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
  r := concat(construct([explaa]),r)
  iflist := showIntensityFunctions(ode)$ODEIntensityFunctionsTable
  iflist case "failed" => r
  concat(iflist2Result iflist, r)$ExpertSystemToolsPackage

solve(ode:NumericalODEProblem):Result == solve(ode,routines()$RT)

solve(f:VEF,xStart:F,xEnd:F,yInitial:LF,G:EF,intVals:LF,epsabs:F,epsrel:F):Result ==
  d:ODEA := [f2df xStart,f2df xEnd,vector([ef2edf e for e in members f])$VEDF,
             [f2df i for i in yInitial], [f2df j for j in intVals],
              ef2edf G,f2df epsabs,f2df epsrel]
  solve(d::NumericalODEProblem,routines()$RT)

solve(f:VEF,xStart:F,xEnd:F,yInitial:LF,G:EF,intVals:LF,tol:F):Result ==
  solve(f,xStart,xEnd,yInitial,G,intVals,tol,tol)
```

```
  solve(f:VEF,xStart:F,xEnd:F,yInitial:LF,intVals:LF,tol:F):Result ==
    solve(f,xStart,xEnd,yInitial,1$EF,intVals,tol)

  solve(f:VEF,xStart:F,xEnd:F,y:LF,G:EF,tol:F):Result ==
    solve(f,xStart,xEnd,y,G,empty()$LF,tol)

  solve(f:VEF,xStart:F,xEnd:F,yInitial:LF,tol:F):Result ==
    solve(f,xStart,xEnd,yInitial,1$EF,empty()$LF,tol)

  solve(f:VEF,xStart:F,xEnd:F,yInitial:LF):Result == solve(f,xStart,xEnd,yInitial,1.0e-4)
```

———-

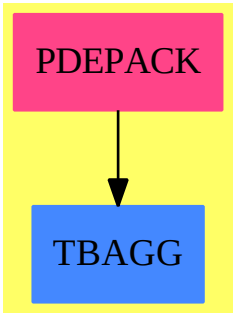
— **ODEPACK.dotabb** —

```
"ODEPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPACK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ODEPACK" -> "ALIST"
```

———-

2.14 package PDEPACK AnnaPartialDifferentialEquationPackage

2.14.1 AnnaPartialDifferentialEquationPackage (PDEPACK)



Exports:
measure solve

— **package PDEPACK AnnaPartialDifferentialEquationPackage** —

```
)abbrev package PDEPACK AnnaPartialDifferentialEquationPackage
++ Author: Brian Dupee
++ Date Created: June 1996
++ Date Last Updated: December 1997
++ Basic Operations:
++ Description:
++ AnnaPartialDifferentialEquationPackage is an uncompleted
++ package for the interface to NAG PDE routines.  It has been realised that
++ a new approach to solving PDEs will need to be created.

AnnaPartialDifferentialEquationPackage(): EE == II where
 LEDF  ==> List Expression DoubleFloat
 EDF  ==> Expression DoubleFloat
 LDF  ==> List DoubleFloat
 MDF  ==> Matrix DoubleFloat
 DF  ==> DoubleFloat
 LEF  ==> List Expression Float
 EF  ==> Expression Float
 MEF  ==> Matrix Expression Float
 LF  ==> List Float
 F  ==> Float
 LS  ==> List Symbol
 ST  ==> String
 LST  ==> List String
 INT  ==> Integer
 NNI  ==> NonNegativeInteger
 RT  ==> RoutinesTable
 PDEC  ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
                  dStart:MDF, dFinish:MDF)
 PDEB  ==> Record(pde:LEDF, constraints:List PDEC,
                  f:List LEDF, st:ST, tol:DF)
 IFL  ==> List(Record(ifail:INT,instruction:ST))
 Entry  ==> Record(chapter:ST, type:ST, domainName: ST,
                   defaultMin:F, measure:F, failList:IFL, explList:LST)
 Measure  ==> Record(measure:F,name:ST, explanations:LST)

 EE ==> with
  solve:(NumericalPDEProblem) -> Result
    ++ solve(PDEProblem) is a top level ANNA function to solve numerically
    ++ a system of partial differential equations.
    ++
    ++ The method used to perform the numerical
    ++ process will be one of the routines contained in the NAG numerical
    ++ Library.  The function predicts the likely most effective routine
    ++ by checking various attributes of the system of PDE's and calculating
    ++ a measure of compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
    ++
    ++ ** At the moment, only Second Order Elliptic Partial Differential
```

```
  ++ Equations are solved **
solve:(NumericalPDEProblem,RT) -> Result
  ++ solve(PDEProblem,routines) is a top level ANNA function to solve numerically a system
  ++ of partial differential equations.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library.  The function predicts the likely most effective routine
  ++ by checking various attributes of the system of PDE's and calculating
  ++ a measure of compatibility of each routine to these attributes.
  ++
  ++ It then calls the resulting 'best' routine.
  ++
  ++ ** At the moment, only Second Order Elliptic Partial Differential
  ++ Equations are solved **
solve:(F,F,F,F,NNI,NNI,LEF,List LEF,ST,DF) -> Result
  ++ solve(xmin,ymin,xmax,ymax,ngx,ngy,pde,bounds,st,tol) is a top level
  ++ ANNA function to solve numerically a system of partial differential
  ++ equations.  This is defined as a list of coefficients (\axiom{pde}),
  ++ a grid (\axiom{xmin}, \axiom{ymin}, \axiom{xmax}, \axiom{ymax},
  ++ \axiom{ngx}, \axiom{ngy}), the boundary values (\axiom{bounds}) and a
  ++ tolerance requirement (\axiom{tol}).  There is also a parameter
  ++ (\axiom{st}) which should contain the value "elliptic" if the PDE is
  ++ known to be elliptic, or "unknown" if it is uncertain.  This causes the
  ++ routine to check whether the PDE is elliptic.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library.  The function predicts the likely most effective routine
  ++ by checking various attributes of the system of PDE's and calculating
  ++ a measure of compatibility of each routine to these attributes.
  ++
  ++ It then calls the resulting 'best' routine.
  ++
  ++ ** At the moment, only Second Order Elliptic Partial Differential
  ++ Equations are solved **
solve:(F,F,F,F,NNI,NNI,LEF,List LEF,ST) -> Result
  ++ solve(xmin,ymin,xmax,ymax,ngx,ngy,pde,bounds,st) is a top level
  ++ ANNA function to solve numerically a system of partial differential
  ++ equations.  This is defined as a list of coefficients (\axiom{pde}),
  ++ a grid (\axiom{xmin}, \axiom{ymin}, \axiom{xmax}, \axiom{ymax},
  ++ \axiom{ngx}, \axiom{ngy}) and the boundary values (\axiom{bounds}).
  ++ A default value for tolerance is used.  There is also a parameter
  ++ (\axiom{st}) which should contain the value "elliptic" if the PDE is
  ++ known to be elliptic, or "unknown" if it is uncertain.  This causes the
  ++ routine to check whether the PDE is elliptic.
  ++
  ++ The method used to perform the numerical
  ++ process will be one of the routines contained in the NAG numerical
  ++ Library.  The function predicts the likely most effective routine
```

```
    ++ by checking various attributes of the system of PDE's and calculating
    ++ a measure of compatibility of each routine to these attributes.
    ++
    ++ It then calls the resulting 'best' routine.
    ++
    ++ ** At the moment, only Second Order Elliptic Partial Differential
    ++ Equations are solved **
  measure:(NumericalPDEProblem) -> Measure
    ++ measure(prob) is a top level ANNA function for identifying the most
    ++ appropriate numerical routine from those in the routines table
    ++ provided for solving the numerical PDE
    ++ problem defined by \axiom{prob}.
    ++
    ++ It calls each \axiom{domain} of \axiom{category}
    ++ \axiomType{PartialDifferentialEquationsSolverCategory} in turn to
    ++ calculate all measures and returns the best i.e. the name of
    ++ the most appropriate domain and any other relevant information.
    ++ It predicts the likely most effective NAG numerical
    ++ Library routine to solve the input set of PDEs
    ++ by checking various attributes of the system of PDEs and calculating
    ++ a measure of compatibility of each routine to these attributes.
  measure:(NumericalPDEProblem,RT) -> Measure
    ++ measure(prob,R) is a top level ANNA function for identifying the most
    ++ appropriate numerical routine from those in the routines table
    ++ provided for solving the numerical PDE
    ++ problem defined by \axiom{prob}.
    ++
    ++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
    ++ \axiomType{PartialDifferentialEquationsSolverCategory} in turn to
    ++ calculate all measures and returns the best i.e. the name of
    ++ the most appropriate domain and any other relevant information.
    ++ It predicts the likely most effective NAG numerical
    ++ Library routine to solve the input set of PDEs
    ++ by checking various attributes of the system of PDEs and calculating
    ++ a measure of compatibility of each routine to these attributes.


 II ==> add

  import PDEB, d03AgentsPackage, ExpertSystemToolsPackage, NumericalPDEProblem

  zeroMeasure:Measure -> Result
  measureSpecific:(ST,RT,PDEB) -> Record(measure:F,explanations:ST)
  solveSpecific:(PDEB,ST) -> Result
  changeName:(Result,ST) -> Result
  recoverAfterFail:(PDEB,RT,Measure,Integer,Result) -> Record(a:Result,b:Measure)

  zeroMeasure(m:Measure):Result ==
    a := coerce(0$F)$AnyFunctions1(F)
    text := coerce("No available routine appears appropriate")$AnyFunctions1(ST)
```

```
    r := construct([[result@Symbol,a],[method@Symbol,text]])$Result
    concat(measure2Result m,r)$ExpertSystemToolsPackage

  measureSpecific(name:ST,R:RT,p:PDEB):Record(measure:F,explanations:ST) ==
    name = "d03eefAnnaType" => measure(R,p)$d03eefAnnaType
    --name = "d03fafAnnaType" => measure(R,p)$d03fafAnnaType
    error("measureSpecific","invalid type name: " name)$ErrorFunctions

  measure(P:NumericalPDEProblem,R:RT):Measure ==
    p:PDEB := retract(P)$NumericalPDEProblem
    sofar := 0$F
    best := "none" :: ST
    routs := copy R
    routs := selectPDERoutines(routs)$RT
    empty?(routs)$RT =>
      error("measure", "no routines found")$ErrorFunctions
    rout := inspect(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    meth := empty()$LST
    for i in 1..# routs repeat
      rout := extract!(routs)$RT
      e := retract(rout.entry)$AnyFunctions1(Entry)
      n := e.domainName
      if e.defaultMin > sofar then
        m := measureSpecific(n,R,p)
        if m.measure > sofar then
          sofar := m.measure
          best := n
        str:LST := [string(rout.key)$Symbol "measure: "
                    outputMeasure(m.measure)$ExpertSystemToolsPackage " - "
                     m.explanations]
      else
        str := [string(rout.key)$Symbol " is no better than other routines"]
      meth := append(meth,str)$LST
    [sofar,best,meth]

  measure(P:NumericalPDEProblem):Measure == measure(P,routines()$RT)

  solveSpecific(p:PDEB,n:ST):Result ==
    n = "d03eefAnnaType" => PDESolve(p)$d03eefAnnaType
    --n = "d03fafAnnaType" => PDESolve(p)$d03fafAnnaType
    error("solveSpecific","invalid type name: " n)$ErrorFunctions

  changeName(ans:Result,name:ST):Result ==
    sy:Symbol := coerce(name "Answer")$Symbol
    anyAns:Any := coerce(ans)$AnyFunctions1(Result)
    construct([[sy,anyAns]])$Result

  recoverAfterFail(p:PDEB,routs:RT,m:Measure,iint:Integer,r:Result):
                                          Record(a:Result,b:Measure) ==
```

```
    while positive?(iint) repeat
      routineName := m.name
      s := recoverAfterFail(routs,routineName(1..6),iint)$RT
      s case "failed" => iint := 0
      (s = "no action")@Boolean => iint := 0
      fl := coerce(s)$AnyFunctions1(ST)
      flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
      m2 := measure(p::NumericalPDEProblem,routs)
      zero?(m2.measure) => iint := 0
      r2:Result := solveSpecific(p,m2.name)
      m := m2
      insert!(flrec,r2)$Result
      r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
      iany := search(ifail@Symbol,r2)$Result
      iany case "failed" => iint := 0
      iint := retract(iany)$AnyFunctions1(Integer)
    [r,m]

  solve(P:NumericalPDEProblem,t:RT):Result ==
    routs := copy(t)$RT
    m := measure(P,routs)
    p:PDEB := retract(P)$NumericalPDEProblem
    zero?(m.measure) => zeroMeasure m
    r := solveSpecific(p,n := m.name)
    iany := search(ifail@Symbol,r)$Result
    iint := 0$Integer
    if (iany case Any) then
      iint := retract(iany)$AnyFunctions1(Integer)
    if positive?(iint) then
      tu:Record(a:Result,b:Measure) := recoverAfterFail(p,routs,m,iint,r)
      r := tu.a
      m := tu.b
    expl := getExplanations(routs,n(1..6))$RoutinesTable
    expla := coerce(expl)$AnyFunctions1(LST)
    explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
    r := concat(construct([explaa]),r)
    concat(measure2Result m,r)$ExpertSystemToolsPackage

  solve(P:NumericalPDEProblem):Result == solve(P,routines()$RT)

  solve(xmi:F,xma:F,ymi:F,yma:F,nx:NNI,ny:NNI,pe:LEF,bo:List
                LEF,s:ST,to:DF):Result ==
    cx:PDEC := [f2df xmi, f2df xma, nx, 1, empty()$MDF, empty()$MDF]
    cy:PDEC := [f2df ymi, f2df yma, ny, 1, empty()$MDF, empty()$MDF]
    p:PDEB := [[ef2edf e for e in pe],[cx,cy],
                [[ef2edf u for u in w] for w in bo],s,to]
    solve(p::NumericalPDEProblem,routines()$RT)

  solve(xmi:F,xma:F,ymi:F,yma:F,nx:NNI,ny:NNI,pe:LEF,bo:List
                LEF,s:ST):Result ==
```

```
    solve(xmi,xma,ymi,yma,nx,ny,pe,bo,s,0.0001::DF)
```

———-

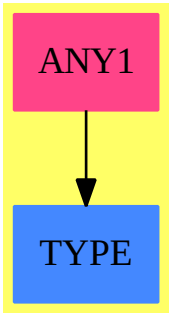
— **PDEPACK.dotabb** —

```
"PDEPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PDEPACK"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"PDEPACK" -> "TBAGG"
```

———-

2.15 package ANY1 AnyFunctions1

2.15.1 AnyFunctions1 (ANY1)



Exports:
coerce retract retractable? retractIfCan

— **package ANY1 AnyFunctions1** —

```
)abbrev package ANY1 AnyFunctions1
++ Author:
++ Date Created:
++ Change History:
++ Basic Functions:  coerce, retractIfCan, retractable?, retract
++ Related Constructors: Any
++ Also See:
++ AMS Classification:
++ Keywords:
++ Description:
++ \spadtype{AnyFunctions1} implements several utility functions for
```

```
++ working with \spadtype{Any}. These functions are used to go back
++ and forth between objects of \spadtype{Any} and objects of other
++ types.

AnyFunctions1(S:Type): with
        coerce      : S -> Any
          ++ coerce(s) creates an object of \spadtype{Any} from the
          ++ object \spad{s} of type \spad{S}.
        retractIfCan: Any -> Union(S, "failed")
          ++ retractIfCan(a) tries change \spad{a} into an object
          ++ of type \spad{S}. If it can, then such an object is
          ++ returned. Otherwise, "failed" is returned.
        retractable?: Any -> Boolean
          ++ retractable?(a) tests if \spad{a} can be converted
          ++ into an object of type \spad{S}.
        retract     : Any -> S
          ++ retract(a) tries to convert \spad{a} into an object of
          ++ type \spad{S}. If possible, it returns the object.
          ++ Error: if no such retraction is possible.

    == add
        import NoneFunctions1(S)

        Sexpr:SExpression := devaluate(S)$Lisp

        retractable? a  == dom(a) = Sexpr
        coerce(s:S):Any == any(Sexpr, s::None)

        retractIfCan a ==
            retractable? a => obj(a) pretend S
            "failed"

        retract a ==
            retractable? a => obj(a) pretend S
            error "Cannot retract value."
```

———-

— ANY1.dotabb —

```
"ANY1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ANY1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ANY1" -> "TYPE"
```

———-

2.16 package API ApplicationProgramInterface

— ApplicationProgramInterface.input —

```
)set break resume
)sys rm -f ApplicationProgramInterface.output
)spool ApplicationProgramInterface.output
)set message test on
)set message auto off
)clear all
--S 1 of 6
getDomains 'Collection
--R
--R   (1)
--R   {AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList,
--R    DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet,
--R    GeneralSparseTable, GeneralTriangularSet, HashTable, IndexedBits,
--R    IndexedFlexibleArray, IndexedList, IndexedOneDimensionalArray,
--R    IndexedString, IndexedVector, InnerTable, KeyedAccessFile, Library, List,
--R    ListMultiDictionary, Multiset, NeitherSparseOrDensePowerSeries,
--R    OneDimensionalArray, Point, PrimitiveArray, RegularChain,
--R    RegularTriangularSet, Result, RoutinesTable, Set, SparseTable,
--R    SquareFreeRegularTriangularSet, Stream, String, StringTable, Table,
--R    U32Vector, Vector, WuWenTsunTriangularSet}
--R                                                          Type: Set(Symbol)
--E 1

--S 2 of 6
difference(getDomains 'IndexedAggregate,getDomains 'Collection)
--R
--R   (2)
--R   {DirectProduct, DirectProductMatrixModule, DirectProductModule,
--R    HomogeneousDirectProduct, OrderedDirectProduct,
--R    SplitHomogeneousDirectProduct}
--R                                                          Type: Set(Symbol)
--E 2

--S 3 of 6
credits()
--R
--RAn alphabetical listing of contributors to AXIOM:
--RMichael Albaugh        Cyril Alberga          Roy Adler
--RChristian Aistleitner  Richard Anderson       George Andrews
--RS.J. Atkins
--RHenry Baker            Martin Baker           Stephen Balzac
--RYurij Baransky         David R. Barton        Gerald Baumgartner
--RGilbert Baumslag       Michael Becker         Nelson H. F. Beebe
--RJay Belanger           David Bindel           Fred Blair
```

```
--RVladimir Bondarenko    Mark Botch              Alexandre Bouyer
--RPeter A. Broadbery     Martin Brock            Manuel Bronstein
--RStephen Buchwald       Florian Bundschuh       Luanne Burns
--RWilliam Burge
--RQuentin Carpent        Robert Caviness         Bruce Char
--ROndrej Certik          Cheekai Chin            David V. Chudnovsky
--RGregory V. Chudnovsky  Mark Clements           James Cloos
--RJosh Cohen             Christophe Conil        Don Coppersmith
--RGeorge Corliss         Robert Corless          Gary Cornell
--RMeino Cramer           Claire Di Crescenzo     David Cyganski
--RNathaniel Daly         Timothy Daly Sr.        Timothy Daly Jr.
--RJames H. Davenport     Didier Deshommes        Michael Dewar
--RJean Della Dora        Gabriel Dos Reis        Claire DiCrescendo
--RSam Dooley             Lionel Ducos            Lee Duhem
--RMartin Dunstan         Brian Dupee             Dominique Duval
--RRobert Edwards         Heow Eide-Goodman       Lars Erickson
--RRichard Fateman        Bertfried Fauser        Stuart Feldman
--RJohn Fletcher          Brian Ford              Albrecht Fortenbacher
--RGeorge Frances         Constantine Frangos     Timothy Freeman
--RKorrinn Fu
--RMarc Gaetano           Rudiger Gebauer         Kathy Gerber
--RPatricia Gianni        Samantha Goldrich       Holger Gollan
--RTeresa Gomez-Diaz      Laureano Gonzalez-Vega  Stephen Gortler
--RJohannes Grabmeier     Matt Grayson            Klaus Ebbe Grue
--RJames Griesmer         Vladimir Grinberg       Oswald Gschnitzer
--RJocelyn Guidry
--RGaetan Hache           Steve Hague             Satoshi Hamaguchi
--RMike Hansen            Richard Harke           Bill Hart
--RVilya Harvey           Martin Hassner          Arthur S. Hathaway
--RDan Hatton             Waldek Hebisch          Karl Hegbloom
--RRalf Hemmecke          Henderson               Antoine Hersen
--RRoger House            Gernot Hueber
--RPietro Iglio
--RAlejandro Jakubi       Richard Jenks
--RKai Kaminski           Grant Keady             Wilfrid Kendall
--RTony Kennedy           Ted Kosan               Paul Kosinski
--RKlaus Kusche           Bernhard Kutzler
--RTim Lahey              Larry Lambe             Kaj Laurson
--RFranz Lehner           Frederic Lehobey        Michel Levaud
--RHoward Levy            Liu Xiaojun             Rudiger Loos
--RMichael Lucks          Richard Luczak
--RCamm Maguire           Francois Maltey         Alasdair McAndrew
--RBob McElrath           Michael McGettrick      Ian Meikle
--RDavid Mentre           Victor S. Miller        Gerard Milmeister
--RMohammed Mobarak       H. Michael Moeller      Michael Monagan
--RMarc Moreno-Maza       Scott Morrison          Joel Moses
--RMark Murray
--RWilliam Naylor         Patrice Naudin          C. Andrew Neff
--RJohn Nelder            Godfrey Nolan           Arthur Norman
--RJinzhong Niu
```

```
--RMichael O'Connor       Summat Oemrawsingh     Kostas Oikonomou
--RHumberto Ortiz-Zuazaga
--RJulian A. Padget       Bill Page              David Parnas
--RSusan Pelzel           Michel Petitot         Didier Pinchon
--RAyal Pinkus            Jose Alfredo Portes
--RClaude Quitte
--RArthur C. Ralfs        Norman Ramsey          Anatoly Raportirenko
--RAlbert D. Rich         Michael Richardson     Guilherme Reis
--RRenaud Rioboo          Jean Rivlin            Nicolas Robidoux
--RSimon Robinson         Raymond Rogers         Michael Rothstein
--RMartin Rubey
--RPhilip Santas          Alfred Scheerhorn      William Schelter
--RGerhard Schneider      Martin Schoenert       Marshall Schor
--RFrithjof Schulze       Fritz Schwarz          Steven Segletes
--RNick Simicich          William Sit            Elena Smirnova
--RJonathan Steinbach     Fabio Stumbo           Christine Sundaresan
--RRobert Sutor           Moss E. Sweedler       Eugene Surowitz
--RMax Tegmark            T. Doug Telford        James Thatcher
--RBalbir Thomas          Mike Thomas            Dylan Thurston
--RSteve Toleque          Barry Trager           Themos T. Tsikas
--RGregory Vanuxem
--RBernhard Wall          Stephen Watt           Jaap Weel
--RJuergen Weiss          M. Weller              Mark Wegman
--RJames Wen              Thorsten Werther       Michael Wester
--RJohn M. Wiley          Berhard Will           Clifton J. Williamson
--RStephen Wilson         Shmuel Winograd        Robert Wisbauer
--RSandra Wityak          Waldemar Wiwianka      Knut Wolf
--RClifford Yapp          David Yun
--RVadim Zhytnikov        Richard Zippel         Evelyn Zoernack
--RBruno Zuercher         Dan Zwillinger
--R                                                              Type: Void
--E 3
```

———-

The summary command simply types out the contents of the file $AXIOM/lib/summary so no output is captured in the spool file.

— **ApplicationProgramInterface.input** —

```
--S 4 of 6
summary()
--R
--R                                                              Type: Void
--E 4

--S 5 of 6
)show API
--R
--R ApplicationProgramInterface  is a package constructor
```

```
--R Abbreviation for ApplicationProgramInterface is API
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for API
--R
--R------------------------------ Operations --------------------------------
--R credits : () -> Void                 getDomains : Symbol -> Set(Symbol)
--R summary : () -> Void
--R
--E 5

--S 1 of 5
getAncestors 'IndexedAggregate
--R
--R   (3)
--R   {Aggregate, BasicType, CoercibleTo, Eltable, EltableAggregate, Evalable,
--R    HomogeneousAggregate, InnerEvalable, SetCategory, Type}
--R                                                          Type: Set(Symbol)
--E 6
)spool
)lisp (bye)
```

———-

— ApplicationProgramInterface.help —

```
=====================================================================
ApplicationProgramInterface examples
=====================================================================

The ApplicationProgramInterface exposes Axiom internal functions
which might be useful for understanding, debugging, or creating
tools.

The getDomains function takes the name of a category and returns
a set of domains which inherit from that category:

  getDomains 'Collection

   {AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList,
    DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet,
    GeneralSparseTable, GeneralTriangularSet, HashTable, IndexedBits,
    IndexedFlexibleArray, IndexedList, IndexedOneDimensionalArray,
    IndexedString, IndexedVector, InnerTable, KeyedAccessFile, Library, List,
    ListMultiDictionary, Multiset, NeitherSparseOrDensePowerSeries,
    OneDimensionalArray, Point, PrimitiveArray, RegularChain,
    RegularTriangularSet, Result, RoutinesTable, Set, SparseTable,
    SquareFreeRegularTriangularSet, Stream, String, StringTable, Table,
    U32Vector, Vector, WuWenTsunTriangularSet}
                                                        Type: Set Symbol
```

```
This can be used to form the set-difference of two categories:

  difference(getDomains 'IndexedAggregate, getDomains 'Collection)

   {DirectProduct, DirectProductMatrixModule, DirectProductModule,
    HomogeneousDirectProduct, OrderedDirectProduct,
    SplitHomogeneousDirectProduct}
                                                          Type: Set Symbol

The credits function prints a list of the people who have contributed
to the development of Axiom. This is equivalent to the )credits command.

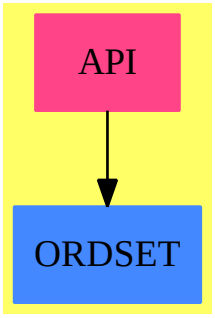
The summary function prints a short list of useful console commands.

The getAncestors function takes the name of a domain or category
and returns a list of categories from which it inherits.

getAncestors 'IndexedAggregate

   {Aggregate, BasicType, CoercibleTo, Eltable, EltableAggregate, Evalable,
    HomogeneousAggregate, InnerEvalable, SetCategory, Type}
                                                    Type: Set(Symbol)
```

2.16.1 ApplicationProgramInterface (API)



Exports:

— package API ApplicationProgramInterface —

```
)abbrev package API ApplicationProgramInterface
++ Author: Timothy Daly, Martin Rubey
++ Date Created: 3 March 2009
++ Date Last Updated: 24 March 2012
++ Description:
++ This package contains useful functions that expose Axiom system internals

ApplicationProgramInterface(): Exports == Implementation where
  Exports ==> with
    getDomains : Symbol -> Set Symbol
      ++ The getDomains(s) takes a category and returns the list of domains
      ++ that have that category
      ++
      ++X getDomains 'IndexedAggregate
    getAncestors : Symbol -> Set Symbol
      ++ The getAncestor(s) takes a category and returns the list of domains
      ++ that have that category as ancestors
      ++
      ++X getAncestors 'IndexedAggregate
    credits : () -> Void
      ++ credits() prints a list of people who contributed to Axiom
      ++
      ++X credits()
    summary : () -> Void
      ++ summary() prints a short list of useful console commands
      ++
      ++X summary()
  Implementation ==> add
    getDomains(cat:Symbol):Set(Symbol) ==
      set [symbol car first destruct a _
        for a in (destruct domainsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

    getAncestors(cat:Symbol):Set(Symbol) ==
      set [symbol car first destruct a _
        for a in (destruct ancestorsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

    credits() == ( credits()$Lisp ; void() )

    summary() == ( summary()$Lisp ; void() )
```

———-

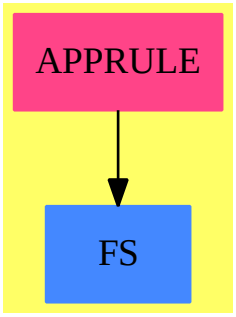
— API.dotabb —

```
"API" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPRULE"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"API" -> "ORDSET"
```

———-

2.17 package APPRULE ApplyRules

2.17.1 ApplyRules (APPRULE)



Exports:
applyRules localUnquote

— package APPRULE ApplyRules —

```
)abbrev package APPRULE ApplyRules
++ Applications of rules to expressions
++ Author: Manuel Bronstein
++ Date Created: 20 Mar 1990
++ Date Last Updated: 5 Jul 1990
++ Keywords: pattern, matching, rule.
++ Description:
++ This package apply rewrite rules to expressions, calling
++ the pattern matcher.

ApplyRules(Base, R, F): Exports == Implementation where
  Base   : SetCategory
  R      : Join(Ring, PatternMatchable Base, OrderedSet,
                                ConvertibleTo Pattern Base)
  F      : Join(FunctionSpace R, PatternMatchable Base,
                                ConvertibleTo Pattern Base)

  P  ==> Pattern Base
  PR ==> PatternMatchResult(Base, F)
  RR ==> RewriteRule(Base, R, F)
  K  ==> Kernel F

  Exports ==> with
    applyRules  : (List RR, F) -> F
```

```
    ++ applyRules([r1,...,rn], expr) applies the rules
    ++ r1,...,rn to f an unlimited number of times, i.e. until
    ++ none of r1,...,rn is applicable to the expression.
  applyRules  : (List RR, F, PositiveInteger) -> F
    ++ applyRules([r1,...,rn], expr, n) applies the rules
    ++ r1,...,rn to f a most n times.
  localUnquote: (F, List Symbol) -> F
    ++ localUnquote(f,ls) is a local function.

Implementation ==> add
  import PatternFunctions1(Base, F)

  splitRules: List RR -> Record(lker: List K,lval: List F,rl: List RR)
  localApply  : (List K, List F, List RR, F, PositiveInteger) -> F
  rewrite     : (F, PR, List Symbol) -> F
  app         : (List RR, F) -> F
  applist     : (List RR, List F) -> List F
  isit        : (F, P) -> PR
  isitwithpred: (F, P, List P, List PR) -> PR

  applist(lrule, arglist)  == [app(lrule, arg) for arg in arglist]

  splitRules l ==
    ncr := empty()$List(RR)
    lk  := empty()$List(K)
    lv  := empty()$List(F)
    for r in l repeat
      if (u := retractIfCan(r)@Union(Equation F, "failed"))
        case "failed" then ncr := concat(r, ncr)
        else
          lk := concat(retract(lhs(u::Equation F))@K, lk)
          lv := concat(rhs(u::Equation F), lv)
    [lk, lv, ncr]

  applyRules(l, s) ==
    rec := splitRules l
    repeat
      (new:= localApply(rec.lker,rec.lval,rec.rl,s,1)) = s => return s
      s := new

  applyRules(l, s, n) ==
    rec := splitRules l
    localApply(rec.lker, rec.lval, rec.rl, s, n)

  localApply(lk, lv, lrule, subject, n) ==
    for i in 1..n repeat
      for k in lk for v in lv repeat
        subject := eval(subject, k, v)
      subject := app(lrule, subject)
    subject
```

```
rewrite(f, res, l) ==
  lk := empty()$List(K)
  lv := empty()$List(F)
  for rec in destruct res repeat
    lk := concat(kernel(rec.key), lk)
    lv := concat(rec.entry, lv)
  localUnquote(eval(f, lk, lv), l)

if R has ConvertibleTo InputForm then
  localUnquote(f, l) ==
    empty? l => f
    eval(f, l)
else
  localUnquote(f, l) == f

isitwithpred(subject, pat, vars, bad) ==
  failed?(u := patternMatch(subject, pat, new()$PR)) => u
  satisfy?(u, pat)::Boolean => u
  member?(u, bad) => failed()
  for v in vars repeat addBadValue(v, getMatch(v, u)::F)
  isitwithpred(subject, pat, vars, concat(u, bad))

isit(subject, pat) ==
  hasTopPredicate? pat =>
    for v in (l := variables pat) repeat resetBadValues v
    isitwithpred(subject, pat, l, empty())
  patternMatch(subject, pat, new()$PR)

app(lrule, subject) ==
  for r in lrule repeat
    not failed?(u := isit(subject, pattern r)) =>
      return rewrite(rhs r, u, quotedOperators r)
  (k := retractIfCan(subject)@Union(K, "failed")) case K =>
    operator(k::K) applist(lrule, argument(k::K))
  (l := isPlus  subject) case List(F) => +/applist(lrule,l::List(F))
  (l := isTimes subject) case List(F) => */applist(lrule,l::List(F))
  (e := isPower subject) case Record(val:F, exponent:Integer) =>
    ee := e::Record(val:F, exponent:Integer)
    f  := app(lrule, ee.val)
    positive?(ee.exponent) => f ** (ee.exponent)::NonNegativeInteger
    recip(f)::F ** (- ee.exponent)::NonNegativeInteger
  subject
```

———-

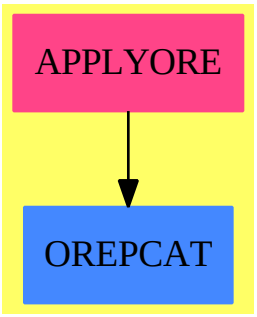
— APPRULE.dotabb —

```
"APPRULE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPRULE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"APPRULE" -> "FS"
```

———-

2.18 package APPLYORE ApplyUnivariateSkewPolynomial

2.18.1 ApplyUnivariateSkewPolynomial (APPLYORE)



Exports:
apply

— package APPLYORE ApplyUnivariateSkewPolynomial —

```
)abbrev package APPLYORE ApplyUnivariateSkewPolynomial
++ Author: Manuel Bronstein
++ Date Created: 7 December 1993
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{ApplyUnivariateSkewPolynomial} (internal) allows univariate
++ skew polynomials to be applied to appropriate modules.

ApplyUnivariateSkewPolynomial(R:Ring, M: LeftModule R,
    P: UnivariateSkewPolynomialCategory R): with
      apply: (P, M -> M, M) -> M
        ++ apply(p, f, m) returns \spad{p(m)} where the action is given
        ++ by \spad{x m = f(m)}.
        ++ \spad{f} must be an R-pseudo linear map on M.
   == add
      apply(p, f, m) ==
        w:M  := 0
```

```
        mn:M := m
        for i in 0..degree p repeat
          w  := w + coefficient(p, i) * mn
          mn := f mn
        w
```

———-

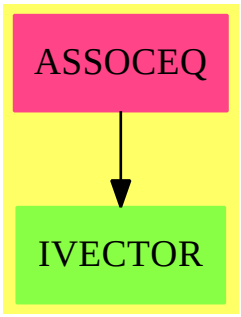
— **APPLYORE.dotabb** —

```
"APPLYORE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPLYORE"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"APPLYORE" -> "OREPCAT"
```

———-

2.19 package ASSOCEQ AssociatedEquations

2.19.1 AssociatedEquations (ASSOCEQ)



Exports:
associatedEquations associatedSystem uncouplingMatrices

— **package ASSOCEQ AssociatedEquations** —

```
)abbrev package ASSOCEQ AssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 10 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{AssociatedEquations} provides functions to compute the
++ associated equations needed for factoring operators
```

```
AssociatedEquations(R, L):Exports == Implementation where
  R: IntegralDomain
  L: LinearOrdinaryDifferentialOperatorCategory R

  PI  ==> PositiveInteger
  N   ==> NonNegativeInteger
  MAT ==> Matrix R
  REC ==> Record(minor: List PI, eq: L, minors: List List PI, ops: List L)

  Exports ==> with
    associatedSystem: (L, PI) -> Record(mat: MAT, vec:Vector List PI)
      ++ associatedSystem(op, m) returns \spad{[M,w]} such that the
      ++ m-th associated equation system to L is \spad{w' = M w}.
    uncouplingMatrices: MAT -> Vector MAT
      ++ uncouplingMatrices(M) returns \spad{[A_1,...,A_n]} such that if
      ++ \spad{y = [y_1,...,y_n]} is a solution of \spad{y' = M y}, then
      ++ \spad{[$y_j',y_j'',...,y_j^{(n)}$] = $A_j y$} for all j's.
    if R has Field then
        associatedEquations: (L, PI) -> REC
          ++ associatedEquations(op, m) returns \spad{[w, eq, lw, lop]}
          ++ such that \spad{eq(w) = 0} where w is the given minor, and
          ++ \spad{lw_i = lop_i(w)} for all the other minors.

  Implementation ==> add
    makeMatrix: (Vector MAT, N) -> MAT

    diff:L := D()

    makeMatrix(v, n) == matrix [parts row(v.i, n) for i in 1..#v]

    associatedSystem(op, m) ==
      eq: Vector R
      S := SetOfMIntegersInOneToN(m, n := degree(op)::PI)
      w := enumerate()$S
      s := size()$S
      ww:Vector List PI := new(s, empty())
      M:MAT := new(s, s, 0)
      m1 := (m::Integer - 1)::PI
      an := leadingCoefficient op
      a:Vector(R) := [- (coefficient(op, j) exquo an)::R for j in 0..n - 1]
      for i in 1..s repeat
          eq := new(s, 0)
          wi := w.i
          ww.i := elements wi
          for k in 1..m1 repeat
              u := incrementKthElement(wi, k::PI)$S
              if u case S then eq(lookup(u::S)) := 1
          if member?(n, wi) then
              for j in 1..n | a.j ^= 0 repeat
```

```
                u := replaceKthElement(wi, m, j::PI)
                if u case S then
                  eq(lookup(u::S)) := (odd? delta(wi, m, j::PI) => -a.j; a.j)
        else
            u := incrementKthElement(wi, m)$S
            if u case S then eq(lookup(u::S)) := 1
        setRow_!(M, i, eq)
    [M, ww]

uncouplingMatrices m ==
    n := nrows m
    v:Vector MAT := new(n, zero(1, 0)$MAT)
    v.1 := mi := m
    for i in 2..n repeat v.i := mi := map((z1:R):R +-> diff z1, mi) + mi * m
    [makeMatrix(v, i) for i in 1..n]

if R has Field then
    import PrecomputedAssociatedEquations(R, L)

    makeop:    Vector R -> L
    makeeq:    (Vector List PI, MAT, N, N) -> REC
    computeIt: (L, PI, N) -> REC

    makeeq(v, m, i, n) ==
      [v.i, makeop row(m, i) - 1, [v.j for j in 1..n | j ^= i],
                                  [makeop row(m, j) for j in 1..n | j ^= i]]

    associatedEquations(op, m) ==
      (u := firstUncouplingMatrix(op, m)) case "failed" => computeIt(op,m,1)
      (v := inverse(u::MAT)) case "failed" => computeIt(op, m, 2)
      S := SetOfMIntegersInOneToN(m, degree(op)::PI)
      w := enumerate()$S
      s := size()$S
      ww:Vector List PI := new(s, empty())
      for i in 1..s repeat ww.i := elements(w.i)
      makeeq(ww, v::MAT, 1, s)

    computeIt(op, m, k) ==
      rec := associatedSystem(op, m)
      a := uncouplingMatrices(rec.mat)
      n := #a
      for i in k..n repeat
        (u := inverse(a.i)) case MAT => return makeeq(rec.vec,u::MAT,i,n)
      error "associatedEquations: full degenerate case"

    makeop v ==
      op:L := 0
      for i in 1..#v repeat op := op + monomial(v i, i)
      op
```

———-

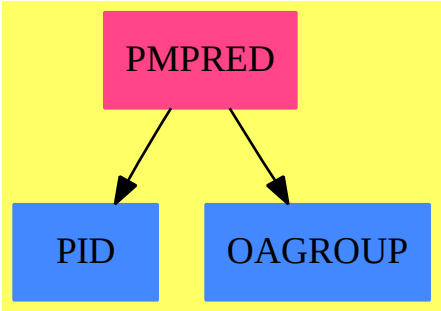
— **ASSOCEQ.dotabb** —

```
"ASSOCEQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ASSOCEQ"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ASSOCEQ" -> "IVECTOR"
```

———-

2.20 package PMPRED AttachPredicates

2.20.1 AttachPredicates (PMPRED)



Exports:
suchThat

— **package PMPRED AttachPredicates** —

```
)abbrev package PMPRED AttachPredicates
++ Predicates for pattern-matching
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Keywords: pattern, matching.
++ Description:
++ Attaching predicates to symbols for pattern matching.

AttachPredicates(D:Type): Exports == Implementation where
  FE ==> Expression Integer

  Exports ==> with
    suchThat: (Symbol, D -> Boolean) -> FE
```

```
      ++ suchThat(x, foo) attaches the predicate foo to x.
    suchThat: (Symbol, List(D -> Boolean)) -> FE
      ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
      ++ f1 and f2 and ... and fn to x.

  Implementation ==> add
    import FunctionSpaceAttachPredicates(Integer, FE, D)

    suchThat(p:Symbol, f:D -> Boolean)       == suchThat(p::FE, f)
    suchThat(p:Symbol, l:List(D -> Boolean)) == suchThat(p::FE, l)
```

———-

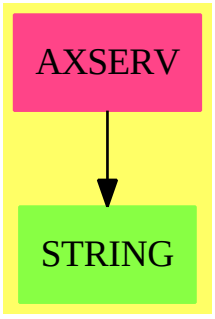
— **PMPRED.dotabb** —

```
"PMPRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPRED"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PMPRED" -> "PID"
"PMPRED" -> "OAGROUP"
```

———-

2.21 package AXSERV AxiomServer

2.21.1 AxiomServer (AXSERV)



Exports:
multiServ axServer getDatabase

— **package AXSERV AxiomServer** —

```
)abbrev package AXSERV AxiomServer
++ Author: Arthur Ralfs, Timothy Daly, Alfredo Portes
++ Description:
++ This package provides a functions to support a web server for the
++ new Axiom Browser functions.

AxiomServer: public == private where

 public == with

   axServer: (Integer, SExpression->Void) -> Void
   multiServ: SExpression -> Void
   getDatabase: (String,String) -> String

 private == add

   getFile: (SExpression,String) -> Void
   getCommand: (SExpression,String) -> Void
   getDescription: String -> String
   getInterp: (SExpression,String) -> Void
   getLisp:   (SExpression,String) -> Void
   getShow:   (SExpression,String) -> Void
   lastStep: () -> String
   lastType: () -> String
   formatMessages: String -> String
   makeErrorPage: String -> String
   getSourceFile: (String,String,String) -> String
   makeDBPage: String -> String
   getContentType: String -> String
   readTheFile: SExpression -> String
   outputToSocket: (SExpression,String,String) -> Void

   getDatabase(constructor:String, key:String):String ==
     answer:=string GETDATABASE(INTERN$Lisp constructor,INTERN$Lisp key)$Lisp
--     WriteLine$Lisp concat ["getDatabase: ",constructor," ",key," ",answer]
     answer
```

———-

The axServer function handles the socket connection on the given port. When it gets a input on the socket it calls the server function on the socket input.

— **package AXSERV AxiomServer** —

```
   axServer(port:Integer,serverfunc:SExpression->Void):Void ==
     WriteLine$Lisp "listening on port 8085"
     s := SiSock(port,serverfunc)$Lisp
     -- To listen for just one connection and then close the socket
     -- uncomment i := 0.
```

```
     i:Integer := 1
     while (i > 0) repeat
       if not null?(SiListen(s)$Lisp)$SExpression then
         w := SiAccept(s)$Lisp
         serverfunc(w)
--        i := 0
```

The multiServ function parses the socket input. It expects either a GET or POST request. A GET request fetches a new page, calling "getFile". A POST request starts with

- "command=" which expects axiom interpreter commands. When this is recognized we call the "getCommand" function.
- "lispcall=" which expects lisp interpreter input When this is recognized we call the "getLisp" function.

— package AXSERV AxiomServer —

```
   multiServ(s:SExpression):Void ==
--     WriteLine("multiServ begin")$Lisp
     headers:String := ""
     char:String
     -- read in the http headers
     while (char := _
       STRING(READ_-CHAR_-NO_-HANG(s,NIL$Lisp,'EOF)$Lisp)$Lisp) ^= "EOF"_
        repeat
         headers := concat [headers,char]
--     sayTeX$Lisp headers
     StringMatch("([^ ]*)", headers)$Lisp
     u:UniversalSegment(Integer)
     u := segment(MatchBeginning(1)$Lisp+1,_
                  MatchEnd(1)$Lisp)$UniversalSegment(Integer)
     reqtype:String := headers.u
--     sayTeX$Lisp  concat ["request type: ",reqtype]
     if  reqtype = "GET" then
         StringMatch("GET ([^ ]*)",headers)$Lisp
         u:UniversalSegment(Integer)
         u := segment(MatchBeginning(1)$Lisp+1,_
                      MatchEnd(1)$Lisp)$UniversalSegment(Integer)
         getFile(s,headers.u)
     if reqtype = "POST" and StringMatch("command=(.*)$",headers)$Lisp > 0
      then
         u:UniversalSegment(Integer)
         u := segment(MatchBeginning(1)$Lisp+1,_
```

```
                    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
        getCommand(s,headers.u)
    if reqtype = "POST" and StringMatch("interpcall=(.*)$",headers)$Lisp > 0
     then
        u:UniversalSegment(Integer)
        u := segment(MatchBeginning(1)$Lisp+1,_
                    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
        getInterp(s,headers.u)
    if reqtype = "POST" and StringMatch("lispcall=(.*)$",headers)$Lisp > 0
     then
        u:UniversalSegment(Integer)
        u := segment(MatchBeginning(1)$Lisp+1,_
                    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
        getLisp(s,headers.u)
    if reqtype = "POST" and StringMatch("showcall=(.*)$",headers)$Lisp > 0
     then
        u:UniversalSegment(Integer)
        u := segment(MatchBeginning(1)$Lisp+1,_
                    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
        getShow(s,headers.u)
--     WriteLine("multiServ end")$Lisp
--     WriteLine("")$Lisp
```

———-

getFile

Given a socket and the URL of the file we create an input stream that contains the file. If the filename contains a question mark then we need to parse the parameters and dynamically construct the file contents.

— package AXSERV AxiomServer —

```
   getFile(s:SExpression,pathvar:String):Void ==
--     WriteLine("")$Lisp
     WriteLine$Lisp concat ["getFile: ",pathvar]
     params:=split(pathvar,char "?")
     if #params = 1
      then if not null? PATHNAME_-NAME(PATHNAME(pathvar)$Lisp)$Lisp
       then
         contentType:String := getContentType(pathvar)
         q:=Open(pathvar)$Lisp
         if null? q
           then
             q := MAKE_-STRING_-INPUT_-STREAM(_
                   makeErrorPage("File doesn't exist"))$Lisp
       else
         q:=MAKE_-STRING_-INPUT_-STREAM(_
             makeErrorPage("Problem with file path"))$Lisp
```

```
      else
       q:=MAKE_-STRING_-INPUT_-STREAM(makeDBPage(pathvar))$Lisp
     outputToSocket(s,readTheFile(q),contentType)
```

———-

makeErrorPage

— **package AXSERV AxiomServer** —

```
   makeErrorPage(msg:String):String ==
     page:String:="<!DOCTYPE html PUBLIC "
     page:=page "_"-//W3C//DTD XHTML 1.0 Strict//EN_" "
     page:=page "_"http://www.w3.org/TR/xthml1/DTD/xhtml1-strict.dtd_">"
     page:=page "<html xmlns=_"http://www.w3.org/1999/xhtml_">"
     page:=page "<head><title>Error</title></head><body>" msg "</body></html>"
--     WriteLine(page)$Lisp
     page
```

———-

getDescription

We need to fish around in the data structure to return the piece of documentation for the domain. We have to call the lisp version of GETDATABASE because the version above returns a string object. The string object is missing quotes and cannot be properly read. So we need to get the lisp object and work with it in native form first.

The doc string also contains spad markup which we need to replace with html.

— **package AXSERV AxiomServer** —

```
   getDescription(dom:String):String ==
    d:=CADR(CADAR(GETDATABASE(INTERN(dom)$Lisp,'DOCUMENTATION)$Lisp)$Lisp)$Lisp
    string d
```

———-

getSourceFile

During build we construct a hash table that takes the chunk name as the key and returns the filename. We reconstruct the chunk name here and do a lookup for the source file.

— **package AXSERV AxiomServer** —

```
   getSourceFile(constructorkind:String,_
                 abbreviation:String,_
```

```
                dom:String):String ==
    sourcekey:="@<<" constructorkind " " abbreviation " " dom ">>"
--     WriteLine(sourcekey)$Lisp
    sourcefile:=lowerCase last split(getDatabase(dom,"SOURCEFILE"),char "/")
    sourcefile:=sourcefile ".pamphlet"
```

———-

makeDBPage

— **package AXSERV AxiomServer** —

```
makeDBPage(pathvar:String):String ==
  params:List(String):=split(pathvar,char "?")
  for i in 1..#params repeat WriteLine$Lisp concat ["params: ",params.i]
  pathparts:List(String):=split(params.1,char "/")
  for i in 1..#pathparts repeat
    WriteLine$Lisp concat ["pathparts: ",pathparts.i]
  pagename:=last pathparts
  WriteLine$Lisp concat ["pagename: ",pagename]
  cmd:=first split(pagename,char ".")
  WriteLine$Lisp concat ["cmd: ",cmd]
  args:List(String):=split(params.2, char "&")
  for i in 1..#args repeat WriteLine$Lisp concat ["args: ",args.i]
  page:String:="<!DOCTYPE html PUBLIC "
  page:=page "_"-//W3C//DTD XHTML 1.0 Strict//EN_" "
  page:=page "_"http://www.w3.org/TR/xthml1/DTD/xhtml1-strict.dtd_">"
  page:=page "<html xmlns=_"http://www.w3.org/1999/xhtml_">"
  page:=page "<head>"
  page:=page "<meta http-equiv=_"Content-Type_" content=_"text/html_""
  page:=page " charset=_"us-ascii_"/>"
  page:=page "<title>" cmd " " args.1 "</title></head>"
  page:=page "<style> html { background-color: #ECEA81; } </style>"
  page:=page "<body>"
  cmd = "db" =>
   dom:=args.1
   domi:=INTERN(dom)$Lisp
   -- category, domain, or package?
   constructorkind:=getDatabase(dom,"CONSTRUCTORKIND")
   abbreviation:=getDatabase(dom, "ABBREVIATION")
   sourcefile:=getDatabase(dom, "SOURCEFILE")
   constructorkind.1:=upperCase constructorkind.1
   description:=getDescription(dom)
   page:=page "<div align=_"center_">"
   page:=page "<img align=_"middle_" src=_"doctitle.png_"/></div><hr/>"
   page:=page "<div align=_"center_">" constructorkind " " dom "</div><hr/>"
   page:=page "<table>"
```

```
 page:=page "<tr><td valign=_"top_">Description:  </td>"
 page:=page "<td>" description  "</td></tr>"
 page:=page "<tr><td>Abbreviation: </td><td>" abbreviation "</td></tr>"
 page:=page "<tr><td>Source File:  </td><td>" sourcefile   "</td></tr>"
 page:=page "</table><hr/>"
 page:=page "<table>"
 page:=page "<tr>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Ancestors_">Ancestors</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Dependents_">Dependents</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Exports_">Exports</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Parents_">Parents</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Users_">Users</a>"
 page:=page "</td>"
 page:=page "</tr>"
 page:=page "<tr>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Attributes_">Attributes</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Examples_">Examples</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Operations_">Operations</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=SearchPath_">Search Path</a>"
 page:=page "</td>"
 page:=page "<td>"
 page:=page "<a href=_"?" dom "&lookup=Uses_">Uses</a>"
 page:=page "</td>"
 page:=page "</tr>"
 page:=page "</table>"
cmd = "op" =>
 dom:=args.1
 domi:=INTERN(dom)$Lisp
 -- category, domain, or package?
 constructorkind:=getDatabase(dom,"CONSTRUCTORKIND")
 abbreviation:=getDatabase(dom, "ABBREVIATION")
 sourcefile:=getDatabase(dom, "SOURCEFILE")
 constructorkind.1:=upperCase constructorkind.1
 description:=getDescription(dom)
```

```
      page:=page "<div align=_"center_">"
      page:=page "<img align=_"middle_" src=_"doctitle.png_"/></div><hr/>"
      page:=page "<div align=_"center_">" constructorkind " " dom "</div><hr/>"
      page:=page "<table>"
      page:=page "<tr><td valign=_"top_">Description:  </td>"
      page:=page "<td>" description  "</td></tr>"
      page:=page "<tr><td>Abbreviation: </td><td>" abbreviation "</td></tr>"
      page:=page "<tr><td>Source File:  </td><td>" sourcefile   "</td></tr>"
      page:=page "</table><hr/>"
      page:=page "<table>"
      page:=page "<tr>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Ancestors_">Ancestors</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Dependents_">Dependents</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Exports_">Exports</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Parents_">Parents</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Users_">Users</a>"
      page:=page "</td>"
      page:=page "</tr>"
      page:=page "<tr>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Attributes_">Attributes</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Examples_">Examples</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Operations_">Operations</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=SearchPath_">Search Path</a>"
      page:=page "</td>"
      page:=page "<td>"
      page:=page "<a href=_"?" dom "&lookup=Uses_">Uses</a>"
      page:=page "</td>"
      page:=page "</tr>"
      page:=page "</table>"
--      WriteLine(page)$Lisp
     page:=page "</body></html>"
     page
```

———-

readTheFile

We have q which is a stream which contains the file. We read the file into a string-stream to get it all into one string. We return the string.

— package AXSERV AxiomServer —

```
   readTheFile(q:SExpression):String ==
--     WriteLine("begin reading file")$Lisp
     r := MAKE_-STRING_-OUTPUT_-STREAM()$Lisp
     SiCopyStream(q,r)$Lisp
     filestream:String := GET_-OUTPUT_-STREAM_-STRING(r)$Lisp
     CLOSE(r)$Lisp
     CLOSE(q)$Lisp
--     WriteLine("end reading file")$Lisp
     filestream
```

———-

outputToSocket

We have "s" which is the socket, "filestream" which is the text of the file to output, and "contentType" which is the HTML Content-Type. We construct the HTML header information according to the standard and prepend it to the file. The resulting string is output to the socket.

— package AXSERV AxiomServer —

```
   outputToSocket(s:SExpression,filestream:String,contentType:String):Void ==
     filelength:String := string(#filestream)
     file:String := ""
     nl:String:=STRING(NewLine$Lisp)$Lisp
     file := concat ["Content-Length: ",filelength,nl,nl,file]
     file := concat ["Connection: close",nl,file]
     file := concat ["Content-Type: ",contentType,nl,file]
     file := concat ["HTTP/1.1 200 OK",nl,file]
     file := concat [file,filestream]
--     WriteLine(file)$Lisp
     f:=MAKE_-STRING_-INPUT_-STREAM(file)$Lisp
     SiCopyStream(f,s)$Lisp
     CLOSE(f)$Lisp
     CLOSE(s)$Lisp
```

———-

getCommand

The getCommand function is invoked when the HTTP request is a POST and contains the string "command". Essentially the game here is to rebind the various output streams used by Axiom so we can capture the normal output. This function returns a set of HTML 5 div blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

— **package AXSERV AxiomServer** —

```
getCommand(s:SExpression,command:String):Void ==
    WriteLine$Lisp concat ["getCommand: ",command]
    SETQ(tmpmathml$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
    SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
    ans := string parseAndEvalToStringEqNum$Lisp command
    SETQ(resultmathml$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)$Lisp
    SETQ(resultalgebra$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
    CLOSE(tmpmathml$Lisp)$Lisp
    CLOSE(tmpalgebra$Lisp)$Lisp
    -- Since strings returned from axiom are going to be displayed in html I
    -- should really check for the characters &,<,> and replace them with
    -- &amp;,&lt;,&gt;.
    -- At present I only check for ampersands in formatMessages.
    mathml:String := string(resultmathml$Lisp)
    algebra:String := string(resultalgebra$Lisp)
    algebra := formatMessages(algebra)
    -- At this point mathml contains the mathml for the output but does not
    -- include step number or type information.
    -- We should also save the command.
    -- I get the type and step number from the $internalHistoryTable
    axans:String := _
```

```
        concat ["<div class=_"stepnum_">", lastStep(), "</div>_
                <div class=_"command_">", command, "</div>_
                <div class=_"algebra_">",algebra,"</div>_
                <div class=_"mathml_">",mathml,"</div>_
                <div class=_"type_">",lastType(),"</div>"]
--      WriteLine$Lisp concat ["mathml answer: ",mathml]
--      WriteLine$Lisp concat ["algebra answer: ",algebra]
      q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
      SiCopyStream(q,s)$Lisp
      CLOSE(q)$Lisp
      CLOSE(s)$Lisp
```

———-

getInterp

The getInterp function is invoked when the HTTP request is a POST and contains the string "command". Essentially the game here is to rebind the various output streams used by Axiom so we can capture the normal output. This function returns a set of HTML 5 div blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

— **package AXSERV AxiomServer** —

```
getInterp(s:SExpression,command:String):Void ==
    WriteLine$Lisp concat ["getInterp: ",command]
    SETQ(tmpmathml$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
    SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
    ans := string parseAndEvalToStringEqNum$Lisp command
    SETQ(resultmathml$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)$Lisp
    SETQ(resultalgebra$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
```

```
      SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
      SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
      CLOSE(tmpmathml$Lisp)$Lisp
      CLOSE(tmpalgebra$Lisp)$Lisp
      -- Since strings returned from axiom are going to be displayed in html I
      -- should really check for the characters &,<,> and replace them with
      -- &amp;,&lt;,&gt;.
      -- At present I only check for ampersands in formatMessages.
      mathml:String := string(resultmathml$Lisp)
      algebra:String := string(resultalgebra$Lisp)
      algebra := formatMessages(algebra)
      -- At this point mathml contains the mathml for the output but does not
      -- include step number or type information.
      -- We should also save the command.
      -- I get the type and step number from the $internalHistoryTable
      axans:String := _
        concat ["<div class=_"stepnum_">", lastStep(), "</div>_
                 <div class=_"command_">", command, "</div>_
                 <div class=_"algebra_">",algebra,"</div>_
                 <div class=_"mathml_">",mathml,"</div>_
                 <div class=_"type_">",lastType(),"</div>"]
--      WriteLine$Lisp concat ["mathml answer: ",mathml]
--      WriteLine$Lisp concat ["algebra answer: ",algebra]
      q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
      SiCopyStream(q,s)$Lisp
      CLOSE(q)$Lisp
      CLOSE(s)$Lisp
```

———-

getLisp

The getLisp function is invoked when the HTTP request is a POST and contains the string "lispcall".

— **package AXSERV AxiomServer** —

```
   getLisp(s:SExpression,command:String):Void ==
      WriteLine$Lisp concat ["getLisp: ",command]
      evalresult:=EVAL(READ_-FROM_-STRING(command)$Lisp)$Lisp
      mathml:String:=string(evalresult)
--      WriteLine$Lisp concat ["getLisp: after ",mathml]
--      WriteLine$Lisp concat ["getLisp output: ",mathml]
      SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
      SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
      SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
      SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
      SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
      SETQ(resultalgebra$Lisp,_
```

```
          GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
      SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
      SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
      CLOSE(tmpalgebra$Lisp)$Lisp
      -- Since strings returned from axiom are going to be displayed in html I
      -- should really check for the characters &,<,> and replace them with
      -- &amp;,&lt;,&gt;.
      -- At present I only check for ampersands in formatMessages.
      algebra:String := string(resultalgebra$Lisp)
      algebra := formatMessages(algebra)
      -- At this point mathml contains the mathml for the output but does not
      -- include step number or type information.
      -- We should also save the command.
      -- I get the type and step number from the $internalHistoryTable
      axans:String := _
       concat ["<div class=_"stepnum_">", lastStep(), "</div>_
                <div class=_"command_">", command, "</div>_
                <div class=_"algebra_">",algebra,"</div>_
                <div class=_"mathml_">",mathml,"</div>_
                <div class=_"type_">",lastType(),"</div>"]
--      WriteLine$Lisp concat ["mathml answer: ",mathml]
--      WriteLine$Lisp concat ["algebra answer: ",algebra]
      q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
      SiCopyStream(q,s)$Lisp
      CLOSE(q)$Lisp
      CLOSE(s)$Lisp
```

———-

getShow

The getShow function is invoked when the HTTP request is a POST and contains the string "showcall". The)show command generates output to lisp's *standard-output* so we wrap that stream to capture it. The resulting string needs to be transformed into html-friendly form. This is done in the call to replace-entitites (see http.lisp)

— package AXSERV AxiomServer —

```
   getShow(s:SExpression,showarg:String):Void ==
       WriteLine$Lisp concat ["getShow: ",showarg]
       realarg:=SUBSEQ(showarg,6)$Lisp
       show:=_
        "(progn (setq |$options| '((|operations|))) (|show| '|" realarg "|))"
--       WriteLine$Lisp concat ["getShow: ",show]
       SETQ(SAVESTREAM$Lisp,_*STANDARD_-OUTPUT_*$Lisp)$Lisp
       SETQ(_*STANDARD_-OUTPUT_*$Lisp,_
             MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
       evalresult:=EVAL(READ_-FROM_-STRING(show)$Lisp)$Lisp
       SETQ(evalresult,_
```

```
      GET_-OUTPUT_-STREAM_-STRING(_*STANDARD_-OUTPUT_*$Lisp)$Lisp)$Lisp
    SETQ(_*STANDARD_-OUTPUT_*$Lisp,SAVESTREAM$Lisp)$Lisp
    mathml:String:=string(REPLACE_-ENTITIES(evalresult)$Lisp)
    SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
    SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
    SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
    SETQ(resultalgebra$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
    SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
    SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
    CLOSE(tmpalgebra$Lisp)$Lisp
    -- Since strings returned from axiom are going to be displayed in html I
    -- should really check for the characters &,<,> and replace them with
    -- &amp;,&lt;,&gt;.
    -- At present I only check for ampersands in formatMessages.
    algebra:String := string(resultalgebra$Lisp)
    algebra := formatMessages(algebra)
    -- At this point mathml contains the mathml for the output but does not
    -- include step number or type information.
    -- We should also save the command.
    -- I get the type and step number from the $internalHistoryTable
    axans:String := _
     concat ["<div class=_"stepnum_">", lastStep(), "</div>_
              <div class=_"command_">", showarg, "</div>_
              <div class=_"algebra_">",algebra,"</div>_
              <div class=_"mathml_">",mathml,"</div>_
              <div class=_"type_">",lastType(),"</div>"]
--    WriteLine$Lisp concat ["mathml answer: ",mathml]
    q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
    SiCopyStream(q,s)$Lisp
    CLOSE(q)$Lisp
    CLOSE(s)$Lisp
```

———-

lastType

To examine the $internalHistoryTable use the following line

```
)lisp |$internalHistoryTable|
```

We need to pick out first member of internalHistoryTable and then pick out the element with % as first element. Here is an example showing just the first element of the list, which correponds to the last command.

Note that the last command does not necessarily correspond to the last element of the first element of $internalHistoryTable as it is in this example.

```
(
 (4 NIL
 (x (value (BasicOperator) WRAPPED . #<vector 09a93bd0>))
 (y (value (BasicOperator) WRAPPED . #<vector 09a93bb4>))
 (% (value (Matrix (Polynomial (Integer))) WRAPPED . #<vector 0982e0e0>))
 )
...
)
```

We also need to check for input error in which case the $internalHistoryTable is not changed and the type retrieved would be that for the last correct input.

— package AXSERV AxiomServer —

```
lastType():String ==
  SETQ(first$Lisp,FIRST(_$internalHistoryTable$Lisp)$Lisp)$Lisp
  count:Integer := 0
  hisLength:Integer := LIST_-LENGTH(_$internalHistoryTable$Lisp)$Lisp
  length:Integer := LIST_-LENGTH(first$Lisp)$Lisp
  -- This initializes stepSav.  The test is a bit of a hack, maybe I'll
  -- figure out the right way to do it later.
  if string stepSav$Lisp = "#<OBJNULL>" then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
  -- If hisLength = 0 then the history table has been reset to NIL
  -- and we're starting numbering over
  if hisLength = 0 then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
  if hisLength > 0 and
    car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp ^= stepSav$Lisp then
     SETQ(stepSav$Lisp,car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp)$Lisp
     while count < length  repeat
      position(char "%",string FIRST(first$Lisp)$Lisp) = 2 =>
        count := length+1
      count := count +1
      SETQ(first$Lisp,REST(first$Lisp)$Lisp)$Lisp
  count = length + 1 =>
      string SECOND(SECOND(FIRST(first$Lisp)$Lisp)$Lisp)$Lisp
  ""


lastStep():String ==
    string car(car(_$internalHistoryTable$Lisp)$Lisp)$Lisp

formatMessages(str:String):String ==
--      WriteLine("formatMessages")$Lisp
      -- I need to replace any ampersands with &amp; and may also need to
      -- replace < and > with &lt; and &gt;
      strlist:List String
--      WriteLine(str)$Lisp
      strlist := split(str,char "&")
      str := ""
      -- oops, if & is the last character in the string this method
      -- will eliminate it.  Need to redo this.
```

```
      for s in strlist repeat
          str := concat [str,s,"&amp;"]
      strlen:Integer := #str
      str := str.(1..(#str - 5))
--      WriteLine(str)$Lisp
      -- Here I split the string into lines and put each line in a "div".
      strlist := split(str, char string NewlineChar$Lisp)
      str := ""
--      WriteLine("formatMessages1")$Lisp
--      WriteLine(concat strlist)$Lisp
      for s in strlist repeat
--          WriteLine(s)$Lisp
          str := concat [str,"<div>",s,"</div>"]
      str

   getContentType(pathvar:String):String ==
--      WriteLine("getContentType begin")$Lisp
      -- set default content type
      contentType:String := "text/plain"
      -- need to test for successful match?
      StringMatch(".*\.(.*)$", pathvar)$Lisp
      u:UniversalSegment(Integer)
      u := segment(MatchBeginning(1)$Lisp+1,_
                   MatchEnd(1)$Lisp)$UniversalSegment(Integer)
      extension:String := pathvar.u
--      WriteLine$Lisp concat ["file extension: ",extension]
      -- test for extensions: html, htm, xml, xhtml, js, css
      if extension = "html" then
          contentType:String := "text/html"
      else if extension = "htm" then
          contentType:String := "text/html"
      else if extension = "xml" then
          contentType:String := "text/xml"
      else if extension = "xhtml" then
          contentType:String := "application/xhtml+xml"
      else if extension = "js" then
          contentType:String := "text/javascript"
      else if extension = "css" then
          contentType:String := "text/css"
      else if extension = "png" then
          contentType:String := "image/png"
      else if extension = "jpg" then
          contentType:String := "image/jpeg"
      else if extension = "jpeg" then
          contentType:String := "image/jpeg"
--      WriteLine$Lisp concat ["Content-Type: ",contentType]
--      WriteLine("getContentType end")$Lisp
      contentType
```

———-

— **AXSERV.dotabb** —

```
"AXSERV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AXSERV"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"AXSERV" -> "STRING"
```

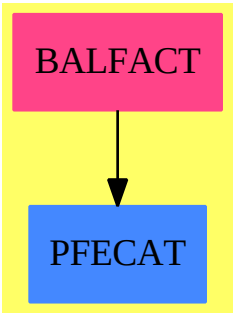
———-

Chapter 3

Chapter B

3.1 package BALFACT BalancedFactorisation

3.1.1 BalancedFactorisation (BALFACT)



Exports:
balancedFactorisation

— package BALFACT BalancedFactorisation —

```
)abbrev package BALFACT BalancedFactorisation
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 11 October 1991
++ Description:
++ This package provides balanced factorisations of polynomials.

BalancedFactorisation(R, UP): Exports == Implementation where
  R  : Join(GcdDomain, CharacteristicZero)
```

```
UP : UnivariatePolynomialCategory R

Exports ==> with
  balancedFactorisation: (UP, UP) -> Factored UP
    ++ balancedFactorisation(a, b) returns
    ++ a factorisation \spad{a = p1^e1 ... pm^em} such that each
    ++ \spad{pi} is balanced with respect to b.
  balancedFactorisation: (UP, List UP) -> Factored UP
    ++ balancedFactorisation(a, [b1,...,bn]) returns
    ++ a factorisation \spad{a = p1^e1 ... pm^em} such that each
    ++ pi is balanced with respect to \spad{[b1,...,bm]}.

Implementation ==> add
  balSqfr : (UP, Integer, List UP) -> Factored UP
  balSqfr1: (UP, Integer,      UP) -> Factored UP

  balancedFactorisation(a:UP, b:UP) == balancedFactorisation(a, [b])

  balSqfr1(a, n, b) ==
    g := gcd(a, b)
    fa := sqfrFactor((a exquo g)::UP, n)
    ground? g => fa
    fa * balSqfr1(g, n, (b exquo (g ** order(b, g)))::UP)

  balSqfr(a, n, l) ==
    b := first l
    empty? rest l => balSqfr1(a, n, b)
    */[balSqfr1(f.factor, n, b) for f in factors balSqfr(a,n,rest l)]

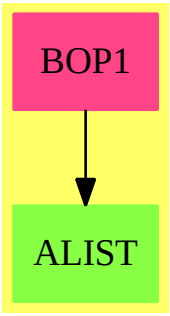
  balancedFactorisation(a:UP, l:List UP) ==
    empty?(ll := select(z1 +-> z1 ^= 0, l)) =>
      error "balancedFactorisation: 2nd argument is empty or all 0"
    sa := squareFree a
    unit(sa) * */[balSqfr(f.factor,f.exponent,ll) for f in factors sa])
```

— BALFACT.dotabb —

```
"BALFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BALFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BALFACT" -> "PFECAT"
```

3.2 package BOP1 BasicOperatorFunctions1

3.2.1 BasicOperatorFunctions1 (BOP1)



Exports:
constantOpIfCan constantOperator derivative evaluate

— package BOP1 BasicOperatorFunctions1 —

```
)abbrev package BOP1 BasicOperatorFunctions1
++ Author: Manuel Bronstein
++ Date Created: 28 Mar 1988
++ Date Last Updated: 15 May 1990
++ Keywords: operator.
++ Description:
++ This package exports functions to set some commonly used properties
++ of operators, including properties which contain functions.

BasicOperatorFunctions1(A:SetCategory): Exports == Implementation where
  OP   ==> BasicOperator
  EVAL    ==> "%eval"
  CONST   ==> "%constant"
  DIFF    ==> "%diff"
  OUT     ==> OutputForm
  IN      ==> InputForm

  Exports ==> with
    evaluate        : (OP, List A)      -> Union(A, "failed")
      ++ evaluate(op, [a1,...,an]) checks if op has an "%eval"
      ++ property f. If it has, then \spad{f(a1,...,an)} is returned, and
      ++ "failed" otherwise.
    evaluate        : (OP, List A -> A) -> OP
      ++ evaluate(op, foo) attaches foo as the "%eval" property
      ++ of op. If op has an "%eval" property f, then applying op
      ++ to \spad{(a1,...,an)} returns the result of \spad{f(a1,...,an)}.
    evaluate        : (OP, A -> A)      -> OP
```

```
    ++ evaluate(op, foo) attaches foo as the "%eval" property
    ++ of op. If op has an "%eval" property f, then applying op
    ++ to a returns the result of \spad{f(a)}. Argument op must be unary.
  evaluate        : OP                -> Union(List A -> A, "failed")
    ++ evaluate(op) returns the value of the "%eval" property of
    ++ op if it has one, and "failed" otherwise.
  derivative      : (OP, List (List A -> A)) -> OP
    ++ derivative(op, [foo1,...,foon]) attaches [foo1,...,foon] as
    ++ the "%diff" property of op. If op has an "%diff" property
    ++ \spad{[f1,...,fn]} then applying a derivation D
    ++ to \spad{op(a1,...,an)}
    ++ returns \spad{f1(a1,...,an) * D(a1) + ... + fn(a1,...,an) * D(an)}.
  derivative      : (OP, A -> A) -> OP
    ++ derivative(op, foo) attaches foo as the "%diff" property
    ++ of op. If op has an "%diff" property f, then applying a
    ++ derivation D to op(a) returns \spad{f(a) * D(a)}.
    ++ Argument op must be unary.
  derivative      : OP -> Union(List(List A -> A), "failed")
    ++ derivative(op) returns the value of the "%diff" property of
    ++ op if it has one, and "failed" otherwise.
  if A has OrderedSet then
    constantOperator: A -> OP
      ++ constantOperator(a) returns a nullary operator op
      ++ such that \spad{op()} always evaluate to \spad{a}.
    constantOpIfCan : OP -> Union(A, "failed")
      ++ constantOpIfCan(op) returns \spad{a} if op is the constant
      ++ nullary operator always returning \spad{a}, "failed" otherwise.

Implementation ==> add
  evaluate(op:OP, func:A -> A) ==
     evaluate(op, (ll:List(A)):A +-> func first ll)

  evaluate op ==
    (func := property(op, EVAL)) case "failed" => "failed"
    (func::None) pretend (List A -> A)

  evaluate(op:OP, args:List A) ==
    (func := property(op, EVAL)) case "failed" => "failed"
    ((func::None) pretend (List A -> A)) args

  evaluate(op:OP, func:List A -> A) ==
    setProperty(op, EVAL, func pretend None)

  derivative op ==
    (func := property(op, DIFF)) case "failed" => "failed"
    ((func::None) pretend List(List A -> A))

  derivative(op:OP, grad:List(List A -> A)) ==
    setProperty(op, DIFF, grad pretend None)
```

```
derivative(op:OP, f:A -> A) ==
  unary? op or nary? op =>
    derivative(op, [(ll:List(A)):A +-> f first ll]$List(List A -> A))
  error "Operator is not unary"

if A has OrderedSet then
  cdisp   : (OUT, List OUT) -> OUT
  csex    : (IN,  List IN) -> IN
  eqconst?: (OP, OP) -> Boolean
  ltconst?: (OP, OP) -> Boolean
  constOp : A -> OP

  opconst:OP :=
    comparison(equality(operator("constant"::Symbol, 0), eqconst?),
                                                          ltconst?)

  cdisp(a, l) == a
  csex(a, l)  == a

  eqconst?(a, b) ==
    (va := property(a, CONST)) case "failed" => not has?(b, CONST)
    ((vb := property(b, CONST)) case None) and
       ((va::None) pretend A) = ((vb::None) pretend A)

  ltconst?(a, b) ==
    (va := property(a, CONST)) case "failed" => has?(b, CONST)
    ((vb := property(b, CONST)) case None) and
       ((va::None) pretend A) < ((vb::None) pretend A)

  constOp a ==
    setProperty(
      display(copy opconst, (ll:List(OUT)):OUT +-> cdisp(a::OUT, ll)),
        CONST, a pretend None)

  constantOpIfCan op ==
    is?(op, "constant"::Symbol) and
      ((u := property(op, CONST)) case None) => (u::None) pretend A
    "failed"

  if A has ConvertibleTo IN then
    constantOperator a ==
      input(constOp a, (ll:List(IN)):IN +-> csex(convert a, ll))
  else
    constantOperator a == constOp a
```

———-

— **BOP1.dotabb** —

```
"BOP1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BOP1"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"BOP1" -> "ALIST"
```

3.3 package BEZIER Bezier

— Bezier.input —

```
)set break resume
)sys rm -f Bezier.output
)spool Bezier.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
n:=linearBezier([2.0,2.0],[4.0,4.0])
--R
--I   (1)  theMap(BEZIER;linearBezier;2LM;1!0,707)
--R                                                Type: (Float -> List(Float))
--E 1

--S 2 of 9
[n(t/10.0) for t in 0..10 by 1]
--R
--R   (2)
--R   [[2.0,2.0], [2.2,2.2], [2.4,2.4], [2.6,2.6], [2.8,2.8], [3.0,3.0],
--R    [3.2,3.2], [3.4,3.4], [3.6,3.6], [3.8,3.8], [4.0,4.0]]
--R                                                     Type: List(List(Float))
--E 2

--S 3 of 9
n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
--R
--I   (3)  theMap(BEZIER;quadraticBezier;3LM;2!0,291)
--R                                                Type: (Float -> List(Float))
--E 3

--S 4 of 9
[n(t/10.0) for t in 0..10 by 1]
--R
--R   (4)
--R   [[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0],
--R    [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]
--R                                                     Type: List(List(Float))
```

```
--E 4

--S 5 of 9
n:=cubicBezier([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])
--R
--I   (5)  theMap(BEZIER;cubicBezier;4LM;3!0,915)
--R                                                Type: (Float -> List(Float))
--E 5

--S 6 of 9
[n(t/10.0) for t in 0..10 by 1]
--R
--R   (6)
--R   [[2.0,2.0], [2.112,2.54], [2.416,2.96], [2.864,3.26], [3.408,3.44],
--R    [4.0,3.5], [4.592,3.44], [5.136,3.26], [5.584,2.96], [5.888,2.54],
--R    [6.0,2.0]]
--R                                                     Type: List(List(Float))
--E 6

--S 7 of 9
line:=[[i::Float,4.0] for i in -4..4 by 1]
--R
--R
--R   (7)
--R   [[- 4.0,4.0], [- 3.0,4.0], [- 2.0,4.0], [- 1.0,4.0], [0.0,4.0], [1.0,4.0],
--R    [2.0,4.0], [3.0,4.0], [4.0,4.0]]
--R                                                     Type: List(List(Float))
--E 7

--S 8 of 9
functions:=[quadraticBezier([2.0,2.0],m,[6.0,2.0]) for m in line]
--R
--R
--R   (8)
--R   [theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0),
--R    theMap(BEZIER;quadraticBezier;3LM;2!0,0)]
--R                                          Type: List((Float -> List(Float)))
--E 8

--S 9 of 9
graphs:=[[point(((functions.i)(j/100.0))::LIST(DFLOAT)) _
          for j in 0..100] for i in 1..9]
--R
```

```
--R
--R   (9)
--R   [
--R     [[2.,2.], [1.8815999999999999,2.0396000000000001],
--R      [1.7664,2.0783999999999998], [1.6544000000000003,2.1164000000000005],
--R      [1.5456000000000003,2.1536], [1.4399999999999999,2.1899999999999999],
--R      [1.3375999999999999,2.2256], [1.2383999999999999,2.2604000000000006],
--R      [1.1424000000000003,2.2944], [1.0496000000000003,2.3275999999999999],
--R      [0.95999999999999996,2.3599999999999999],
--R      [0.87360000000000015,2.3915999999999999],
--R      [0.79039999999999999,2.4224000000000006],
--R      [0.71040000000000014,2.4523999999999999],
--R      [0.63360000000000016,2.4815999999999998],
--R      [0.56000000000000005,2.5099999999999998],
--R      [0.48959999999999998,2.5375999999999999], [0.4224,2.5644],
--R      [0.3584,2.5903999999999998], [0.29759999999999998,2.6156000000000006],
--R      [0.23999999999999999,2.6400000000000006],
--R      [0.18559999999999999,2.6636000000000006],
--R      [0.13439999999999999,2.6863999999999999],
--R      [8.6400000000000005E-2,2.7084000000000001],
--R      [4.1599999999999998E-2,2.7296000000000005], [0.,2.75],
--R      [- 3.8400000000000004E-2,2.7696000000000005],
--R      [- 7.3599999999999999E-2,2.7884000000000002], [- 0.1056,2.8064],
--R      [- 0.13440000000000002,2.8235999999999999], [- 0.16,2.8399999999999999],
--R      [- 0.18240000000000001,2.8555999999999999],
--R      [- 0.2016,2.8704000000000001],
--R      [- 0.21760000000000002,2.8843999999999999],
--R      [- 0.23039999999999999,2.8976000000000006],
--R      [- 0.23999999999999999,2.9100000000000001],
--R      [- 0.24640000000000001,2.9216000000000006],
--R      [- 0.24959999999999999,2.9323999999999999],
--R      [- 0.24959999999999999,2.9424000000000001],
--R      [- 0.24640000000000001,2.9516], [- 0.23999999999999999,2.96],
--R      [- 0.23039999999999999,2.9676],
--R      [- 0.21760000000000002,2.9744000000000002],
--R      [- 0.2016,2.9803999999999999],
--R      [- 0.18240000000000001,2.9855999999999998], [- 0.16,2.9900000000000002],
--R      [- 0.13440000000000002,2.9935999999999998], [- 0.1056,2.9964],
--R      [- 7.3599999999999999E-2,2.9984000000000002],
--R      [- 3.8400000000000004E-2,2.9996], [0.,3.],
--R      [4.1599999999999998E-2,2.9996],
--R      [8.6400000000000005E-2,2.9984000000000002], [0.13439999999999999,2.9964],
--R      [0.18559999999999999,2.9935999999999998],
--R      [0.23999999999999999,2.9900000000000002],
--R      [0.29759999999999998,2.9855999999999998], [0.3584,2.9803999999999999],
--R      [0.4224,2.9744000000000002], [0.48959999999999998,2.9676],
--R      [0.56000000000000005,2.96], [0.63360000000000016,2.9516],
--R      [0.71040000000000014,2.9424000000000001],
--R      [0.79039999999999999,2.9323999999999999],
--R      [0.87360000000000015,2.9216000000000006],
```

```
--R      [0.95999999999999996,2.9100000000000001],
--R      [1.0496000000000003,2.8976000000000006],
--R      [1.1424000000000003,2.8843999999999999],
--R      [1.2383999999999999,2.8704000000000001],
--R      [1.3375999999999999,2.8555999999999999],
--R      [1.4399999999999999,2.8399999999999999],
--R      [1.5456000000000003,2.8235999999999999], [1.6544000000000003,2.8064],
--R      [1.7664,2.7884000000000002], [1.8815999999999999,2.7696000000000005],
--R      [2.,2.75], [2.1215999999999999,2.7296000000000005],
--R      [2.2464,2.7084000000000001], [2.3744000000000005,2.6863999999999999],
--R      [2.5055999999999998,2.6636000000000006],
--R      [2.6400000000000006,2.6400000000000006],
--R      [2.7776000000000005,2.6156000000000006],
--R      [2.9184000000000001,2.5903999999999998], [3.0623999999999998,2.5644],
--R      [3.2096,2.5375999999999999], [3.3599999999999999,2.5099999999999998],
--R      [3.5135999999999998,2.4815999999999998],
--R      [3.6703999999999999,2.4523999999999999], [3.8304,2.4224000000000006],
--R      [3.9935999999999998,2.3915999999999999],
--R      [4.1600000000000001,2.3599999999999999],
--R      [4.329600000000001,2.3275999999999999], [4.5023999999999997,2.2944],
--R      [4.6783999999999999,2.2604000000000006], [4.8575999999999997,2.2256],
--R      [5.0400000000000009,2.1899999999999999], [5.2256,2.1536],
--R      [5.4143999999999997,2.1164000000000005],
--R      [5.6063999999999998,2.0783999999999998],
--R      [5.8015999999999996,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [1.9014,2.0396000000000001],
--R      [1.8056000000000001,2.0783999999999998],
--R      [1.7125999999999999,2.1164000000000005], [1.6224000000000003,2.1536],
--R      [1.5349999999999999,2.1899999999999999], [1.4503999999999999,2.2256],
--R      [1.3686000000000003,2.2604000000000006], [1.2896000000000001,2.2944],
--R      [1.2134,2.3275999999999999], [1.1399999999999999,2.3599999999999999],
--R      [1.0693999999999999,2.3915999999999999],
--R      [1.0016000000000003,2.4224000000000006],
--R      [0.93659999999999999,2.4523999999999999],
--R      [0.87439999999999996,2.4815999999999998],
--R      [0.81499999999999995,2.5099999999999998],
--R      [0.75839999999999996,2.5375999999999999], [0.70460000000000012,2.5644],
--R      [0.65359999999999996,2.5903999999999998],
--R      [0.60540000000000016,2.6156000000000006],
--R      [0.56000000000000005,2.6400000000000006],
--R      [0.51739999999999997,2.6636000000000006],
--R      [0.47760000000000002,2.6863999999999999],
--R      [0.44059999999999999,2.7084000000000001],
--R      [0.40639999999999998,2.7296000000000005], [0.375,2.75],
--R      [0.34639999999999999,2.7696000000000005], [0.3206,2.7884000000000002],
--R      [0.29759999999999998,2.8064], [0.27739999999999998,2.8235999999999999],
--R      [0.26000000000000001,2.8399999999999999],
--R      [0.24540000000000001,2.8555999999999999],
```

```
--R      [0.23360000000000003,2.8704000000000001],
--R      [0.22459999999999999,2.8843999999999999],
--R      [0.21840000000000004,2.8976000000000006], [0.215,2.9100000000000001],
--R      [0.21440000000000003,2.9216000000000006],
--R      [0.21659999999999999,2.9323999999999999],
--R      [0.22159999999999999,2.9424000000000001], [0.22939999999999999,2.9516],
--R      [0.23999999999999999,2.96], [0.25340000000000007,2.9676],
--R      [0.26960000000000006,2.9744000000000002],
--R      [0.28860000000000008,2.9803999999999999],
--R      [0.31040000000000001,2.9855999999999998],
--R      [0.33500000000000008,2.9900000000000002], [0.3624,2.9935999999999998],
--R      [0.39260000000000006,2.9964], [0.42559999999999998,2.9984000000000002],
--R      [0.46139999999999998,2.9996], [0.5,3.], [0.54139999999999999,2.9996],
--R      [0.58560000000000012,2.9984000000000002], [0.63260000000000005,2.9964],
--R      [0.68240000000000012,2.9935999999999998],
--R      [0.73499999999999999,2.9900000000000002],
--R      [0.79039999999999999,2.9855999999999998],
--R      [0.84860000000000002,2.9803999999999999],
--R      [0.90959999999999996,2.9744000000000002], [0.97340000000000004,2.9676],
--R      [1.04,2.96], [1.1093999999999999,2.9516], [1.1816,2.9424000000000001],
--R      [1.2565999999999999,2.9323999999999999], [1.3344,2.9216000000000006],
--R      [1.415,2.9100000000000001], [1.4984,2.8976000000000006],
--R      [1.5846,2.8843999999999999], [1.6736,2.8704000000000001],
--R      [1.7654000000000001,2.8555999999999999],
--R      [1.8600000000000003,2.8399999999999999],
--R      [1.9574000000000003,2.8235999999999999], [2.0575999999999999,2.8064],
--R      [2.1606000000000005,2.7884000000000002], [2.2664,2.7696000000000005],
--R      [2.375,2.75], [2.4864000000000006,2.7296000000000005],
--R      [2.6006,2.7084000000000001], [2.7176,2.6863999999999999],
--R      [2.8374000000000006,2.6636000000000006], [2.96,2.6400000000000006],
--R      [3.0853999999999999,2.6156000000000006],
--R      [3.2136000000000005,2.5903999999999998], [3.3445999999999998,2.5644],
--R      [3.4784000000000006,2.5375999999999999],
--R      [3.6150000000000002,2.5099999999999998], [3.7544,2.4815999999999998],
--R      [3.8965999999999998,2.4523999999999999],
--R      [4.0415999999999999,2.4224000000000006],
--R      [4.1894000000000009,2.3915999999999999],
--R      [4.3399999999999999,2.3599999999999999],
--R      [4.4934000000000012,2.3275999999999999], [4.6496000000000013,2.2944],
--R      [4.8086000000000002,2.2604000000000006], [4.9703999999999997,2.2256],
--R      [5.1349999999999998,2.1899999999999999], [5.3023999999999996,2.1536],
--R      [5.4725999999999999,2.1164000000000005], [5.6456,2.0783999999999998],
--R      [5.8213999999999997,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [1.9212000000000002,2.0396000000000001],
--R      [1.8448,2.0783999999999998], [1.7707999999999999,2.1164000000000005],
--R      [1.6992000000000003,2.1536], [1.6299999999999999,2.1899999999999999],
--R      [1.5631999999999999,2.2256], [1.4987999999999999,2.2604000000000006],
--R      [1.4368000000000003,2.2944], [1.3772,2.3275999999999999],
```

```
--R      [1.3200000000000003,2.3599999999999999],
--R      [1.2652000000000001,2.3915999999999999],
--R      [1.2128000000000001,2.4224000000000006],
--R      [1.1628000000000003,2.4523999999999999], [1.1152,2.4815999999999998],
--R      [1.0700000000000003,2.5099999999999998],
--R      [1.0271999999999999,2.5375999999999999], [0.98680000000000012,2.5644],
--R      [0.94879999999999998,2.5903999999999998],
--R      [0.91320000000000001,2.6156000000000006],
--R      [0.88000000000000012,2.6400000000000006],
--R      [0.84919999999999995,2.6636000000000006],
--R      [0.82079999999999997,2.6863999999999999],
--R      [0.79479999999999995,2.7084000000000001], [0.7712,2.7296000000000005],
--R      [0.75,2.75], [0.73119999999999996,2.7696000000000005],
--R      [0.71479999999999999,2.7884000000000002], [0.70079999999999998,2.8064],
--R      [0.68920000000000003,2.8235999999999999],
--R      [0.68000000000000016,2.8399999999999999],
--R      [0.67320000000000002,2.8555999999999999],
--R      [0.66879999999999995,2.8704000000000001],
--R      [0.66679999999999995,2.8843999999999999],
--R      [0.66720000000000002,2.8976000000000006],
--R      [0.67000000000000015,2.9100000000000001],
--R      [0.67520000000000002,2.9216000000000006],
--R      [0.68279999999999996,2.9323999999999999],
--R      [0.69279999999999997,2.9424000000000001], [0.70520000000000005,2.9516],
--R      [0.71999999999999997,2.96], [0.73719999999999997,2.9676],
--R      [0.75680000000000014,2.9744000000000002],
--R      [0.77880000000000016,2.9803999999999999],
--R      [0.80320000000000014,2.9855999999999998],
--R      [0.82999999999999996,2.9900000000000002],
--R      [0.85919999999999996,2.9935999999999998], [0.89080000000000004,2.9964],
--R      [0.92479999999999996,2.9984000000000002], [0.96120000000000005,2.9996],
--R      [1.,3.], [1.0411999999999999,2.9996], [1.0848,2.9984000000000002],
--R      [1.1308000000000002,2.9964], [1.1792000000000002,2.9935999999999998],
--R      [1.23,2.9900000000000002], [1.2831999999999999,2.9855999999999998],
--R      [1.3388,2.9803999999999999], [1.3968000000000003,2.9744000000000002],
--R      [1.4572000000000003,2.9676], [1.52,2.96], [1.5851999999999999,2.9516],
--R      [1.6528,2.9424000000000001], [1.7228000000000003,2.9323999999999999],
--R      [1.7951999999999999,2.9216000000000006],
--R      [1.8700000000000001,2.9100000000000001], [1.9472,2.8976000000000006],
--R      [2.0268000000000006,2.8843999999999999],
--R      [2.1088000000000005,2.8704000000000001], [2.1932,2.8555999999999999],
--R      [2.2799999999999998,2.8399999999999999],
--R      [2.3692000000000002,2.8235999999999999], [2.4607999999999999,2.8064],
--R      [2.5548000000000002,2.7884000000000002],
--R      [2.6511999999999998,2.7696000000000005], [2.75,2.75],
--R      [2.8512,2.7296000000000005], [2.9548000000000005,2.7084000000000001],
--R      [3.0608,2.6863999999999999], [3.1692,2.6636000000000006],
--R      [3.2799999999999998,2.6400000000000006],
--R      [3.3932000000000002,2.6156000000000006],
--R      [3.5087999999999999,2.5903999999999998], [3.6267999999999998,2.5644],
```

```
--R      [3.7471999999999999,2.5375999999999999],
--R      [3.8700000000000001,2.5099999999999998],
--R      [3.9952000000000005,2.4815999999999998],
--R      [4.1227999999999998,2.4523999999999999],
--R      [4.2527999999999997,2.4224000000000006],
--R      [4.3852000000000011,2.3915999999999999],
--R      [4.5199999999999996,2.3599999999999999],
--R      [4.6571999999999996,2.3275999999999999], [4.7968000000000011,2.2944],
--R      [4.9387999999999996,2.2604000000000006], [5.0831999999999997,2.2256],
--R      [5.2300000000000004,2.1899999999999999], [5.3792,2.1536],
--R      [5.530800000000001,2.1164000000000005],
--R      [5.684800000000001,2.0783999999999998],
--R      [5.8411999999999997,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [1.9410000000000003,2.0396000000000001],
--R      [1.8839999999999999,2.0783999999999998], [1.829,2.1164000000000005],
--R      [1.7760000000000002,2.1536], [1.7250000000000001,2.1899999999999999],
--R      [1.6759999999999999,2.2256], [1.629,2.2604000000000006],
--R      [1.5840000000000001,2.2944], [1.5409999999999999,2.3275999999999999],
--R      [1.5,2.3599999999999999], [1.4610000000000003,2.3915999999999999],
--R      [1.4239999999999999,2.4224000000000006],
--R      [1.3890000000000002,2.4523999999999999],
--R      [1.3560000000000003,2.4815999999999998], [1.325,2.5099999999999998],
--R      [1.2960000000000003,2.5375999999999999], [1.2689999999999999,2.5644],
--R      [1.244,2.5903999999999998], [1.2210000000000001,2.6156000000000006],
--R      [1.2,2.6400000000000006], [1.181,2.6636000000000006],
--R      [1.1639999999999999,2.6863999999999999], [1.149,2.7084000000000001],
--R      [1.1359999999999999,2.7296000000000005], [1.125,2.75],
--R      [1.1160000000000001,2.7696000000000005], [1.109,2.7884000000000002],
--R      [1.1040000000000001,2.8064], [1.101,2.8235999999999999],
--R      [1.1000000000000001,2.8399999999999999], [1.101,2.8555999999999999],
--R      [1.1040000000000001,2.8704000000000001], [1.109,2.8843999999999999],
--R      [1.1160000000000001,2.8976000000000006], [1.125,2.9100000000000001],
--R      [1.1359999999999999,2.9216000000000006], [1.149,2.9323999999999999],
--R      [1.1639999999999999,2.9424000000000001], [1.181,2.9516], [1.2,2.96],
--R      [1.2210000000000001,2.9676], [1.244,2.9744000000000002],
--R      [1.2689999999999999,2.9803999999999999],
--R      [1.2960000000000003,2.9855999999999998], [1.325,2.9900000000000002],
--R      [1.3560000000000003,2.9935999999999998], [1.3890000000000002,2.9964],
--R      [1.4239999999999999,2.9984000000000002], [1.4610000000000003,2.9996],
--R      [1.5,3.], [1.5409999999999999,2.9996],
--R      [1.5840000000000001,2.9984000000000002], [1.629,2.9964],
--R      [1.6759999999999999,2.9935999999999998],
--R      [1.7250000000000001,2.9900000000000002],
--R      [1.7760000000000002,2.9855999999999998], [1.829,2.9803999999999999],
--R      [1.8839999999999999,2.9744000000000002], [1.9410000000000003,2.9676],
--R      [2.,2.96], [2.0609999999999999,2.9516],
--R      [2.1240000000000006,2.9424000000000001],
--R      [2.1890000000000001,2.9323999999999999],
```

```
--R      [2.2559999999999998,2.9216000000000006],
--R      [2.3250000000000002,2.9100000000000001],
--R      [2.3959999999999999,2.8976000000000006],
--R      [2.4689999999999999,2.8843999999999999],
--R      [2.5440000000000005,2.8704000000000001], [2.621,2.8555999999999999],
--R      [2.7000000000000002,2.8399999999999999],
--R      [2.7810000000000006,2.8235999999999999], [2.8639999999999999,2.8064],
--R      [2.9489999999999998,2.7884000000000002],
--R      [3.0360000000000005,2.7696000000000005], [3.125,2.75],
--R      [3.2160000000000002,2.7296000000000005],
--R      [3.3090000000000002,2.7084000000000001],
--R      [3.4039999999999999,2.6863999999999999],
--R      [3.5009999999999999,2.6636000000000006],
--R      [3.6000000000000005,2.6400000000000006],
--R      [3.7010000000000005,2.6156000000000006],
--R      [3.8039999999999998,2.5903999999999998], [3.9089999999999998,2.5644],
--R      [4.016,2.5375999999999999], [4.125,2.5099999999999998],
--R      [4.2359999999999998,2.4815999999999998],
--R      [4.3490000000000002,2.4523999999999999],
--R      [4.4640000000000004,2.4224000000000006],
--R      [4.5810000000000013,2.3915999999999999],
--R      [4.7000000000000011,2.3599999999999999],
--R      [4.8209999999999997,2.3275999999999999], [4.944,2.2944],
--R      [5.069,2.2604000000000006], [5.1959999999999997,2.2256],
--R      [5.3250000000000011,2.1899999999999999], [5.4560000000000013,2.1536],
--R      [5.5890000000000004,2.1164000000000005],
--R      [5.7240000000000002,2.0783999999999998],
--R      [5.8609999999999998,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [1.9608000000000003,2.0396000000000001],
--R      [1.9232,2.0783999999999998], [1.8872,2.1164000000000005],
--R      [1.8528000000000002,2.1536], [1.8200000000000003,2.1899999999999999],
--R      [1.7887999999999999,2.2256], [1.7592000000000003,2.2604000000000006],
--R      [1.7312000000000003,2.2944], [1.7048000000000001,2.3275999999999999],
--R      [1.6799999999999999,2.3599999999999999],
--R      [1.6568000000000001,2.3915999999999999], [1.6352,2.4224000000000006],
--R      [1.6152,2.4523999999999999], [1.5968,2.4815999999999998],
--R      [1.5800000000000001,2.5099999999999998], [1.5648,2.5375999999999999],
--R      [1.5511999999999999,2.5644], [1.5391999999999999,2.5903999999999998],
--R      [1.5287999999999999,2.6156000000000006], [1.52,2.6400000000000006],
--R      [1.5127999999999999,2.6636000000000006],
--R      [1.5072000000000001,2.6863999999999999],
--R      [1.5032000000000001,2.7084000000000001],
--R      [1.5007999999999999,2.7296000000000005], [1.5,2.75],
--R      [1.5007999999999999,2.7696000000000005],
--R      [1.5032000000000001,2.7884000000000002], [1.5072000000000001,2.8064],
--R      [1.5127999999999999,2.8235999999999999], [1.52,2.8399999999999999],
--R      [1.5287999999999999,2.8555999999999999],
--R      [1.5391999999999999,2.8704000000000001],
```

```
--R      [1.5511999999999999,2.8843999999999999], [1.5648,2.8976000000000006],
--R      [1.5800000000000001,2.9100000000000001], [1.5968,2.9216000000000006],
--R      [1.6152,2.9323999999999999], [1.6352,2.9424000000000001],
--R      [1.6568000000000001,2.9516], [1.6799999999999999,2.96],
--R      [1.7048000000000001,2.9676], [1.7312000000000003,2.9744000000000002],
--R      [1.7592000000000003,2.9803999999999999],
--R      [1.7887999999999999,2.9855999999999998],
--R      [1.8200000000000003,2.9900000000000002],
--R      [1.8528000000000002,2.9935999999999998], [1.8872,2.9964],
--R      [1.9232,2.9984000000000002], [1.9608000000000003,2.9996], [2.,3.],
--R      [2.0407999999999999,2.9996], [2.0832000000000006,2.9984000000000002],
--R      [2.1272000000000002,2.9964], [2.1728000000000005,2.9935999999999998],
--R      [2.2200000000000006,2.9900000000000002],
--R      [2.2688000000000006,2.9855999999999998],
--R      [2.3191999999999999,2.9803999999999999], [2.3712,2.9744000000000002],
--R      [2.4247999999999998,2.9676], [2.48,2.96], [2.5367999999999999,2.9516],
--R      [2.5952000000000002,2.9424000000000001],
--R      [2.6551999999999998,2.9323999999999999],
--R      [2.7168000000000001,2.9216000000000006],
--R      [2.7799999999999998,2.9100000000000001],
--R      [2.8448000000000002,2.8976000000000006], [2.9112,2.8843999999999999],
--R      [2.9792000000000005,2.8704000000000001], [3.0488,2.8555999999999999],
--R      [3.1200000000000001,2.8399999999999999],
--R      [3.1928000000000001,2.8235999999999999], [3.2671999999999999,2.8064],
--R      [3.3431999999999999,2.7884000000000002],
--R      [3.4207999999999998,2.7696000000000005], [3.5,2.75],
--R      [3.5808,2.7296000000000005], [3.6631999999999998,2.7084000000000001],
--R      [3.7471999999999999,2.6863999999999999],
--R      [3.8328000000000007,2.6636000000000006],
--R      [3.9199999999999999,2.6400000000000006],
--R      [4.0087999999999999,2.6156000000000006],
--R      [4.0991999999999997,2.5903999999999998], [4.1912000000000003,2.5644],
--R      [4.2847999999999997,2.5375999999999999],
--R      [4.3799999999999999,2.5099999999999998],
--R      [4.4767999999999999,2.4815999999999998],
--R      [4.5751999999999997,2.4523999999999999],
--R      [4.6752000000000002,2.4224000000000006],
--R      [4.7767999999999997,2.3915999999999999],
--R      [4.8799999999999999,2.3599999999999999],
--R      [4.9847999999999999,2.3275999999999999], [5.0911999999999997,2.2944],
--R      [5.1992000000000012,2.2604000000000006], [5.3087999999999997,2.2256],
--R      [5.4199999999999999,2.1899999999999999], [5.5327999999999999,2.1536],
--R      [5.6471999999999998,2.1164000000000005],
--R      [5.7632000000000012,2.0783999999999998],
--R      [5.8807999999999998,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [1.9805999999999999,2.0396000000000001],
--R      [1.9623999999999999,2.0783999999999998],
--R      [1.9454000000000002,2.1164000000000005], [1.9296,2.1536],
```

```
--R      [1.915,2.1899999999999999], [1.9016,2.2256], [1.8894,2.2604000000000006],
--R      [1.8784000000000001,2.2944], [1.8686000000000003,2.3275999999999999],
--R      [1.8600000000000003,2.3599999999999999],
--R      [1.8526000000000002,2.3915999999999999], [1.8464,2.4224000000000006],
--R      [1.8413999999999999,2.4523999999999999],
--R      [1.8375999999999999,2.4815999999999998], [1.835,2.5099999999999998],
--R      [1.8335999999999999,2.5375999999999999], [1.8333999999999999,2.5644],
--R      [1.8344,2.5903999999999998], [1.8366000000000002,2.6156000000000006],
--R      [1.8400000000000003,2.6400000000000006],
--R      [1.8446000000000002,2.6636000000000006], [1.8504,2.6863999999999999],
--R      [1.8573999999999999,2.7084000000000001],
--R      [1.8655999999999999,2.7296000000000005], [1.875,2.75],
--R      [1.8855999999999999,2.7696000000000005], [1.8974,2.7884000000000002],
--R      [1.9104000000000001,2.8064], [1.9246000000000003,2.8235999999999999],
--R      [1.9399999999999999,2.8399999999999999],
--R      [1.9565999999999999,2.8555999999999999],
--R      [1.9743999999999999,2.8704000000000001],
--R      [1.9934000000000003,2.8843999999999999],
--R      [2.0135999999999998,2.8976000000000006],
--R      [2.0350000000000001,2.9100000000000001],
--R      [2.0575999999999999,2.9216000000000006],
--R      [2.0813999999999999,2.9323999999999999],
--R      [2.1063999999999998,2.9424000000000001], [2.1326000000000001,2.9516],
--R      [2.1600000000000001,2.96], [2.1886000000000001,2.9676],
--R      [2.2183999999999999,2.9744000000000002],
--R      [2.2494000000000005,2.9803999999999999],
--R      [2.2816000000000001,2.9855999999999998],
--R      [2.3149999999999999,2.9900000000000002],
--R      [2.3496000000000006,2.9935999999999998], [2.3854000000000006,2.9964],
--R      [2.4224000000000006,2.9984000000000002], [2.4605999999999999,2.9996],
--R      [2.5,3.], [2.5406,2.9996], [2.5823999999999998,2.9984000000000002],
--R      [2.6254,2.9964], [2.6696,2.9935999999999998],
--R      [2.7149999999999999,2.9900000000000002],
--R      [2.7616000000000005,2.9855999999999998],
--R      [2.8094000000000001,2.9803999999999999],
--R      [2.8584000000000005,2.9744000000000002], [2.9085999999999999,2.9676],
--R      [2.96,2.96], [3.0125999999999999,2.9516],
--R      [3.0663999999999998,2.9424000000000001], [3.1214,2.9323999999999999],
--R      [3.1776,2.9216000000000006], [3.2349999999999999,2.9100000000000001],
--R      [3.2936000000000005,2.8976000000000006],
--R      [3.3534000000000006,2.8843999999999999],
--R      [3.4144000000000005,2.8704000000000001],
--R      [3.4765999999999999,2.8555999999999999], [3.54,2.8399999999999999],
--R      [3.6046000000000005,2.8235999999999999], [3.6703999999999999,2.8064],
--R      [3.7374000000000001,2.7884000000000002],
--R      [3.8056000000000001,2.7696000000000005], [3.875,2.75],
--R      [3.9456000000000007,2.7296000000000005],
--R      [4.0174000000000003,2.7084000000000001],
--R      [4.0903999999999998,2.6863999999999999],
--R      [4.1646000000000001,2.6636000000000006],
```

```
--R      [4.2400000000000002,2.6400000000000006],
--R      [4.3166000000000011,2.6156000000000006],
--R      [4.394400000000001,2.5903999999999998], [4.4733999999999998,2.5644],
--R      [4.5536000000000012,2.5375999999999999],
--R      [4.6349999999999998,2.5099999999999998],
--R      [4.7176000000000009,2.4815999999999998],
--R      [4.801400000000001,2.4523999999999999],
--R      [4.8864000000000001,2.4224000000000006],
--R      [4.9725999999999999,2.3915999999999999],
--R      [5.0599999999999996,2.3599999999999999],
--R      [5.1486000000000001,2.3275999999999999], [5.2384000000000004,2.2944],
--R      [5.3293999999999997,2.2604000000000006], [5.4215999999999998,2.2256],
--R      [5.5149999999999997,2.1899999999999999], [5.6096000000000004,2.1536],
--R      [5.7054000000000009,2.1164000000000005],
--R      [5.8023999999999996,2.0783999999999998],
--R      [5.9005999999999998,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [2.0004,2.0396000000000001],
--R      [2.0015999999999998,2.0783999999999998],
--R      [2.0036000000000005,2.1164000000000005], [2.0064000000000002,2.1536],
--R      [2.0099999999999998,2.1899999999999999], [2.0144000000000002,2.2256],
--R      [2.0196000000000005,2.2604000000000006], [2.0255999999999998,2.2944],
--R      [2.0324,2.3275999999999999], [2.04,2.3599999999999999],
--R      [2.0484,2.3915999999999999], [2.0575999999999999,2.4224000000000006],
--R      [2.0676000000000005,2.4523999999999999],
--R      [2.0783999999999998,2.4815999999999998],
--R      [2.0899999999999999,2.5099999999999998],
--R      [2.1023999999999998,2.5375999999999999], [2.1156000000000006,2.5644],
--R      [2.1295999999999999,2.5903999999999998],
--R      [2.1444000000000001,2.6156000000000006],
--R      [2.1600000000000001,2.6400000000000006],
--R      [2.1764000000000001,2.6636000000000006], [2.1936,2.6863999999999999],
--R      [2.2115999999999998,2.7084000000000001],
--R      [2.2303999999999999,2.7296000000000005], [2.25,2.75],
--R      [2.2704,2.7696000000000005], [2.2915999999999999,2.7884000000000002],
--R      [2.3136000000000001,2.8064], [2.3363999999999998,2.8235999999999999],
--R      [2.3599999999999999,2.8399999999999999],
--R      [2.3843999999999999,2.8555999999999999],
--R      [2.4096000000000002,2.8704000000000001], [2.4356,2.8843999999999999],
--R      [2.4624000000000006,2.8976000000000006],
--R      [2.4900000000000002,2.9100000000000001],
--R      [2.5184000000000006,2.9216000000000006],
--R      [2.5476000000000001,2.9323999999999999],
--R      [2.5775999999999999,2.9424000000000001], [2.6084000000000005,2.9516],
--R      [2.6400000000000006,2.96], [2.6724000000000006,2.9676],
--R      [2.7056000000000004,2.9744000000000002],
--R      [2.7395999999999998,2.9803999999999999], [2.7744,2.9855999999999998],
--R      [2.8100000000000005,2.9900000000000002], [2.8464,2.9935999999999998],
--R      [2.8835999999999999,2.9964], [2.9216000000000006,2.9984000000000002],
```

```
--R      [2.9603999999999999,2.9996], [3.,3.], [3.0404,2.9996],
--R      [3.0815999999999999,2.9984000000000002], [3.1236000000000006,2.9964],
--R      [3.1663999999999999,2.9935999999999998], [3.21,2.9900000000000002],
--R      [3.2544,2.9855999999999998], [3.2995999999999999,2.9803999999999999],
--R      [3.3456000000000001,2.9744000000000002], [3.3923999999999999,2.9676],
--R      [3.4399999999999999,2.96], [3.4883999999999999,2.9516],
--R      [3.5375999999999999,2.9424000000000001],
--R      [3.5876000000000001,2.9323999999999999],
--R      [3.6383999999999999,2.9216000000000006],
--R      [3.6899999999999999,2.9100000000000001],
--R      [3.7423999999999999,2.8976000000000006],
--R      [3.7955999999999999,2.8843999999999999],
--R      [3.8496000000000006,2.8704000000000001],
--R      [3.9043999999999999,2.8555999999999999], [3.96,2.8399999999999999],
--R      [4.0164,2.8235999999999999], [4.0735999999999999,2.8064],
--R      [4.1315999999999997,2.7884000000000002],
--R      [4.1904000000000003,2.7696000000000005], [4.25,2.75],
--R      [4.3103999999999996,2.7296000000000005],
--R      [4.3715999999999999,2.7084000000000001],
--R      [4.4336000000000002,2.6863999999999999],
--R      [4.4964000000000013,2.6636000000000006],
--R      [4.5599999999999996,2.6400000000000006],
--R      [4.6243999999999996,2.6156000000000006],
--R      [4.6896000000000004,2.5903999999999998], [4.7556000000000012,2.5644],
--R      [4.8224,2.5375999999999999], [4.8899999999999997,2.5099999999999998],
--R      [4.958400000000001,2.4815999999999998],
--R      [5.0275999999999996,2.4523999999999999],
--R      [5.0975999999999999,2.4224000000000006],
--R      [5.1684000000000001,2.3915999999999999],
--R      [5.2400000000000002,2.3599999999999999],
--R      [5.3124000000000002,2.3275999999999999], [5.3856000000000002,2.2944],
--R      [5.4596,2.2604000000000006], [5.5343999999999998,2.2256],
--R      [5.6100000000000012,2.1899999999999999], [5.6863999999999999,2.1536],
--R      [5.7636000000000003,2.1164000000000005],
--R      [5.8415999999999997,2.0783999999999998],
--R      [5.9203999999999999,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [2.0202,2.0396000000000001],
--R      [2.0407999999999999,2.0783999999999998],
--R      [2.0617999999999999,2.1164000000000005], [2.0832000000000006,2.1536],
--R      [2.105,2.1899999999999999], [2.1272000000000002,2.2256],
--R      [2.1497999999999999,2.2604000000000006], [2.1728000000000005,2.2944],
--R      [2.1962000000000002,2.3275999999999999],
--R      [2.2200000000000006,2.3599999999999999],
--R      [2.2442000000000002,2.3915999999999999],
--R      [2.2688000000000006,2.4224000000000006],
--R      [2.2938000000000001,2.4523999999999999],
--R      [2.3191999999999999,2.4815999999999998],
--R      [2.3450000000000006,2.5099999999999998], [2.3712,2.5375999999999999],
```

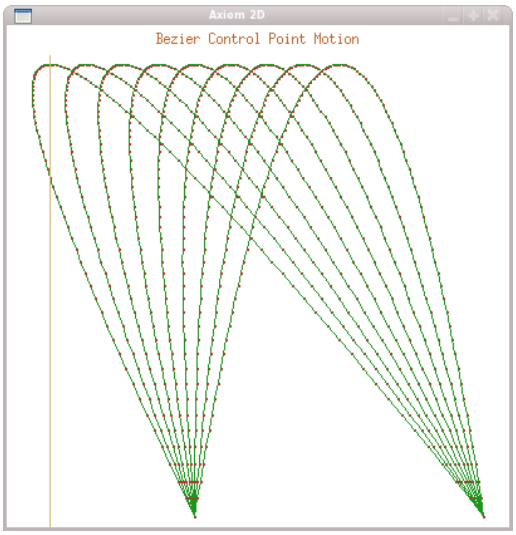
```
--R      [2.3978000000000002,2.5644], [2.4247999999999998,2.5903999999999998],
--R      [2.4521999999999999,2.6156000000000006], [2.48,2.6400000000000006],
--R      [2.5082,2.6636000000000006], [2.5367999999999999,2.6863999999999999],
--R      [2.5657999999999999,2.7084000000000001],
--R      [2.5952000000000002,2.7296000000000005], [2.625,2.75],
--R      [2.6551999999999998,2.7696000000000005], [2.6858,2.7884000000000002],
--R      [2.7168000000000001,2.8064], [2.7482000000000006,2.8235999999999999],
--R      [2.7799999999999998,2.8399999999999999],
--R      [2.8121999999999998,2.8555999999999999],
--R      [2.8448000000000002,2.8704000000000001],
--R      [2.8778000000000006,2.8843999999999999], [2.9112,2.8976000000000006],
--R      [2.9449999999999998,2.9100000000000001],
--R      [2.9792000000000005,2.9216000000000006],
--R      [3.0137999999999998,2.9323999999999999], [3.0488,2.9424000000000001],
--R      [3.0842000000000001,2.9516], [3.1200000000000001,2.96],
--R      [3.1562000000000001,2.9676], [3.1928000000000001,2.9744000000000002],
--R      [3.2298,2.9803999999999999], [3.2671999999999999,2.9855999999999998],
--R      [3.3050000000000006,2.9900000000000002],
--R      [3.3431999999999999,2.9935999999999998], [3.3818000000000001,2.9964],
--R      [3.4207999999999998,2.9984000000000002], [3.4601999999999999,2.9996],
--R      [3.5,3.], [3.5402000000000005,2.9996], [3.5808,2.9984000000000002],
--R      [3.6217999999999999,2.9964], [3.6631999999999998,2.9935999999999998],
--R      [3.7050000000000001,2.9900000000000002],
--R      [3.7471999999999999,2.9855999999999998],
--R      [3.7898000000000005,2.9803999999999999],
--R      [3.8328000000000007,2.9744000000000002], [3.8761999999999999,2.9676],
--R      [3.9199999999999999,2.96], [3.9641999999999999,2.9516],
--R      [4.0087999999999999,2.9424000000000001],
--R      [4.0537999999999998,2.9323999999999999],
--R      [4.0991999999999997,2.9216000000000006],
--R      [4.1449999999999996,2.9100000000000001],
--R      [4.1912000000000003,2.8976000000000006], [4.2378,2.8843999999999999],
--R      [4.2847999999999997,2.8704000000000001],
--R      [4.3322000000000003,2.8555999999999999],
--R      [4.3799999999999999,2.8399999999999999],
--R      [4.4282000000000004,2.8235999999999999], [4.4767999999999999,2.8064],
--R      [4.5258000000000003,2.7884000000000002],
--R      [4.5751999999999997,2.7696000000000005], [4.625,2.75],
--R      [4.6752000000000002,2.7296000000000005],
--R      [4.7257999999999996,2.7084000000000001],
--R      [4.7767999999999997,2.6863999999999999],
--R      [4.8281999999999998,2.6636000000000006],
--R      [4.8799999999999999,2.6400000000000006],
--R      [4.9321999999999999,2.6156000000000006],
--R      [4.9847999999999999,2.5903999999999998], [5.0377999999999998,2.5644],
--R      [5.0911999999999997,2.5375999999999999],
--R      [5.1449999999999996,2.5099999999999998],
--R      [5.1992000000000012,2.4815999999999998], [5.2538,2.4523999999999999],
--R      [5.3087999999999997,2.4224000000000006],
--R      [5.3642000000000003,2.3915999999999999],
```

```
--R      [5.4199999999999999,2.3599999999999999],
--R      [5.4762000000000004,2.3275999999999999], [5.5327999999999999,2.2944],
--R      [5.5898000000000003,2.2604000000000006], [5.6471999999999998,2.2256],
--R      [5.7050000000000001,2.1899999999999999], [5.7632000000000012,2.1536],
--R      [5.8217999999999996,2.1164000000000005],
--R      [5.8807999999999998,2.0783999999999998],
--R      [5.9401999999999999,2.0396000000000001], [6.,2.]]
--R     ,
--R
--R     [[2.,2.], [2.04,2.0396000000000001],
--R      [2.0800000000000001,2.0783999999999998],
--R      [2.1200000000000001,2.1164000000000005], [2.1600000000000001,2.1536],
--R      [2.2000000000000002,2.1899999999999999], [2.2400000000000002,2.2256],
--R      [2.2799999999999998,2.2604000000000006], [2.3199999999999998,2.2944],
--R      [2.3599999999999999,2.3275999999999999],
--R      [2.3999999999999999,2.3599999999999999],
--R      [2.4399999999999999,2.3915999999999999], [2.48,2.4224000000000006],
--R      [2.5200000000000005,2.4523999999999999],
--R      [2.5600000000000005,2.4815999999999998],
--R      [2.6000000000000005,2.5099999999999998],
--R      [2.6400000000000006,2.5375999999999999], [2.6800000000000006,2.5644],
--R      [2.7200000000000006,2.5903999999999998],
--R      [2.7599999999999998,2.6156000000000006],
--R      [2.7999999999999998,2.6400000000000006],
--R      [2.8399999999999999,2.6636000000000006],
--R      [2.8799999999999999,2.6863999999999999],
--R      [2.9199999999999999,2.7084000000000001], [2.96,2.7296000000000005],
--R      [3.,2.75], [3.04,2.7696000000000005],
--R      [3.0800000000000001,2.7884000000000002], [3.1200000000000001,2.8064],
--R      [3.1600000000000001,2.8235999999999999],
--R      [3.2000000000000002,2.8399999999999999],
--R      [3.2400000000000002,2.8555999999999999],
--R      [3.2799999999999998,2.8704000000000001],
--R      [3.3199999999999998,2.8843999999999999],
--R      [3.3599999999999999,2.8976000000000006],
--R      [3.3999999999999999,2.9100000000000001],
--R      [3.4399999999999999,2.9216000000000006], [3.48,2.9323999999999999],
--R      [3.5200000000000005,2.9424000000000001], [3.5600000000000005,2.9516],
--R      [3.6000000000000005,2.96], [3.6400000000000006,2.9676],
--R      [3.6800000000000006,2.9744000000000002],
--R      [3.7200000000000006,2.9803999999999999],
--R      [3.7599999999999998,2.9855999999999998],
--R      [3.7999999999999998,2.9900000000000002],
--R      [3.8399999999999999,2.9935999999999998], [3.8799999999999999,2.9964],
--R      [3.9199999999999999,2.9984000000000002], [3.96,2.9996], [4.,3.],
--R      [4.0400000000000009,2.9996], [4.0800000000000001,2.9984000000000002],
--R      [4.120000000000001,2.9964], [4.1600000000000001,2.9935999999999998],
--R      [4.2000000000000011,2.9900000000000002],
--R      [4.2400000000000002,2.9855999999999998],
--R      [4.2800000000000011,2.9803999999999999],
```

```
--R      [4.3200000000000003,2.9744000000000002], [4.3600000000000012,2.9676],
--R      [4.4000000000000004,2.96], [4.4400000000000013,2.9516],
--R      [4.4800000000000004,2.9424000000000001],
--R      [4.5199999999999996,2.9323999999999999],
--R      [4.5599999999999996,2.9216000000000006],
--R      [4.5999999999999996,2.9100000000000001],
--R      [4.6399999999999997,2.8976000000000006],
--R      [4.6799999999999997,2.8843999999999999],
--R      [4.7199999999999998,2.8704000000000001],
--R      [4.7599999999999998,2.8555999999999999],
--R      [4.7999999999999998,2.8399999999999999],
--R      [4.8399999999999999,2.8235999999999999], [4.8799999999999999,2.8064],
--R      [4.9199999999999999,2.7884000000000002], [4.96,2.7696000000000005],
--R      [5.,2.75], [5.0400000000000009,2.7296000000000005],
--R      [5.0800000000000001,2.7084000000000001],
--R      [5.120000000000001,2.6863999999999999],
--R      [5.1600000000000001,2.6636000000000006],
--R      [5.2000000000000011,2.6400000000000006],
--R      [5.2400000000000002,2.6156000000000006],
--R      [5.2800000000000011,2.5903999999999998], [5.3200000000000003,2.5644],
--R      [5.3600000000000012,2.5375999999999999],
--R      [5.4000000000000004,2.5099999999999998],
--R      [5.4400000000000013,2.4815999999999998],
--R      [5.4800000000000004,2.4523999999999999],
--R      [5.5199999999999996,2.4224000000000006],
--R      [5.5599999999999996,2.3915999999999999],
--R      [5.5999999999999996,2.3599999999999999],
--R      [5.6399999999999997,2.3275999999999999], [5.6799999999999997,2.2944],
--R      [5.7199999999999998,2.2604000000000006], [5.7599999999999998,2.2256],
--R      [5.7999999999999998,2.1899999999999999], [5.8399999999999999,2.1536],
--R      [5.8799999999999999,2.1164000000000005],
--R      [5.9199999999999999,2.0783999999999998], [5.96,2.0396000000000001],
--R      [6.,2.]]
--R     ]
--R                                          Type: List(List(Point(DoubleFloat)))
--E 9

-- We do not do these during testing since graphics is not available
-- The resulting image is in the Bezier section of Volume 10.4

--d1:=draw(graphs.1,title=="Bezier Control Point Motion")
--others:=[graphs.i for i in 2..9]
--for i in 2..9 for graph in others repeat putGraph(d1,[graph],i)
--vp:=makeViewport2D(d1)
```



— Bezier.help —

```
========================================================================
Bezier Curve examples
========================================================================

A linear Bezier curve is a simple interpolation between the
starting point and the ending point based on a parameter t.

Given a start point a=[x1,y1] and an endpoint b=[x2,y2]
f(t) == [(1-t)*x1 + t*x2, (1-t)*y1 + t*y2]

n:=linearBezier([2.0,2.0],[4.0,4.0])
   theMap(BEZIER;linearBezier;2LM;1!0,707)

[n(t/10.0) for t in 0..10 by 1]
   [[2.0,2.0], [2.2,2.2], [2.4,2.4], [2.6,2.6], [2.8,2.8], [3.0,3.0],
    [3.2,3.2], [3.4,3.4], [3.6,3.6], [3.8,3.8], [4.0,4.0]]


A quadratic Bezier curve is a simple interpolation between the
starting point, a middle point, and the ending point based on
a parameter t.

Given a start point a=[x1,y1], a middle point b=[x2,y2],
and an endpoint c=[x3,y3]

f(t) == [(1-t)^2 x1 + 2t(1-t) x2 + t^2 x3,
         (1-t)^2 y1 + 2t(1-t) y2 + t^2 y3]

n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
   theMap(BEZIER;quadraticBezier;3LM;2!0,291)
```

```
[n(t/10.0) for t in 0..10 by 1]
   [[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0],
    [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]

A cubic Bezier curve is a simple interpolation between the
starting point, a left-middle point,, a right-middle point,
and the ending point based on a parameter t.

Given a start point a=[x1,y1], the left-middle point b=[x2,y2],
the right-middle point c=[x3,y3] and an endpoint d=[x4,y4]

f(t) == [(1-t)^3 x1 + 3t(1-t)^2 x2 + 3t^2 (1-t) x3 + t^3 x4,
         (1-t)^3 y1 + 3t(1-t)^2 y2 + 3t^2 (1-t) y3 + t^3 y4]

n:=cubicBezier([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])
   theMap(BEZIER;cubicBezier;4LM;3!0,915)

[n(t/10.0) for t in 0..10 by 1]
   [[2.0,2.0], [2.112,2.54], [2.416,2.96], [2.864,3.26], [3.408,3.44],
    [4.0,3.5], [4.592,3.44], [5.136,3.26], [5.584,2.96], [5.888,2.54],
    [6.0,2.0]]

Bezier curves "move" based on moving their control points, which
in the case of the three components of a quadratic Bezier curve
are the three points given. To see this motion we can show what
happens when you "drag" the middle control point along the line
from [-4,4] to [4,4] by increments of 1.

First, we form the line as a list of Floats, 9 in all.

   line:=[[i::Float,4.0] for i in -4..4 by 1]

Next, we form a list of functions, each with a different center control point.
Notice that the endpoints remain fized so we expect that the curve will
begin and end at the same point but that the midpoint is pulled around.

  functions:=[quadraticBezier([2.0,2.0],m,[6.0,2.0]) for m in line]

Then we form a list of the graphs by calling each function in the above
list. Each function call happens 101 times (to include both endpoints).
Thus we get a List of Lists of Points of DoubleFloats

  graphs:=[ [ point( ( (functions.i)(j/100.0) )::LIST(DFLOAT) ) _
    for j in 0..100] for i in 1..9]

We draw the first graph to see if it looks reasonable:

  d1:=draw(graphs.1)
```

```
Since it does we add the other 8 graphs. The 2D graphs can hold up to
9 simultaneous graphs.

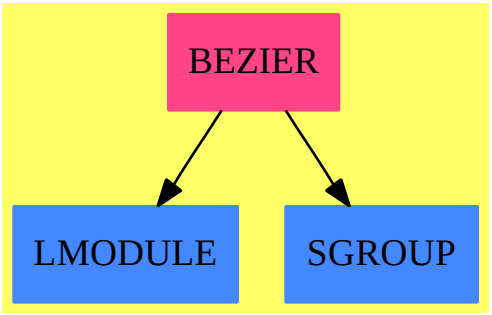
  others:=[graphs.i for i in 2..9]
  for i in 2..9 for graph in others repeat putGraph(d1,[graph],i)

and now we display them all on one graph.

  vp:=makeViewport2D(d1)

See Also:
o )show Bezier
o )show TwoDimensionalViewport
o )d op draw
o )d op point
o )d op putGraph
o )d op makeViewport2D
```

3.3.1 Bezier (BEZIER)



Exports:
linearBezier quadraticBezier cubicBezier

— package BEZIER Bezier —

```
)abbrev package BEZIER Bezier
++ Author: Timothy Daly
++ Date Created: 14 April 2009
++ Description:
++ Provide linear, quadratic, and cubic spline bezier curves
```

```
Bezier(R:Ring): with
   linearBezier: (x:List R,y:List R) -> Mapping(List R,R)
    ++ A linear Bezier curve is a simple interpolation between the
    ++ starting point and the ending point based on a parameter t.
    ++ Given a start point a=[x1,y1] and an endpoint b=[x2,y2]
    ++ f(t) == [(1-t)*x1 + t*x2, (1-t)*y1 + t*y2]
    ++
    ++X n:=linearBezier([2.0,2.0],[4.0,4.0])
    ++X [n(t/10.0) for t in 0..10 by 1]
   quadraticBezier: (x:List R,y:List R,z:List R) -> Mapping(List R,R)
    ++ A quadratic Bezier curve is a simple interpolation between the
    ++ starting point, a middle point, and the ending point based on
    ++ a parameter t.
    ++ Given a start point a=[x1,y1], a middle point b=[x2,y2],
    ++ and an endpoint c=[x3,y3]
    ++ f(t) == [(1-t)^2 x1 + 2t(1-t) x2 + t^2 x3,
    ++          (1-t)^2 y1 + 2t(1-t) y2 + t^2 y3]
    ++
    ++X n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
    ++X [n(t/10.0) for t in 0..10 by 1]
   cubicBezier: (w:List R,x:List R,y:List R,z:List R) -> Mapping(List R,R)
    ++ A cubic Bezier curve is a simple interpolation between the
    ++ starting point, a left-middle point,, a right-middle point,
    ++ and the ending point based on a parameter t.
    ++ Given a start point a=[x1,y1], the left-middle point b=[x2,y2],
    ++ the right-middle point c=[x3,y3] and an endpoint d=[x4,y4]
    ++ f(t) == [(1-t)^3 x1 + 3t(1-t)^2 x2 + 3t^2 (1-t) x3 + t^3 x4,
    ++          (1-t)^3 y1 + 3t(1-t)^2 y2 + 3t^2 (1-t) y3 + t^3 y4]
    ++
    ++X n:=cubicBezier([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])
    ++X [n(t/10.0) for t in 0..10 by 1]
 == add
   linearBezier(a,b) ==
    t +-> [(1-t)*(a.1) + t*(b.1), (1-t)*(a.2) + t*(b.2)]

   quadraticBezier(a,b,c) ==
    t +-> [(1-t)**2*(a.1) + 2*t*(1-t)*(b.1) + t**2*(c.1),
           (1-t)**2*(a.2) + 2*t*(1-t)*(b.2) + t**2*(c.2)]

   cubicBezier(a,b,c,d) ==
    t +-> [(1-t)**3*(a.1) + 3*t*(1-t)**2*(b.1)
             + 3*t**2*(1-t)*(c.1) + t**3*(d.1),
           (1-t)**3*(a.2) + 3*t*(1-t)**2*(b.2)
             + 3*t**2*(1-t)*(c.2) + t**3*(d.2)]
```

———-

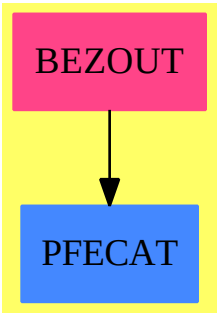
— BEZIER.dotabb —

```
"BEZIER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BEZIER"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"BEZIER" -> "LMODULE"
"BEZIER" -> "SGROUP"
```

———-

3.4 package BEZOUT BezoutMatrix

3.4.1 BezoutMatrix (BEZOUT)



Exports:
bezoutDiscriminant bezoutMatrix bezoutResultant sylvesterMatrix

— package BEZOUT BezoutMatrix —

```
)abbrev package BEZOUT BezoutMatrix
++ Author: Clifton J. Williamson
++ Date Created: 2 August 1988
++ Date Last Updated: 3 November 1993
++ Basic Operations: bezoutMatrix, bezoutResultant, bezoutDiscriminant
++ Related Domains
++ Also See:
++ AMS Classifiactions:
++ Keywords: Bezout matrix, resultant, discriminant
++ Examples:
++ Reference: Knuth, The Art of Computer Programming, 2nd edition,
++            Vol. 2, p. 619, problem 12.
++ Description:
++ \spadtype{BezoutMatrix} contains functions for computing resultants and
++ discriminants using Bezout matrices.

BezoutMatrix(R,UP,M,Row,Col): Exports == Implementation where
```

```
R    : Ring
UP   : UnivariatePolynomialCategory R
Row  : FiniteLinearAggregate R
Col  : FiniteLinearAggregate R
M    : MatrixCategory(R,Row,Col)
I  ==> Integer
lc ==> leadingCoefficient

Exports ==> with
  sylvesterMatrix: (UP,UP) -> M
    ++ sylvesterMatrix(p,q) returns the Sylvester matrix for the two
    ++ polynomials p and q.
  bezoutMatrix: (UP,UP) -> M
    ++ bezoutMatrix(p,q) returns the Bezout matrix for the two
    ++ polynomials p and q.

  if R has commutative("*") then
    bezoutResultant: (UP,UP) -> R
     ++ bezoutResultant(p,q) computes the resultant of the two
     ++ polynomials p and q by computing the determinant of a Bezout matrix.

    bezoutDiscriminant: UP -> R
     ++ bezoutDiscriminant(p) computes the discriminant of a polynomial p
     ++ by computing the determinant of a Bezout matrix.

Implementation ==> add

  sylvesterMatrix(p,q) ==
    n1 := degree p; n2 := degree q; n := n1 + n2
    sylmat : M := new(n,n,0)
    minR := minRowIndex sylmat; minC := minColIndex sylmat
    maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
    p0 := p
    -- fill in coefficients of 'p'
    while not zero? p0 repeat
      coef := lc p0; deg := degree p0; p0 := reductum p0
      -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
      for i in 0..n2 - 1 repeat
        qsetelt_!(sylmat,minR + i,minC + n1 - deg + i,coef)
    q0 := q
    -- fill in coefficients of 'q'
    while not zero? q0 repeat
      coef := lc q0; deg := degree q0; q0 := reductum q0
      for i in 0..n1-1 repeat
        qsetelt_!(sylmat,minR + n2 + i,minC + n2 - deg + i,coef)
    sylmat

  bezoutMatrix(p,q) ==
  -- This function computes the Bezout matrix for 'p' and 'q'.
  -- See Knuth, The Art of Computer Programming, Vol. 2, p. 619, # 12.
```

```
-- One must have deg(p) >= deg(q), so the arguments are reversed
-- if this is not the case.
  n1 := degree p; n2 := degree q; n := n1 + n2
  n1 < n2 => bezoutMatrix(q,p)
  m1 : I := n1 - 1; m2 : I := n2 - 1; m : I := n - 1
  -- 'sylmat' will be a matrix consisting of the first n1 columns
  -- of the standard Sylvester matrix for 'p' and 'q'
  sylmat : M := new(n,n1,0)
  minR := minRowIndex sylmat; minC := minColIndex sylmat
  maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
  p0 := p
  -- fill in coefficients of 'p'
  while not ground? p0 repeat
    coef := lc p0; deg := degree p0; p0 := reductum p0
    -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
    -- for i = 0...
    -- quit when i > m2 or when i + (n1 - k) > m1, whichever happens first
    for i in 0..min(m2,deg - 1) repeat
      qsetelt_!(sylmat,minR + i,minC + n1 - deg + i,coef)
  q0 := q
  -- fill in coefficients of 'q'
  while not zero? q0 repeat
    coef := lc q0; deg := degree q0; q0 := reductum q0
    -- put ak = coef(q,k) in sylmat(minR + n1 + i,minC + i + (n2 - k))
    -- for i = 0...
    -- quit when i > m1 or when i + (n2 - k) > m1, whichever happens first
    -- since n2 - k >= 0, we quit when i + (n2 - k) > m1
    for i in 0..(deg + n1 - n2 - 1) repeat
      qsetelt_!(sylmat,minR + n2 + i,minC + n2 - deg + i,coef)
  -- 'bezmat' will be the 'Bezout matrix' as described in Knuth
  bezmat : M := new(n1,n1,0)
  for i in 0..m2 repeat
    -- replace A_i by (b_0 A_i + ... + b_{n_2-1-i} A_{n_2 - 1}) -
    -- (a_0 B_i + ... + a_{n_2-1-i} B_{n_2-1}), as in Knuth
    bound : I := n2 - i; q0 := q
    while not zero? q0 repeat
      deg := degree q0
      if (deg < bound) then
        -- add b_deg A_{n_2 - deg} to the new A_i
        coef := lc q0
        for k in minC..maxC repeat
          c := coef * qelt(sylmat,minR + m2 - i - deg,k) +
                      qelt(bezmat,minR + m2 - i,k)
          qsetelt_!(bezmat,minR + m2 - i,k,c)
      q0 := reductum q0
    p0 := p
    while not zero? p0 repeat
      deg := degree p0
      if deg < bound then
        coef := lc p0
```

```
            -- subtract a_deg B_{n_2 - deg} from the new A_i
            for k in minC..maxC repeat
              c := -coef * qelt(sylmat,minR + m - i - deg,k) +
                           qelt(bezmat,minR + m2 - i,k)
              qsetelt_!(bezmat,minR + m2 - i,k,c)
          p0 := reductum p0
      for i in n2..m1 repeat for k in minC..maxC repeat
        qsetelt_!(bezmat,minR + i,k,qelt(sylmat,minR + i,k))
      bezmat

    if R has commutative("*") then

      bezoutResultant(f,g) == determinant bezoutMatrix(f,g)

      if R has IntegralDomain then

        bezoutDiscriminant f ==
          degMod4 := (degree f) rem 4
          (degMod4 = 0) or (degMod4 = 1) =>
            (bezoutResultant(f,differentiate f) exquo (lc f)) :: R
          -((bezoutResultant(f,differentiate f) exquo (lc f)) :: R)

        else

          bezoutDiscriminant f ==
            lc f = 1 =>
              degMod4 := (degree f) rem 4
              (degMod4 = 0) or (degMod4 = 1) =>
                bezoutResultant(f,differentiate f)
              -bezoutResultant(f,differentiate f)
            error "bezoutDiscriminant: leading coefficient must be 1"
```

———-

— BEZOUT.dotabb —

```
"BEZOUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BEZOUT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BEZOUT" -> "PFECAT"
```

———-

3.5 package BLUPPACK BlowUpPackage

— BlowUpPackage.input —

```
)set break resume
)sys rm -f BlowUpPackage.output
)spool BlowUpPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BlowUpPackage
--R
--R BlowUpPackage(K: Field,symb: List(Symbol),PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategor
--R Abbreviation for BlowUpPackage is BLUPPACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for BLUPPACK
--R
--R------------------------------ Operations --------------------------------
--R applyTransform : (PolyRing,BLMET) -> PolyRing
--R biringToPolyRing : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),BLMET) -> PolyRing
--R newtonPolySlope : DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K) -> List(List(NonNegati
--R polyRingToBlUpRing : (PolyRing,BLMET) -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R quadTransform : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),NonNegativeInteger,BLME
--R stepBlowUp : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),AffinePlane(K),BLMET,K) ->
--R
--E 1

)spool
)lisp (bye)
```

———-

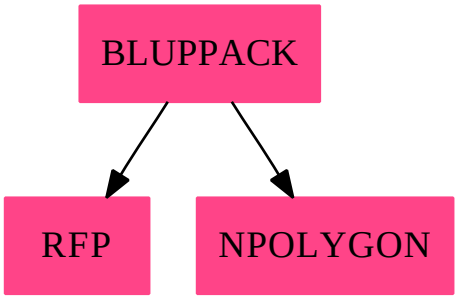
— BlowUpPackage.help —

```
=====================================================================
BlowUpPackage examples
=====================================================================

See Also:
o )show BlowUpPackage
```

———-

3.5.1 BlowUpPackage (BLUPPACK)



Exports:

applyTransform	biringToPolyRing	newtonPolySlope
polyRingToBlUpRing	quadTransform	stepBlowUp

— package BLUPPACK BlowUpPackage —

```
)abbrev package BLUPPACK BlowUpPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
BlowUpPackage(K,symb,PolyRing,E,  BLMET):Exports == Implementation where
  K         : Field
  symb      : List Symbol
  PolyRing  : FiniteAbelianMonoidRing(K,E)
  E         : DirectProductCategory(#symb,NonNegativeInteger)
  BLMET     : BlowUpMethodCategory

  NNI      ==> NonNegativeInteger
  RFP      ==> RootsFindingPackage
  NP       ==> NewtonPolygon( K, BlUpRing, E2 , #bls)
  PackPoly ==> PackageForPoly(K,BlUpRing,E2,#bls)
  bls      ==>  ['X,'Y]
  BlUpRing ==> DistributedMultivariatePolynomial( bls , K )
  E2       ==> DirectProduct(#bls , NNI)
  AFP      ==> AffinePlane(K)

  blowUpRec    ==> Record(recTransStr:BlUpRing,recPoint:AFP,recChart:BLMET,_
                          definingExtension:K)
  blowUpReturn ==> Record(mult:NonNegativeInteger,subMult: NNI, _
                             blUpRec:List(blowUpRec))
  recStr    ==> Record( sM: NNI , blRec:List blowUpRec)

  Exports ==  with
```

```
    applyTransform: (PolyRing,BLMET) -> PolyRing
      ++ quadTransform(pol,chart) apply the quadratique transformation to
      ++ pol specified by chart which consist of 3 integers. The last one
      ++ indicates which varibles is set to 1, the first on indicates
      ++ which variable remains unchange, and the second one indicates
      ++ which variable oon which the transformation is applied.
      ++ For example, [2,3,1] correspond to the following:
      ++ x -> 1, y -> y, z -> yz (here the variable are [x,y,z] in BlUpRing).

    quadTransform: (BlUpRing,NNI,BLMET) -> BlUpRing  -- CHH
      ++ quadTransform(pol,n,chart) apply the quadratique transformation
      ++ to pol specified by chart has in quadTransform(pol,chart) and
      ++ extract x**n to it, where x is the variable specified by the
      ++ first integer in chart (blow-up exceptional coordinate).

    stepBlowUp:(BlUpRing,AFP,BLMET,K) -> blowUpReturn  -- CHH
      ++ stepBlowUp(pol,pt,n) blow-up the point pt on the curve defined
      ++ by pol in the affine neighbourhood specified by n.

    newtonPolySlope: BlUpRing -> List List(NNI)

    polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing

    biringToPolyRing: (BlUpRing, BLMET) -> PolyRing

Implementation ==> add

  import BlUpRing
  import AFP
  import RFP(K)
  import PackPoly
  import NP

  makeAff( l:List(K) , chart: BLMET ):AFP ==
        (excepCoord chart) = 1 => affinePoint( l )$AFP
        affinePoint( reverse l )$AFP

  blowExp: (E2, NNI, BLMET ) -> E2

  maxOf: (K,K) -> K

  getStrTrans: ( BlUpRing , List BlUpRing , BLMET, K ) -> recStr

  stepBlowUp(crb:BlUpRing,pt:AFP,chart:BLMET,actualExtension:K) ==
    -- next is with Hamburger-Noether method
    BLMET has HamburgerNoether =>
      nV:Integer:= chartCoord chart
      crbTrans:BlUpRing:=translate(crb, list(pt))$PackPoly
      newtPol:= newtonPolygon( crbTrans, quotValuation chart, _
```

```
                          ramifMult chart, type chart )$NP
  multPt:= multiplicity(newtPol)$NP
  one?(multPt) =>
    [multPt, 0 , empty() ]$blowUpReturn
  listOfgetTr:List recStr:= _
    [ getStrTrans( crbTrans , edge , chart , actualExtension ) _
      for edge in newtPol ]
  lsubM: List NNI :=  [ ll.sM for ll in listOfgetTr]
  subM := reduce( "+" , lsubM )
  llistOfRec: List List blowUpRec :=  [ ll.blRec for ll in listOfgetTr]
  listOfRec:= concat llistOfRec
  [ multPt, subM ,listOfRec]$blowUpReturn
-- next is with usual quadratic transform.

BLMET has QuadraticTransform =>
  nV:Integer:= chartCoord chart
  lpt:List(K) := list(pt)$AFP
  crbTrans:=translate(crb,lpt)
  minForm:=minimalForm(crbTrans)
  multPt:=totalDegree( minForm)$PackPoly
  listRec:List(blowUpRec):=empty()
  one?(multPt) => [multPt, 0 , listRec]$blowUpReturn
  -- now pt is singular !!!!
  lstInd:=[i::PositiveInteger for i in 1..2 ]
  -- la ligne suivante fait un choix judicieux pour minimiser le
  -- degre' du transforme' stricte.
  if degree( crbTrans , 2 )$PackPoly < degree( crbTrans , 1 )$PackPoly _
     then  lstInd := reverse lstInd
  ptInf:List(K):=[0$K,0$K]
  laCarte:BLMET:=
    ([last(lstInd), first(lstInd),nV] @  List Integer) :: BLMET
  laCarteInf:BLMET:=
    ([first(lstInd),last(lstInd),nV] @ List Integer ) :: BLMET
  transStricte   :=quadTransform(crbTrans,multPt,laCarte)
  transStricteInf:=quadTransform(crbTrans,multPt,laCarteInf)
  listPtsSingEcl:List(AFP):=empty()
  transStricteZero:BlUpRing:= replaceVarByOne(minForm,excepCoord laCarte)
  recOfZeros:=_
    distinguishedRootsOf(univariate(transStricteZero)$PackPoly ,_
                                    actualExtension )$RFP(K)
  degExt:=recOfZeros.extDegree
  ^one?(degExt) =>
    print(("You need an extension of degree")::OutputForm)
    print(degExt::OutputForm)
    error("Have a nice day")
  listPtsSingEcl:=[makeAff([0$K,a]::List(K),laCarte) _
                   for a in recOfZeros.zeros]
  listRec:=[
              [  transStricte,_
                 ptS,laCarte,_
```

```
      maxOf(a,actualExtension)]$blowUpRec_
   for ptS in listPtsSingEcl_
        for a in recOfZeros.zeros]
      if zero?(constant(transStricteInf))$K then
        listRec:= concat(listRec,[transStricteInf,_
                            affinePoint(ptInf)$AFP,_
   laCarteInf,_
   actualExtension]$blowUpRec)
      empty?(listRec) =>
        error "Something is very wrong in blowing up!!!!!!"
      [multPt, 0 ,listRec]$blowUpReturn
    error "Desingularisation is not implemented for the blowing up method chosen, see BlowingUpMethodCateg

  getStrTrans( crb , inedge , actChart, actualExtension ) ==
    edge:= copy inedge
    s := slope(edge)$NP
    sden:Integer
    snum:Integer
    i1:Integer
    i2:Integer
    if s.type case "right"  then
      sden:= s.base
      snum:=s.height
      i1:=1
      i2:=2
    else -- interchange les roles de X et Y .
      sden:= s.height
      snum:= s.base
      i1:=2
      i2:=1
      edge := copy reverse inedge
    ee := entries( degree first edge) pretend List Integer
    euclq: Integer
    if one?(snum) then
      euclq:=1
    else
      euclq   := s.quotient
    -- sMult est la somme des  multiplicite des  points infiniment
    -- voisin par une trans. quadratique
    sMult: NNI :=  ( ( euclq - 1 )   * ee.i2 ) pretend NNI
    -- extMult est egal a la plus grande puissance de X que l'on peut
    --extraire de la transformee.
    extMult := (ee.i1 + ee.i2 * euclq) pretend NonNegativeInteger
    ch: BLMET
    trStr:BlUpRing
    listBlRec: List blowUpRec
    ^zero?(s.reste ) =>
       ch:= createHN( i1 , i2 , chartCoord actChart, euclq , s.reste , _
                     false , s.type)$BLMET
       trStr:= quadTransform(crb, extMult , ch )
```

```
      listBlRec:= [ [trStr,origin()$AFP,ch,actualExtension ]$blowUpRec ]
      [ sMult , listBlRec  ]$recStr
  polEdge := reduce( "+" , edge )
  unipol:= univariate( replaceVarByOne( polEdge , i1 )$PackPoly )$PackPoly
  recOfZeros:= distinguishedRootsOf( unipol , actualExtension )$RFP(K)
  degExt:=recOfZeros.extDegree
  ^one?(degExt) =>
      print(("You need an extension of degree")::OutputForm)
      print(degExt::OutputForm)
      error("Have a nice day")
  listOfZeroes:List K:= [ z for z in recOfZeros.zeros | ^zero?(z) ]
  empty? listOfZeroes => _
    error " The curve is not absolutely irreducible since the Newton polygon has no sides "
  ch:=_
    createHN( i1 , i2, chartCoord actChart, euclq, 0, false, s.type)$BLMET
  lsTr:BlUpRing:= quadTransform(crb, extMult , ch )
  lAff:List AFP:=[makeAff([ 0$K, z]:: List K , ch) for z in listOfZeroes ]
  listBlRec := [ [ lsTr,p,ch,maxOf( actualExtension , z) ]$blowUpRec_
     for p in lAff for z in listOfZeroes ]
  [sMult, listBlRec ]$recStr

blowExp(exp,mult,chart)== -- CHH
  zero?( excepCoord chart) => exp
  lexp:List NNI:=parts(exp)
  ch1:Integer:= excepCoord chart
  ch2:Integer:= transCoord chart
  e1:Integer := lexp(ch1) pretend Integer
  e2:Integer := lexp(ch2) pretend Integer
  quotVal:Integer := quotValuation chart
  lbexp:=[0,0] :: List(NNI)
  lbexp(ch1):= ( e1 + quotVal * e2  - mult ) pretend NonNegativeInteger
  lbexp(ch2):=lexp(ch2)
  directProduct(vector(lbexp)$Vector(NNI))$E2

quadTransform(pol,mult,chart)==  -- CHH
  mapExponents(blowExp(#1,mult,chart),pol)

polyRingToBlUpRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  ll:= [ d.i for i in 1..3 | ^( i = chartCoord(chart) ) ]
  e:= directProduct( vector( ll)$Vector(NNI) )$E2
  monomial(lc , e )$BlUpRing + polyRingToBlUpRing( reductum pol, chart )

biringToPolyRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  nV:= chartCoord chart
```

```
      ll:List NNI:=
        nV = 1 => [ 0$NNI , d.1  , d.2 ]
        nV = 2 => [ d.1  , 0$NNI , d.2 ]
        [d.1 , d.2 , 0$NNI ]
      e:= directProduct( vector( ll)$Vector(NNI) )$E
      monomial(lc , e )$PolyRing  + biringToPolyRing( reductum pol, chart )

    applyTransform(pol,chart)==
      biringToPolyRing( quadTransform( polyRingToBlUpRing( pol, chart ) ,_
                        0 , chart) , chart )

-- K has PseudoAlgebraicClosureOfFiniteFieldCategory => maxTower([a,b])$K
-- K has PseudoAlgebraicClosureOfRationalNumberCategory  => maxTower([a,b])$K
    maxOf(a:K,b:K):K ==
      K has PseudoAlgebraicClosureOfPerfectFieldCategory  => maxTower([a,b])$K
      1$K
```

———-

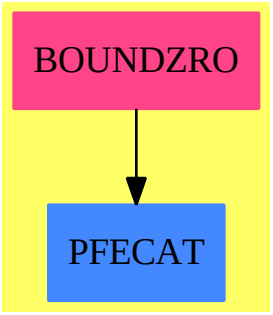
— **BLUPPACK.dotabb** —

```
"BLUPPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BLUPPACK"]
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"NPOLYGON" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPOLYGON"]
"BLUPPACK" -> "RFP"
"BLUPPACK" -> "NPOLYGON"
```

———-

3.6 package BOUNDZRO BoundIntegerRoots

3.6.1 BoundIntegerRoots (BOUNDZRO)



Exports:
integerBound

— package BOUNDZRO BoundIntegerRoots —

```
)abbrev package BOUNDZRO BoundIntegerRoots
++ Author: Manuel Bronstein
++ Date Created: 11 March 1991
++ Date Last Updated: 18 November 1991
++ Description:
++ \spadtype{BoundIntegerRoots} provides functions to
++ find lower bounds on the integer roots of a polynomial.

BoundIntegerRoots(F, UP): Exports == Implementation where
  F  : Join(Field, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

  Z   ==> Integer
  Q   ==> Fraction Z
  K   ==> Kernel F
  UPQ ==> SparseUnivariatePolynomial Q
  ALGOP ==> "%alg"

  Exports ==> with
    integerBound: UP -> Z
      ++ integerBound(p) returns a lower bound on the negative integer
      ++ roots of p, and 0 if p has no negative integer roots.

  Implementation ==> add
    import RationalFactorize(UPQ)
    import UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)

    qbound : (UP, UPQ) -> Z
    zroot1 : UP -> Z
    qzroot1: UPQ -> Z
    negint : Q -> Z

-- returns 0 if p has no integer root < 0, its negative integer root otherwise
    qzroot1 p == negint(- leadingCoefficient(reductum p) / leadingCoefficient p)

-- returns 0 if p has no integer root < 0, its negative integer root otherwise
    zroot1 p ==
      z := - leadingCoefficient(reductum p) / leadingCoefficient p
      (r := retractIfCan(z)@Union(Q, "failed")) case Q => negint(r::Q)
      0

-- returns 0 if r is not a negative integer, r otherwise
    negint r ==
      ((u := retractIfCan(r)@Union(Z, "failed")) case Z) and (u::Z < 0) => u::Z
```

```
      0

    if F has ExpressionSpace then
      bringDown: F -> Q

-- the random substitution used by bringDown is NOT always a ring-homorphism
-- (because of potential algebraic kernels), but is ALWAYS a Z-linear map.
-- this guarantees that bringing down the coefficients of (x + n) q(x) for an
-- integer n yields a polynomial h(x) which is divisible by x + n
-- the only problem is that evaluating with random numbers can cause a
-- division by 0. We should really be able to trap this error later and
-- reevaluate with a new set of random numbers    MB 11/91
      bringDown f ==
        t := tower f
        retract eval(f, t, [random()$Q :: F for k in t])

      integerBound p ==
--        one? degree p => zroot1 p
        (degree p) = 1 => zroot1 p
        q1 := map(bringDown, p)
        q2 := map(bringDown, p)
        qbound(p, gcd(q1, q2))

    else
      integerBound p ==
--        one? degree p => zroot1 p
        (degree p) = 1 => zroot1 p
        qbound(p, map((z1:F):Q +-> retract(z1)@Q, p))

-- we can probably do better here (i.e. without factoring)
    qbound(p, q) ==
      bound:Z := 0
      for rec in factors factor q repeat
--        if one?(degree(rec.factor)) and ((r := qzroot1(rec.factor)) < bound)
        if ((degree(rec.factor)) = 1) and ((r := qzroot1(rec.factor)) < bound)
           and zero? p(r::Q::F) then bound := r
      bound
```

———-

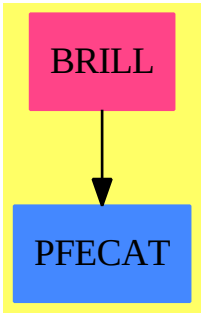
— BOUNDZRO.dotabb —

```
"BOUNDZRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BOUNDZRO"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BOUNDZRO" -> "PFECAT"
```

———-

3.7 package BRILL BrillhartTests

3.7.1 BrillhartTests (BRILL)



Exports:
noLinearFactor? brillhartIrreducible? brillhartTrials

— package BRILL BrillhartTests —

```
)abbrev package BRILL BrillhartTests
++ Author: Frederic Lehobey, James H. Davenport
++ Date Created: 28 June 1994
++ Date Last Updated: 11 July 1997
++ Basic Operations: brillhartIrreducible?
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: factorization
++ Examples:
++ References:
++ [1] John Brillhart, Note on Irreducibility Testing,
++ Mathematics of Computation, vol. 35, num. 35, Oct. 1980, 1379-1381
++ [2] James Davenport, On Brillhart Irreducibility. To appear.
++ [3] John Brillhart, On the Euler and Bernoulli polynomials,
++ J. Reine Angew. Math., v. 234, (1969), pp. 45-64
++ Description:
++ This package has no description

BrillhartTests(UP): Exports == Implementation where
  N ==> NonNegativeInteger
  Z ==> Integer
  UP: UnivariatePolynomialCategory Z

  Exports ==> with
    brillhartIrreducible?: UP -> Boolean -- See [1]
      ++ brillhartIrreducible?(p) returns \spad{true} if p can be shown to be
```

```
    ++ irreducible by a remark of Brillhart, \spad{false} is inconclusive.
  brillhartIrreducible?: (UP,Boolean) -> Boolean -- See [1]
    ++ brillhartIrreducible?(p,noLinears) returns \spad{true} if p can be
    ++ shown to be irreducible by a remark of Brillhart, \spad{false} else.
    ++ If noLinears is \spad{true}, we are being told p has no linear factors
    ++ \spad{false} does not mean that p is reducible.
  brillhartTrials: () -> N
    ++ brillhartTrials() returns the number of tests in
    ++ \spadfun{brillhartIrreducible?}.
  brillhartTrials: N -> N
    ++ brillhartTrials(n) sets to n the number of tests in
    ++ \spadfun{brillhartIrreducible?} and returns the previous value.
  noLinearFactor?: UP -> Boolean -- See [3] p. 47
    ++ noLinearFactor?(p) returns \spad{true} if p can be shown to have no
    ++ linear factor by a theorem of Lehmer, \spad{false} else. I insist on
    ++ the fact that \spad{false} does not mean that p has a linear factor.

Implementation ==> add

  import GaloisGroupFactorizationUtilities(Z,UP,Float)

  squaredPolynomial(p:UP):Boolean ==
    d := degree p
    d = 0 => true
    odd? d => false
    squaredPolynomial reductum p

  primeEnough?(n:Z,b:Z):Boolean ==
     -- checks if n is prime, with the possible exception of
     -- factors whose product is at most b
     import Float
     bb: Float := b::Float
     for i in 2..b repeat
         while (d:= n exquo i) case Integer repeat
               n:=d::Integer
               bb:=bb / i::Float
               bb < 1$Float => return false
               --- we over-divided, so it can't be prime
     prime? n

  brillharttrials: N := 6
  brillhartTrials():N == brillharttrials

  brillhartTrials(n:N):N ==
    (brillharttrials,n) := (n,brillharttrials)
    n

  brillhartIrreducible?(p:UP):Boolean ==
    brillhartIrreducible?(p,noLinearFactor? p)
```

```
    brillhartIrreducible?(p:UP,noLinears:Boolean):Boolean == -- See [1]
      zero? brillharttrials => false
      origBound := (largeEnough := rootBound(p)+1)
      -- see remarks 2 and 4
      even0 := even? coefficient(p,0)
      even1 := even? p(1)
      polyx2 := squaredPolynomial(p)
      prime? p(largeEnough) => true
      not polyx2 and prime? p(-largeEnough) => true
--      one? brillharttrials => false
      (brillharttrials = 1) => false
      largeEnough := largeEnough+1
      primeEnough?(p(largeEnough),if noLinears then 4 else 2) => true
      not polyx2 and
       primeEnough?(p(-largeEnough),if noLinears then 4 else 2) => true
      if odd? largeEnough then
        if even0 then largeEnough := largeEnough+1
      else
        if even1 then largeEnough := largeEnough+1
      count :=(if polyx2 then 2 else 1)*(brillharttrials-2)+largeEnough
      for i in (largeEnough+1)..count repeat
        small := if noLinears then (i-origBound)**2 else (i-origBound)
        primeEnough?(p(i),small) => return true
        not polyx2 and primeEnough?(p(-i),small) => return true
      false

    noLinearFactor?(p:UP):Boolean ==
      (odd? leadingCoefficient p) and (odd? coefficient(p,0)) and (odd? p(1))
```

———-

— BRILL.dotabb —

```
"BRILL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BRILL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BRILL" -> "PFECAT"
```

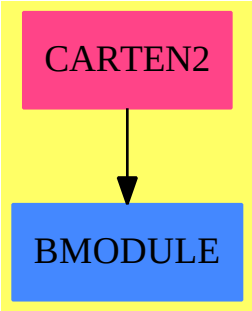
———-

Chapter 4

Chapter C

4.1 package CARTEN2 CartesianTensorFunctions2

4.1.1 CartesianTensorFunctions2 (CARTEN2)



Exports:
map reshape

— package CARTEN2 CartesianTensorFunctions2 —

```
)abbrev package CARTEN2 CartesianTensorFunctions2
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 30, 1991
++ Basic Operations:  reshape, map
++ Related Domains: CartesianTensor
++ Also See:
++ AMS Classifications:
++ Keywords: tensor
```

```
++ Examples:
++ References:
++ Description:
++ This package provides functions to enable conversion of tensors
++ given conversion of the components.

CartesianTensorFunctions2(minix, dim, S, T): CTPcat == CTPdef where
    minix:  Integer
    dim:    NonNegativeInteger
    S, T:   CommutativeRing
    CS  ==> CartesianTensor(minix, dim, S)
    CT  ==> CartesianTensor(minix, dim, T)

    CTPcat == with
        reshape: (List T, CS) -> CT
            ++ reshape(lt,ts) organizes the list of components lt into
            ++ a tensor with the same shape as ts.
        map: (S->T,   CS) -> CT
            ++ map(f,ts) does a componentwise conversion of the tensor ts
            ++ to a tensor with components of type T.
    CTPdef == add
        reshape(l, s) == unravel l
        map(f, s)     == unravel [f e for e in ravel s]
```

———-

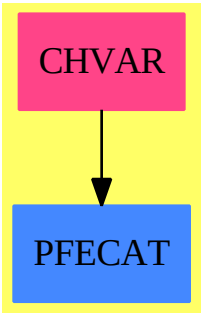
— CARTEN2.dotabb —

```
"CARTEN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CARTEN2"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"CARTEN2" -> "BMODULE"
```

———-

4.2 package CHVAR ChangeOfVariable

4.2.1 ChangeOfVariable (CHVAR)



Exports:

chvar	eval	goodPoint	mkIntegral
radPoly	rootPoly		

— **package CHVAR ChangeOfVariable** —

```
)abbrev package CHVAR ChangeOfVariable
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 Feb 1990
++ Description:
++ Tools to send a point to infinity on an algebraic curve.

ChangeOfVariable(F, UP, UPUP): Exports == Implementation where
  F   : UniqueFactorizationDomain
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP

  N  ==> NonNegativeInteger
  Z  ==> Integer
  Q  ==> Fraction Z
  RF ==> Fraction UP

  Exports ==> with
    mkIntegral: UPUP -> Record(coef:RF, poly:UPUP)
          ++ mkIntegral(p(x,y)) returns \spad{[c(x), q(x,z)]} such that
          ++ \spad{z = c * y} is integral.
          ++ The algebraic relation between x and y is \spad{p(x, y) = 0}.
          ++ The algebraic relation between x and z is \spad{q(x, z) = 0}.
    radPoly   : UPUP -> Union(Record(radicand:RF, deg:N), "failed")
          ++ radPoly(p(x, y)) returns \spad{[c(x), n]} if p is of the form
          ++ \spad{y**n - c(x)}, "failed" otherwise.
    rootPoly  : (RF, N) -> Record(exponent: N, coef:RF, radicand:UP)
```

```
          ++ rootPoly(g, n) returns \spad{[m, c, P]} such that
          ++ \spad{c * g ** (1/n) = P ** (1/m)}
          ++ thus if \spad{y**n = g}, then \spad{z**m = P}
          ++ where \spad{z = c * y}.
    goodPoint : (UPUP,UPUP) -> F
          ++ goodPoint(p, q) returns an integer a such that a is neither
          ++ a pole of \spad{p(x,y)} nor a branch point of \spad{q(x,y) = 0}.
    eval      : (UPUP, RF, RF) -> UPUP
          ++ eval(p(x,y), f(x), g(x)) returns \spad{p(f(x), y * g(x))}.
    chvar : (UPUP,UPUP) -> Record(func:UPUP,poly:UPUP,c1:RF,c2:RF,deg:N)
          ++ chvar(f(x,y), p(x,y)) returns
          ++ \spad{[g(z,t), q(z,t), c1(z), c2(z), n]}
          ++ such that under the change of variable
          ++ \spad{x = c1(z)}, \spad{y = t * c2(z)},
          ++ one gets \spad{f(x,y) = g(z,t)}.
          ++ The algebraic relation between x and y is \spad{p(x, y) = 0}.
          ++ The algebraic relation between z and t is \spad{q(z, t) = 0}.

  Implementation ==> add
    import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

    algPoly     : UPUP           -> Record(coef:RF, poly:UPUP)
    RPrim       : (UP, UP, UPUP) -> Record(coef:RF, poly:UPUP)
    good?       : (F, UP, UP)    -> Boolean
    infIntegral?: (UPUP, UPUP)   -> Boolean

    eval(p, x, y)  == map(s +-> s(x), p)  monomial(y, 1)
    good?(a, p, q) == p(a) ^= 0 and q(a) ^= 0

    algPoly p ==
      ground?(a:= retract(leadingCoefficient(q:=clearDenominator p))@UP)
        => RPrim(1, a, q)
      c := d := squareFreePart a
      q := clearDenominator q monomial(inv(d::RF), 1)
      while not ground?(a := retract(leadingCoefficient q)@UP) repeat
        c := c * (d := gcd(a, d))
        q := clearDenominator q monomial(inv(d::RF), 1)
      RPrim(c, a, q)

    RPrim(c, a, q) ==
--      one? a => [c::RF, q]
      (a = 1) => [c::RF, q]
      [(a * c)::RF, clearDenominator q monomial(inv(a::RF), 1)]

-- always makes the algebraic integral, but does not send a point to infinity
-- if the integrand does not have a pole there (in the case of an nth-root)
    chvar(f, modulus) ==
      r1 := mkIntegral modulus
      f1 := f monomial(r1inv := inv(r1.coef), 1)
      infIntegral?(f1, r1.poly) =>
```

```
        [f1, r1.poly, monomial(1,1)$UP :: RF,r1inv,degree(retract(r1.coef)@UP)]
      x  := (a:= goodPoint(f1,r1.poly))::UP::RF + inv(monomial(1,1)::RF)
      r2c:= retract((r2 := mkIntegral map(s+->s(x), r1.poly)).coef)@UP
      t  := inv((monomial(1, 1)$UP - a::UP)::RF)
      [- inv(monomial(1, 2)$UP :: RF) * eval(f1, x, inv(r2.coef)),
                               r2.poly, t, r1.coef * r2c t, degree r2c]

-- returns true if y is an n-th root, and it can be guaranteed that p(x,y)dx
-- is integral at infinity
-- expects y to be integral.
    infIntegral?(p, modulus) ==
      (r := radPoly modulus) case "failed" => false
      ninv := inv(r.deg::Q)
      degy:Q := degree(retract(r.radicand)@UP) * ninv
      degp:Q := 0
      while p ^= 0 repeat
        c := leadingCoefficient p
        degp := max(degp,
            (2 + degree(numer c)::Z - degree(denom c)::Z)::Q + degree(p) * degy)
        p := reductum p
      degp <= ninv

    mkIntegral p ==
      (r := radPoly p) case "failed" => algPoly p
      rp := rootPoly(r.radicand, r.deg)
      [rp.coef, monomial(1, rp.exponent)$UPUP - rp.radicand::RF::UPUP]

    goodPoint(p, modulus) ==
      q :=
        (r := radPoly modulus) case "failed" =>
                    retract(resultant(modulus, differentiate modulus))@UP
        retract(r.radicand)@UP
      d := commonDenominator p
      for i in 0.. repeat
        good?(a := i::F, q, d) => return a
        good?(-a, q, d)        => return -a

    radPoly p ==
      (r := retractIfCan(reductum p)@Union(RF, "failed")) case "failed"
        => "failed"
      [- (r::RF), degree p]

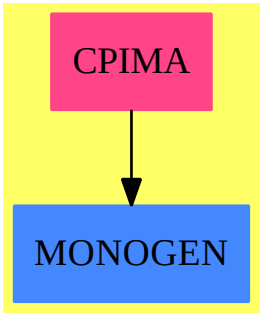
-- we have y**m = g(x) = n(x)/d(x), so if we can write
-- (n(x) * d(x)**(m-1)) ** (1/m)  =  c(x) * P(x) ** (1/n)
-- then z**q = P(x) where z = (d(x) / c(x)) * y
    rootPoly(g, m) ==
      zero? g => error "Should not happen"
      pr := nthRoot(squareFree((numer g) * (d := denom g) ** (m-1)::N),
                                                 m)$FactoredFunctions(UP)
      [pr.exponent, d / pr.coef, */(pr.radicand)]
```

— **CHVAR.dotabb** —

```
"CHVAR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CHVAR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CHVAR" -> "PFECAT"
```

4.3 package CPIMA CharacteristicPolynomialInMonogenicalAlgebra

4.3.1 CharacteristicPolynomialInMonogenicalAlgebra (CPIMA)



Exports:
characteristicPolynomial

— **package CPIMA CharacteristicPolynomialInMonogenicalAlgebra** —

```
)abbrev package CPIMA CharacteristicPolynomialInMonogenicalAlgebra
++ Author: Claude Quitte
++ Date Created: 10/12/93
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ This package implements characteristicPolynomials for monogenic algebras
++ using resultants

CharacteristicPolynomialInMonogenicalAlgebra(R : CommutativeRing,
    PolR : UnivariatePolynomialCategory(R),
    E : MonogenicAlgebra(R, PolR)): with
     characteristicPolynomial : E -> PolR
      ++ characteristicPolynomial(e) returns the characteristic polynomial
      ++ of e using resultants

  == add
    Pol ==> SparseUnivariatePolynomial

    import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, Pol(PolR))
    XtoY(Q : PolR) : Pol(PolR) == map(x+->monomial(x, 0), Q)

    P : Pol(PolR) := XtoY(definingPolynomial()$E)
    X : Pol(PolR) := monomial(monomial(1, 1)$PolR, 0)

    characteristicPolynomial(x : E) : PolR ==
       Qx : PolR := lift(x)
       -- on utilise le fait que resultant_Y (P(Y), X - Qx(Y))
       return resultant(P, X - XtoY(Qx))
```

———-

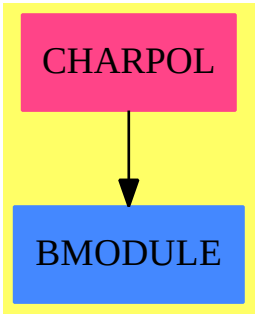
— CPIMA.dotabb —

```
"CPIMA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CPIMA"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"CPIMA" -> "MONOGEN"
```

———-

4.4 package CHARPOL CharacteristicPolynomialPackage

4.4.1 CharacteristicPolynomialPackage (CHARPOL)



Exports:
characteristicPolynomial

— **package CHARPOL CharacteristicPolynomialPackage** —

```
)abbrev package CHARPOL CharacteristicPolynomialPackage
++ Author: Barry Trager
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides a characteristicPolynomial function
++ for any matrix over a commutative ring.

CharacteristicPolynomialPackage(R:CommutativeRing):C == T where
   PI ==> PositiveInteger
   M ==> Matrix R
   C == with
      characteristicPolynomial: (M, R) -> R
        ++ characteristicPolynomial(m,r) computes the characteristic
        ++ polynomial of the matrix m evaluated at the point r.
        ++ In particular, if r is the polynomial 'x, then it returns
        ++ the characteristic polynomial expressed as a polynomial in 'x.
   T == add
```

```
        ---- characteristic polynomial  ----
    characteristicPolynomial(A:M,v:R) : R ==
      dimA :PI := (nrows A):PI
      dimA ^= ncols A => error " The matrix is not square"
      B:M:=zero(dimA,dimA)
      for i in 1..dimA repeat
        for j in 1..dimA repeat  B(i,j):=A(i,j)
        B(i,i) := B(i,i) - v
      determinant B
```

———-

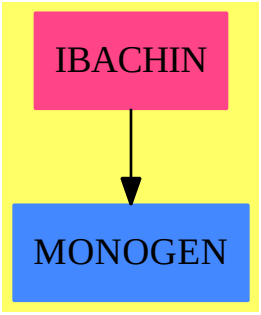
— **CHARPOL.dotabb** —

```
"CHARPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CHARPOL"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"CHARPOL" -> "BMODULE"
```

———-

4.5 package IBACHIN ChineseRemainderToolsForIntegralBases

4.5.1 ChineseRemainderToolsForIntegralBases (IBACHIN)



Exports:
chineseRemainder factorList listConjugateBases

— **package IBACHIN ChineseRemainderToolsForIntegralBases** —

```
)abbrev package IBACHIN ChineseRemainderToolsForIntegralBases
```

```
++ Author: Clifton Williamson
++ Date Created: 9 August 1993
++ Date Last Updated: 3 December 1993
++ Basic Operations: chineseRemainder, factorList
++ Related Domains: PAdicWildFunctionFieldIntegralBasis(K,R,UP,F)
++ Also See: WildFunctionFieldIntegralBasis, FunctionFieldIntegralBasis
++ AMS Classifications:
++ Keywords: function field, finite field, integral basis
++ Examples:
++ References:
++ Description:
++ This package has no description

ChineseRemainderToolsForIntegralBases(K,R,UP): Exports == Implementation where
  K  : FiniteFieldCategory
  R  : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R

  DDFACT ==> DistinctDegreeFactorize
  I      ==> Integer
  L      ==> List
  L2     ==> ListFunctions2
  Mat    ==> Matrix R
  NNI    ==> NonNegativeInteger
  PI     ==> PositiveInteger
  Q      ==> Fraction R
  SAE    ==> SimpleAlgebraicExtension
  SUP    ==> SparseUnivariatePolynomial
  SUP2   ==> SparseUnivariatePolynomialFunctions2
  Result ==> Record(basis: Mat, basisDen: R, basisInv: Mat)

  Exports ==> with
    factorList: (K,NNI,NNI,NNI) -> L SUP K
      ++ factorList(k,n,m,j) \undocumented

    listConjugateBases: (Result,NNI,NNI) -> List Result
      ++ listConjugateBases(bas,q,n) returns the list
      ++ \spad{[bas,bas^Frob,bas^(Frob^2),...bas^(Frob^(n-1))]}, where
      ++ \spad{Frob} raises the coefficients of all polynomials
      ++ appearing in the basis \spad{bas} to the \spad{q}th power.

    chineseRemainder: (List UP, List Result, NNI) -> Result
      ++ chineseRemainder(lu,lr,n) \undocumented

  Implementation ==> add
    import ModularHermitianRowReduction(R)
    import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

    applyFrobToMatrix: (Matrix R,NNI) -> Matrix R
    applyFrobToMatrix(mat,q) ==
```

```
    -- raises the coefficients of the polynomial entries of 'mat'
    -- to the qth power
    m := nrows mat; n := ncols mat
    ans : Matrix R := new(m,n,0)
    for i in 1..m repeat for j in 1..n repeat
      qsetelt_!(ans,i,j,map((k1:K):K +-> k1 ** q,qelt(mat,i,j)))
    ans

  listConjugateBases(bas,q,n) ==
    outList : List Result := list bas
    b := bas.basis; bInv := bas.basisInv; bDen := bas.basisDen
    for i in 1..(n-1) repeat
      b := applyFrobToMatrix(b,q)
      bInv := applyFrobToMatrix(bInv,q)
      bDen := map((k1:K):K +-> k1 ** q,bDen)
      newBasis : Result := [b,bDen,bInv]
      outList := concat(newBasis,outList)
    reverse_! outList

  factorList(a,q,n,k) ==
    coef : SUP K := monomial(a,0); xx : SUP K := monomial(1,1)
    outList : L SUP K := list((xx - coef)**k)
    for i in 1..(n-1) repeat
      coef := coef ** q
      outList := concat((xx - coef)**k,outList)
    reverse_! outList

  basisInfoToPolys: (Mat,R,R) -> L UP
  basisInfoToPolys(mat,lcm,den) ==
    n := nrows(mat) :: I; n1 := n - 1
    outList : L UP := empty()
    for i in 1..n repeat
      pp : UP := 0
      for j in 0..n1 repeat
        pp := pp + monomial((lcm quo den) * qelt(mat,i,j+1),j)
      outList := concat(pp,outList)
    reverse_! outList

  basesToPolyLists: (L Result,R) -> L L UP
  basesToPolyLists(basisList,lcm) ==
    [basisInfoToPolys(b.basis,lcm,b.basisDen) for b in basisList]

  OUT ==> OutputForm

  approximateExtendedEuclidean: (UP,UP,R,NNI) -> Record(coef1:UP,coef2:UP)
  approximateExtendedEuclidean(f,g,p,n) ==
    -- f and g are monic and relatively prime (mod p)
    -- function returns [coef1,coef2] such that
    -- coef1 * f + coef2 * g = 1 (mod p^n)
    sae := SAE(K,R,p)
```

```
      fSUP : SUP R := makeSUP f; gSUP : SUP R := makeSUP g
      fBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,fSUP)$SUP2(R,sae)
      gBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,gSUP)$SUP2(R,sae)
      ee := extendedEuclidean(fBar,gBar)
--      not one?(ee.generator) =>
      not (ee.generator = 1) =>
        error "polynomials aren't relatively prime"
      ss1 := ee.coef1; tt1 := ee.coef2
      s1 : SUP R := map((z1:sae):R +-> convert(z1)@R,ss1)$SUP2(sae,R); s := s1
      t1 : SUP R := map((z1:sae):R +-> convert(z1)@R,tt1)$SUP2(sae,R); t := t1
      pPower := p
      for i in 2..n repeat
        num := 1 - s * fSUP - t * gSUP
        rhs := (num exquo pPower) :: SUP R
        sigma := map((r1:R):R +-> r1 rem p,s1*rhs);
        tau   := map((r1:R):R +-> r1 rem p,t1*rhs)
        s := s + pPower * sigma; t := t + pPower * tau
        quorem := monicDivide(s,gSUP)
        pPower := pPower * p
        s := map((r1:R):R+->r1 rem pPower,quorem.remainder)
        t := map((r1:R):R+->r1 rem pPower,t + fSUP * (quorem.quotient))
      [unmakeSUP s,unmakeSUP t]

    --mapChineseToList: (L SUP Q,L SUP Q,I) -> L SUP Q
    --mapChineseToList(list,polyList,i) ==
    mapChineseToList: (L UP,L UP,I,R) -> L UP
    mapChineseToList(list,polyList,i,den) ==
      -- 'polyList' consists of MONIC polynomials
      -- computes a polynomial p such that p = pp (modulo polyList[i])
      -- and p = 0 (modulo polyList[j]) for j ~= i for each 'pp' in 'list'
      -- create polynomials
      q : UP := 1
      for j in 1..(i-1) repeat
        q := q * first polyList
        polyList := rest polyList
      p := first polyList
      polyList := rest polyList
      for j in (i+1).. while not empty? polyList repeat
        q := q * first polyList
        polyList := rest polyList
      --p := map((numer(#1) rem den)/1, p)
      --q := map((numer(#1) rem den)/1, q)
      -- 'den' is a power of an irreducible polynomial
      --!! make this computation more efficient!!
      factoredDen := factor(den)$DDFACT(K,R)
      prime := nthFactor(factoredDen,1)
      n := nthExponent(factoredDen,1) :: NNI
      invPoly := approximateExtendedEuclidean(q,p,prime,n).coef1
      -- monicDivide may be inefficient?
      [monicDivide(pp * invPoly * q,p * q).remainder for pp in list]
```

```
    polyListToMatrix: (L UP,NNI) -> Mat
    polyListToMatrix(polyList,n) ==
      mat : Mat := new(n,n,0)
      for i in 1..n for poly in polyList repeat
        while not zero? poly repeat
          mat(i,degree(poly) + 1) := leadingCoefficient poly
          poly := reductum poly
      mat

    chineseRemainder(factors,factorBases,n) ==
      denLCM : R := reduce("lcm",[base.basisDen for base in factorBases])
      denLCM = 1 => [scalarMatrix(n,1),1,scalarMatrix(n,1)]
      -- compute local basis polynomials with denominators cleared
      factorBasisPolyLists := basesToPolyLists(factorBases,denLCM)
      -- use Chinese remainder to compute basis polynomials w/o denominators
      basisPolyLists : L L UP := empty()
      for i in 1.. for pList in factorBasisPolyLists repeat
        polyList := mapChineseToList(pList,factors,i,denLCM)
        basisPolyLists := concat(polyList,basisPolyLists)
      basisPolys := concat reverse_! basisPolyLists
      mat := squareTop rowEchelon(polyListToMatrix(basisPolys,n),denLCM)
      matInv := UpTriBddDenomInv(mat,denLCM)
      [mat,denLCM,matInv]
```

———-

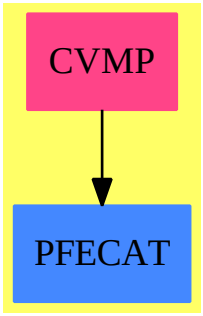
— IBACHIN.dotabb —

```
"IBACHIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBACHIN"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"IBACHIN" -> "MONOGEN"
```

———-

4.6 package CVMP CoerceVectorMatrixPackage

4.6.1 CoerceVectorMatrixPackage (CVMP)



Exports:
coerce coerceP

— package CVMP CoerceVectorMatrixPackage —

```
)abbrev package CVMP CoerceVectorMatrixPackage
++ Author: J. Grabmeier
++ Date Created: 26 June 1991
++ Date Last Updated: 26 June 1991
++ Related Constructors: GenericNonAssociativeAlgebra
++ Keywords:
++ Reference:
++ Description:
++ CoerceVectorMatrixPackage is an unexposed, technical package
++ for data conversions

CoerceVectorMatrixPackage(R : CommutativeRing): public == private where
  M2P ==> MatrixCategoryFunctions2(R, Vector R, Vector R, Matrix R, _
    Polynomial R, Vector Polynomial R, Vector Polynomial R, Matrix Polynomial R)
  M2FP ==> MatrixCategoryFunctions2(R, Vector R, Vector R, Matrix R, _
    Fraction Polynomial R, Vector Fraction Polynomial R, _
    Vector Fraction Polynomial R, Matrix Fraction Polynomial R)
  public ==> with
    coerceP: Vector Matrix R -> Vector Matrix Polynomial R
      ++ coerceP(v) coerces a vector v with entries in \spadtype{Matrix R}
      ++ as vector over \spadtype{Matrix Polynomial R}
    coerce: Vector Matrix R -> Vector Matrix Fraction Polynomial R
      ++ coerce(v) coerces a vector v with entries in \spadtype{Matrix R}
      ++ as vector over \spadtype{Matrix Fraction Polynomial R}
  private ==> add

    imbedFP : R -> Fraction Polynomial R
```

```
    imbedFP r == (r:: Polynomial R) :: Fraction Polynomial R

    imbedP : R -> Polynomial R
    imbedP r == (r:: Polynomial R)

    coerceP(g:Vector Matrix R) : Vector Matrix Polynomial R ==
      m2 : Matrix Polynomial R
      lim : List Matrix R := entries g
      l: List Matrix Polynomial R :=  []
      for m in lim repeat
        m2 :=  map(imbedP,m)$M2P
        l := cons(m2,l)
      vector reverse l

    coerce(g:Vector Matrix R) : Vector Matrix Fraction Polynomial R ==
      m3 : Matrix Fraction Polynomial R
      lim : List Matrix R := entries g
      l: List Matrix Fraction Polynomial R :=  []
      for m in lim repeat
        m3 :=  map(imbedFP,m)$M2FP
        l := cons(m3,l)
      vector reverse l
```

———-

— CVMP.dotabb —

```
"CVMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CVMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CVMP" -> "PFECAT"
```

———-

4.7 package COMBF CombinatorialFunction

— CombinatorialFunction.input —

```
)set break resume
)sys rm -f CombinatorialFunction.output
)spool CombinatorialFunction.output
)set message test on
)set message auto off
)clear all
--S 1 of 6
```

```
f := operator 'f
--R
--R
--R   (1)  f
--R                                                        Type: BasicOperator
--E 1

--S 2 of 6
D(product(f(i,x),i=1..m),x)
--R
--R
--R          m           m    f  (i,x)
--R        ++-++        --+    ,2
--R   (2)   | |   f(i,x)>     --------
--R         | |         --+    f(i,x)
--R        i= 1         i= 1
--R                                                 Type: Expression(Integer)
--E 2

--S 3 of 6
)set expose add constructor OutputForm
--R
--I   OutputForm is already explicitly exposed in frame frame0
--E 3

--S 4 of 6
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
--R
--R                                                                  Type: Void
--E 4

--S 5 of 6
displayRow(n)==output center blankSeparate pascalRow(n)
--R
--R                                                                  Type: Void
--E 5

--S 6 of 6
for i in 0..7 repeat displayRow i
--R
--R   Compiling function pascalRow with type NonNegativeInteger -> List(
--R      OutputForm)
--R   Compiling function displayRow with type NonNegativeInteger -> Void
--R                                      1
--R                                    1   1
--R                                  1   2   1
--R                                1   3   3   1
--R                              1   4   6   4   1
--R                            1   5   10  10   5   1
--R                          1   6   15  20  15   6   1
```

```
--R                  1    7   21   35   35   21    7    1
--R                                                        Type: Void
--E 6


)spool
)lisp (bye)
```

———-

— CombinatorialFunction.help —

```
=====================================================================
CombinatorialFunction examples
=====================================================================

f := operator 'f


   (1)  f
                                                        Type: BasicOperator

D(product(f(i,x),i=1..m),x)


          m           m    f  (i,x)
        ++-++         --+    ,2
   (2)   | |   f(i,x)>     --------
         | |           --+    f(i,x)
        i= 1         i= 1
                                                  Type: Expression Integer


The binomial(n, r) returns the number of subsets of r objects
taken among n objects, i.e. n!/(r! * (n-r)!)

The binomial coefficients are the coefficients of the series expansion
of a power of a binomial, that is

    n
   --+    / n \   k             n
   >     |     | x    =  (1 + x)
   --+    \ k /
   k= 0

This leads to the famous pascal triangle. First we expose the OutputForm
domain, which is normally hidden, so we can use it to format the lines.

  )set expose add constructor OutputForm
```

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:

```
  pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
```

and now we format the whole line so that it looks centered:

```
  displayRow(n)==output center blankSeparate pascalRow(n)
```

and we compute the triangle

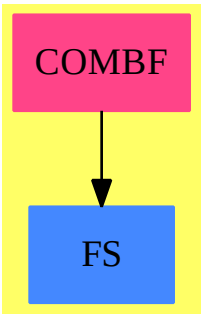
```
  for i in 0..7 repeat displayRow i
```

giving the pretty result:

```
   Compiling function pascalRow with type NonNegativeInteger -> List
      OutputForm
   Compiling function displayRow with type NonNegativeInteger -> Void
                                    1
                                  1   1
                                1   2   1
                              1   3   3   1
                            1   4   6   4   1
                          1   5  10  10   5   1
                        1   6  15  20  15   6   1
                      1   7  21  35  35  21   7   1
```

See Also:
o)show CombinatorialFunction
o)d op binomial
o)show OutputForm
o)help set

4.7.1 CombinatorialFunction (COMBF)



Exports:

belong?	binomial	factorial	factorials	iibinom
iidprod	iidsum	iifact	iiperm	iipow
ipow	permutation	product	summation	operator
product	summation	?**?		

binomial

We currently simplify binomial coefficients only for non-negative integral second argument, using the formula

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{1}{k!} \prod_{i=0..k-1} (n-i),$$

except if the second argument is symbolic: in this case binomial(n,n) is simplified to one.

Note that there are at least two different ways to define binomial coefficients for negative integral second argument. One way, particular suitable for combinatorics, is to set the binomial coefficient equal to zero for negative second argument. This is, partially, also the approach taken in combinat.spad, where we find

```
binomial(n, m) ==
   n < 0 or m < 0 or m > n => 0
   m = 0 => 1
```

Of course, here n and m are integers. This definition agrees with the recurrence

$$\left(\begin{array}{c} n \\ k \end{array} \right) + \left(\begin{array}{c} n \\ k+1 \end{array} \right) = \left(\begin{array}{c} n+1 \\ k+1 \end{array} \right).$$

Alternatively, one can use the formula

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)},$$

and leave the case where $k \in \mathbf{Z}$, $n \in \mathbf{Z}$ and $k \leq n < 0$ undefined, since the limit does not exist in this case:

Since we then have that $n - k + 1 \geq 1$, $\Gamma(n - k + 1)$ is finite. So it is sufficient to consider $\frac{\Gamma(n+1)}{\Gamma(k+1)}$. On the one hand, we have

$$\lim_{n_0 \to n} \lim_{k_0 \to k} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)} = 0,$$

since for any non-integral n_0, $\Gamma(n_0 + 1)$ is finite. On the other hand,

$$\lim_{k_0 \to k} \lim_{n_0 \to n} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)}$$

does not exist, since for non-integral k_0, $\Gamma(k_0 + 1)$ is finite while $\Gamma(n_0 + 1)$ is unbounded.

However, since for $k \in \mathbf{Z}$, $n \in \mathbf{Z}$ and $0 < k < n$ both definitions agree, one could also combine them. This is what, for example, Mathematica does. It seems that MuPAD sets binomial(n,n)=1 for all arguments n, and returns binomial(-2, n) unevaluated. Provisos may help here.

dvsum and dvdsum

The dvsum and dvdsum operations implement differentiation of sums with and without bounds. Note that the function

$$n \mapsto \sum_{k=1}^{n} f(k, n)$$

is well defined only for integral values of n greater than or equal to zero. There is not even consensus how to define this function for $n < 0$. Thus, it is not differentiable. Therefore, we need to check whether we erroneously are differentiating with respect to the upper bound or the lower bound, where the same reasoning holds.

Differentiating a sum with respect to its indexing variable correctly gives zero. This is due to the introduction of dummy variables in the internal representation of a sum: the operator `%defsum` takes 5 arguments, namely

1. the summands, where each occurrence of the indexing variable is replaced by
2. the dummy variable,
3. the indexing variable,
4. the lower bound, and
5. the upper bound.

dvprod and dvdprod

The dvprod and dvdprod operations implement differentiation of products with and without bounds. Note again, that we cannot even properly define products with bounds that are not integral.

To differentiate the product, we use Leibniz rule:

$$\frac{d}{dx}\prod_{i=a}^{b} f(i,x) = \sum_{i=a}^{b} \frac{\frac{d}{dx}f(i,x)}{f(i,x)} \prod_{i=a}^{b} f(i,x)$$

There is one situation where this definition might produce wrong results, namely when the product is zero, but axiom failed to recognize it: in this case,

$$\frac{d}{dx}f(i,x)/f(i,x)$$

is undefined for some i. However, I was not able to come up with an example. The alternative definition

$$\frac{d}{dx}\prod_{i=a}^{b} f(i,x) = \sum_{i=a}^{b} \left(\frac{d}{dx}f(i,x)\right) \prod_{j=a,j\neq i}^{b} f(j,x)$$

has the slight (display) problem that we would have to come up with a new index variable, which looks very ugly. Furthermore, it seems to me that more simplifications will occur with the first definition.

```
f := operator 'f
D(product(f(i,x),i=1..m),x)
```

dvpow2

The dvpow2 operation implements the differentiation of the power operator `%power` with respect to its second argument, i.e., the exponent. It uses the formula

$$\frac{d}{dx}g(y)^x = \frac{d}{dx}e^{x\log g(y)} = \log g(y) g(y)^x.$$

If $g(y)$ equals zero, this formula is not valid, since the logarithm is not defined there. Although strictly speaking 0^x is not differentiable at zero, we return zero for convenience.

— package COMBF CombinatorialFunction —

```
)abbrev package COMBF CombinatorialFunction
++ Author: Manuel Bronstein, Martin Rubey
++ Date Created: 2 Aug 1988
++ Keywords: combinatorial, function, factorial.
++ Date Last Updated: 30 October 2005
```

```
++ Description:
++ Provides combinatorial functions over an integral domain.

CombinatorialFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R

  OP  ==> BasicOperator
  K   ==> Kernel F
  SE  ==> Symbol
  O   ==> OutputForm
  SMP ==> SparseMultivariatePolynomial(R, K)
  Z   ==> Integer

  POWER        ==> "%power"::Symbol
  OPEXP        ==> "exp"::Symbol
  SPECIALDIFF  ==> "%specialDiff"
  SPECIALDISP  ==> "%specialDisp"
  SPECIALEQUAL ==> "%specialEqual"

  Exports ==> with
    belong?    : OP -> Boolean
      ++ belong?(op) is true if op is a combinatorial operator;
    operator   : OP -> OP
      ++ operator(op) returns a copy of op with the domain-dependent
      ++ properties appropriate for F;
      ++ error if op is not a combinatorial operator;
    "**"       : (F, F) -> F
      ++ a ** b is the formal exponential a**b;
    binomial   : (F, F) -> F
      ++ binomial(n, r) returns the number of subsets of r objects
      ++ taken among n objects, i.e. n!/(r! * (n-r)!);
      ++
      ++X [binomial(5,i) for i in 0..5]
    permutation: (F, F) -> F
      ++ permutation(n, r) returns the number of permutations of
      ++ n objects taken r at a time, i.e. n!/(n-r)!;
    factorial  : F -> F
      ++ factorial(n) returns the factorial of n, i.e. n!;
    factorials : F -> F
      ++ factorials(f) rewrites the permutations and binomials in f
      ++ in terms of factorials;
    factorials : (F, SE) -> F
      ++ factorials(f, x) rewrites the permutations and binomials in f
      ++ involving x in terms of factorials;
    summation  : (F, SE) -> F
      ++ summation(f(n), n) returns the formal sum S(n) which verifies
      ++ S(n+1) - S(n) = f(n);
    summation  : (F, SegmentBinding F)  -> F
      ++ summation(f(n), n = a..b) returns f(a) + ... + f(b) as a
```

```
      ++ formal sum;
    product    : (F, SE) -> F
      ++ product(f(n), n) returns the formal product P(n) which verifies
      ++ P(n+1)/P(n) = f(n);
    product    : (F, SegmentBinding  F) -> F
      ++ product(f(n), n = a..b) returns f(a) * ... * f(b) as a
      ++ formal product;
    iifact     : F -> F
      ++ iifact(x) should be local but conditional;
    iibinom    : List F -> F
      ++ iibinom(l) should be local but conditional;
    iiperm     : List F -> F
      ++ iiperm(l) should be local but conditional;
    iipow      : List F -> F
      ++ iipow(l) should be local but conditional;
    iidsum     : List F -> F
      ++ iidsum(l) should be local but conditional;
    iidprod    : List F -> F
      ++ iidprod(l) should be local but conditional;
    ipow       : List F -> F
      ++ ipow(l) should be local but conditional;

  Implementation ==> add
    ifact     : F -> F
    iiipow    : List F -> F
    iperm     : List F -> F
    ibinom    : List F -> F
    isum      : List F -> F
    idsum     : List F -> F
    iprod     : List F -> F
    idprod    : List F -> F
    dsum      : List F -> O
    ddsum     : List F -> O
    dprod     : List F -> O
    ddprod    : List F -> O
    equalsumprod  : (K, K) -> Boolean
    equaldsumprod : (K, K) -> Boolean
    fourth    : List F -> F
    dvpow1    : List F -> F
    dvpow2    : List F -> F
    summand   : List F -> F
    dvsum     : (List F, SE) -> F
    dvdsum    : (List F, SE) -> F
    dvprod    : (List F, SE) -> F
    dvdprod   : (List F, SE) -> F
    facts     : (F, List SE) -> F
    K2fact    : (K, List SE) -> F
    smpfact   : (SMP, List SE) -> F

-- This macro will be used in product and summation, both the 5 and 3
```

```
-- argument forms. It is used to introduce a dummy variable in place of the
-- summation index within the summands. This in turn is necessary to keep the
-- indexing variable local, circumventing problems, for example, with
-- differentiation.

    dummy == new()$SE :: F

    opfact  := operator("factorial"::Symbol)$CommonOperators
    opperm  := operator("permutation"::Symbol)$CommonOperators
    opbinom := operator("binomial"::Symbol)$CommonOperators
    opsum   := operator("summation"::Symbol)$CommonOperators
    opdsum  := operator("%defsum"::Symbol)$CommonOperators
    opprod  := operator("product"::Symbol)$CommonOperators
    opdprod := operator("%defprod"::Symbol)$CommonOperators
    oppow   := operator(POWER::Symbol)$CommonOperators

    factorial x          == opfact x
    binomial(x, y)       == opbinom [x, y]
    permutation(x, y)    == opperm [x, y]

    import F
    import Kernel F

    number?(x:F):Boolean ==
      if R has RetractableTo(Z) then
        ground?(x) or
         ((retractIfCan(x)@Union(Fraction(Z),"failed")) case Fraction(Z))
      else
        ground?(x)

    x ** y               ==
      -- Do some basic simplifications
      is?(x,POWER) =>
        args : List F := argument first kernels x
        not(#args = 2) => error "Too many arguments to **"
        number?(first args) and number?(y) =>
          oppow [first(args)**y, second args]
        oppow [first args, (second args)* y]
      -- Generic case
      exp : Union(Record(val:F,exponent:Z),"failed") := isPower x
      exp case Record(val:F,exponent:Z) =>
        expr := exp::Record(val:F,exponent:Z)
        oppow [expr.val, (expr.exponent)*y]
      oppow [x, y]

    belong? op           == has?(op, "comb")
    fourth l             == third rest l
    dvpow1 l             == second(l) * first(l) ** (second l - 1)
    factorials x         == facts(x, variables x)
    factorials(x, v)     == facts(x, [v])
```

```
    facts(x, l)          == smpfact(numer x, l) / smpfact(denom x, l)
    summand l            == eval(first l, retract(second l)@K, third l)

    product(x:F, i:SE) ==
      dm := dummy
      opprod [eval(x, k := kernel(i)$K, dm), dm, k::F]

    summation(x:F, i:SE) ==
      dm := dummy
      opsum [eval(x, k := kernel(i)$K, dm), dm, k::F]

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.

    dvsum(l, x) ==
      opsum [differentiate(first l, x), second l, third l]

    dvdsum(l, x) ==
      x = retract(y := third l)@SE => 0
      if member?(x, variables(h := third rest rest l)) or
         member?(x, variables(g := third rest l)) then
        error "a sum cannot be differentiated with respect to a bound"
      else
        opdsum [differentiate(first l, x), second l, y, g, h]

    dvprod(l, x) ==
      dm := retract(dummy)@SE
      f := eval(first l, retract(second l)@K, dm::F)
      p := product(f, dm)

      opsum [differentiate(first l, x)/first l * p, second l, third l]


    dvdprod(l, x) ==
      x = retract(y := third l)@SE => 0
      if member?(x, variables(h := third rest rest l)) or
         member?(x, variables(g := third rest l)) then
        error "a product cannot be differentiated with respect to a bound"
      else
        opdsum cons(differentiate(first l, x)/first l, rest l) * opdprod l

-- These four operations handle the conversion of sums and products to
-- OutputForm

    dprod l ==
      prod(summand(l)::O, third(l)::O)

    ddprod l ==
      prod(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)
```

```
    dsum l ==
      sum(summand(l)::O, third(l)::O)

    ddsum l ==
      sum(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

-- The two operations handle the testing for equality of sums and products.
-- The corresponding property \verb|%specialEqual| set below is checked in
-- Kernel. Note that we can assume that the operators are equal, since this is
-- checked in Kernel itself.

    equalsumprod(s1, s2) ==
      l1 := argument s1
      l2 := argument s2
      (eval(first l1, retract(second l1)@K, second l2) = first l2)

    equaldsumprod(s1, s2) ==
      l1 := argument s1
      l2 := argument s2
      ((third rest l1 = third rest l2) and
       (third rest rest l1 = third rest rest l2) and
       (eval(first l1, retract(second l1)@K, second l2) = first l2))

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.

    product(x:F, s:SegmentBinding F) ==
      k := kernel(variable s)$K
      dm := dummy
      opdprod [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

    summation(x:F, s:SegmentBinding F) ==
      k := kernel(variable s)$K
      dm := dummy
      opdsum [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

    smpfact(p, l) ==
      map(x +-> K2fact(x, l), y+->y::F, p)_
        $PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)

    K2fact(k, l) ==
      empty? [v for v in variables(kf := k::F) | member?(v, l)] => kf
      empty?(args:List F := [facts(a, l) for a in argument k]) => kf
      is?(k, opperm) =>
        factorial(n := first args) / factorial(n - second args)
      is?(k, opbinom) =>
        n := first args
        p := second args
        factorial(n) / (factorial(p) * factorial(n-p))
      (operator k) args
```

```
    operator op ==
      is?(op, "factorial"::Symbol)   => opfact
      is?(op, "permutation"::Symbol) => opperm
      is?(op, "binomial"::Symbol)    => opbinom
      is?(op, "summation"::Symbol)   => opsum
      is?(op, "%defsum"::Symbol)     => opdsum
      is?(op, "product"::Symbol)     => opprod
      is?(op, "%defprod"::Symbol)    => opdprod
      is?(op, POWER)                 => oppow
      error "Not a combinatorial operator"

    iprod l ==
      zero? first l => 0
--      one? first l => 1
      (first l = 1) => 1
      kernel(opprod, l)

    isum l ==
      zero? first l => 0
      kernel(opsum, l)

    idprod l ==
      member?(retract(second l)@SE, variables first l) =>
        kernel(opdprod, l)
      first(l) ** (fourth rest l - fourth l + 1)

    idsum l ==
      member?(retract(second l)@SE, variables first l) =>
        kernel(opdsum, l)
      first(l) * (fourth rest l - fourth l + 1)

    ifact x ==
--      zero? x or one? x => 1
      zero? x or (x = 1) => 1
      kernel(opfact, x)

    ibinom l ==
      n := first l
      ((p := second l) = 0) or (p = n) => 1
--      one? p or (p = n - 1) => n
      (p = 1) or (p = n - 1) => n
      kernel(opbinom, l)

    iperm l ==
      zero? second l => 1
      kernel(opperm, l)

    if R has RetractableTo Z then
      iidsum l ==
```

```
        (r1:=retractIfCan(fourth l)@Union(Z,"failed"))
         case "failed" or
          (r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
            case "failed" or
             (k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
               => idsum l
        +/[eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

      iidprod l ==
        (r1:=retractIfCan(fourth l)@Union(Z,"failed"))
         case "failed" or
          (r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
            case "failed" or
             (k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
               => idprod l
        */[eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

      iiipow l ==
          (u := isExpt(x := first l, OPEXP)) case "failed" => kernel(oppow, l)
          rec := u::Record(var: K, exponent: Z)
          y := first argument(rec.var)
          (r := retractIfCan(y)@Union(Fraction Z, "failed")) case
              "failed" => kernel(oppow, l)
          (operator(rec.var)) (rec.exponent * y * second l)

      if F has RadicalCategory then
        ipow l ==
          (r := retractIfCan(second l)@Union(Fraction Z,"failed"))
            case "failed" => iiipow l
          first(l) ** (r::Fraction(Z))
      else
        ipow l ==
          (r := retractIfCan(second l)@Union(Z, "failed"))
            case "failed" => iiipow l
          first(l) ** (r::Z)

    else
      ipow l ==
        zero?(x := first l) =>
          zero? second l => error "0 ** 0"
          0
--        one? x or zero?(n := second l) => 1
        (x = 1) or zero?(n: F := second l) => 1
--        one? n => x
        (n = 1) => x
        (u := isExpt(x, OPEXP)) case "failed" => kernel(oppow, l)
        rec := u::Record(var: K, exponent: Z)
--        one?(y := first argument(rec.var)) or y = -1 =>
        ((y := first argument(rec.var))=1) or y = -1 =>
            (operator(rec.var)) (rec.exponent * y * n)
```

```
        kernel(oppow, l)

    if R has CombinatorialFunctionCategory then
      iifact x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => ifact x
        factorial(r::R)::F

      iiperm l ==
        (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
          (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
            => iperm l
        permutation(r1::R, r2::R)::F

      if R has RetractableTo(Z) and F has Algebra(Fraction(Z)) then
         iibinom l ==
           (s:=retractIfCan(second l)@Union(R,"failed")) case R and
              (t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
                ans:=1::F
                for i in 0..t-1 repeat
                    ans:=ans*(first l - i::R::F)
                (1/factorial t) * ans
           (s:=retractIfCan(first l-second l)@Union(R,"failed")) case R and
             (t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
                ans:=1::F
                for i in 1..t repeat
                    ans:=ans*(second l+i::R::F)
                (1/factorial t) * ans
           (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
             (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
               => ibinom l
           binomial(r1::R, r2::R)::F

-- iibinom checks those cases in which the binomial coefficient may
-- be evaluated explicitly. Currently, the naive iterative algorithm is
-- used to calculate the coefficient, there is room for improvement here.

      else
         iibinom l ==
           (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
             (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
               => ibinom l
           binomial(r1::R, r2::R)::F

    else
      iifact x  == ifact x
      iibinom l == ibinom l
      iiperm l  == iperm l

    if R has ElementaryFunctionCategory then
      iipow l ==
```

```
        (r1:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
          (r2:=retractIfCan(second l)@Union(R,"failed")) case "failed"
            => ipow l
        (r1::R ** r2::R)::F
    else
      iipow l == ipow l

    if F has ElementaryFunctionCategory then
      dvpow2 l == if zero?(first l) then
                    0
                  else
                    log(first l) * first(l) ** second(l)

    evaluate(opfact, iifact)$BasicOperatorFunctions1(F)
    evaluate(oppow, iipow)
    evaluate(opperm, iiperm)
    evaluate(opbinom, iibinom)
    evaluate(opsum, isum)
    evaluate(opdsum, iidsum)
    evaluate(opprod, iprod)
    evaluate(opdprod, iidprod)
    derivative(oppow, [dvpow1, dvpow2])

-- These four properties define special differentiation rules for sums and
-- products.

    setProperty(opsum,   SPECIALDIFF, dvsum@((List F, SE) -> F) pretend None)
    setProperty(opdsum,  SPECIALDIFF, dvdsum@((List F, SE)->F) pretend None)
    setProperty(opprod,  SPECIALDIFF, dvprod@((List F, SE)->F) pretend None)
    setProperty(opdprod, SPECIALDIFF, dvdprod@((List F, SE)->F) pretend None)

-- Set the properties for displaying sums and products and testing for
-- equality.

    setProperty(opsum,   SPECIALDISP, dsum@(List F -> O) pretend None)
    setProperty(opdsum,  SPECIALDISP, ddsum@(List F -> O) pretend None)
    setProperty(opprod,  SPECIALDISP, dprod@(List F -> O) pretend None)
    setProperty(opdprod, SPECIALDISP, ddprod@(List F -> O) pretend None)
    setProperty(opsum,   SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
    setProperty(opdsum,  SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)
    setProperty(opprod,  SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
    setProperty(opdprod, SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)
```

— COMBF.dotabb —

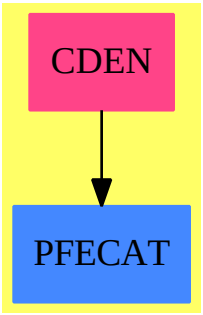
```
"COMBF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMBF"]
```

```
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMBF" -> "FS"
```

———-

4.8 package CDEN CommonDenominator

4.8.1 CommonDenominator (CDEN)



Exports:
clearDenominator commonDenominator splitDenominator

— package CDEN CommonDenominator —

```
)abbrev package CDEN CommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Keywords: gcd, quotient, common, denominator.
++ Description:
++ CommonDenominator provides functions to compute the
++ common denominator of a finite linear aggregate of elements of
++ the quotient field of an integral domain.

CommonDenominator(R, Q, A): Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R
  A: FiniteLinearAggregate Q

  Exports ==> with
    commonDenominator: A -> R
      ++ commonDenominator([q1,...,qn]) returns a common denominator
      ++ d for q1,...,qn.
    clearDenominator : A -> A
```

```
      ++ clearDenominator([q1,...,qn]) returns \spad{[p1,...,pn]} such that
      ++ \spad{qi = pi/d} where d is a common denominator for the qi's.
    splitDenominator : A -> Record(num: A, den: R)
      ++ splitDenominator([q1,...,qn]) returns
      ++ \spad{[[p1,...,pn], d]} such that
      ++ \spad{qi = pi/d} and d is a common denominator for the qi's.

  Implementation ==> add
    clearDenominator l ==
      d := commonDenominator l
      map(x+->numer(d*x)::Q, l)

    splitDenominator l ==
      d := commonDenominator l
      [map(x+->numer(d*x)::Q, l), d]

    if R has GcdDomain then
      qlcm: (Q, Q) -> Q

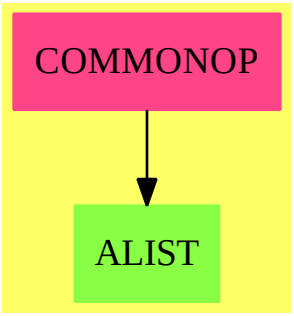
      qlcm(a, b)          == lcm(numer a, numer b)::Q
      commonDenominator l == numer reduce(qlcm, map(x+->denom(x)::Q, l), 1)
    else
      commonDenominator l == numer reduce("*", map(x+->denom(x)::Q, l), 1)
```

— CDEN.dotabb —

```
"CDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CDEN" -> "PFECAT"
```

4.9 package COMMONOP CommonOperators

4.9.1 CommonOperators (COMMONOP)



Exports:
operator

— package COMMONOP CommonOperators —

```
)abbrev package COMMONOP CommonOperators
++ Author: Manuel Bronstein
++ Date Created: 25 Mar 1988
++ Date Last Updated: 2 December 1994
++ Keywords: operator.
++ Description:
++ This package exports the elementary operators, with some semantics
++ already attached to them. The semantics that is attached here is not
++ dependent on the set in which the operators will be applied.

CommonOperators(): Exports == Implementation where
  OP  ==> BasicOperator
  O   ==> OutputForm
  POWER ==> "%power"::Symbol
  ALGOP ==> "%alg"
  EVEN  ==> "even"
  ODD   ==> "odd"
  DUMMYVAR ==> "%dummyVar"

  Exports ==> with
    operator: Symbol -> OP
        ++ operator(s) returns an operator with name s, with the
        ++ appropriate semantics if s is known. If s is not known,
        ++ the result has no semantics.

  Implementation ==> add
    dpi        : List O -> O
```

```
dgamma     : List O -> O
dquote     : List O -> O
dexp       : O -> O
dfact      : O -> O
startUp    : Boolean -> Void
setDummyVar: (OP, NonNegativeInteger) -> OP

brandNew?:Reference(Boolean) := ref true

opalg   := operator("rootOf"::Symbol, 2)$OP
oproot  := operator("nthRoot"::Symbol, 2)
oppi    := operator("pi"::Symbol, 0)
oplog   := operator("log"::Symbol, 1)
opexp   := operator("exp"::Symbol, 1)
opabs   := operator("abs"::Symbol, 1)
opsin   := operator("sin"::Symbol, 1)
opcos   := operator("cos"::Symbol, 1)
optan   := operator("tan"::Symbol, 1)
opcot   := operator("cot"::Symbol, 1)
opsec   := operator("sec"::Symbol, 1)
opcsc   := operator("csc"::Symbol, 1)
opasin  := operator("asin"::Symbol, 1)
opacos  := operator("acos"::Symbol, 1)
opatan  := operator("atan"::Symbol, 1)
opacot  := operator("acot"::Symbol, 1)
opasec  := operator("asec"::Symbol, 1)
opacsc  := operator("acsc"::Symbol, 1)
opsinh  := operator("sinh"::Symbol, 1)
opcosh  := operator("cosh"::Symbol, 1)
optanh  := operator("tanh"::Symbol, 1)
opcoth  := operator("coth"::Symbol, 1)
opsech  := operator("sech"::Symbol, 1)
opcsch  := operator("csch"::Symbol, 1)
opasinh := operator("asinh"::Symbol, 1)
opacosh := operator("acosh"::Symbol, 1)
opatanh := operator("atanh"::Symbol, 1)
opacoth := operator("acoth"::Symbol, 1)
opasech := operator("asech"::Symbol, 1)
opacsch := operator("acsch"::Symbol, 1)
opbox   := operator("%box"::Symbol)$OP
oppren  := operator("%paren"::Symbol)$OP
opquote := operator("applyQuote"::Symbol)$OP
opdiff  := operator("%diff"::Symbol, 3)
opsi    := operator("Si"::Symbol, 1)
opci    := operator("Ci"::Symbol, 1)
opei    := operator("Ei"::Symbol, 1)
opli    := operator("li"::Symbol, 1)
operf   := operator("erf"::Symbol, 1)
opli2   := operator("dilog"::Symbol, 1)
opfis   := operator("fresnelS"::Symbol, 1)
```

```
    opfic   := operator("fresnelC"::Symbol, 1)
    opGamma     := operator("Gamma"::Symbol, 1)
    opGamma2    := operator("Gamma2"::Symbol, 2)
    opBeta      := operator("Beta"::Symbol, 2)
    opdigamma   := operator("digamma"::Symbol, 1)
    oppolygamma := operator("polygamma"::Symbol, 2)
    opBesselJ   := operator("besselJ"::Symbol, 2)
    opBesselY   := operator("besselY"::Symbol, 2)
    opBesselI   := operator("besselI"::Symbol, 2)
    opBesselK   := operator("besselK"::Symbol, 2)
    opAiryAi    := operator("airyAi"::Symbol,  1)
    opAiryBi    := operator("airyBi"::Symbol , 1)
    opint   := operator("integral"::Symbol, 3)
    opdint  := operator("%defint"::Symbol, 5)
    opfact  := operator("factorial"::Symbol, 1)
    opperm  := operator("permutation"::Symbol, 2)
    opbinom := operator("binomial"::Symbol, 2)
    oppow   := operator(POWER, 2)
    opsum   := operator("summation"::Symbol, 3)
    opdsum  := operator("%defsum"::Symbol, 5)
    opprod  := operator("product"::Symbol, 3)
    opdprod := operator("%defprod"::Symbol, 5)

    algop   := [oproot, opalg]$List(OP)
    rtrigop := [opsin, opcos, optan, opcot, opsec, opcsc,
                        opasin, opacos, opatan, opacot, opasec, opacsc]
    htrigop := [opsinh, opcosh, optanh, opcoth, opsech, opcsch,
                    opasinh, opacosh, opatanh, opacoth, opasech, opacsch]
    trigop  := concat(rtrigop, htrigop)
    elemop  := concat(trigop, [oppi, oplog, opexp])
    primop  := [opei, opli, opsi, opci, operf, opli2, opint, opdint,
                opfis, opfic]
    combop  := [opfact, opperm, opbinom, oppow,
                                        opsum, opdsum, opprod, opdprod]
    specop  := [opGamma, opGamma2, opBeta, opdigamma, oppolygamma, opabs,
                opBesselJ, opBesselY, opBesselI, opBesselK, opAiryAi,
                 opAiryBi]
    anyop   := [oppren, opdiff, opbox, opquote]
    allop   := concat(concat(concat(concat(concat(
                            algop,elemop),primop),combop),specop),anyop)

-- odd and even operators, must be maintained current!
    evenop := [opcos, opsec, opcosh, opsech, opabs]
    oddop  := [opsin, opcsc, optan, opcot, opasin, opacsc, opatan,
               opsinh, opcsch, optanh, opcoth, opasinh, opacsch,opatanh,opacoth,
                opsi, operf]

-- operators whose second argument is a dummy variable
    dummyvarop1 := [opdiff,opalg, opint, opsum, opprod]
-- operators whose second and third arguments are dummy variables
```

```
dummyvarop2 := [opdint, opdsum, opdprod]

operator s ==
  if (deref brandNew?) then startUp false
  for op in allop repeat
    is?(op, s) => return copy op
  operator(s)$OP

dpi l    == "%pi"::Symbol::O
dfact x  == postfix("!"::Symbol::O, (ATOM(x)$Lisp => x; paren x))
dquote l == prefix(quote(first(l)::O), rest l)
dgamma l == prefix(hconcat("|"::Symbol::O, overbar(" "::Symbol::O)), l)
setDummyVar(op, n) == setProperty(op, DUMMYVAR, n pretend None)

dexp x ==
  e := "%e"::Symbol::O
  x = 1::O => e
  e ** x

fsupersub(x:List O):O == supersub("A"::Symbol::O, x)
fbinomial(x:List O):O == binomial(first x, second x)
fpower(x:List O):O    == first(x) ** second(x)
fsum(x:List O):O      == sum(first x, second x, third x)
fprod(x:List O):O     == prod(first x, second x, third x)

fint(x:List O):O      ==
   int(first x * hconcat("d"::Symbol::O, second x),empty(), third x)

fpren(x:List InputForm):InputForm  ==
   convert concat(convert("("::Symbol)@InputForm,
                        concat(x, convert(")"::Symbol)@InputForm))

fpow(x:List InputForm):InputForm  ==
   convert concat(convert("**"::Symbol)@InputForm, x)

froot(x:List InputForm):InputForm  ==
   convert [convert("**"::Symbol)@InputForm, first x, 1 / second x]

startUp b ==
  brandNew?() := b
  display(oppren,   paren)
  display(opbox,    commaSeparate)
  display(oppi,     dpi)
  display(opexp,    dexp)
  display(opGamma,  dgamma)
  display(opGamma2, dgamma)
  display(opfact,   dfact)
  display(opquote,  dquote)
  display(opperm,   fsupersub)
  display(opbinom,  fbinomial)
```

```
display(oppow,    fpower)
display(opsum,    fsum)
display(opprod,   fprod)
display(opint,    fint)
input(oppren,     fpren)
input(oppow,      fpow)
input(oproot,     froot)
for op in algop   repeat assert(op, ALGOP)
for op in rtrigop repeat assert(op, "rtrig")
for op in htrigop repeat assert(op, "htrig")
for op in trigop  repeat assert(op, "trig")
for op in elemop  repeat assert(op, "elem")
for op in primop  repeat assert(op, "prim")
for op in combop  repeat assert(op, "comb")
for op in specop  repeat assert(op, "special")
for op in anyop   repeat assert(op, "any")
for op in evenop  repeat assert(op, EVEN)
for op in oddop   repeat assert(op, ODD)
for op in dummyvarop1 repeat setDummyVar(op, 1)
for op in dummyvarop2 repeat setDummyVar(op, 2)
assert(oppren, "linear")
void
```

———-

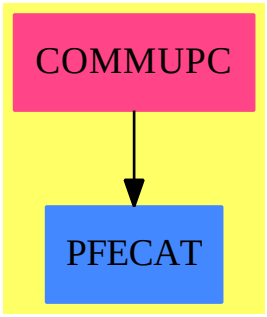
— **COMMONOP.dotabb** —

```
"COMMONOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMMONOP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"COMMONOP" -> "ALIST"
```

———-

4.10 package COMMUPC CommuteUnivariatePolynomialCategory

4.10.1 CommuteUnivariatePolynomialCategory (COMMUPC)



Exports:
swap

— package COMMUPC CommuteUnivariatePolynomialCategory —

```
)abbrev package COMMUPC CommuteUnivariatePolynomialCategory
++ Author: Manuel Bronstein
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ A package for swapping the order of two variables in a tower of two
++ UnivariatePolynomialCategory extensions.

CommuteUnivariatePolynomialCategory(R, UP, UPUP): Exports == Impl where
  R   : Ring
  UP  : UnivariatePolynomialCategory R
  UPUP: UnivariatePolynomialCategory UP

  N ==> NonNegativeInteger

  Exports ==> with
    swap: UPUP -> UPUP
      ++ swap(p(x,y)) returns p(y,x).
```

```
  Impl ==> add
    makePoly: (UP, N) -> UPUP

-- converts P(x,y) to P(y,x)
    swap poly ==
      ans:UPUP := 0
      while poly ^= 0 repeat
        ans  := ans + makePoly(leadingCoefficient poly, degree poly)
        poly := reductum poly
      ans

    makePoly(poly, d) ==
      ans:UPUP := 0
      while poly ^= 0 repeat
        ans  := ans +
             monomial(monomial(leadingCoefficient poly, d), degree poly)
        poly := reductum poly
      ans
```

———-

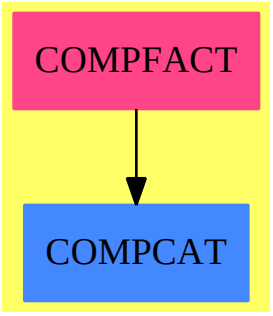
— **COMMUPC.dotabb** —

```
"COMMUPC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMMUPC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"COMMUPC" -> "PFECAT"
```

———-

4.11 package COMPFACT ComplexFactorization

4.11.1 ComplexFactorization (COMPFACT)



Exports:
factor

— package COMPFACT ComplexFactorization —

```
)abbrev package COMPFACT ComplexFactorization
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: Complex, UnivariatePolynomial
++ Also See:
++ AMS Classifications:
++ Keywords: complex, polynomial factorization, factor
++ References:
++ Description:
++ This package has no description

ComplexFactorization(RR,PR) : C == T where
  RR   :    EuclideanDomain   -- R is Z or Q
  PR   :    UnivariatePolynomialCategory Complex RR
  R    ==>  Complex RR
  I    ==>  Integer
  RN   ==>  Fraction I
  GI   ==>  Complex I
  GRN  ==>  Complex RN


  C  == with

     factor        :   PR   ->  Factored PR
       ++ factor(p) factorizes the polynomial p with complex coefficients.

  T  == add
     SUP    ==> SparseUnivariatePolynomial
     fUnion ==> Union("nil", "sqfr", "irred", "prime")
     FF     ==> Record(flg:fUnion, fctr:PR, xpnt:Integer)
     SAEF   :=  SimpleAlgebraicExtensionAlgFactor(SUP RN,GRN,SUP GRN)
     UPCF2  :=  UnivariatePolynomialCategoryFunctions2(R,PR,GRN,SUP GRN)
     UPCFB  :=  UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,R,PR)

     myMap(r:R) : GRN ==
       R is GI   =>
         cr :GI := r pretend GI
         complex((real cr)::RN,(imag cr)::RN)
       R is GRN  => r pretend GRN

     compND(cc:GRN):Record(cnum:GI,cden:Integer) ==
       ccr:=real cc
```

```
    cci:=imag cc
    dccr:=denom ccr
    dcci:=denom cci
    ccd:=lcm(dccr,dcci)
    [complex(((ccd exquo dccr)::Integer)*numer ccr,
             ((ccd exquo dcci)::Integer)*numer cci),ccd]

  conv(f:SUP GRN) :Record(convP:SUP GI, convD:RN) ==
    pris:SUP GI :=0
    dris:Integer:=1
    dris1:Integer:=1
    pdris:Integer:=1
    for i in 0..(degree f) repeat
      (cf:= coefficient(f,i)) = 0 => "next i"
      cdf:=compND cf
      dris:=lcm(cdf.cden,dris1)
      pris:=((dris exquo dris1)::Integer)*pris +
            ((dris exquo cdf.cden)::Integer)*
              monomial(cdf.cnum,i)$(SUP GI)
      dris1:=dris
    [pris,dris::RN]

  backConv(ffr:Factored SUP GRN) : Factored PR ==
    R is GRN =>
      makeFR((unit ffr) pretend PR,[[f.flg,(f.fctr) pretend PR,f.xpnt]
                                    for f in factorList ffr])
    R is GI  =>
      const:=unit ffr
      ris: List FF :=[]
      for ff in factorList ffr repeat
        fact:=primitivePart(conv(ff.fctr).convP)
        expf:=ff.xpnt
        ris:=cons([ff.flg,fact pretend PR,expf],ris)
        lc:GRN := myMap leadingCoefficient(fact pretend PR)
        const:= const*(leadingCoefficient(ff.fctr)/lc)**expf
      uconst:GI:= compND(coefficient(const,0)).cnum
      makeFR((uconst pretend R)::PR,ris)


  factor(pol : PR)  : Factored PR ==
    ratPol:SUP GRN := 0
    ratPol:=map(myMap,pol)$UPCF2
    ffr:=factor ratPol
    backConv ffr
```

———-

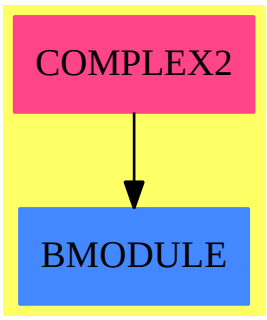
— COMPFACT.dotabb —

```
"COMPFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMPFACT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"COMPFACT" -> "COMPCAT"
```

———-

4.12 package COMPLEX2 ComplexFunctions2

4.12.1 ComplexFunctions2 (COMPLEX2)



Exports:
map

— package COMPLEX2 ComplexFunctions2 —

```
)abbrev package COMPLEX2 ComplexFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package extends maps from underlying rings to maps between
++ complex over those rings.

ComplexFunctions2(R:CommutativeRing, S:CommutativeRing): with
    map:     (R -> S, Complex R) -> Complex S
      ++ map(f,u) maps f onto real and imaginary parts of u.
 == add
    map(fn, gr) == complex(fn real gr, fn imag gr)
```

———-

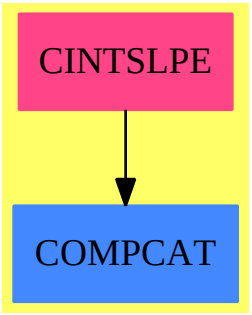
— **COMPLEX2.dotabb** —

```
"COMPLEX2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMPLEX2"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"COMPLEX2" -> "BMODULE"
```

———-

4.13 package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation

4.13.1 ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE)



Exports:
solveLinearPolynomialEquation

— **package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation** —

```
)abbrev package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation
++ Author: James Davenport
++ Date Created: 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ This package provides the generalized euclidean algorithm which is
++ needed as the basic step for factoring polynomials.

ComplexIntegerSolveLinearPolynomialEquation(R,CR): C == T
 where
  CP ==> SparseUnivariatePolynomial CR
  R:IntegerNumberSystem
  CR:ComplexCategory(R)
  C == with
      solveLinearPolynomialEquation: (List CP,CP) -> Union(List CP,"failed")
                  ++ solveLinearPolynomialEquation([f1, ..., fn], g)
                  ++ where (fi relatively prime to each other)
                  ++ returns a list of ai such that
                  ++ g = sum ai prod fj (j \= i) or
                  ++ equivalently g/prod fj = sum (ai/fi)
                  ++ or returns "failed" if no such list exists
  T == add
      oldlp:List CP := []
      slpePrime:R:=(2::R)
      oldtable:Vector List CP := empty()
      solveLinearPolynomialEquation(lp,p) ==
         if (oldlp ^= lp) then
            -- we have to generate a new table
            deg:= _+/[degree u for u in lp]
            ans:Union(Vector List CP,"failed"):="failed"
            slpePrime:=67108859::R   -- 2**26 -5 : a prime
                 -- a good test case for this package is
                 --  (good question?)
            while (ans case "failed") repeat
              ans:=tablePow(deg,complex(slpePrime,0),lp)$GenExEuclid(CR,CP)
              if (ans case "failed") then
                 slpePrime:=  slpePrime-4::R
                 while not prime?(slpePrime)$IntegerPrimesPackage(R) repeat
                   slpePrime:= slpePrime-4::R
            oldtable:=(ans:: Vector List CP)
         answer:=solveid(p,complex(slpePrime,0),oldtable)
         answer
```

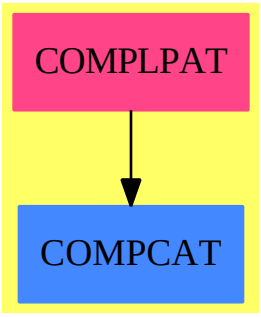
— CINTSLPE.dotabb —

```
"CINTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CINTSLPE"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CINTSLPE" -> "COMPCAT"
```

———-

4.14 package COMPLPAT ComplexPattern

4.14.1 ComplexPattern (COMPLPAT)



Exports:
convert

— package COMPLPAT ComplexPattern —

```
)abbrev package COMPLPAT ComplexPattern
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: complex, patterns
++ References:
++ Description:
++ This package supports converting complex expressions to patterns

ComplexPattern(R, S, CS) : C == T where
    R: SetCategory
    S: Join(ConvertibleTo Pattern R, CommutativeRing)
    CS: ComplexCategory S
    C == with
       convert: CS -> Pattern R
        ++ convert(cs) converts the complex expression cs to a pattern

    T == add

       ipat : Pattern R := patternVariable("%i"::Symbol, true, false, false)
```

```
        convert(cs) ==
           zero? imag cs => convert real cs
           convert real cs + ipat * convert imag cs
```

———-

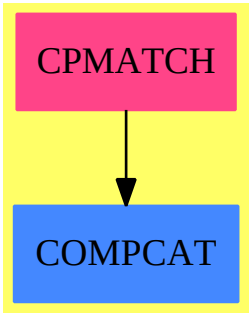
— **COMPLPAT.dotabb** —

```
"COMPLPAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMPLPAT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"COMPLPAT" -> "COMPCAT"
```

———-

4.15 package CPMATCH ComplexPatternMatch

4.15.1 ComplexPatternMatch (CPMATCH)



Exports:
patternMatch

— **package CPMATCH ComplexPatternMatch** —

```
)abbrev package CPMATCH ComplexPatternMatch
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
```

```
++ Keywords: complex, pattern matching
++ References:
++ Description:
++ This package supports matching patterns involving complex expressions

ComplexPatternMatch(R, S, CS) : C == T where
    R: SetCategory
    S: Join(PatternMatchable R, CommutativeRing)
    CS: ComplexCategory S
    PMRS ==> PatternMatchResult(R, CS)
    PS   ==> Polynomial S
    C == with
       if PS has PatternMatchable(R) then
           patternMatch: (CS, Pattern R, PMRS) -> PMRS
             ++ patternMatch(cexpr, pat, res) matches the pattern pat to the
             ++ complex expression cexpr. res contains the variables of pat
             ++ which are already matched and their matches.

    T == add

       import PatternMatchPushDown(R, S, CS)
       import PatternMatchResultFunctions2(R, PS, CS)
       import PatternMatchResultFunctions2(R, CS, PS)

       ivar : PS := "%i"::Symbol::PS

       makeComplex(p:PS):CS ==
          up := univariate p
          degree up > 1 => error "not linear in %i"
          icoef:=leadingCoefficient(up)
          rcoef:=leadingCoefficient(reductum p)
          complex(rcoef,icoef)

       makePoly(cs:CS):PS == real(cs)*ivar + imag(cs)::PS

       if PS has PatternMatchable(R) then
          patternMatch(cs, pat, result) ==
             zero? imag cs =>
                patternMatch(real cs, pat, result)
             map(makeComplex,
                patternMatch(makePoly cs, pat, map(makePoly, result)))
```

———-

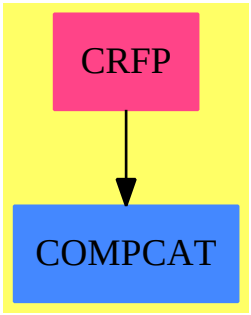
— CPMATCH.dotabb —

```
"CPMATCH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CPMATCH"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
```

```
"CPMATCH" -> "COMPCAT"
```

4.16 package CRFP ComplexRootFindingPackage

4.16.1 ComplexRootFindingPackage (CRFP)



Exports:

complexZeros	divisorCascade	factor	graeffe	norm
pleskenSplit	reciprocalPolynomial	rootRadius	schwerpunkt	setErrorBound
startPolynomial				

— package CRFP ComplexRootFindingPackage —

```
)abbrev package CRFP ComplexRootFindingPackage
++ Author: J. Grabmeier
++ Date Created: 31 January 1991
++ Date Last Updated: 12 April 1991
++ Basic Operations: factor, pleskenSplit
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: complex zeros, roots
++ References: J. Grabmeier: On Plesken's root finding algorithm,
++  in preparation
++  A. Schoenhage: The fundamental theorem of algebra in terms of computational
++  complexity, preliminary report, Univ. Tuebingen, 1982
++ Description:
++ \spadtype{ComplexRootFindingPackage} provides functions to
++ find all roots of a polynomial p over the complex number by
++ using Plesken's idea to calculate in the polynomial ring
++ modulo f and employing the Chinese Remainder Theorem.
++ In this first version, the precision (see digits)
```

```
++ is not increased when this is necessary to
++ avoid rounding errors. Hence it is the user's responsibility to
++ increase the precision if necessary.
++ Note also, if this package is called with e.g. \spadtype{Fraction Integer},
++ the precise calculations could require a lot of time.
++ Also note that evaluating the zeros is not necessarily a good check
++ whether the result is correct: already evaluation can cause
++ rounding errors.

ComplexRootFindingPackage(R, UP): public == private where
   -- R   : Join(Field, OrderedRing, CharacteristicZero)
   -- Float not in CharacteristicZero !|
   R   : Join(Field, OrderedRing)
   UP  : UnivariatePolynomialCategory Complex R

   C      ==> Complex R
   FR     ==> Factored
   I      ==> Integer
   L      ==> List
   FAE    ==> Record(factors : L UP, error : R)
   NNI    ==> NonNegativeInteger
   OF     ==> OutputForm
   ICF    ==> IntegerCombinatoricFunctions(I)

   public ==> with
     complexZeros : UP -> L C
       ++ complexZeros(p) tries to determine all complex zeros
       ++ of the polynomial p with accuracy given by the package
       ++ constant globalEps which you may change by setErrorBound.
     complexZeros : (UP, R) -> L C
       ++ complexZeros(p, eps) tries to determine all complex zeros
       ++ of the polynomial p with accuracy given by eps.
     divisorCascade : (UP,UP, Boolean) -> L FAE
       ++ divisorCascade(p,tp) assumes that degree of polynomial tp
       ++ is smaller than degree of polynomial p, both monic.
       ++ A sequence of divisions are calculated
       ++ using the remainder, made monic, as divisor
       ++ for the the next division. The result contains also the error of the
       ++ factorizations, i.e. the norm of the remainder polynomial.
       ++ If info is true, then information messages are issued.
     divisorCascade : (UP,UP) -> L FAE
       ++ divisorCascade(p,tp) assumes that degree of polynomial tp
       ++ is smaller than degree of polynomial p, both monic.
       ++ A sequence of divisions is calculated
       ++ using the remainder, made monic, as divisor
       ++ for the  the next division. The result contains also the error of the
       ++ factorizations, i.e. the norm of the remainder polynomial.
     factor: (UP,R,Boolean)  ->  FR UP
       ++ factor(p, eps, info) tries to factor p into linear factors
       ++ with error atmost eps. An overall error bound
```

```
  ++ eps0 is determined and iterated tree-like calls
  ++ to pleskenSplit are used to get the factorization.
  ++ If info is true, then information messages are given.
factor: (UP,R)  ->  FR UP
  ++ factor(p, eps) tries to factor p into linear factors
  ++ with error atmost eps. An overall error bound
  ++ eps0 is determined and iterated tree-like calls
  ++ to pleskenSplit are used to get the factorization.
factor: UP  ->  FR UP
  ++ factor(p) tries to factor p into linear factors
  ++ with error atmost globalEps, the internal error bound,
  ++ which can be set by setErrorBound. An overall error bound
  ++ eps0 is determined and iterated tree-like calls
  ++ to pleskenSplit are used to get the factorization.
graeffe : UP -> UP
  ++ graeffe p determines q such that \spad{q(-z**2) = p(z)*p(-z)}.
  ++ Note that the roots of q are the squares of the roots of p.
norm : UP -> R
  ++ norm(p) determines sum of absolute values of coefficients
  ++ Note that this function depends on abs.
pleskenSplit: (UP, R, Boolean)  ->  FR UP
  ++ pleskenSplit(poly,eps,info) determines a start polynomial start
  ++ by using "startPolynomial then it increases the exponent
  ++ n of start ** n mod poly to get an approximate factor of
  ++ poly, in general of degree "degree poly -1". Then a divisor
  ++ cascade is calculated and the best splitting is chosen, as soon
  ++ as the error is small enough.
  --++ In a later version we plan
  --++ to use the whole information to get a split into more than 2
  --++ factors.
  ++ If info is true, then information messages are issued.
pleskenSplit: (UP, R)  ->  FR UP
  ++ pleskenSplit(poly, eps)  determines a start polynomial start
  ++ by using "startPolynomial then it increases the exponent
  ++ n of start ** n mod poly to get an approximate factor of
  ++ poly, in general of degree "degree poly -1". Then a divisor
  ++ cascade is calculated and the best splitting is chosen, as soon
  ++ as the error is small enough.
  --++ In a later version we plan
  --++ to use the whole information to get a split into more than 2
  --++ factors.
reciprocalPolynomial: UP  -> UP
  ++ reciprocalPolynomial(p) calulates a polynomial which has exactly
  ++ the inverses of the non-zero roots of p as roots, and the same
  ++ number of 0-roots.
rootRadius: (UP,R) -> R
  ++ rootRadius(p,errQuot) calculates the root radius of p with a
  ++ maximal error quotient of errQuot.
rootRadius: UP -> R
  ++ rootRadius(p) calculates the root radius of p with a
```

```
      ++ maximal error quotient of 1+globalEps, where
      ++ globalEps is the internal error bound, which can be
      ++ set by setErrorBound.
    schwerpunkt: UP ->  C
      ++ schwerpunkt(p) determines the 'Schwerpunkt' of the roots of the
      ++ polynomial p of degree n, i.e. the center of gravity, which is
      ++ coeffient of \spad{x**(n-1)} divided by
      ++ n times coefficient of \spad{x**n}.
    setErrorBound : R -> R
      ++ setErrorBound(eps) changes the internal error bound,
      -- by default being 10 ** (-20) to eps, if R is
      ++ by default being 10 ** (-3) to eps, if R is
      ++ a member in the category \spadtype{QuotientFieldCategory Integer}.
      ++ The internal globalDigits is set to
      ++ \em ceiling(1/r)**2*10 being 10**7 by default.
    startPolynomial: UP  -> Record(start: UP, factors: FR UP)
      ++ startPolynomial(p) uses the ideas of Schoenhage's
      ++ variant of Graeffe's method to construct circles which separate
      ++ roots to get a good start polynomial, i.e. one whose
      ++ image under the Chinese Remainder Isomorphism has both entries
      ++ of norm smaller and greater or equal to 1. In case the
      ++ roots are found during internal calculations.
      ++ The corresponding factors
      ++ are in factors which are otherwise 1.

  private ==> add


    Rep := ModMonic(C, UP)

    -- constants
    c : C
    r : R
    --globalDigits : I := 10 ** 41
    globalDigits : I := 10 ** 7
    globalEps : R :=
      --a : R := (10000000000000000000 :: I) :: R
      a : R := (1000 :: I) :: R
      1/a
    emptyLine : OF := "  "
    dashes : OF := center "--------------------------------------------------"
    dots : OF :=   center ".................................................."
    one : R := 1$R
    two : R := 2 * one
    ten : R := 10 * one
    eleven : R := 11 * one
    weakEps := eleven/ten
    --invLog2 : R := 1/log10 (2*one)

    -- signatures of local functions
```

```
absC : C -> R
  --
absR : R -> R
  --
calculateScale : UP -> R
  --
makeMonic : UP -> UP
  -- 'makeMonic p' divides 'p' by the leading coefficient,
  -- to guarantee new leading coefficient to be 1$R  we cannot
  -- simply divide the leading monomial by the leading coefficient
  -- because of possible rounding errors
min: (FAE, FAE) -> FAE
  -- takes factorization with smaller error
nthRoot : (R, NNI) -> R
  -- nthRoot(r,n) determines an approximation to the n-th
  -- root of r, if \spadtype{R} has ?**?: (R,Fraction Integer)->R
  -- we use this, otherwise we use approxNthRoot via
  -- \spadtype{Integer}
shift: (UP,C) ->  UP
  -- shift(p,c) changes p(x) into p(x+c), thereby modifying the
  -- roots u_j of p to the roots (u_j - c)  of shift(p,c)
scale: (UP,C) -> UP
  -- scale(p,c) changes p(x) into p(cx), thereby modifying the
  -- roots u_j of p to the roots ((1/c) u_j)  of scale(p,c)


-- implementation of exported functions


complexZeros(p,eps) ==
  --r1 : R := rootRadius(p,weakEps)
  --eps0 : R = r1 * nthRoot(eps, degree p)
  -- right now we are content with
  eps0 : R := eps/(ten ** degree p)
  facs : FR UP := factor(p,eps0)
  [-coefficient(linfac.factor,0) for linfac in factors facs]

complexZeros p == complexZeros(p,globalEps)
setErrorBound r ==
  r <= 0 => error "setErrorBound: need error bound greater 0"
  globalEps := r
  if R has QuotientFieldCategory Integer then
    rd : Integer := ceiling(1/r)
    globalDigits := rd * rd * 10
    lof : List OF := _
      ["setErrorBound: internal digits set to",globalDigits::OF]
    print hconcat lof
  messagePrint  "setErrorBound: internal error bound set to"
  globalEps
```

```
    pleskenSplit(poly,eps,info) ==
      p := makeMonic poly
      fp : FR UP
      if not zero? (md := minimumDegree p) then
        fp : FR UP := irreducibleFactor(monomial(1,1)$UP,md)$(FR UP)
        p := p quo monomial(1,md)$UP
      sP : Record(start: UP, factors: FR UP) := startPolynomial p
      fp : FR UP := sP.factors
--      if not one? fp then
      if not (fp = 1) then
        qr: Record(quotient: UP, remainder: UP):= divide(p,makeMonic expand fp)
        p := qr.quotient
      st := sP.start
      zero? degree st => fp
        -- we calculate in ModMonic(C, UP),
        -- next line defines the polynomial, which is used for reducing
      setPoly p
      nm : R := eps
      split : FAE
      sR : Rep := st :: Rep
      psR : Rep := sR ** (degree poly)

      notFoundSplit : Boolean := true
      while notFoundSplit repeat
      --  if info then
      --    lof : L OF := ["not successfull, new exponent:", nn::OF]
      --    print hconcat lof
        psR := psR * psR * sR   -- exponent (2*d +1)
        -- be careful, too large exponent results in rounding errors
        -- tp is the first approximation of a divisor of poly:
        tp : UP  := lift psR
        zero? degree tp  =>
          if info then print "we leave as we got constant factor"
          nilFactor(poly,1)$(FR UP)
        -- this was the case where we don't find a non-trivial factorization
        -- we refine tp by repeated polynomial division and hope that
        -- the norm of the remainder gets small  from time to time
        splits : L FAE :=  divisorCascade(p, makeMonic tp, info)
        split := reduce(min,splits)
        notFoundSplit := (eps <=  split.error)

      for fac in split.factors repeat
        fp :=
--           one? degree fac => fp * nilFactor(fac,1)$(FR UP)
          (degree fac = 1) => fp * nilFactor(fac,1)$(FR UP)
          fp * irreducibleFactor(fac,1)$(FR UP)
      fp

    startPolynomial p == -- assume minimumDegree is 0
```

```
      --print (p :: OF)
      fp : FR UP := 1
--      one? degree p =>
      (degree p = 1) =>
        p := makeMonic p
        [p,irreducibleFactor(p,1)]
      startPoly : UP := monomial(1,1)$UP
      eps : R := weakEps   -- 10 per cent errors allowed
      r1 : R := rootRadius(p, eps)
      rd : R := 1/rootRadius(reciprocalPolynomial p, eps)
      (r1 > (2::R)) and (rd < 1/(2::R)) => [startPoly,fp] -- unit circle splitting!
      -- otherwise the norms of the roots are too closed so we
      -- take the center of gravity as new origin:
      u  : C := schwerpunkt p
      startPoly := startPoly-monomial(u,0)
      p := shift(p,-u)
      -- determine new rootRadius:
      r1 : R := rootRadius(p, eps)
      startPoly := startPoly/(r1::C)
      -- use one of the 4 points r1*zeta, where zeta is a 4th root of unity
      -- as new origin, this could be changed to an arbitrary list
      -- of elements of norm 1.
      listOfCenters : L C := [complex(r1,0), complex(0,r1), _
        complex(-r1,0), complex(0,-r1)]
      lp   : L UP := [shift(p,v) for v in listOfCenters]
      -- next we check if one of these centers is a root
      centerIsRoot : Boolean := false
      for i in 1..maxIndex lp repeat
        if (mD := minimumDegree lp.i) > 0 then
          pp : UP := monomial(1,1)-monomial(listOfCenters.i-u,0)
          centerIsRoot := true
          fp := fp * irreducibleFactor(pp,mD)
      centerIsRoot =>
        p := shift(p,u) quo expand fp
        --print (p::OF)
        zero? degree p => [p,fp]
        sP:= startPolynomial(p)
        [sP.start,fp]
      -- choose the best one w.r.t. maximal quotient of norm of largest
      -- root and norm of smallest root
      lpr1 : L R := [rootRadius(q,eps) for  q in lp]
      lprd : L R := [1/rootRadius(reciprocalPolynomial q,eps) for  q in lp]
      -- later we should check here of an rd is smaller than globalEps
      lq : L R := []
      for i in 1..maxIndex lpr1 repeat
        lq := cons(lpr1.i/lprd.i, lq)
      --lq : L R := [(l/s)::R for l in lpr1 for s in lprd])
      lq := reverse lq
      po := position(reduce(max,lq),lq)
      --p := lp.po
```

```
    --lrr : L R := [rootRadius(p,i,1+eps) for i in 2..(degree(p)-1)]
    --lrr := concat(concat(lpr1.po,lrr),lprd.po)
    --lu : L R := [(lrr.i + lrr.(i+1))/2 for i in 1..(maxIndex(lrr)-1)]
    [startPoly - monomial(listOfCenters.po,0),fp]

  norm p ==
   -- reduce(_+$R,map(absC,coefficients p))
   nm : R := 0
   for c in  coefficients p repeat
     nm := nm + absC c
   nm

  pleskenSplit(poly,eps) == pleskenSplit(poly,eps,false)

  graeffe p ==
    -- If  p = ao x**n + a1 x**(n-1) + ... + a<n-1> x + an
    -- and q = bo x**n + b1 x**(n-1) + ... + b<n-1> x + bn
    -- are such that q(-x**2) = p(x)p(-x), then
    -- bk := ak**2 + 2 * ((-1) * a<k-1>*a<k+1> + ... +
    --                    (-1)**l * a<l>*a<l>) where l = min(k, n-k).
    -- graeffe(p) constructs q using these identities.
    n   : NNI  := degree p
    aForth : L C := []
    for k in 0..n repeat  --  aForth = [a0, a1, ..., a<n-1>, an]
      aForth := cons(coefficient(p, k::NNI), aForth)
    aBack  : L C := [] --  after k steps
                       --  aBack = [ak, a<k-1>, ..., a1, a0]
    gp : UP := 0$UP
    for k in 0..n repeat
      ak : C := first aForth
      aForth := rest aForth
      aForthCopy : L C := aForth  -- we iterate over aForth and
      aBackCopy  : L C := aBack   -- aBack but do not want to
                                  -- destroy them
      sum        :   C := 0
      const : I  := -1  --  after i steps const = (-1)**i
      for aminus in aBack for aplus in aForth repeat
        -- after i steps aminus = a<k-i> and aplus = a<k+i>
        sum := sum + const * aminus * aplus
        aForthCopy := rest aForthCopy
        aBackCopy  := rest aBackCopy
        const := -const
      gp := gp + monomial(ak*ak + 2 * sum, (n-k)::NNI)
      aBack := cons(ak, aBack)
    gp



  rootRadius(p,errorQuotient) ==
    errorQuotient <= 1$R =>
```

```
    error "rootRadius: second Parameter must be greater than 1"
  pp   : UP  := p
  rho  : R   := calculateScale makeMonic pp
  rR   : R   := rho
  pp := makeMonic scale(pp,complex(rho,0$R))
  expo : NNI := 1
  d    : NNI := degree p
  currentError:  R   := nthRoot(2::R, 2)
  currentError     := d*20*currentError
  while nthRoot(currentError, expo) >= errorQuotient repeat
    -- if info then print (expo :: OF)
    pp := graeffe pp
    rho := calculateScale pp
    expo := 2 * expo
    rR := nthRoot(rho, expo) * rR
    pp :=  makeMonic scale(pp,complex(rho,0$R))
  rR

rootRadius(p) == rootRadius(p, 1+globalEps)

schwerpunkt p ==
  zero? p => 0$C
  zero? (d := degree p) => error _
  "schwerpunkt: non-zero const. polynomial has no roots and no schwerpunkt"
  -- coeffient of x**d and x**(d-1)
  lC : C :=  coefficient(p,d)  -- ^= 0
  nC : C :=  coefficient(p,(d-1) pretend NNI)
  (denom := recip ((d::I::C)*lC)) case "failed" => error  "schwerpunkt: _
    degree * leadingCoefficient not invertible in ring of coefficients"
  - (nC*(denom::C))

reciprocalPolynomial p ==
  zero? p => 0
  d : NNI := degree p
  md : NNI := d+minimumDegree p
  lm : L UP := [monomial(coefficient(p,i),(md-i) :: NNI) for i in 0..d]
  sol := reduce(_+, lm)

divisorCascade(p, tp, info) ==
  lfae : L FAE :=  nil()
  for i in 1..degree tp while (degree tp > 0)  repeat
    -- USE monicDivide !!!
    qr  : Record(quotient: UP, remainder: UP)  :=  divide(p,tp)
    factor1 : UP := tp
    factor2 : UP := makeMonic qr.quotient
    -- refinement of tp:
    tp := qr.remainder
    nm : R := norm tp
    listOfFactors  : L UP := cons(factor2,nil()$(L UP))
    listOfFactors := cons(factor1,listOfFactors)
```

```
        lfae := cons( [listOfFactors,nm], lfae)
        if info then
          --lof : L OF :=  [i :: OF,"-th division:"::OF]
          --print center box hconcat lof
          print emptyLine
          lof : L OF :=  ["error polynomial has degree " ::OF,_
            (degree tp)::OF, " and norm " :: OF, nm :: OF]
          print center hconcat lof
          lof : L OF := ["degrees of factors:" ::OF,_
            (degree factor1)::OF,"  ", (degree factor2)::OF]
          print center hconcat lof
      if info then print emptyLine
      reverse lfae

    divisorCascade(p, tp) == divisorCascade(p, tp, false)

    factor(poly,eps) == factor(poly,eps,false)
    factor(p) == factor(p, globalEps)

    factor(poly,eps,info) ==
      result : FR  UP := coerce monomial(leadingCoefficient poly,0)
      d : NNI := degree poly
      --should be
      --den : R := (d::I)::R * two**(d::Integer) * norm poly
      --eps0 : R := eps / den
      -- for now only
      eps0 : R := eps / (ten*ten)
--      one? d  => irreducibleFactor(poly,1)$(FR UP)
      (d = 1) => irreducibleFactor(poly,1)$(FR UP)
      listOfFactors : L Record(factor: UP,exponent: I) :=_
        list [makeMonic poly,1]
      if info then
        lof : L OF := [dashes,dots,"list of Factors:",dots,listOfFactors::OF, _
          dashes, "list of Linear Factors:", dots, result::OF, _
          dots,dashes]
        print vconcat lof
      while not null listOfFactors  repeat
        p : UP := (first listOfFactors).factor
        exponentOfp : I := (first listOfFactors).exponent
        listOfFactors := rest listOfFactors
        if info then
          lof : L OF := ["just now we try to split the polynomial:",p::OF]
          print vconcat lof
        split : FR UP  := pleskenSplit(p, eps0, info)
--        one? numberOfFactors split =>
        (numberOfFactors split = 1) =>
          -- in a later version we will change error bound and
          -- accuracy here to deal this case as well
          lof : L OF := ["factor: couldn't split factor",_
            center(p :: OF), "with required error bound"]
```

```
          print vconcat lof
          result := result * nilFactor(p, exponentOfp)
        -- now we got 2 good factors of p, we drop p and continue
        -- with the factors, if they are not linear, or put a
        -- linear factor to the result
        for rec in factors(split)$(FR UP) repeat
          newFactor : UP := rec.factor
          expOfFactor := exponentOfp * rec.exponent
--          one? degree newFactor =>
          (degree newFactor = 1) =>
            result := result * nilFactor(newFactor,expOfFactor)
          listOfFactors:=cons([newFactor,expOfFactor],_
            listOfFactors)
      result

    -- implementation of local functions

    absC c == nthRoot(norm(c)$C,2)
    absR r ==
      r < 0 => -r
      r
    min(fae1,fae2) ==
      fae2.error <  fae1.error => fae2
      fae1
    calculateScale p ==
      d  := degree p
      maxi :R := 0
      for j in 1..d for cof in rest coefficients p repeat
        -- here we need abs: R -> R
        rc :  R := absR real cof
        ic :  R := absR imag cof
        locmax: R := max(rc,ic)
        maxi := max( nthRoot( locmax/(binomial(d,j)$ICF::R), j), maxi)
      -- Maybe I should use some type of logarithm for the following:
      maxi = 0$R => error("Internal Error: scale cannot be 0")
      rho  :R := one
      rho < maxi =>
        while rho < maxi repeat rho := ten * rho
        rho / ten
      while maxi < rho repeat rho := rho / ten
      rho = 0 => one
      rho
    makeMonic p  ==
      p = 0 => p
      monomial(1,degree p)$UP + (reductum p)/(leadingCoefficient p)

    scale(p, c) ==
      -- eval(p,cx) is missing !!
      eq : Equation UP := equation(monomial(1,1), monomial(c,1))
      eval(p,eq)
```

```
      -- improvement?: direct calculation of the new coefficients

    shift(p,c) ==
      rhs : UP := monomial(1,1) + monomial(c,0)
      eq : Equation UP := equation(monomial(1,1), rhs)
      eval(p,eq)
      -- improvement?: direct calculation of the new coefficients

    nthRoot(r,n) ==
      R has RealNumberSystem =>  r ** (1/n)
      R has QuotientFieldCategory Integer =>
        den : I := approxNthRoot(globalDigits * denom r ,n)$IntegerRoots(I)
        num : I := approxNthRoot(globalDigits * numer r ,n)$IntegerRoots(I)
        num/den
      -- the following doesn't compile
      --R has coerce: % -> Fraction Integer =>
      --  q : Fraction Integer := coerce(r)@Fraction(Integer)
      --  den : I := approxNthRoot(globalDigits * denom q ,n)$IntegerRoots(I)
      --  num : I := approxNthRoot(globalDigits * numer q ,n)$IntegerRoots(I)
      --  num/den
      r -- this is nonsense, perhaps a Newton iteration for x**n-r here

)fin
    -- for late use:

    graeffe2 p ==
      -- substitute x by -x :
      eq : Equation UP := equation(monomial(1,1), monomial(-1$C,1))
      pp : UP := p*eval(p,eq)
      gp : UP :=  0$UP
      while pp ^= 0 repeat
         i:NNI := (degree pp) quo (2::NNI)
         coef:C:=
           even? i => leadingCoefficient pp
           - leadingCoefficient pp
         gp    := gp + monomial(coef,i)
         pp    := reductum pp
      gp
    shift2(p,c) ==
      d := degree p
      cc : C := 1
      coef := List C := [cc := c * cc for i in 1..d]
      coef := cons(1,coef)
      coef := [coefficient(p,i)*coef.(1+i) for i in 0..d]
      res : UP := 0
      for j in 0..d repeat
        cc := 0
        for i in j..d repeat
          cc := cc + coef.i * (binomial(i,j)$ICF :: R)
        res := res + monomial(cc,j)$UP
```

```
      res
    scale2(p,c) ==
      d := degree p
      cc : C := 1
      coef := List C := [cc := c * cc for i in 1..d]
      coef := cons(1,coef)
      coef := [coefficient(p,i)*coef.(i+1) for i in 0..d]
      res : UP := 0
      for i in 0..d repeat  res := res + monomial(coef.(i+1),i)$UP
      res
    scale2: (UP,C) -> UP
    shift2: (UP,C) ->  UP
    graeffe2 : UP -> UP
      ++ graeffe2 p determines q such that \spad{q(-z**2) = p(z)*p(-z)}.
      ++ Note that the roots of q are the squares of the roots of p.
```

———-

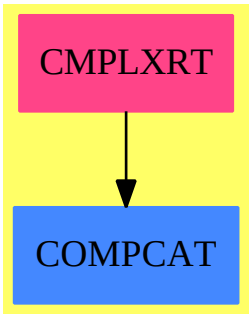
— **CRFP.dotabb** —

```
"CRFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CRFP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CRFP" -> "COMPCAT"
```

———-

4.17 package CMPLXRT ComplexRootPackage

4.17.1 ComplexRootPackage (CMPLXRT)



Exports:
complexZeros

— package CMPLXRT ComplexRootPackage —

```
)abbrev package CMPLXRT ComplexRootPackage
++ Author: P. Gianni
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: Complex, Float, Fraction, UnivariatePolynomial
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides functions complexZeros
++ for finding the complex zeros
++ of univariate polynomials with complex rational number coefficients.
++ The results are to any user specified precision and are returned
++ as either complex rational number or complex floating point numbers
++ depending on the type of the second argument which specifies the
++ precision.


ComplexRootPackage(UP,Par) : T == C where
   RN     ==> Fraction Integer
   I      ==> Integer
   NF     ==> Float
   UP   :   UnivariatePolynomialCategory Complex Integer
   Par  :   Join(Field, OrderedRing) -- will be Float or RN
   CP   ==> Complex Par
   PCI  ==> Polynomial Complex Integer

   T == with
        complexZeros:(UP,Par)  -> List CP
          ++ complexZeros(poly, eps) finds the complex zeros of the
          ++ univariate polynomial poly to precision eps with
          ++ solutions returned as complex floats or rationals
          ++ depending on the type of eps.

   C == add
    complexZeros(p:UP,eps:Par):List CP ==
      x1:Symbol():=new()
      x2:Symbol():=new()
      vv:Symbol():=new()
      lpf:=factors factor(p)$ComplexFactorization(I,UP)
      ris:List CP:=empty()
      for pf in lpf repeat
          pp:=pf.factor pretend SparseUnivariatePolynomial Complex Integer
          q:PCI :=multivariate(pp,vv)
          q:=eval(q,vv,x1::PCI+complex(0,1)*(x2::PCI))
```

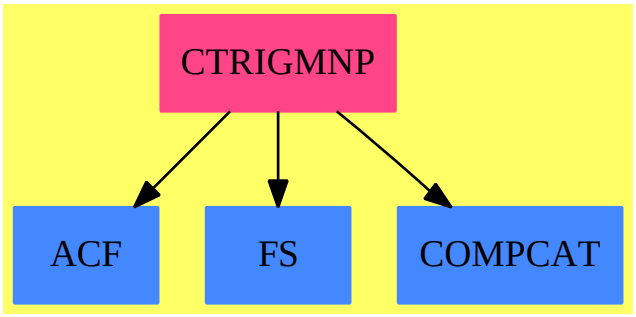
```
        p1:=map(real,q)$PolynomialFunctions2(Complex I,I)
        p2:=map(imag,q)$PolynomialFunctions2(Complex I,I)
        lz:=innerSolve([p1,p2],[],[x1,x2],
                       eps)$InnerNumericFloatSolvePackage(I,Par,Par)
        ris:=append([complex(first z,second z) for z in lz],ris)
    ris
```

— **CMPLXRT.dotabb** —

```
"CMPLXRT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CMPLXRT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CMPLXRT" -> "COMPCAT"
```

4.18 package CTRIGMNP ComplexTrigonometricManipulations

4.18.1 ComplexTrigonometricManipulations (CTRIGMNP)



Exports:

complexElementary	complexForm	complexNormalize	imag
real	trigs		

— **package CTRIGMNP ComplexTrigonometricManipulations** —

```
)abbrev package CTRIGMNP ComplexTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 11 June 1993
++ Date Last Updated: 14 June 1993
```

```
++ Keywords: complex, function, manipulation.
++ Description:
++ \spadtype{ComplexTrigonometricManipulations} provides function that
++ compute the real and imaginary parts of complex functions.

ComplexTrigonometricManipulations(R, F): Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace Complex R)


  SY  ==> Symbol
  FR  ==> Expression R
  K   ==> Kernel F


  Exports ==> with
    complexNormalize: F -> F
      ++ complexNormalize(f) rewrites \spad{f} using the least possible number
      ++ of complex independent kernels.
    complexNormalize: (F, SY) -> F
      ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
      ++ number of complex independent kernels involving \spad{x}.
    complexElementary: F -> F
      ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
      ++ complex transcendental elementary functions: \spad{log, exp}.
    complexElementary: (F, SY) -> F
      ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
      ++ \spad{x} in terms of the 2 fundamental complex
      ++ transcendental elementary functions: \spad{log, exp}.
    real   : F -> FR
      ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
      ++ function.
    imag   : F -> FR
      ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
      ++ is a complex function.
    real?  : F -> Boolean
      ++ real?(f) returns \spad{true} if \spad{f = real f}.
    trigs  : F -> F
      ++ trigs(f) rewrites all the complex logs and exponentials
      ++ appearing in \spad{f} in terms of trigonometric functions.
    complexForm: F -> Complex FR
      ++ complexForm(f) returns \spad{[real f, imag f]}.

  Implementation ==> add
    import InnerTrigonometricManipulations(R, FR, F)
    import ElementaryFunctionStructurePackage(Complex R, F)

    rreal?: Complex R -> Boolean
    kreal?: Kernel F -> Boolean
```

```
localexplogs  : (F, F, List SY) -> F

real f        == real complexForm f
imag f        == imag complexForm f
rreal? r      == zero? imag r
kreal? k      == every?(real?, argument k)$List(F)
complexForm f == explogs2trigs f

trigs f ==
  GF2FG explogs2trigs f

real? f ==
  every?(rreal?, coefficients numer f)
    and every?(rreal?, coefficients denom f) and every?(kreal?, kernels f)

localexplogs(f, g, lx) ==
  trigs2explogs(g, [k for k in tower f
                      | is?(k, "tan"::SY) or is?(k, "cot"::SY)], lx)

complexElementary f ==
  any?(x +-> has?(x, "rtrig"),
    operators(g := realElementary f))$List(BasicOperator) =>
      localexplogs(f, g, variables g)
  g

complexElementary(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
   [k for k in tower(g := realElementary(f, x))
              | member?(x, variables(k::F))]$List(K))$List(K) =>
                  localexplogs(f, g, [x])
  g

complexNormalize(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
   [k for k in tower(g := realElementary(f, x))
            | member?(x, variables(k::F))]$List(K))$List(K) =>
               (rischNormalize(localexplogs(f, g, [x]), x).func)
  rischNormalize(g, x).func

complexNormalize f ==
  l := variables(g := realElementary f)
  any?(y +-> has?(y, "rtrig"), operators g)$List(BasicOperator) =>
    h := localexplogs(f, g, l)
    for x in l repeat h := rischNormalize(h, x).func
    h
  for x in l repeat g := rischNormalize(g, x).func
  g
```

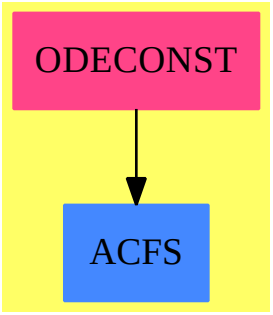
— CTRIGMNP.dotabb —

```
"CTRIGMNP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CTRIGMNP"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CTRIGMNP" -> "ACF"
"CTRIGMNP" -> "FS"
"CTRIGMNP" -> "COMPCAT"
```

———-

4.19 package ODECONST ConstantLODE

4.19.1 ConstantLODE (ODECONST)



Exports:
constDsolve

— package ODECONST ConstantLODE —

```
)abbrev package ODECONST ConstantLODE
++ Author: Manuel Bronstein
++ Date Created: 18 March 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ Solution of linear ordinary differential equations,
++ constant coefficient case.

ConstantLODE(R, F, L): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
```

```
F: Join(AlgebraicallyClosedFunctionSpace R,
        TranscendentalFunctionCategory, PrimitiveFunctionCategory)
L: LinearOrdinaryDifferentialOperatorCategory F

Z   ==> Integer
SY  ==> Symbol
K   ==> Kernel F
V   ==> Vector F
M   ==> Matrix F
SUP ==> SparseUnivariatePolynomial F

Exports ==> with
  constDsolve: (L, F, SY) -> Record(particular:F, basis:List F)
    ++ constDsolve(op, g, x) returns \spad{[f, [y1,...,ym]]}
    ++ where f is a particular solution of the equation \spad{op y = g},
    ++ and the \spad{yi}'s form a basis for the solutions of \spad{op y = 0}.

Implementation ==> add
  import ODETools(F, L)
  import ODEIntegration(R, F)
  import ElementaryFunctionSign(R, F)
  import AlgebraicManipulations(R, F)
  import FunctionSpaceIntegration(R, F)
  import FunctionSpaceUnivariatePolynomialFactor(R, F, SUP)

  homoBasis: (L, F) -> List F
  quadSol  : (SUP, F) -> List F
  basisSqfr: (SUP, F) -> List F
  basisSol : (SUP, Z, F) -> List F

  constDsolve(op, g, x) ==
    b := homoBasis(op, x::F)
    [particularSolution(op, g, b, (f1:F):F +-> int(f1, x))::F, b]

  homoBasis(op, x) ==
    p:SUP := 0
    while op ^= 0 repeat
        p  := p + monomial(leadingCoefficient op, degree op)
        op := reductum op
    b:List(F) := empty()
    for ff in factors ffactor p repeat
      b := concat_!(b, basisSol(ff.factor, dec(ff.exponent), x))
    b

  basisSol(p, n, x) ==
    l := basisSqfr(p, x)
    zero? n => l
    ll := copy l
    xn := x::F
    for i in 1..n repeat
```

```
        l := concat_!(l, [xn * f for f in ll])
        xn := x * xn
      l

    basisSqfr(p, x) ==
--      one?(d := degree p) =>
      ((d := degree p) = 1) =>
        [exp(- coefficient(p, 0) * x / leadingCoefficient p)]
      d = 2 => quadSol(p, x)
      [exp(a * x) for a in rootsOf p]

    quadSol(p, x) ==
      (u := sign(delta := (b := coefficient(p, 1))**2 - 4 *
        (a := leadingCoefficient p) * (c := coefficient(p, 0))))
          case Z and negative?(u::Z) =>
            y := x / (2 * a)
            r := - b * y
            i := rootSimp(sqrt(-delta)) * y
            [exp(r) * cos(i), exp(r) * sin(i)]
      [exp(a * x) for a in zerosOf p]
```

———-

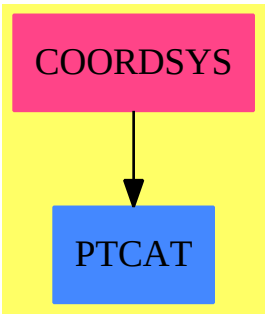
— **ODECONST.dotabb** —

```
"ODECONST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODECONST"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODECONST" -> "ACFS"
```

———-

4.20 package COORDSYS CoordinateSystems

4.20.1 CoordinateSystems (COORDSYS)



Exports:

bipolar	bipolarCylindrical	cartesian	conical	cylindrical
elliptic	ellipticCylindrical	oblateSpheroidal	parabolic	parabolicCylindrical
paraboloidal	polar	prolateSpheroidal	spherical	toroidal

— package COORDSYS CoordinateSystems —

```
)abbrev package COORDSYS CoordinateSystems
++ Author: Jim Wen
++ Date Created: 12 March 1990
++ Date Last Updated: 19 June 1990, Clifton J. Williamson
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ CoordinateSystems provides coordinate transformation functions
++ for plotting.  Functions in this package return conversion functions
++ which take points expressed in other coordinate systems and return points
++ with the corresponding Cartesian coordinates.

CoordinateSystems(R): Exports == Implementation where

  R : Join(Field,TranscendentalFunctionCategory,RadicalCategory)
  Pt ==> Point R

  Exports ==> with
    cartesian : Pt -> Pt
      ++ cartesian(pt) returns the Cartesian coordinates of point pt.
    polar: Pt -> Pt
      ++ polar(pt) transforms pt from polar coordinates to Cartesian
      ++ coordinates: the function produced will map the point \spad{(r,theta)}
      ++ to \spad{x = r * cos(theta)} , \spad{y = r * sin(theta)}.
    cylindrical: Pt -> Pt
      ++ cylindrical(pt) transforms pt from polar coordinates to Cartesian
      ++ coordinates: the function produced will map the point
      ++ \spad{(r,theta,z)}
      ++ to \spad{x = r * cos(theta)}, \spad{y = r * sin(theta)}, \spad{z}.
    spherical: Pt -> Pt
      ++ spherical(pt) transforms pt from spherical coordinates to Cartesian
      ++ coordinates: the function produced will map the point
      ++ \spad{(r,theta,phi)}
      ++ to \spad{x = r*sin(phi)*cos(theta)}, \spad{y = r*sin(phi)*sin(theta)},
      ++ \spad{z = r*cos(phi)}.
    parabolic: Pt -> Pt
      ++ parabolic(pt) transforms pt from parabolic coordinates to Cartesian
      ++ coordinates: the function produced will map the point \spad{(u,v)} to
```

```
    ++ \spad{x = 1/2*(u**2 - v**2)}, \spad{y = u*v}.
  parabolicCylindrical: Pt -> Pt
    ++ parabolicCylindrical(pt) transforms pt from parabolic cylindrical
    ++ coordinates to Cartesian coordinates: the function produced will
    ++ map the point \spad{(u,v,z)} to \spad{x = 1/2*(u**2 - v**2)},
    ++ \spad{y = u*v}, \spad{z}.
  paraboloidal: Pt -> Pt
    ++ paraboloidal(pt) transforms pt from paraboloidal coordinates to
    ++ Cartesian coordinates: the function produced will map the point
    ++ \spad{(u,v,phi)} to \spad{x = u*v*cos(phi)}, \spad{y = u*v*sin(phi)},
    ++ \spad{z = 1/2 * (u**2 - v**2)}.
  elliptic: R -> (Pt -> Pt)
    ++ elliptic(a) transforms from elliptic coordinates to Cartesian
    ++ coordinates: \spad{elliptic(a)} is a function which will map the
    ++ point \spad{(u,v)} to \spad{x = a*cosh(u)*cos(v)}, \spad{y = a*sinh(u)*sin(v)}.
  ellipticCylindrical: R -> (Pt -> Pt)
    ++ ellipticCylindrical(a) transforms from elliptic cylindrical coordinates
    ++ to Cartesian coordinates: \spad{ellipticCylindrical(a)} is a function
    ++ which will map the point \spad{(u,v,z)} to \spad{x = a*cosh(u)*cos(v)},
    ++ \spad{y = a*sinh(u)*sin(v)}, \spad{z}.
  prolateSpheroidal: R -> (Pt -> Pt)
    ++ prolateSpheroidal(a) transforms from prolate spheroidal coordinates to
    ++ Cartesian coordinates: \spad{prolateSpheroidal(a)} is a function
    ++ which will map the point \spad{(xi,eta,phi)} to
    ++ \spad{x = a*sinh(xi)*sin(eta)*cos(phi)}, \spad{y = a*sinh(xi)*sin(eta)*sin(phi)},
    ++ \spad{z = a*cosh(xi)*cos(eta)}.
  oblateSpheroidal: R -> (Pt -> Pt)
    ++ oblateSpheroidal(a) transforms from oblate spheroidal coordinates to
    ++ Cartesian coordinates: \spad{oblateSpheroidal(a)} is a function which
    ++ will map the point \spad{(xi,eta,phi)} to \spad{x = a*sinh(xi)*sin(eta)*cos(phi)},
    ++ \spad{y = a*sinh(xi)*sin(eta)*sin(phi)}, \spad{z = a*cosh(xi)*cos(eta)}.
  bipolar: R -> (Pt -> Pt)
    ++ bipolar(a) transforms from bipolar coordinates to Cartesian coordinates:
    ++ \spad{bipolar(a)} is a function which will map the point \spad{(u,v)} to
    ++ \spad{x = a*sinh(v)/(cosh(v)-cos(u))}, \spad{y = a*sin(u)/(cosh(v)-cos(u))}.
  bipolarCylindrical: R -> (Pt -> Pt)
    ++ bipolarCylindrical(a) transforms from bipolar cylindrical coordinates
    ++ to Cartesian coordinates: \spad{bipolarCylindrical(a)} is a function which
    ++ will map the point \spad{(u,v,z)} to \spad{x = a*sinh(v)/(cosh(v)-cos(u))},
    ++ \spad{y = a*sin(u)/(cosh(v)-cos(u))}, \spad{z}.
  toroidal: R -> (Pt -> Pt)
    ++ toroidal(a) transforms from toroidal coordinates to Cartesian
    ++ coordinates: \spad{toroidal(a)} is a function which will map the point
    ++ \spad{(u,v,phi)} to \spad{x = a*sinh(v)*cos(phi)/(cosh(v)-cos(u))},
    ++ \spad{y = a*sinh(v)*sin(phi)/(cosh(v)-cos(u))}, \spad{z = a*sin(u)/(cosh(v)-cos(u))}.
  conical: (R,R) -> (Pt -> Pt)
    ++ conical(a,b) transforms from conical coordinates to Cartesian coordinates:
    ++ \spad{conical(a,b)} is a function which will map the point \spad{(lambda,mu,nu)} to
    ++ \spad{x = lambda*mu*nu/(a*b)},
    ++ \spad{y = lambda/a*sqrt((mu**2-a**2)*(nu**2-a**2)/(a**2-b**2))},
```

```
    ++ \spad{z = lambda/b*sqrt((mu**2-b**2)*(nu**2-b**2)/(b**2-a**2))}.

  Implementation ==> add

    cartesian pt ==
      -- we just want to interpret the cartesian coordinates
      -- from the first N elements of the point - so the
      -- identity function will do
      pt

    polar pt0 ==
      pt := copy pt0
      r := elt(pt0,1); theta := elt(pt0,2)
      pt.1 := r * cos(theta); pt.2 := r * sin(theta)
      pt

    cylindrical pt0 == polar pt0
    -- apply polar transformation to first 2 coordinates

    spherical pt0 ==
      pt := copy pt0
      r := elt(pt0,1); theta := elt(pt0,2); phi := elt(pt0,3)
      pt.1 := r * sin(phi) * cos(theta); pt.2 := r * sin(phi) * sin(theta)
      pt.3 := r * cos(phi)
      pt

    parabolic pt0 ==
      pt := copy pt0
      u := elt(pt0,1); v := elt(pt0,2)
      pt.1 := (u*u - v*v)/(2::R) ; pt.2 := u*v
      pt

    parabolicCylindrical pt0 == parabolic pt0
    -- apply parabolic transformation to first 2 coordinates

    paraboloidal pt0 ==
      pt := copy pt0
      u := elt(pt0,1); v := elt(pt0,2); phi := elt(pt0,3)
      pt.1 := u*v*cos(phi); pt.2 := u*v*sin(phi); pt.3 := (u*u - v*v)/(2::R)
      pt

    elliptic a ==
     x+->
      pt := copy(x)
      u := elt(x,1); v := elt(x,2)
      pt.1 := a*cosh(u)*cos(v); pt.2 := a*sinh(u)*sin(v)
      pt

    ellipticCylindrical a == elliptic a
    -- apply elliptic transformation to first 2 coordinates
```

```
prolateSpheroidal a ==
 x+->
  pt := copy(x)
  xi := elt(x,1); eta := elt(x,2); phi := elt(x,3)
  pt.1 := a*sinh(xi)*sin(eta)*cos(phi)
  pt.2 := a*sinh(xi)*sin(eta)*sin(phi)
  pt.3 := a*cosh(xi)*cos(eta)
  pt

oblateSpheroidal a ==
 x+->
  pt := copy(x)
  xi := elt(x,1); eta := elt(x,2); phi := elt(x,3)
  pt.1 := a*sinh(xi)*sin(eta)*cos(phi)
  pt.2 := a*cosh(xi)*cos(eta)*sin(phi)
  pt.3 := a*sinh(xi)*sin(eta)
  pt

bipolar a ==
 x+->
  pt := copy(x)
  u := elt(x,1); v := elt(x,2)
  pt.1 := a*sinh(v)/(cosh(v)-cos(u))
  pt.2 := a*sin(u)/(cosh(v)-cos(u))
  pt

bipolarCylindrical a == bipolar a
-- apply bipolar transformation to first 2 coordinates

toroidal a ==
 x+->
  pt := copy(x)
  u := elt(x,1); v := elt(x,2); phi := elt(x,3)
  pt.1 := a*sinh(v)*cos(phi)/(cosh(v)-cos(u))
  pt.2 := a*sinh(v)*sin(phi)/(cosh(v)-cos(u))
  pt.3 := a*sin(u)/(cosh(v)-cos(u))
  pt

conical(a,b) ==
 x+->
  pt := copy(x)
  lambda := elt(x,1); mu := elt(x,2); nu := elt(x,3)
  pt.1 := lambda*mu*nu/(a*b)
  pt.2 := lambda/a*sqrt((mu**2-a**2)*(nu**2-a**2)/(a**2-b**2))
  pt.3 := lambda/b*sqrt((mu**2-b**2)*(nu**2-b**2)/(b**2-a**2))
  pt
```

———-

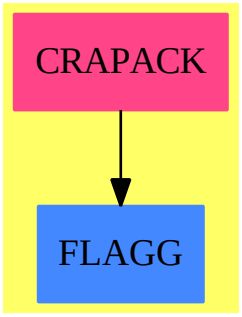
— COORDSYS.dotabb —

```
"COORDSYS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COORDSYS"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"COORDSYS" -> "PTCAT"
```

———-

4.21 package CRAPACK CRApackage

4.21.1 CRApackage (CRAPACK)



Exports:
modTree chineseRemainder multiEuclideanTree

— package CRAPACK CRApackage —

```
)abbrev package CRAPACK CRApackage
++ Description:
++ This package has no documentation

CRApackage(R:EuclideanDomain): Exports == Implementation where
  Exports == with
    modTree: (R,List R) -> List R
      ++ modTree(r,l) \undocumented{}
    chineseRemainder: (List R, List R) -> R
      ++ chineseRemainder(lv,lm) returns a value \axiom{v} such that, if
      ++ x is \axiom{lv.i} modulo \axiom{lm.i} for all \axiom{i}, then
      ++ x is \axiom{v} modulo \axiom{lm(1)*lm(2)*...*lm(n)}.
    chineseRemainder: (List List R, List R) -> List R
      ++ chineseRemainder(llv,lm) returns a list of values, each of which
      ++ corresponds to the Chinese remainder of the associated element of
```

```
    ++ \axiom{llv} and axiom{lm}.  This is more efficient than applying
    ++ chineseRemainder several times.
  multiEuclideanTree: (List R, R) -> List R
    ++ multiEuclideanTree(l,r) \undocumented{}
Implementation == add

  BB:=BalancedBinaryTree(R)
  x:BB

  -- Definition for modular reduction mapping with several moduli
  modTree(a,lm) ==
    t := balancedBinaryTree(#lm, 0$R)
    setleaves_!(t,lm)
    mapUp_!(t,"*")
    leaves mapDown_!(t, a, "rem")

  chineseRemainder(lv:List(R), lm:List(R)):R ==
    #lm ^= #lv => error "lists of moduli and values not of same length"
    x := balancedBinaryTree(#lm, 0$R)
    x := setleaves_!(x, lm)
    mapUp_!(x,"*")
    y := balancedBinaryTree(#lm, 1$R)
    y := mapUp_!(copy y,x,(a,b,c,d)+->a*d + b*c)
    (u := extendedEuclidean(value y, value x,1)) case "failed" =>
      error "moduli not relatively prime"
    inv := u . coef1
    linv := modTree(inv, lm)
    l := [(u*v) rem m for v in lv for u in linv for m in lm]
    y := setleaves_!(y,l)
    value(mapUp_!(y, x, (a,b,c,d)+->a*d + b*c)) rem value(x)

  chineseRemainder(llv:List List(R), lm:List(R)):List(R) ==
    x := balancedBinaryTree(#lm, 0$R)
    x := setleaves_!(x, lm)
    mapUp_!(x,"*")
    y := balancedBinaryTree(#lm, 1$R)
    y := mapUp_!(copy y,x,(a,b,c,d)+->a*d + b*c)
    (u := extendedEuclidean(value y, value x,1)) case "failed" =>
      error "moduli not relatively prime"
    inv := u . coef1
    linv := modTree(inv, lm)
    retVal:List(R) := []
    for lv in llv repeat
      l := [(u3*v) rem m for v in lv for u3 in linv for m in lm]
      y := setleaves!(y,l)
      retVal :=
        cons(value(mapUp!(y, x, (a,b,c,d)+->a*d+b*c)) rem value(x),retVal)
    reverse retVal

  extEuclidean: (R, R, R) -> List R
```

```
    extEuclidean(a, b, c) ==
      u := extendedEuclidean(a, b, c)
      u case "failed" => error [c, " not spanned by ", a, " and ",b]
      [u.coef2, u.coef1]

    multiEuclideanTree(fl, rhs) ==
      x := balancedBinaryTree(#fl, rhs)
      x := setleaves_!(x, fl)
      mapUp_!(x,"*")
      leaves mapDown_!(x, rhs, extEuclidean)
```

———-

— CRAPACK.dotabb —

```
"CRAPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CRAPACK"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"CRAPACK" -> "FLAGG"
```

———-

4.22 package CYCLES CycleIndicators

— CycleIndicators.input —

```
)set break resume
)sys rm -f CycleIndicators.output
)spool CycleIndicators.output
)set message test on
)set message auto off
)clear all
--S 1 of 47
complete 1
--R
--R
--R   (1)  (1)
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 1

--S 2 of 47
complete 2
--R
--R
--R        1       1   2
```

```
--R   (2)  - (2) + - (1 )
--R        2       2
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 2

--S 3 of 47
complete 3
--R
--R
--R        1       1         1   3
--R   (3)  - (3) + - (2 1) + - (1 )
--R        3       2         6
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 3

--S 4 of 47
complete 7
--R
--R
--R   (4)
--R     1       1          1          1     2     1         1            1     3
--R     - (7) + - (6 1) + -- (5 2) + -- (5 1 ) + -- (4 3) + - (4 2 1) + -- (4 1 )
--R     7       6         10         10          12         8           24
--R   +
--R      1   2      1     2     1       2     1     4     1   3      1   2 3
--R     -- (3 1) + -- (3 2 ) + -- (3 2 1 ) + -- (3 1 ) + -- (2 1) + -- (2 1 )
--R     18         24          12            72          48         48
--R   +
--R      1      5      1    7
--R     --- (2 1 ) + ---- (1 )
--R     240          5040
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 4

--S 5 of 47
elementary 7
--R
--R
--R   (5)
--R     1       1          1          1     2     1         1            1     3
--R     - (7) - - (6 1) - -- (5 2) + -- (5 1 ) - -- (4 3) + - (4 2 1) - -- (4 1 )
--R     7       6         10         10          12         8           24
--R   +
--R      1   2      1     2     1       2     1     4     1   3      1   2 3
--R     -- (3 1) + -- (3 2 ) - -- (3 2 1 ) + -- (3 1 ) - -- (2 1) + -- (2 1 )
--R     18         24          12            72          48         48
--R   +
--R        1      5      1    7
--R     - --- (2 1 ) + ---- (1 )
--R       240          5040
```

```
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 5

--S 6 of 47
alternating 7
--R
--R
--R   (6)
--R     2       1     2    1           1   2      1     2     1     4     1   2 3
--R     - (7) + - (5 1 ) + - (4 2 1) + - (3 1) + -- (3 2 ) + -- (3 1 ) + -- (2 1 )
--R     7       5          4           9         12          36          24
--R   +
--R       1    7
--R     ---- (1 )
--R     2520
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 6

--S 7 of 47
cyclic 7
--R
--R
--R        6       1   7
--R   (7)  - (7) + - (1 )
--R        7       7
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 7

--S 8 of 47
dihedral 7
--R
--R
--R        3       1   3      1   7
--R   (8)  - (7) + - (2 1) + -- (1 )
--R        7       2         14
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 8

--S 9 of 47
graphs 5
--R
--R
--R   (9)
--R   1           1   2    1   2     1   3     1   4 2     1   3 4     1    10
--R   - (6 3 1) + - (5 ) + - (4 2) + - (3 1) + - (2 1 ) + -- (2 1 ) + --- (1  )
--R   6           5        4         6         8          12          120
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 9

--S 10 of 47
```

```
cap(complete 2**2, complete 2*complete 1**2)
--R
--R
--R   (10)  4
--R                                                    Type: Fraction(Integer)
--E 10

--S 11 of 47
cap(elementary 2**2, complete 2*complete 1**2)
--R
--R
--R   (11)  2
--R                                                    Type: Fraction(Integer)
--E 11

--S 12 of 47
cap(complete 3*complete 2*complete 1,complete 2**2*complete 1**2)
--R
--R
--R   (12)  24
--R                                                    Type: Fraction(Integer)
--E 12

--S 13 of 47
cap(elementary 3*elementary 2*elementary 1,complete 2**2*complete 1**2)
--R
--R
--R   (13)  8
--R                                                    Type: Fraction(Integer)
--E 13

--S 14 of 47
cap(complete 3*complete 2*complete 1,elementary 2**2*elementary 1**2)
--R
--R
--R   (14)  8
--R                                                    Type: Fraction(Integer)
--E 14

--S 15 of 47
eval(cup(complete 3*complete 2*complete 1, _
     cup(complete 2**2*complete 1**2,complete 2**3)))
--R
--R
--R   (15)  1500
--R                                                    Type: Fraction(Integer)
--E 15

--S 16 of 47
square:=dihedral 4
```

```
--R
--R
--R         1       3   2    1     2    1   4
--R   (16)  - (4) + - (2 ) + - (2 1 ) + - (1 )
--R         4       8        4          8
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 16

--S 17 of 47
cap(complete 2**2,square)
--R
--R
--R   (17)  2
--R                                                        Type: Fraction(Integer)
--E 17

--S 18 of 47
cap(complete 3*complete 2**2,dihedral 7)
--R
--R
--R   (18)  18
--R                                                        Type: Fraction(Integer)
--E 18

--S 19 of 47
cap(graphs 5,complete 7*complete 3)
--R
--R
--R   (19)  4
--R                                                        Type: Fraction(Integer)
--E 19

--S 20 of 47
s(x) == powerSum(x)
--R
--R                                                                    Type: Void
--E 20

--S 21 of 47
cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)
--R
--R   Compiling function s with type PositiveInteger ->
--R      SymmetricPolynomial(Fraction(Integer))
--R
--R         1   2    1   2 2    3   4     1   8
--R   (21)  - (4 ) + - (3 1 ) + - (2 ) + -- (1 )
--R         4        3          8        24
--R                                  Type: SymmetricPolynomial(Fraction(Integer))
--E 21
```

```
--S 22 of 47
cap(complete 4**2,cube)
--R
--R
--R   (22)  7
--R                                                    Type: Fraction(Integer)
--E 22

--S 23 of 47
cap(complete 2**3*complete 1**2,wreath(elementary 4,elementary 2))
--R
--R
--R   (23)  7
--R                                                    Type: Fraction(Integer)
--E 23

--S 24 of 47
cap(complete 2**3*complete 1**2,wreath(elementary 4,complete 2))
--R
--R
--R   (24)  17
--R                                                    Type: Fraction(Integer)
--E 24

--S 25 of 47
cap(complete 2**3*complete 1**2,wreath(complete 4,elementary 2))
--R
--R
--R   (25)  10
--R                                                    Type: Fraction(Integer)
--E 25

--S 26 of 47
cap(complete 2**3*complete 1**2,wreath(complete 4,complete 2))
--R
--R
--R   (26)  23
--R                                                    Type: Fraction(Integer)
--E 26

--S 27 of 47
x: ULS(FRAC INT,'x,0) := 'x
--R
--R
--R   (27)  x
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 27

--S 28 of 47
ZeroOrOne: INT -> ULS(FRAC INT, 'x, 0)
```

```
--R
--R                                                                    Type: Void
--E 28

--S 29 of 47
Integers: INT -> ULS(FRAC INT, 'x, 0)
--R
--R                                                                    Type: Void
--E 29

--S 30 of 47
ZeroOrOne n == 1+x**n
--R
--R                                                                    Type: Void
--E 30

--S 31 of 47
ZeroOrOne 5
--R
--R   Compiling function ZeroOrOne with type Integer ->
--R      UnivariateLaurentSeries(Fraction(Integer),x,0)
--R
--R              5
--R   (31)  1 + x
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 31

--S 32 of 47
Integers n == 1/(1-x**n)
--R
--R                                                                    Type: Void
--E 32

--S 33 of 47
Integers 5
--R
--R   Compiling function Integers with type Integer ->
--R      UnivariateLaurentSeries(Fraction(Integer),x,0)
--R
--R              5    10      11
--R   (33)  1 + x  + x   + O(x  )
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 33

--S 34 of 47
)expose EVALCYC
--R
--I   EvaluateCycleIndicators is now explicitly exposed in frame frame0
--E 34
```

```
--S 35 of 47
eval(ZeroOrOne, graphs 5)
--R
--R
--R                   2     3     4     5     6     7     8    9    10      11
--R   (34)  1 + x + 2x  + 4x  + 6x  + 6x  + 6x  + 4x  + 2x  + x  + x   + O(x  )
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 35

--S 36 of 47
eval(ZeroOrOne,dihedral 8)
--R
--R
--R                   2     3     4     5     6    7    8
--R   (35)  1 + x + 4x  + 5x  + 8x  + 5x  + 4x  + x  + x
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 36

--S 37 of 47
eval(Integers,complete 4)
--R
--R
--R   (36)
--R             2     3     4     5     6      7      8      9      10      11
--R   1 + x + 2x  + 3x  + 5x  + 6x  + 9x  + 11x  + 15x  + 18x  + 23x   + O(x  )
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 37

--S 38 of 47
eval(Integers,elementary 4)
--R
--R
--R   (37)
--R      6    7     8     9     10     11     12      13      14      15      16
--R     x  + x  + 2x  + 3x  + 5x   + 6x   + 9x   + 11x   + 15x   + 18x   + 23x
--R   +
--R        17
--R     O(x  )
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 38

--S 39 of 47
eval(ZeroOrOne,cube)
--R
--R
--R                   2     3     4     5     6    7    8
--R   (38)  1 + x + 3x  + 3x  + 7x  + 3x  + 3x  + x  + x
--R                         Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 39
```

```
--S 40 of 47
eval(Integers,cube)
--R
--R
--R   (39)
--R               2     3      4      5      6       7       8       9       10
--R     1 + x + 4x  + 7x  + 21x  + 37x  + 85x  + 151x  + 292x  + 490x  + 848x
--R   +
--R        11
--R     O(x  )
--R                          Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 40

--S 41 of 47
eval(Integers,graphs 5)
--R
--R
--R   (40)
--R               2     3      4      5      6       7       8       9       10
--R     1 + x + 3x  + 7x  + 17x  + 35x  + 76x  + 149x  + 291x  + 539x  + 974x
--R   +
--R        11
--R     O(x  )
--R                          Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 41

--S 42 of 47
eval(ZeroOrOne ,graphs 15)
--R
--R
--R   (41)
--R               2     3      4      5      6       7       8        9        10
--R     1 + x + 2x  + 5x  + 11x  + 26x  + 68x  + 177x  + 496x  + 1471x  + 4583x
--R   +
--R        11
--R     O(x  )
--R                          Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 42

--S 43 of 47
cap(dihedral 30,complete 7*complete 8*complete 5*complete 10)
--R
--R
--R   (42)  49958972383320
--R                                                   Type: Fraction(Integer)
--E 43

--S 44 of 47
sf3221:= SFunction [3,2,2,1]
--R
```

```
--R
--R   (43)
--R      1          1     2     1   2     1            1     4     1   2
--R     -- (6 2) - -- (6 1 ) - -- (4 ) + -- (4 3 1) + -- (4 1 ) - -- (3 2)
--R     12         12          16        12           24          36
--R   +
--R      1   2 2     1     2      1       3     1     5     1    4     1   3 2
--R     -- (3 1 ) - -- (3 2 1) - -- (3 2 1 ) - -- (3 1 ) - --- (2 ) + -- (2 1 )
--R     36          24           36            72          192        48
--R   +
--R      1   2 4     1      6     1    8
--R     -- (2 1 ) - --- (2 1 ) + --- (1 )
--R     96          144          576
--R                                Type: SymmetricPolynomial(Fraction(Integer))
--E 44

--S 45 of 47
cap(sf3221,complete 2**4)
--R
--R
--R   (44)  3
--R                                                      Type: Fraction(Integer)
--E 45

--S 46 of 47
cap(sf3221, powerSum 1**8)
--R
--R
--R   (45)  70
--R                                                      Type: Fraction(Integer)
--E 46

--S 47 of 47
eval(Integers, sf3221)
--R
--R
--R   (46)
--R      9     10     11      12      13      14      15       16       17       18
--R     x  + 3x   + 7x   + 14x   + 27x   + 47x   + 79x   + 126x   + 196x   + 294x
--R   +
--R         19      20
--R     432x   + O(x  )
--R                          Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 47
)spool
)lisp (bye)
```

— CycleIndicators.help —

```
====================================================================
CycleIndicators examples
====================================================================

This section is based upon the paper J. H. Redfield, ‘‘The Theory of
Group-Reduced Distributions’’, American J. Math.,49 (1927) 433-455,
and is an application of group theory to enumeration problems.  It is
a development of the work by P. A. MacMahon on the application of
symmetric functions and Hammond operators to combinatorial theory.

The theory is based upon the power sum symmetric functions s(i) which
are the sum of the i-th powers of the variables.  The cycle index of a
permutation is an expression that specifies the sizes of the cycles of
a permutation, and may be represented as a partition.  A partition of
a non-negative integer n is a collection of positive integers called
its parts whose sum is n.  For example, the partition (3^2 2 1^2) will
be used to represent s^2_3 s_2 s^2_1 and will indicate that the
permutation has two cycles of length 3, one of length 2 and two of
length 1.  The cycle index of a permutation group is the sum of the
cycle indices of its permutations divided by the number of
permutations.  The cycle indices of certain groups are provided.

The operation complete returns the cycle index of the symmetric group
of order n for argument n.  Alternatively, it is the n-th complete
homogeneous symmetric function expressed in terms of power sum
symmetric functions.

  complete 1
   (1)
                      Type: SymmetricPolynomial Fraction Integer

  complete 2
   1       1   2
   - (2) + - (1 )
   2       2
                      Type: SymmetricPolynomial Fraction Integer

  complete 3
   1       1         1   3
   - (3) + - (2 1) + - (1 )
   3       2         6
                      Type: SymmetricPolynomial Fraction Integer

  complete 7
     1       1          1          1     2     1         1            1     3
     - (7) + - (6 1) + -- (5 2) + -- (5 1 ) + -- (4 3) + - (4 2 1) + -- (4 1 )
     7       6          10         10          12         8           24
   +
```

```
     1   2      1     2     1       2     1     4     1   3      1   2 3
    -- (3 1) + -- (3 2 ) + -- (3 2 1 ) + -- (3 1 ) + -- (2 1) + -- (2 1 )
    18         24          12            72          48         48
  +
     1      5      1    7
    --- (2 1 ) + ---- (1 )
    240          5040
                  Type: SymmetricPolynomial Fraction Integer
```

The operation elementary computes the n-th elementary symmetric function for argument n.

```
  elementary 7
     1       1          1          1     2     1         1            1     3
    - (7) - - (6 1) - -- (5 2) + -- (5 1 ) - -- (4 3) + - (4 2 1) - -- (4 1 )
    7       6          10         10          12         8           24
  +
     1   2      1     2     1       2     1     4     1   3      1   2 3
    -- (3 1) + -- (3 2 ) - -- (3 2 1 ) + -- (3 1 ) - -- (2 1) + -- (2 1 )
    18         24          12            72          48         48
  +
       1      5      1    7
    - --- (2 1 ) + ---- (1 )
      240          5040
                  Type: SymmetricPolynomial Fraction Integer
```

The operation alternating returns the cycle index of the alternating group having an even number of even parts in each cycle partition.

```
  alternating 7
     2       1     2    1           1   2      1     2     1     4     1   2 3
    - (7) + - (5 1 ) + - (4 2 1) + - (3 1) + -- (3 2 ) + -- (3 1 ) + -- (2 1 )
    7       5          4           9         12          36          24
  +
      1    7
    ---- (1 )
    2520
                  Type: SymmetricPolynomial Fraction Integer
```

The operation cyclic returns the cycle index of the cyclic group.

```
  cyclic 7
   6       1   7
   - (7) + - (1 )
   7       7
                  Type: SymmetricPolynomial Fraction Integer
```

The operation dihedral is the cycle index of the dihedral group.

```
  dihedral 7
```

```
  3       1   3      1   7
  - (7) + - (2 1) + -- (1 )
  7       2         14
                Type: SymmetricPolynomial Fraction Integer
```

The operation graphs for argument n returns the cycle index of the group of permutations on the edges of the complete graph with n nodes induced by applying the symmetric group to the nodes.

```
  graphs 5
   1           1   2    1   2     1   3     1   4 2     1   3 4     1    10
  - (6 3 1) + - (5 ) + - (4 2) + - (3 1) + - (2 1 ) + -- (2 1 ) + --- (1  )
  6           5        4         6         8          12          120
                Type: SymmetricPolynomial Fraction Integer
```

The cycle index of a direct product of two groups is the product of the cycle indices of the groups. Redfield provided two operations on two cycle indices which will be called "cup" and "cap" here. The cup of two cycle indices is a kind of scalar product that combines monomials for permutations with the same cycles. The cap operation provides the sum of the coefficients of the result of the cup operation which will be an integer that enumerates what Redfield called group-reduced distributions.

We can, for example, represent complete 2 * complete 2 as the set of objects a a b b and complete 2 * complete 1 * complete 1 as c c d e.

This integer is the number of different sets of four pairs.

```
  cap(complete 2**2, complete 2*complete 1**2)
    4
                Type: Fraction Integer
```

For example,

```
  a a b b     a a b b    a a b b   a a b b
  c c d e     c d c e    c e c d   d e c c
```

This integer is the number of different sets of four pairs no two pairs being equal.

```
  cap(elementary 2**2, complete 2*complete 1**2)
    2
                Type: Fraction Integer
```

For example,

```
  a a b b    a a b b
  c d c e    c e c d
```

In this case the configurations enumerated are easily constructed,

```
however the theory merely enumerates them providing little help in
actually constructing them.

Here are the number of 6-pairs, first from a a a b b c, second
from d d e e f g.

  cap(complete 3*complete 2*complete 1,complete 2**2*complete 1**2)
    24
                    Type: Fraction Integer

Here it is again, but with no equal pairs.

  cap(elementary 3*elementary 2*elementary 1,complete 2**2*complete 1**2)
    8
                    Type: Fraction Integer

  cap(complete 3*complete 2*complete 1,elementary 2**2*elementary 1**2)
    8
                    Type: Fraction Integer

The number of 6-triples, first from a a a b b c, second from
d d e e f g, third from h h i i j j.

  eval(cup(complete 3*complete 2*complete 1, cup(complete 2**2*complete 1**2,complete 2**3)))
    1500
                    Type: Fraction Integer

The cycle index of vertices of a square is dihedral 4.

  square:=dihedral 4
   1       3   2    1     2    1   4
   - (4) + - (2 ) + - (2 1 ) + - (1 )
   4       8        4          8
                    Type: SymmetricPolynomial Fraction Integer

The number of different squares with 2 red vertices and 2 blue vertices.

  cap(complete 2**2,square)
    2
                    Type: Fraction Integer

The number of necklaces with 3 red beads, 2 blue beads and 2 green beads.

  cap(complete 3*complete 2**2,dihedral 7)
    18
                    Type: Fraction Integer

The number of graphs with 5 nodes and 7 edges.

  cap(graphs 5,complete 7*complete 3)
```

```
   4
                    Type: Fraction Integer
```

The cycle index of rotations of vertices of a cube.

```
  s(x) == powerSum(x)
                    Type: Void

  cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)
   1   2    1   2 2    3   4     1   8
   - (4 ) + - (3 1 ) + - (2 ) + -- (1 )
   4        3          8        24
                    Type: SymmetricPolynomial Fraction Integer
```

The number of cubes with 4 red vertices and 4 blue vertices.

```
  cap(complete 4**2,cube)
   7
                    Type: Fraction Integer
```

The number of labeled graphs with degree sequence 2 2 2 1 1 with no loops or multiple edges.

```
  cap(complete 2**3*complete 1**2,wreath(elementary 4,elementary 2))
   7
                    Type: Fraction Integer
```

Again, but with loops allowed but not multiple edges.

```
  cap(complete 2**3*complete 1**2,wreath(elementary 4,complete 2))
   17
                    Type: Fraction Integer
```

Again, but with multiple edges allowed, but not loops

```
  cap(complete 2**3*complete 1**2,wreath(complete 4,elementary 2))
   10
                    Type: Fraction Integer
```

Again, but with both multiple edges and loops allowed

```
  cap(complete 2**3*complete 1**2,wreath(complete 4,complete 2))
   23
                    Type: Fraction Integer
```

Having constructed a cycle index for a configuration we are at liberty to evaluate the s_i components any way we please. For example we can produce enumerating generating functions. This is done by providing a function f on an integer i to the value required of s_i, and then evaluating eval(f, cycleindex).

```
  x: ULS(FRAC INT,'x,0) := 'x
   x
                    Type: UnivariateLaurentSeries(Fraction Integer,x,0)

  ZeroOrOne: INT -> ULS(FRAC INT, 'x, 0)
                    Type: Void

  Integers: INT -> ULS(FRAC INT, 'x, 0)
                    Type: Void
```

For the integers 0 and 1, or two colors.

```
  ZeroOrOne n == 1+x**n
                    Type: Void

  ZeroOrOne 5
         5
    1 + x
                    Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

For the integers 0, 1, 2, ... we have this.

```
  Integers n == 1/(1-x**n)
                    Type: Void

  Integers 5
         5    10      11
    1 + x  + x   + O(x  )
                    Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of graphs with 5 nodes and n edges.

Note that there is an eval function that takes two arguments. It has the signature:

```
  ((Integer -> D1),SymmetricPolynomial Fraction Integer) -> D1
    from EvaluateCycleIndicators D1 if D1 has ALGEBRA FRAC INT
```

This function is not normally exposed (it will not normally be considered in the list of eval functions) as it is only useful for this particular domain. To use it we ask that it be considered thus:

```
  )expose EVALCYC
```

and now we can use it:

```
  eval(ZeroOrOne, graphs 5)
              2     3     4     5     6     7     8    9    10      11
    1 + x + 2x  + 4x  + 6x  + 6x  + 6x  + 4x  + 2x  + x  + x   + O(x  )
```

```
                  Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of necklaces with n red beads
and n-8 green beads.

```
  eval(ZeroOrOne,dihedral 8)
             2     3     4     5     6    7    8
    1 + x + 4x  + 5x  + 8x  + 5x  + 4x  + x  + x
                  Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of partitions of n into 4 or fewer parts.

```
  eval(Integers,complete 4)
            2     3     4     5     6      7      8      9      10      11
   1 + x + 2x  + 3x  + 5x  + 6x  + 9x  + 11x  + 15x  + 18x  + 23x   + O(x  )
                  Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of partitions of n into 4 boxes
containing ordered distinct parts.

```
  eval(Integers,elementary 4)
      6    7     8     9     10     11     12      13      14      15      16
     x  + x  + 2x  + 3x  + 5x   + 6x   + 9x   + 11x   + 15x   + 18x   + 23x
   +
        17
     O(x  )
                  Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of different cubes with n red
vertices and 8-n green ones.

```
  eval(ZeroOrOne,cube)
             2     3     4     5     6    7    8
    1 + x + 3x  + 3x  + 7x  + 3x  + 3x  + x  + x
                  Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of different cubes with integers
on the vertices whose sum is n.

```
  eval(Integers,cube)
              2     3      4      5      6       7       8       9       10
     1 + x + 4x  + 7x  + 21x  + 37x  + 85x  + 151x  + 292x  + 490x  + 848x
   +
        11
     O(x  )
                        Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

The coefficient of x^n is the number of graphs with 5 nodes and with
integers on the edges whose sum is n. In other words, the enumeration
is of multigraphs with 5 nodes and n edges.

```
  eval(Integers,graphs 5)
              2     3      4      5      6       7       8       9       10
    1 + x + 3x  + 7x  + 17x  + 35x  + 76x  + 149x  + 291x  + 539x  + 974x
 +
       11
    O(x  )
                    Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

Graphs with 15 nodes enumerated with respect to number of edges.

```
  eval(ZeroOrOne ,graphs 15)
              2     3      4      5      6       7       8        9        10
    1 + x + 2x  + 5x  + 11x  + 26x  + 68x  + 177x  + 496x  + 1471x  + 4583x
 +
       11
    O(x  )
                    Type: UnivariateLaurentSeries(Fraction Integer,x,0)
```

Necklaces with 7 green beads, 8 white beads, 5 yellow beads and 10
red beads.

```
  cap(dihedral 30,complete 7*complete 8*complete 5*complete 10)
    49958972383320
                    Type: Fraction Integer
```

The operation SFunction is the S-function or Schur function of a
partition written as a descending list of integers expressed in terms
of power sum symmetric functions.

In this case the argument partition represents a tableau shape. For
example 3,2,2,1 represents a tableau with three boxes in the first
row, two boxes in the second and third rows, and one box in the fourth
row. SFunction [3,2,2,1] counts the number of different tableaux of
shape 3, 2, 2, 1 filled with objects with an ascending order in the
columns and a non-descending order in the rows.

```
  sf3221:= SFunction [3,2,2,1]
      1          1     2     1   2     1            1     4     1   2
    -- (6 2) - -- (6 1 ) - -- (4 ) + -- (4 3 1) + -- (4 1 ) - -- (3 2)
    12         12          16        12           24          36
 +
     1   2 2     1     2      1       3     1     5     1    4     1   3 2
    -- (3 1 ) - -- (3 2 1) - -- (3 2 1 ) - -- (3 1 ) - --- (2 ) + -- (2 1 )
    36          24           36            72          192        48
 +
     1   2 4     1      6     1    8
    -- (2 1 ) - --- (2 1 ) + --- (1 )
    96          144          576
                    Type: SymmetricPolynomial Fraction Integer
```

```
This is the number filled with a a b b c c d d.

  cap(sf3221,complete 2**4)
    3
                    Type: Fraction Integer

The configurations enumerated above are:

  a a b    a a c    a a d
  b c      b b      b b
  c d      c d      c c
  d        d        d

This is the number of tableaux filled with 1..8.

  cap(sf3221, powerSum 1**8)
    70
                    Type: Fraction Integer

The coefficient of x^n is the number of column strict reverse plane
partitions of n of shape 3 2 2 1.

  eval(Integers, sf3221)
      9     10     11      12      13      14      15       16       17
     x  + 3x   + 7x   + 14x   + 27x   + 47x   + 79x   + 126x   + 196x
   +
        18      19      20
    294x  + 432x   + O(x  )
                         Type: UnivariateLaurentSeries(Fraction Integer,x,0)

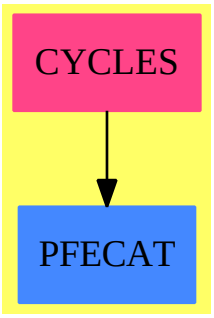
The smallest is

  0 0 0
  1 1
  2 2
  3

See Also:
o )show CycleIndicators
```

———-

4.22.1 CycleIndicators (CYCLES)



Exports:

alternating	cap	complete	cup	cyclic
dihedral	elementary	eval	graphs	powerSum
SFunction	skewSFunction	wreath		

— package CYCLES CycleIndicators —

```
)abbrev package CYCLES CycleIndicators
++ Author: William H. Burge
++ Date Created: 1986
++ Date Last Updated: 11 Feb 1992
++ Keywords:Polya, Redfield, enumeration
++ Examples:
++ References: J.H.Redfield, 'The Theory of Group-Reduced Distributions',
++             American J. Math., 49 (1927) 433-455.
++             G.Polya, 'Kombinatorische Anzahlbestimmungen fur Gruppen,
++               Graphen und chemische Verbindungen', Acta Math. 68
++                (1937) 145-254.
++ Description:
++ Polya-Redfield enumeration by cycle indices.

CycleIndicators: Exports == Implementation where
  I    ==> Integer
  L    ==> List
  B    ==> Boolean
  SPOL ==> SymmetricPolynomial
  PTN  ==> Partition
  RN   ==> Fraction Integer
  FR   ==> Factored Integer
  h ==> complete
  s ==> powerSum
  --a ==> elementary
  alt ==> alternating
  cyc ==> cyclic
  dih ==> dihedral
```

```
  ev == eval
  Exports ==> with

    complete: I -> SPOL RN
      ++\spad{complete n} is the \spad{n} th complete homogeneous
      ++ symmetric function expressed in terms of power sums.
      ++ Alternatively it is the cycle index of the symmetric
      ++ group of degree n.

    powerSum: I -> SPOL RN
      ++\spad{powerSum n} is the \spad{n} th power sum symmetric
      ++ function.

    elementary: I -> SPOL RN
      ++\spad{elementary n} is the \spad{n} th elementary symmetric
      ++ function expressed in terms of power sums.

 -- s2h: I -> SPOL RN--s to h

    alternating: I -> SPOL RN
      ++\spad{alternating n} is the cycle index of the
      ++ alternating group of degree n.

    cyclic: I -> SPOL RN    --cyclic group
      ++\spad{cyclic n} is the cycle index of the
      ++ cyclic group of degree n.

    dihedral: I -> SPOL RN    --dihedral group
      ++\spad{dihedral n} is the cycle index of the
      ++ dihedral group of degree n.

    graphs: I -> SPOL RN
      ++\spad{graphs n} is the cycle index of the group induced on
      ++ the edges of a graph by applying the symmetric function to the
      ++ n nodes.

    cap: (SPOL RN,SPOL RN) -> RN
      ++\spad{cap(s1,s2)}, introduced by Redfield,
      ++ is the scalar product of two cycle indices.

    cup: (SPOL RN,SPOL RN) -> SPOL RN
      ++\spad{cup(s1,s2)}, introduced by Redfield,
      ++ is the scalar product of two cycle indices, in which the
      ++ power sums are retained to produce a cycle index.

    eval: SPOL RN -> RN
      ++\spad{eval s} is the sum of the coefficients of a cycle index.

    wreath: (SPOL RN,SPOL RN) -> SPOL RN
      ++\spad{wreath(s1,s2)} is the cycle index of the wreath product
```

```
    ++ of the two groups whose cycle indices are \spad{s1} and
    ++ \spad{s2}.

  SFunction:L I -> SPOL RN
    ++\spad{SFunction(li)} is the S-function of the partition \spad{li}
    ++ expressed in terms of power sum symmetric functions.

  skewSFunction:(L I,L I) -> SPOL RN
    ++\spad{skewSFunction(li1,li2)} is the S-function
    ++ of the partition difference \spad{li1 - li2}
    ++ expressed in terms of power sum symmetric functions.

Implementation ==> add
  import PartitionsAndPermutations
  import IntegerNumberTheoryFunctions

  trm: PTN -> SPOL RN
  trm pt == monomial(inv(pdct(pt) :: RN),pt)

  list: Stream L I -> L L I
  list st == entries complete st

  complete i ==
         if i=0
         then 1
         else if i<0
              then 0
              else
                 _+/[trm(partition pt) for pt in list(partitions i)]


  even?: L I -> B
  even? li == even?( #([i for i in li | even? i]))

  alt i ==
    2 * _+/[trm(partition li) for li in list(partitions i) | even? li]
  elementary i ==
         if i=0
         then 1
         else if i<0
              then 0
              else
                _+/[(spol := trm(partition pt); even? pt => spol; -spol)
                        for pt in list(partitions i)]

  divisors: I -> L I
  divisors n ==
    b := factors(n :: FR)
    c := concat(1,"append"/
               [[a.factor**j for j in 1..a.exponent] for a in b]);
```

```
    if #(b) = 1 then c else concat(n,c)

ss: (I,I) -> SPOL RN
ss(n,m) ==
  li : L I := [n for j in 1..m]
  monomial(1,partition li)

s n == ss(n,1)

cyc n ==
  n = 1 => s 1
  _+/[(eulerPhi(i) / n) * ss(i,numer(n/i)) for i in divisors n]

dih n ==
  k := n quo 2
  odd? n => (1/2) * cyc n + (1/2) * ss(2,k) * s 1
  (1/2) * cyc n + (1/4) * ss(2,k) + (1/4) * ss(2,k-1) * ss(1,2)

trm2: L I -> SPOL RN
trm2 li ==
  lli := powers(li)$PTN
  xx := 1/(pdct partition li)
  prod : SPOL RN := 1
  for ll in lli repeat
    ll0 := first ll; ll1 := second ll
    k := ll0 quo 2
    c :=
      odd? ll0 => ss(ll0,ll1 * k)
      ss(k,ll1) * ss(ll0,ll1 * (k - 1))
    c := c * ss(ll0,ll0 * ((ll1*(ll1 - 1)) quo 2))
    prod2 : SPOL RN := 1
    for r in lli | first(r) < ll0 repeat
      r0 := first r; r1 := second r
      prod2 := ss(lcm(r0,ll0),gcd(r0,ll0) * r1 * ll1) * prod2
    prod := c * prod2 * prod
  xx * prod

graphs n == _+/[trm2 li for li in list(partitions n)]

cupp: (PTN,SPOL RN) -> SPOL RN
cupp(pt,spol) ==
  zero? spol => 0
  (dg := degree spol) < pt => 0
  dg = pt => (pdct pt) * monomial(leadingCoefficient spol,dg)
  cupp(pt,reductum spol)

cup(spol1,spol2) ==
  zero? spol1 => 0
  p := leadingCoefficient(spol1) * cupp(degree spol1,spol2)
  p + cup(reductum spol1,spol2)
```

```
    ev spol ==
      zero? spol => 0
      leadingCoefficient(spol) + ev(reductum spol)

    cap(spol1,spol2) == ev cup(spol1,spol2)

    mtpol: (I,SPOL RN) -> SPOL RN
    mtpol(n,spol)==
      zero? spol => 0
      deg := partition [n*k for k in (degree spol)::L(I)]
      monomial(leadingCoefficient spol,deg) + mtpol(n,reductum spol)

    fn2: I -> SPOL RN
    evspol: ((I -> SPOL RN),SPOL RN) -> SPOL RN
    evspol(fn2,spol) ==
      zero? spol => 0
      lc := leadingCoefficient spol
      prod := _*/[fn2 i for i in (degree spol)::L(I)]
      lc * prod + evspol(fn2,reductum spol)

    wreath(spol1,spol2) == evspol(x+->mtpol(x,spol2),spol1)

    hh: I -> SPOL RN      --symmetric group
    hh n == if n=0 then 1 else if n<0 then 0 else h n
    SFunction li==
      a:Matrix SPOL RN:=matrix [[hh(k -j+i) for k in li for j in 1..#li]
                    for i in 1..#li]
      determinant a

    roundup:(L I,L I)-> L I
    roundup(li1,li2)==
              #li1 > #li2 => roundup(li1,concat(li2,0))
              li2

    skewSFunction(li1,li2)==
      #li1 < #li2 =>
        error "skewSFunction: partition1 does not include partition2"
      li2:=roundup (li1,li2)
      a:Matrix SPOL RN:=matrix [[hh(k-li2.i-j+i)
               for k in li1 for j in 1..#li1]  for i in 1..#li1]
      determinant a
```

———-

— CYCLES.dotabb —

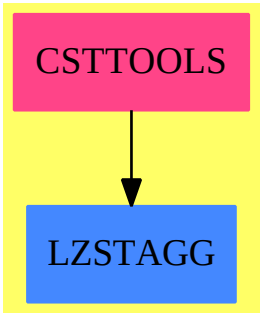
```
"CYCLES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLES"]
```

```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLES" -> "PFECAT"
```

———-

4.23 package CSTTOOLS CyclicStreamTools

4.23.1 CyclicStreamTools (CSTTOOLS)



Exports:
computeCycleEntry computeCycleLength cycleElt

— **package CSTTOOLS CyclicStreamTools** —

```
)abbrev package CSTTOOLS CyclicStreamTools
++ Author: Clifton J. Williamson
++ Date Created: 5 December 1989
++ Date Last Updated: 5 December 1989
++ Keywords: stream, cyclic
++ Description:
++ This package provides tools for working with cyclic streams.

CyclicStreamTools(S,ST): Exports == Implementation where
  S  : Type
  ST : LazyStreamAggregate S

  Exports ==> with

    cycleElt: ST -> Union(ST,"failed")
      ++ cycleElt(s) returns a pointer to a node in the cycle if the stream
      ++ s is cyclic and returns "failed" if s is not cyclic
      ++
      ++X p:=repeating([1,2,3])
      ++X q:=cons(4,p)
```

```
    ++X cycleElt q
    ++X r:=[1,2,3]::Stream(Integer)
    ++X cycleElt r

  computeCycleLength: ST -> NonNegativeInteger
    ++ computeCycleLength(s) returns the length of the cycle of a
    ++ cyclic stream t, where s is a pointer to a node in the
    ++ cyclic part of t.
    ++
    ++X p:=repeating([1,2,3])
    ++X q:=cons(4,p)
    ++X computeCycleLength(cycleElt(q))

  computeCycleEntry: (ST,ST) -> ST
    ++ computeCycleEntry(x,cycElt), where cycElt is a pointer to a
    ++ node in the cyclic part of the cyclic stream x, returns a
    ++ pointer to the first node in the cycle
    ++
    ++X p:=repeating([1,2,3])
    ++X q:=cons(4,p)
    ++X computeCycleEntry(q,cycleElt(q))

Implementation ==> add

  cycleElt x ==
    y := x
    for i in 0.. repeat
      (explicitlyEmpty? y) or (lazy? y) => return "failed"
      y := rst y
      if odd? i then x := rst x
      eq?(x,y) => return y

  computeCycleLength cycElt ==
    i : NonNegativeInteger
    y := cycElt
    for i in 1.. repeat
      y := rst y
      eq?(y,cycElt) => return i

  computeCycleEntry(x,cycElt) ==
    y := rest(x, computeCycleLength cycElt)
    repeat
      eq?(x,y) => return x
      x := rst x ; y := rst y
```

———-

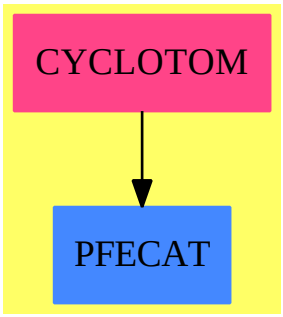
— CSTTOOLS.dotabb —

```
"CSTTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CSTTOOLS"]
"LZSTAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LZSTAGG"]
"CSTTOOLS" -> "LZSTAGG"
```

———-

4.24 package CYCLOTOM CyclotomicPolynomialPackage

4.24.1 CyclotomicPolynomialPackage (CYCLOTOM)



Exports:
cyclotomic cyclotomicDecomposition cyclotomicFactorization

— **package CYCLOTOM CyclotomicPolynomialPackage** —

```
)abbrev package CYCLOTOM CyclotomicPolynomialPackage
++ Description:
++ This package has no description

CyclotomicPolynomialPackage: public == private where
  SUP  ==> SparseUnivariatePolynomial(Integer)
  LSUP ==> List(SUP)
  NNI  ==> NonNegativeInteger
  FR   ==> Factored SUP
  IFP  ==> IntegerFactorizationPackage Integer

  public == with
    cyclotomicDecomposition: Integer -> LSUP
      ++ cyclotomicDecomposition(n) \undocumented{}
    cyclotomic:              Integer -> SUP
      ++ cyclotomic(n) \undocumented{}
    cyclotomicFactorization: Integer -> FR
```

```
      ++ cyclotomicFactorization(n) \undocumented{}

  private == add
    cyclotomic(n:Integer): SUP ==
      x,y,z,l: SUP
      g := factors factor(n)$IFP
      --Now, for each prime in the factorization apply recursion
      l := monomial(1,1) - monomial(1,0)
      for u in g repeat
         l := (monicDivide(multiplyExponents(l,u.factor::NNI),l)).quotient
         if u.exponent>1 then
            l := multiplyExponents(l,((u.factor)**((u.exponent-1)::NNI))::NNI)
      l

    cyclotomicDecomposition(n:Integer):LSUP ==
      x,y,z: SUP
      l,ll,m: LSUP
      rr: Integer
      g := factors factor(n)$IFP
      l := [monomial(1,1) - monomial(1,0)]
      --Now, for each prime in the factorization apply recursion
      for u in g repeat
         m := [(monicDivide(
            multiplyExponents(z,u.factor::NNI),z)).quotient for z in l]
         for rr in 1..(u.exponent-1) repeat
            l := append(l,m)
            m := [multiplyExponents(z,u.factor::NNI) for z in m]
         l := append(l,m)
      l

    cyclotomicFactorization(n:Integer):FR ==
      f :  SUP
      fr : FR := 1$FR
      for f in cyclotomicDecomposition(n) repeat
        fr := fr * primeFactor(f,1$Integer)
      fr
```

———-

— CYCLOTOM.dotabb —

```
"CYCLOTOM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLOTOM"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLOTOM" -> "PFECAT"
```

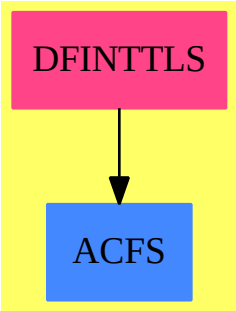
———-

Chapter 5

Chapter D

5.1 package DFINTTLS DefiniteIntegrationTools

5.1.1 DefiniteIntegrationTools (DFINTTLS)



Exports:
ignore? checkForZero computeInt

— package DFINTTLS DefiniteIntegrationTools —

```
)abbrev package DFINTTLS DefiniteIntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 15 April 1992
++ Date Last Updated: 24 February 1993
++ Description:
++ \spadtype{DefiniteIntegrationTools} provides common tools used
++ by the definite integration of both rational and elementary functions.

DefiniteIntegrationTools(R, F): Exports == Implementation where
```

```
R : Join(GcdDomain, OrderedSet, RetractableTo Integer,
         LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory,
         AlgebraicallyClosedFunctionSpace R)

B   ==> Boolean
Z   ==> Integer
Q   ==> Fraction Z
SE  ==> Symbol
P   ==> Polynomial R
RF  ==> Fraction P
UP  ==> SparseUnivariatePolynomial F
K   ==> Kernel F
OFE ==> OrderedCompletion F
UPZ ==> SparseUnivariatePolynomial Z
UPQ ==> SparseUnivariatePolynomial Q
REC ==> Record(left:Q, right:Q)
REC2==> Record(endpoint:Q, dir:Z)
U   ==> Union(fin:REC, halfinf:REC2, all:"all", failed:"failed")
IGNOR ==> "noPole"

Exports ==> with
  ignore?: String -> B
    ++ ignore?(s) is true if s is the string that tells the integrator
    ++ to assume that the function has no pole in the integration interval.
  computeInt: (K, F, OFE, OFE, B) -> Union(OFE, "failed")
    ++ computeInt(x, g, a, b, eval?) returns the integral of \spad{f} for x
    ++ between a and b, assuming that g is an indefinite integral of
    ++ \spad{f} and \spad{f} has no pole between a and b.
    ++ If \spad{eval?} is true, then \spad{g} can be evaluated safely
    ++ at \spad{a} and \spad{b}, provided that they are finite values.
    ++ Otherwise, limits must be computed.
  checkForZero: (P,  SE, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, x, a, b, incl?) is true if p has a zero for x between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.
  checkForZero: (UP, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, a, b, incl?) is true if p has a zero between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.

Implementation ==> add
  import RealZeroPackage UPZ
  import InnerPolySign(F, UP)
  import ElementaryFunctionSign(R, F)
  import PowerSeriesLimitPackage(R, F)
  import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)

  mkLogPos    : F -> F
  keeprec?    : (Q, REC) -> B
```

```
negative    : F -> Union(B, "failed")
mkKerPos    : K -> Union(F, "positive")
posRoot     : (UP, B) -> Union(B, "failed")
realRoot    : UP -> Union(B, "failed")
var         : UP -> Union(Z, "failed")
maprat      : UP -> Union(UPZ, "failed")
variation   : (UP, F) -> Union(Z, "failed")
infeval     : (UP, OFE) -> Union(F, "failed")
checkHalfAx : (UP, F, Z, B) -> Union(B, "failed")
findLimit   : (F, K, OFE, String, B) -> Union(OFE, "failed")
checkBudan  : (UP, OFE, OFE, B) -> Union(B, "failed")
checkDeriv  : (UP, OFE, OFE) -> Union(B, "failed")
sameSign    : (UP, OFE, OFE) -> Union(B, "failed")
intrat      : (OFE, OFE) -> U
findRealZero: (UPZ, U, B) -> List REC

variation(p, a)      == var p(monomial(1, 1)$UP - a::UP)
keeprec?(a, rec)     == (a > rec.right) or (a < rec.left)

checkHalfAx(p, a, d, incl?) ==
  posRoot(p(d * (monomial(1, 1)$UP - a::UP)), incl?)

ignore? str ==
  str = IGNOR => true
  error "integrate: last argument must be 'noPole'"

computeInt(k, f, a, b, eval?) ==
  is?(f, "integral"::SE) => "failed"
  if not eval? then f := mkLogPos f
  ((ib := findLimit(f, k, b, "left", eval?)) case "failed") or
      ((ia := findLimit(f, k, a, "right", eval?)) case "failed") => "failed"
  infinite?(ia::OFE) and (ia::OFE = ib::OFE) => "failed"
  ib::OFE - ia::OFE

findLimit(f, k, a, dir, eval?) ==
  r := retractIfCan(a)@Union(F, "failed")
  r case F =>
    eval? => mkLogPos(eval(f, k, r::F))::OFE
    (u := limit(f, equation(k::F, r::F), dir)) case OFE => u::OFE
    "failed"
  (u := limit(f, equation(k::F::OFE, a))) case OFE => u::OFE
  "failed"

mkLogPos f ==
  lk := empty()$List(K)
  lv := empty()$List(F)
  for k in kernels f | is?(k, "log"::SE) repeat
    if (v := mkKerPos k) case F then
      lk := concat(k, lk)
      lv := concat(v::F, lv)
```

```
  eval(f, lk, lv)

mkKerPos k ==
  (u := negative(f := first argument k)) case "failed" =>
                                                log(f**2) / (2::F)
  u::B => log(-f)
  "positive"

negative f ==
  (u := sign f) case "failed" => "failed"
  u::Z < 0

checkForZero(p, x, a, b, incl?) ==
  checkForZero(
    map(s+->s::F, univariate(p, x))_
     $SparseUnivariatePolynomialFunctions2(P, F),
        a, b, incl?)

checkForZero(q, a, b, incl?) ==
  ground? q => false
  (d := maprat q) case UPZ and not((i := intrat(a, b)) case failed) =>
      not empty? findRealZero(d::UPZ, i, incl?)
  (u := checkBudan(q, a, b, incl?)) case "failed" =>
     incl? => checkDeriv(q, a, b)
     "failed"
  u::B

maprat p ==
  ans:UPQ := 0
  while p ^= 0 repeat
    (r := retractIfCan(c := leadingCoefficient p)@Union(Q,"failed"))
      case "failed"  => return "failed"
    ans := ans + monomial(r::Q, degree p)
    p   := reductum p
  map(numer,(splitDenominator ans).num
     )$SparseUnivariatePolynomialFunctions2(Q, Z)

intrat(a, b) ==
  (n := whatInfinity a) ^= 0 =>
    (r := retractIfCan(b)@Union(F,"failed")) case "failed" => ["all"]
    (q := retractIfCan(r::F)@Union(Q, "failed")) case "failed" =>
      ["failed"]
    [[q::Q, n]]
  (q := retractIfCan(retract(a)@F)@Union(Q,"failed")) case "failed"
    => ["failed"]
  (n := whatInfinity b) ^= 0 => [[q::Q, n]]
  (t := retractIfCan(retract(b)@F)@Union(Q,"failed")) case "failed"
    => ["failed"]
  [[q::Q, t::Q]]
```

```
    findRealZero(p, i, incl?) ==
      i case fin =>
        l := realZeros(p, r := i.fin)
        incl? => l
        select_!(s+->keeprec?(r.left, s) and keeprec?(r.right, s), l)
      i case all => realZeros p
      i case halfinf =>
        empty?(l := realZeros p) => empty()
        bounds:REC :=
          i.halfinf.dir > 0 => [i.halfinf.endpoint, "max"/[t.right for t in l]]
          ["min"/[t.left for t in l], i.halfinf.endpoint]
        l := [u::REC for t in l | (u := refine(p, t, bounds)) case REC]
        incl? => l
        ep := i.halfinf.endpoint
        select_!(s+->keeprec?(ep, s), l)
      error "findRealZero: should not happpen"

    checkBudan(p, a, b, incl?) ==
      r := retractIfCan(b)@Union(F, "failed")
      (n := whatInfinity a) ^= 0 =>
        r case "failed" => realRoot p
        checkHalfAx(p, r::F, n, incl?)
      (za? := zero? p(aa := retract(a)@F)) and incl? => true
      (n := whatInfinity b) ^= 0 => checkHalfAx(p, aa, n, incl?)
      (zb? := zero? p(bb := r::F)) and incl? => true
      (va := variation(p, aa)) case "failed" or
                    (vb := variation(p, bb)) case "failed" => "failed"
      m:Z := 0
      if za? then m := inc m
      if zb? then m := inc m
      odd?(v := va::Z - vb::Z) =>          -- p has an odd number of roots
        incl? or even? m => true
--        one? v => false
        (v = 1) => false
        "failed"
      zero? v => false                      -- p has no roots
--      one? m => true                     -- p has an even number > 0 of roots
      (m = 1) => true                     -- p has an even number > 0 of roots
      "failed"

    checkDeriv(p, a, b) ==
      (r := retractIfCan(p)@Union(F, "failed")) case F => zero?(r::F)
      (s := sameSign(p, a, b)) case "failed" => "failed"
      s::B =>                   -- p has the same nonzero sign at a and b
        (u := checkDeriv(differentiate p,a,b)) case "failed" => "failed"
        u::B => "failed"
        false
      true

    realRoot p ==
```

```
      (b := posRoot(p, true)) case "failed" => "failed"
      b::B => true
      posRoot(p(p - monomial(1, 1)$UP), true)

    sameSign(p, a, b) ==
      (ea := infeval(p, a)) case "failed" => "failed"
      (eb := infeval(p, b)) case "failed" => "failed"
      (s := sign(ea::F * eb::F)) case "failed" => "failed"
      s::Z > 0

-- returns true if p has a positive root. Include 0 is incl0? is true
    posRoot(p, incl0?) ==
      (z0? := zero?(coefficient(p, 0))) and incl0? => true
      (v := var p) case "failed" => "failed"
      odd?(v::Z) =>            -- p has an odd number of positive roots
        incl0? or not(z0?) => true
--        one?(v::Z) => false
        (v::Z) = 1 => false
        "failed"
      zero?(v::Z) => false     -- p has no positive roots
      z0? => true              -- p has an even number > 0 of positive roots
      "failed"

    infeval(p, a) ==
      zero?(n := whatInfinity a) => p(retract(a)@F)
      (u := signAround(p, n, sign)) case "failed" => "failed"
      u::Z::F

    var q ==
      i:Z := 0
      (lastCoef := negative leadingCoefficient q) case "failed" =>
        "failed"
      while ((q := reductum q) ^= 0) repeat
        (next := negative leadingCoefficient q) case "failed" =>
          return "failed"
        if ((not(lastCoef::B)) and next::B) or
                        ((not(next::B)) and lastCoef::B) then i := i + 1
        lastCoef := next
      i
```

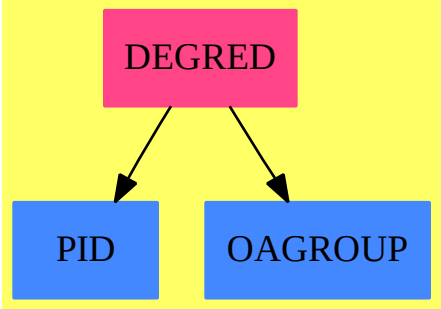
— DFINTTLS.dotabb —

```
"DFINTTLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DFINTTLS"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"DFINTTLS" -> "ACFS"
```

———-

5.2 package DEGRED DegreeReductionPackage

5.2.1 DegreeReductionPackage (DEGRED)



Exports:
expand reduce

— **package DEGRED DegreeReductionPackage** —

```
)abbrev package DEGRED DegreeReductionPackage
++ Description:
++ This package has no description

DegreeReductionPackage(R1, R2): Cat == Capsule where
    R1: Ring
    R2: Join(IntegralDomain,OrderedSet)

    I    ==> Integer
    PI   ==> PositiveInteger
    UP   ==> SparseUnivariatePolynomial
    RE   ==> Expression R2

    Cat == with
        reduce:  UP R1    ->  Record(pol: UP R1, deg: PI)
          ++ reduce(p) \undocumented{}
        expand:  (RE, PI) ->  List RE
          ++ expand(f,n) \undocumented{}

    Capsule == add


        degrees(u: UP R1): List Integer ==
            l: List Integer := []
```

```
            while u ^= 0 repeat
              l := concat(degree u,l)
              u := reductum u
            l
        reduce(u: UP R1) ==
            g := "gcd"/[d for d in degrees u]
            u := divideExponents(u, g:PI)::(UP R1)
            [u, g:PI]

        import Fraction Integer

        rootOfUnity(j:I,n:I):RE ==
            j = 0 => 1
            arg:RE := 2*j*pi()/(n::RE)
            cos arg + (-1)**(1/2) * sin arg

        expand(s, g) ==
            g = 1 => [s]
            [rootOfUnity(i,g)*s**(1/g) for i in 0..g-1]
```

———-

— DEGRED.dotabb —

```
"DEGRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEGRED"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"DEGRED" -> "PID"
"DEGRED" -> "OAGROUP"
```

———-

5.3 package DTP DesingTreePackage

— DesingTreePackage.input —

```
)set break resume
)sys rm -f DesingTreePackage.output
)spool DesingTreePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
```

```
)show DesingTreePackage
--R
--R DesingTreePackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)
--R Abbreviation for DesingTreePackage is DTP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DTP
--R
--R------------------------------ Operations --------------------------------
--R fullParamInit : DesTree -> Void       genusNeg : PolyRing -> Integer
--R initParLocLeaves : DesTree -> Void
--R adjunctionDivisor : DesTree -> DIVISOR
--R blowUp : InfClsPoint -> List(InfClsPoint)
--R blowUpWithExcpDiv : DesTree -> Void
--R desingTree : PolyRing -> List(DesTree)
--R desingTreeAtPoint : (ProjPt,PolyRing) -> DesTree
--R divisorAtDesingTree : (PolyRing,DesTree) -> DIVISOR
--R genus : PolyRing -> NonNegativeInteger
--R genusTree : (NonNegativeInteger,List(DesTree)) -> NonNegativeInteger
--R genusTreeNeg : (NonNegativeInteger,List(DesTree)) -> Integer
--R inBetweenExcpDiv : DesTree -> DIVISOR
--R initializeParamOfPlaces : DesTree -> Void
--R initializeParamOfPlaces : (DesTree,List(PolyRing)) -> Void
--R
--E 1

)spool
)lisp (bye)
```

———-

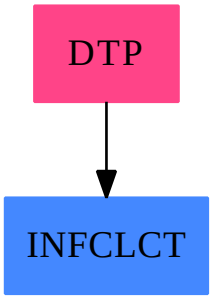
— DesingTreePackage.help —

```
=====================================================================
DesingTreePackage examples
=====================================================================

See Also:
o )show DesingTreePackage
```

———-

5.3.1 DesingTreePackage (DTP)



Exports:

adjunctionDivisor	blowUp	blowUpWithExcpDiv
desingTree	desingTreeAtPoint	divisorAtDesingTree
fullParamInit	genus	genusNeg
genusTree	genusTreeNeg	inBetweenExcpDiv
initParLocLeaves	initializeParamOfPlaces	

— package DTP DesingTreePackage —

```
)abbreviation package DTP DesingTreePackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: 31 jan 95
++ Keywords:
++ Description:  The following is all the categories, domains and package
++ used for the desingularisation be means of
++ monoidal transformation (Blowing-up)
DesingTreePackage(K,
                  symb,
                  PolyRing,
                  E,
                  ProjPt,
                  PCS,
                  Plc,
                  DIVISOR,
                  InfClsPoint,
                  DesTree,
                  BLMET
                   ):Exports == Implementation where
  K:Field
  symb: List(Symbol)

  OV ==> OrderedVariableList(symb)

  E           : DirectProductCategory(#symb,NonNegativeInteger)
```

```
PolyRing    : PolynomialCategory(K,E,OV)
ProjPt      : ProjectiveSpaceCategory(K)
PCS         : LocalPowerSeriesCategory(K)
Plc         : PlacesCategory(K,PCS)
DIVISOR     : DivisorCategory(Plc)

bls      ==> ['X,'Y]
BlUpRing ==> DistributedMultivariatePolynomial( bls , K)
E2       ==> DirectProduct( #bls , NonNegativeInteger )
AFP      ==> AffinePlane(K)
OV2      ==> OrderedVariableList( bls )
PI       ==> PositiveInteger
INT      ==> Integer
NNI      ==> NonNegativeInteger
LPARSPT  ==> LocalParametrizationOfSimplePointPackage
PARAMP   ==> ParametrizationPackage
PRALGPK  ==> ProjectiveAlgebraicSetPackage


InfClsPoint : InfinitlyClosePointCategory(K,symb,PolyRing,E,ProjPt,_
                                          PCS,Plc,DIVISOR,BLMET)
DesTree     : DesingTreeCategory(InfClsPoint)
BLMET       : BlowUpMethodCategory

PackPoly    ==> PackageForPoly(K,PolyRing,E,#symb)
PACKBL      ==> PackageForPoly( K , BlUpRing , E2 , #bls )
NP          ==> NewtonPolygon(K,BlUpRing,E2,#bls)
PPFC1       ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
PPFC2       ==> BlowUpPackage(K,symb,PolyRing,E, BLMET)
ParamPackFC ==> LPARSPT(K,symb,PolyRing,E,ProjPt,PCS,Plc)
ParamPack   ==> PARAMP(K,symb,PolyRing,E,ProjPt,PCS,Plc)
PrjAlgPack  ==>  PRALGPK(K,symb,PolyRing,E,ProjPt)

Exports ==>  with

  blowUp: InfClsPoint ->  List InfClsPoint

  divisorAtDesingTree: (PolyRing,DesTree) -> DIVISOR
    ++ divisorAtDesingTree(f,tr) computes the local
    ++ divisor of f at a desingularisation tree tr of
    ++ a singular point.

  adjunctionDivisor: DesTree -> DIVISOR
    ++ adjunctionDivisor(tr) compute the local
    ++ adjunction divisor of a desingularisation tree tr of
    ++ a singular point.

  blowUpWithExcpDiv: DesTree -> Void -- DesTree

  desingTreeAtPoint: (ProjPt,PolyRing) -> DesTree
```

```
    ++ desingTreeAtPoint(pt,pol) computes
    ++ the desingularisation tree at the point pt
    ++ on the curve defined by pol.
    ++ This function recursively compute the tree.

  desingTree: PolyRing -> List DesTree
    ++ desingTree(pol) returns all the desingularisation
    ++ trees of all singular points on the curve
    ++ defined by pol.

  fullParamInit: DesTree -> Void
    ++ fullParamInit(tr) initialize the local
    ++ parametrization at all places (leaves of tr),
    ++ computes the local exceptional divisor
    ++ at each infinytly close points in the tree.
    ++ This function is equivalent to the following called:
    ++ initParLocLeaves(tr)
    ++ initializeParamOfPlaces(tr)
    ++ blowUpWithExcpDiv(tr)

  initParLocLeaves:  DesTree -> Void
    ++ initParLocLeaves(tr) initialize the local
    ++ parametrization at simple points corresponding to
    ++ the leaves of tr.

  initializeParamOfPlaces: DesTree -> Void
    ++ initParLocLeaves(tr) initialize the local
    ++ parametrization at places corresponding to
    ++ the leaves of tr.

  initializeParamOfPlaces: (DesTree,List PolyRing) -> Void
    ++ initParLocLeaves(tr,listOfFnc) initialize
    ++ the local parametrization at places corresponding to
    ++ the leaves of tr according to the given
    ++ list of functions in listOfFnc.

  genus: PolyRing -> NNI
    ++ genus(pol) computes the genus of the curve defined by pol.

  genusNeg: PolyRing -> INT
    ++ genusNeg(pol) computes the "genus" of a curve
    ++ that may be not absolutly irreducible.
    ++ A "negative" genus means that
    ++ the curve is reducible !!.

  genusTree: (NNI,List(DesTree)) -> NNI
    ++ genusTree(n,listOfTrees) computes the genus of a curve,
    ++ where n is the degree of a  polynomial pol
    ++ defining the curve and listOfTrees is all
    ++ the desingularisation trees at all singular points
```

```
      ++ on the curve defined by pol.

    inBetweenExcpDiv: DesTree -> DIVISOR

    genusTreeNeg: (NNI,List(DesTree)) -> INT
      ++ genusTreeNeg(n,listOfTrees)  computes the "genus"
      ++ of a curve that may be not absolutly irreducible,
      ++ where n is the degree of a  polynomial pol
      ++ defining the curve and listOfTrees is all the
      ++ desingularisation trees at all singular points
      ++ on the curve defined by pol.
      ++ A "negative" genus means that
      ++ the curve is reducible !!.

  Implementation ==>  add
    import PackPoly
    import PPFC1
    import PPFC2
    import PolyRing
    import DesTree

    divisorAtDesingTreeLocal: (BlUpRing , DesTree ) -> DIVISOR

    polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing

    makeMono: DesTree -> BlUpRing

    inBetweenExcpDiv( tr )==
      -- trouve le diviseur excp. d'un pt inf voisin PRECEDENT !
      -- qV est egal a : 1 +  nombre de fois que ce point est repete
      -- dans un chaine (le plus un correspond au point d'origine du
      -- point dont il est question ici.
      -- mp est la multiciplicite du point.
      -- cette fonction n'est et ne peut etre qu'utiliser pour
      -- calculer le diviseur d'adjonction ( a cause du mp -1).
      noeud:= value tr
      chart:= chartV noeud
      qV:= quotValuation chart
      one? qV => 0$DIVISOR
      expDiv := divisorAtDesingTreeLocal(makeMono(tr),tr)
      mp:= degree expDiv
      ((qV - 1) * (mp -1)) *$DIVISOR expDiv

    polyRingToBlUpRing(pol,chart)==
      zero? pol => 0
      lc:= leadingCoefficient pol
      d:=entries degree pol
      ll:= [ d.i for i in 1..3 | ^( i = chartCoord(chart) ) ]
      e:= directProduct( vector( ll)$Vector(NNI) )$E2
      monomial(lc , e )$BlUpRing + polyRingToBlUpRing( reductum pol, chart )
```

```
affToProj(pt:AFP, chart:BLMET ):ProjPt==
  nV:= chartCoord chart
  d:List(K) := list(pt)$AFP
  ll:List K:=
    nV = 1 => [ 1$K , d.1  , d.2 ]
    nV = 2 => [ d.1  , 01$K , d.2 ]
    [d.1 , d.2 , 1 ]
  projectivePoint( ll )$ProjPt

biringToPolyRing: (BlUpRing, BLMET) -> PolyRing

biringToPolyRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  nV:= chartCoord chart
  ll:List NNI:=
    nV = 1 => [ 0$NNI , d.1  , d.2 ]
    nV = 2 => [ d.1  , 0$NNI , d.2 ]
    [d.1 , d.2 , 0$NNI ]
  e:= directProduct( vector( ll)$Vector(NNI) )$E
  monomial(lc , e )$PolyRing  + biringToPolyRing( reductum pol, chart )

minus  : (NNI,NNI) -> NNI

minus(a,b)==
    d:=subtractIfCan(a,b)
    d case "failed" => error "cannot substract a-b if b>a for NNI"
    d

-- returns the exceptional coordinate function

makeExcpDiv: List DesTree -> DIVISOR

desingTreeAtPointLocal: InfClsPoint  -> DesTree

subGenus: DesTree -> NNI

lVar:List PolyRing := _
  [monomial(1,index(i pretend PI)$OV,1)$PolyRing for i in 1..#symb]

divisorAtDesingTreeLocal(pol,tr)==
  --  BLMET has QuadraticTransform ; marche aussi avec
  -- Hamburger-Noether mais surement moins efficace
  noeud:=value(tr)
  pt:=localPointV(noeud)
  chart:= chartV noeud
  -- ram:= ramifMult chart -- ???
  -- new way to compute in order not to translate twice pol
```

```
    polTrans:BlUpRing:=translate(pol,list(pt)$AFP)$PACKBL
    multPol:=degreeOfMinimalForm(polTrans)
    chtr:=children(tr)
    parPol:PCS
    ord:Integer
    empty?(chtr) =>
      parPol:=parametrize(biringToPolyRing(pol,chartV(noeud))_
                          ,localParamV(noeud))$ParamPack
      ord:=order(parPol)$PCS
      ord * excpDivV(noeud)  -- Note: le div excp est une fois la place.
    (multPol *$DIVISOR excpDivV(noeud)) +$DIVISOR _
       reduce("+",[divisorAtDesingTreeLocal(_
                    quadTransform(polTrans,multPol,(chartV(value(child)))),_
                      child)_
                        for child in chtr])

  desingTreeAtPointLocal(ipt) ==
    -- crb:PolyRing,pt:ProjPt,lstnV:List(INT),origPoint:ProjPt,actL:K)==
    -- peut etre est-il preferable, avant d'eclater, de tester
    -- si le point est simple avec les derives, et non
    -- verifier si le point est simple ou non apres translation.
    -- ????
    blbl:=blowUp ipt
    multPt:=multV ipt
    one?(multPt) =>
      tree( ipt )$DesTree
    subTree:List DesTree:= [desingTreeAtPointLocal( iipt )  for iipt in blbl]
    tree( ipt, subTree )$DesTree

  blowUp(ipt)==
    crb:=curveV ipt
    pt:= localPointV ipt
    lstnV := chartV ipt   -- CHH  no modif needed
    actL:= actualExtensionV ipt
    origPoint:= pointV ipt
    blbl:=stepBlowUp(crb,pt,lstnV,actL) -- CHH no modif needed
    multPt:=blbl.mult
    sm:= blbl.subMult
    -- la multiplicite et la frontiere du polygone de Newton (ou la forme
    -- minimale selon BLMET) du point ipt est assigne par effet de bord !
    setmult!(ipt,multPt)
    setsubmult!(ipt, sm)
    one?(multPt) => empty()
    [create(origPoint,_
            rec(recTransStr),_
            rec(recPoint) ,_
            0,_
            rec(recChart),_
            0,
            0$DIVISOR,_
```

```
          rec(definingExtension),_
          new(I)$Symbol )$InfClsPoint  for rec in blbl.blUpRec]

makeMono(arb)==
  monomial(1,index(excepCoord(chartV(value(arb))) pretend PI)$OV2,_
              1)$BlUpRing

makeExcpDiv(lstSsArb)==
  reduce("+", _
     [divisorAtDesingTreeLocal(makeMono(arb),arb) for arb in lstSsArb],0)

adjunctionDivisorForQuadTrans: DesTree -> DIVISOR
adjunctionDivisorForHamburgeNoether: DesTree -> DIVISOR

adjunctionDivisor( tr )==
  BLMET has QuadraticTransform => adjunctionDivisorForQuadTrans( tr )
  BLMET has HamburgerNoether =>   adjunctionDivisorForHamburgeNoether( tr )
  error _
   " The algorithm to compute the adjunction divisor is not defined for the blowing method y

adjunctionDivisorForHamburgeNoether( tr )==
  noeud:=value tr
  chtr:= children tr
  empty?(chtr) => 0$DIVISOR  -- on suppose qu'un noeud sans feuille
                             -- est une feulle, donc non singulier. !
  multPt:= multV noeud
  ( minus(multPt,1)  pretend INT)  *$DIVISOR excpDivV(noeud) +$DIVISOR _
     reduce("+",[inBetweenExcpDiv( arb ) for arb in chtr ]) +$DIVISOR _
     reduce("+",[adjunctionDivisorForHamburgeNoether(arb) for arb in chtr])

adjunctionDivisorForQuadTrans(tr)==
  noeud:=value(tr)
  chtr:=children(tr)
  empty?(chtr) => 0$DIVISOR
  multPt:=multV(noeud)
  ( minus(multPt,1)  pretend INT)  *$DIVISOR excpDivV(noeud) +$DIVISOR _
      reduce("+",[adjunctionDivisorForQuadTrans(child) for child in chtr])

divisorAtDesingTree( pol , tr)==
  chart:= chartV value(tr)
  pp:= polyRingToBlUpRing( pol, chart )
  divisorAtDesingTreeLocal( pp, tr )

subGenus(tr)==
  noeud:=value tr
  mult:=multV(noeud)
  chart := chartV noeud
  empty?(chdr:=children(tr)) => 0     -- degree(noeud)* mult* minus(mult,1)
  degree(noeud)* ( mult*minus( mult, 1 ) + subMultV( noeud ) ) +
      reduce("+",[subGenus(ch) for ch in chdr])
```

```
initializeParamOfPlaces(tr,lpol)==
  noeud:=value(tr)
  pt:=localPointV(noeud)
  crb:=curveV(noeud)
  chart:=chartV(noeud) -- CHH
  nV:INT:=chartCoord chart
  chtr:List DesTree:=children(tr)
  plc:Plc
  lParam:List PCS
  dd:PositiveInteger:=degree noeud
  lcoef:List K
  lll:Integer
  lParInf:List(PCS)
  lpar:List PCS
  empty?(chtr) =>
    lPar:=localParamOfSimplePt( affToProj(pt, chart) , _
                          biringToPolyRing(crb, chart),nV)$ParamPackFC
    setlocalParam!(noeud,lPar)
    lParam:=[parametrize( f , lPar)$ParamPack for f in lpol]
    plc:= create( symbNameV(noeud) )$Plc
    setParam!(plc,lParam)
    setDegree!(plc,dd)
    itsALeaf!(plc)
    setexcpDiv!(noeud, plc :: DIVISOR )
    void()
  lpolTrans:List PolyRing:=_
    [translateToOrigin( pol, affToProj(pt, chart) , nV) for pol in lpol]
  lpolBlUp:List PolyRing
  chartBl:BLMET
  for arb in chtr repeat
    chartBl:=chartV value arb
    lpolBlUp:=[applyTransform(pol,chartBl) for pol in lpolTrans]
    initializeParamOfPlaces(arb,lpolBlUp)
  void()

blowUpWithExcpDiv(tr:DesTree)==
  noeud:=value(tr)
  pt:=localPointV(noeud)
  crb:=curveV(noeud)
  chtr:List DesTree:=children(tr)
  empty?(chtr) => void() -- tr
  for arb in chtr repeat
    blowUpWithExcpDiv(arb)
  setexcpDiv!(noeud,makeExcpDiv( chtr  ))
  void()

fullParamInit(tr)==
  initializeParamOfPlaces(tr)
  blowUpWithExcpDiv(tr)
```

```
  void()

initializeParamOfPlaces(tr)==initializeParamOfPlaces(tr,lVar)

desingTreeAtPoint(pt,crb)==
  ipt:= create(pt,crb)$InfClsPoint
  desingTreeAtPointLocal ipt

genus(crb)==
  if BLMET has HamburgerNoether then _
     print(("  BUG BUG corige le bug GH ---- ")::OutputForm)
  degCrb:=totalDegree(crb)$PackPoly
  genusTree(degCrb,desingTree(crb))

genusNeg(crb)==
  degCrb:=totalDegree(crb)$PackPoly
  genusTreeNeg(degCrb,desingTree(crb))

desingTree(crb)==
  [desingTreeAtPoint(pt,crb) for pt in singularPoints(crb)$PrjAlgPack]

genusTree(degCrb,listArbDes)==
  -- le test suivant est necessaire
  -- ( meme s'il n'y a pas de point singulier dans ce cas)
  -- car avec sousNNI on ne peut retourner un entier negatif
  (degCrb <$NNI 3::NNI) and ^empty?(listArbDes) =>
    print(("Too many infinitly near points")::OutputForm)
    print(("The curve may not be absolutely irreducible")::OutputForm)
    error "Have a nice day"
  (degCrb <$NNI 3::NNI)  => 0
  ga:= ( minus(degCrb,1)*minus(degCrb ,2) ) quo$NNI 2
  empty?(listArbDes) => ga
  --calcul du nombre de double point
  dp:= reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2
  (dp >$NNI ga) =>
    print(("Too many infinitly near points")::OutputForm)
    print(("The curve may not be absolutely irreducible")::OutputForm)
    error "Have a nice day"
  minus(ga,dp)

genusTreeNeg(degCrb,listArbDes)==
  -- (degCrb <$NNI 3::NNI) => 0
  ga:= (degCrb-1)*(degCrb-2) quo$INT 2
  empty?(listArbDes) => ga
  ga-( reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2)::INT
```

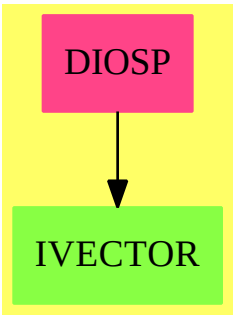
— **DTP.dotabb** —

```
"DTP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DTP"]
"INFCLCT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=INFCLCT"]
"DTP" -> "INFCLCT"
```

————-

5.4 package DIOSP DiophantineSolutionPackage

5.4.1 DiophantineSolutionPackage (DIOSP)



Exports:
dioSolve

— **package DIOSP DiophantineSolutionPackage** —

```
)abbrev package DIOSP DiophantineSolutionPackage
++ Author: A. Fortenbacher
++ Date Created: 29 March 1991
++ Date Last Updated: 29 March 1991
++ Basic Operations: dioSolve
++ Related Constructors: Equation, Vector
++ Also See:
++ AMS Classifications:
++ Keywords: Diophantine equation, nonnegative solutions,
++   basis, depth-first-search
++ Reference:
++   M. Clausen, A. Fortenbacher: Efficient Solution of
++   Linear Diophantine Equations. in JSC (1989) 8, 201-216
++ Description:
++ Any solution of a homogeneous linear Diophantine equation
++ can be represented as a sum of minimal solutions, which
++ form a "basis" (a minimal solution cannot be represented
```

```
++ as a nontrivial sum of solutions)
++ in the case of an inhomogeneous linear Diophantine equation,
++ each solution is the sum of a inhomogeneous solution and
++ any number of homogeneous solutions
++ therefore, it suffices to compute two sets:\br
++ \tab{5}1. all minimal inhomogeneous solutions\br
++ \tab{5}2. all minimal homogeneous solutions\br
++ the algorithm implemented is a completion procedure, which
++ enumerates all solutions in a recursive depth-first-search
++ it can be seen as finding monotone paths in a graph
++ for more details see Reference

DiophantineSolutionPackage(): Cat == Capsule where

  B ==> Boolean
  I ==> Integer
  NI ==> NonNegativeInteger

  LI ==> List(I)
  VI ==> Vector(I)
  VNI ==> Vector(NI)

  POLI ==> Polynomial(I)
  EPOLI ==> Equation(POLI)
  LPOLI ==> List(POLI)

  S ==> Symbol
  LS ==> List(S)

  ListSol ==> List(VNI)
  Solutions ==> Record(varOrder: LS, inhom: Union(ListSol,"failed"),
                       hom: ListSol)

  Node ==> Record(vert: VI, free: B)
  Graph ==> Record(vn: Vector(Node), dim : NI, zeroNode: I)

  Cat ==> with

    dioSolve: EPOLI -> Solutions
      ++ dioSolve(u) computes a basis of all minimal solutions for
      ++ linear homogeneous Diophantine equation u,
      ++ then all minimal solutions of inhomogeneous equation

  Capsule ==> add

    import I
    import POLI

    -- local function specifications
```

```
initializeGraph: (LPOLI, I) -> Graph
createNode: (I, VI, NI, I) -> Node
findSolutions: (VNI, I, I, I, Graph, B) -> ListSol
verifyMinimality: (VNI, Graph, B) -> B
verifySolution: (VNI, I, I, I, Graph) -> B

-- exported functions

dioSolve(eq) ==
  p := lhs(eq) - rhs(eq)
  n := totalDegree(p)
  n = 0 or n > 1 =>
    error "a linear Diophantine equation is expected"
  mon := empty()$LPOLI
  c : I := 0
  for x in monomials(p) repeat
    ground?(x) =>
      c := ground(x) :: I
    mon := cons(x, mon)$LPOLI
  graph := initializeGraph(mon, c)
  sol := zero(graph.dim)$VNI
  hs := findSolutions(sol, graph.zeroNode, 1, 1, graph, true)
  ihs : ListSol :=
    c = 0 => [sol]
    findSolutions(sol, graph.zeroNode + c, 1, 1, graph, false)
  vars := [first(variables(x))$LS for x in mon]
  [vars, if empty?(ihs)$ListSol then "failed" else ihs, hs]

-- local functions

initializeGraph(mon, c) ==
  coeffs := vector([first(coefficients(x))$LI for x in mon])$VI
  k := #coeffs
  m := min(c, reduce(min, coeffs)$VI)
  n := max(c, reduce(max, coeffs)$VI)
  [[createNode(i, coeffs, k, 1 - m) for i in m..n], k, 1 - m]

createNode(ind, coeffs, k, zeroNode) ==
  -- create vertices from node ind to other nodes
  v := zero(k)$VI
  for i in 1..k repeat
    ind > 0 =>
      coeffs.i < 0 =>
        v.i := zeroNode + ind + coeffs.i
    coeffs.i > 0 =>
      v.i := zeroNode + ind + coeffs.i
  [v, true]

findSolutions(sol, ind, m, n, graph, flag) ==
  -- return all solutions (paths) from node ind to node zeroNode
```

```
    sols := empty()$ListSol
    node := graph.vn.ind
    node.free =>
      node.free := false
      v := node.vert
      k := if ind < graph.zeroNode then m else n
      for i in k..graph.dim repeat
        x := sol.i
        v.i > 0 =>  -- vertex exists to other node
          sol.i := x + 1
          v.i = graph.zeroNode =>  -- solution found
            verifyMinimality(sol, graph, flag) =>
              sols := cons(copy(sol)$VNI, sols)$ListSol
              sol.i := x
            sol.i := x
          s :=
            ind < graph.zeroNode =>
              findSolutions(sol, v.i, i, n, graph, flag)
            findSolutions(sol, v.i, m, i, graph, flag)
          sols := append(s, sols)$ListSol
          sol.i := x
      node.free := true
      sols
    sols

  verifyMinimality(sol, graph, flag) ==
    -- test whether sol contains a minimal homogeneous solution
    flag =>  -- sol is a homogeneous solution
      i := 1
      while sol.i = 0 repeat
        i := i + 1
      x := sol.i
      sol.i := (x - 1) :: NI
      flag := verifySolution(sol, graph.zeroNode, 1, 1, graph)
      sol.i := x
      flag
    verifySolution(sol, graph.zeroNode, 1, 1, graph)

  verifySolution(sol, ind, m, n, graph) ==
    -- test whether sol contains a path from ind to zeroNode
    flag := true
    node := graph.vn.ind
    v := node.vert
    k := if ind < graph.zeroNode then m else n
    for i in k..graph.dim while flag repeat
      x := sol.i
      x > 0 and v.i > 0 =>  -- vertex exists to other node
        sol.i := (x - 1) :: NI
        v.i = graph.zeroNode =>  -- solution found
          flag := false
```

```
            sol.i := x
          flag :=
            ind < graph.zeroNode =>
              verifySolution(sol, v.i, i, n, graph)
            verifySolution(sol, v.i, m, i, graph)
          sol.i := x
      flag
```

———-

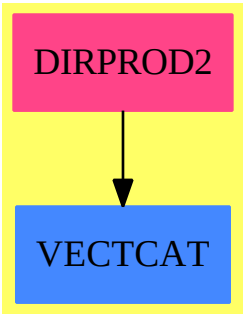
— **DIOSP.dotabb** —

```
"DIOSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DIOSP"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"DIOSP" -> "IVECTOR"
```

———-

5.5 package DIRPROD2 DirectProductFunctions2

5.5.1 DirectProductFunctions2 (DIRPROD2)



Exports:
map reduce scan

— **package DIRPROD2 DirectProductFunctions2** —

```
)abbrev package DIRPROD2 DirectProductFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
```

```
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides operations which all take as arguments direct
++ products of elements of some type \spad{A} and functions from \spad{A}
++ to another type B. The operations all iterate over their vector argument
++ and either return a value of type B or a direct product over B.

DirectProductFunctions2(dim, A, B): Exports == Implementation where
  dim : NonNegativeInteger
  A, B: Type

  DA ==> DirectProduct(dim, A)
  DB ==> DirectProduct(dim, B)
  VA ==> Vector A
  VB ==> Vector B
  O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)

  Exports ==> with
    scan   : ((A, B) -> B, DA, B) -> DB
      ++ scan(func,vec,ident) creates a new vector whose elements are
      ++ the result of applying reduce to the binary function func,
      ++ increasing initial subsequences of the vector vec,
      ++ and the element ident.
    reduce : ((A, B) -> B, DA, B) -> B
      ++ reduce(func,vec,ident) combines the elements in vec using the
      ++ binary function func. Argument ident is returned if the vector is empty.
    map    : (A -> B, DA) -> DB
      ++ map(f, v) applies the function f to every element of the vector v
      ++ producing a new vector containing the values.

  Implementation ==> add
    import FiniteLinearAggregateFunctions2(A, VA, B, VB)

    map(f, v)       == directProduct map(f, v::VA)
    scan(f, v, b)   == directProduct scan(f, v::VA, b)
    reduce(f, v, b) == reduce(f, v::VA, b)
```

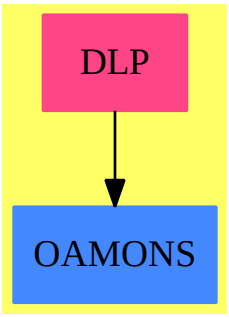
———-

— DIRPROD2.dotabb —

```
"DIRPROD2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DIRPROD2"]
"VECTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=VECTCAT"]
"DIRPROD2" -> "VECTCAT"
```

5.6 package DLP DiscreteLogarithmPackage

5.6.1 DiscreteLogarithmPackage (DLP)



Exports:
shanksDiscLogAlgorithm

— **package DLP DiscreteLogarithmPackage** —

```
)abbrev package DLP DiscreteLogarithmPackage
++ Author: J. Grabmeier, A. Scheerhorn
++ Date Created: 12 March 1991
++ Date Last Updated: 31 March 1991
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: discrete logarithm
++ References:
++  J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++  AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ DiscreteLogarithmPackage implements help functions for discrete logarithms
++ in monoids using small cyclic groups.

DiscreteLogarithmPackage(M): public == private where
  M : Join(Monoid,Finite) with
   "**": (M,Integer) -> M
     ++ x ** n returns x raised to the integer power n
  public ==> with
    shanksDiscLogAlgorithm:(M,M,NonNegativeInteger)->  _
```

```
       Union(NonNegativeInteger,"failed")
     ++ shanksDiscLogAlgorithm(b,a,p) computes s with \spad{b**s = a} for
     ++ assuming that \spad{a} and b are elements in a 'small' cyclic group of
     ++ order p by Shank's algorithm.
     ++ Note that this is a subroutine of the function \spadfun{discreteLog}.
 I   ==> Integer
 PI  ==> PositiveInteger
 NNI ==> NonNegativeInteger
 SUP ==> SparseUnivariatePolynomial
 DLP ==> DiscreteLogarithmPackage

 private ==> add
   shanksDiscLogAlgorithm(logbase,c,p) ==
     limit:Integer:= 30
     -- for logarithms up to cyclic groups of order limit a full
     -- logarithm table is computed
     p < limit =>
       a:M:=1
       disclog:Integer:=0
       found:Boolean:=false
       for i in 0..p-1 while not found repeat
         a = c =>
           disclog:=i
           found:=true
         a:=a*logbase
       not found =>
         messagePrint("discreteLog: second argument not in cyclic group_
generated by first argument")$OutputForm
         "failed"
       disclog pretend NonNegativeInteger
     l:Integer:=length(p)$Integer
     if odd?(l)$Integer then n:Integer:= shift(p,-(l quo 2))
                        else n:Integer:= shift(1,(l quo 2))
     a:M:=1
     exptable : Table(PI,NNI) :=table()$Table(PI,NNI)
     for i in (0::NNI)..(n-1)::NNI repeat
       insert_!([lookup(a),i::NNI]$Record(key:PI,entry:NNI),_
                exptable)$Table(PI,NNI)
       a:=a*logbase
     found := false
     end := (p-1) quo n
     disclog:Integer:=0
     a := c
     b := logbase ** (-n)
     for i in 0..end while not found repeat
       rho:= search(lookup(a),exptable)_
             $Table(PositiveInteger,NNI)
       rho case NNI =>
         found := true
         disclog:= n * i + rho pretend Integer
```

```
        a := a * b
      not found =>
        messagePrint("discreteLog: second argument not in cyclic group_
 generated by first argument")$OutputForm
        "failed"
      disclog pretend NonNegativeInteger
```

———-

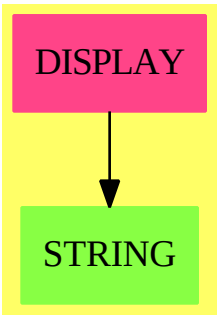
— **DLP.dotabb** —

```
"DLP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DLP"]
"OAMONS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAMONS"]
"DLP" -> "OAMONS"
```

———-

5.7 package DISPLAY DisplayPackage

5.7.1 DisplayPackage (DISPLAY)



Exports:
bright center copies newLine say sayLength

— **package DISPLAY DisplayPackage** —

```
)abbrev package DISPLAY DisplayPackage
++ Author: Robert S. Sutor
++ Date Created: September 1986
++ Date Last Updated:
++ Basic Operations: bright, newLine, copies, center, say, sayLength
++ Related Constructors:
```

```
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ DisplayPackage allows one to print strings in a nice manner,
++ including highlighting substrings.

DisplayPackage: public == private where
  I       ==> Integer
  L       ==> List
  S       ==> String
  RECLR   ==> Record(lhs : S, rhs : S)

  public  == with
    bright:       S           -> L S
      ++ bright(s) sets the font property of the string s to bold-face type.
    bright:       L S         -> L S
      ++ bright(l) sets the font property of a list of strings, l, to
      ++ bold-face type.
    newLine:      ()          -> S
      ++ newLine() sends a new line command to output.

    copies:       (I,S)       -> S
      ++ copies(i,s) will take a string s and create a new string composed of
      ++ i copies of s.
    center:       (S,I,S)     -> S
      ++ center(s,i,s) takes the first string s, and centers it within a string
      ++ of length i, in which the other elements of the string are composed
      ++ of as many replications as possible of the second indicated string, s
      ++ which must have a length greater than that of an empty string.
    center:       (L S,I,S)   -> L S
      ++ center(l,i,s) takes a list of strings l, and centers them within a
      ++ list of strings which is i characters long, in which the remaining
      ++ spaces are filled with strings composed of as many repetitions as
      ++ possible of the last string parameter s.

    say:          S           -> Void
      ++ say(s) sends a string s to output.
    say:          L S         -> Void
      ++ say(l) sends a list of strings l to output.
    sayLength:    S           -> I
      ++ sayLength(s) returns the length of a string s as an integer.
    sayLength:    L S         -> I
      ++ sayLength(l) returns the length of a list of strings l as an integer.

  private == add
    --StringManipulations()

    center0:  (I,I,S) -> RECLR
```

```
    s : S
    l : L S

    HION    : S := "%b"
    HIOFF   : S := "%d"
    NEWLINE : S := "%l"

    bright s == [HION,s,HIOFF]$(L S)
    bright l == cons(HION,append(l,list HIOFF))
    newLine() == NEWLINE

    copies(n : I, s : S) ==
      n < 1 => ""
      n = 1 => s
      t : S := copies(n quo 2, s)
      odd? n => concat [s,t,t]
      concat [t,t]

    center0(len : I, wid : I, fill : S) : RECLR ==
      (wid < 1) or (len >= wid) => ["",""]$RECLR
      m : I := (wid - len) quo 2
      t : S := copies(1 + (m quo (sayLength fill)),fill)
      [t(1..m),t(1..wid-len-m)]$RECLR

    center(s, wid, fill) ==
      wid < 1 => ""
      len : I := sayLength s
      len = wid => s
      len > wid => s(1..wid)
      rec : RECLR := center0(len,wid,fill)
      concat [rec.lhs,s,rec.rhs]

    center(l, wid, fill) ==
      wid < 1 => [""]$(L S)
      len : I := sayLength l
      len = wid => l
--    len > wid => s(1..wid)
      rec : RECLR := center0(len,wid,fill)
      cons(rec.lhs,append(l,list rec.rhs))

    say s ==
      sayBrightly$Lisp s
      void()$Void

    say l ==
      sayBrightly$Lisp l
      void()$Void

    sayLength s == #s
```

```
    sayLength l ==
      sum : I := 0
      for s in l repeat
        s = HION      => sum := sum + 1
        s = HIOFF     => sum := sum + 1
        s = NEWLINE   => sum
        sum := sum + sayLength s
      sum
```

———-

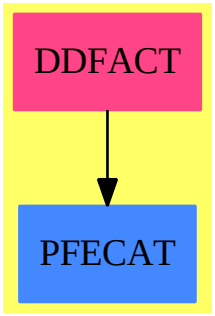
— **DISPLAY.dotabb** —

```
"DISPLAY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DISPLAY"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"DISPLAY" -> "STRING"
```

———-

5.8 package DDFACT DistinctDegreeFactorize

5.8.1 DistinctDegreeFactorize (DDFACT)



Exports:

distdfact	exptMod	factor	factorSquareFree	irreducible?
separateDegrees	separateFactors	trace2PowMod	tracePowMod	

— **package DDFACT DistinctDegreeFactorize** —

```
)abbrev package DDFACT DistinctDegreeFactorize
++ Author: P. Gianni, B.Trager
```

```
++ Date Created: 1983
++ Date Last Updated: 22 November 1993
++ Basic Functions: factor, irreducible?
++ Related Constructors: PrimeField, FiniteField
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Package for the factorization of a univariate polynomial with
++ coefficients in a finite field. The algorithm used is the
++ "distinct degree" algorithm of Cantor-Zassenhaus, modified
++ to use trace instead of the norm and a table for computing
++ Frobenius as suggested by Naudin and Quitte .

DistinctDegreeFactorize(F,FP): C == T
  where
   F  : FiniteFieldCategory
   FP : UnivariatePolynomialCategory(F)

   fUnion ==> Union("nil", "sqfr", "irred", "prime")
   FFE    ==> Record(flg:fUnion, fctr:FP, xpnt:Integer)
   NNI       == NonNegativeInteger
   Z         == Integer
   fact      == Record(deg : NNI,prod : FP)
   ParFact   == Record(irr:FP,pow:Z)
   FinalFact == Record(cont:F,factors:List(ParFact))

   C == with
      factor        :         FP      -> Factored FP
        ++ factor(p) produces the complete factorization of the polynomial p.
      factorSquareFree :         FP      -> Factored FP
        ++ factorSquareFree(p) produces the complete factorization of the
        ++ square free polynomial p.
      distdfact       :   (FP,Boolean)  -> FinalFact
        ++ distdfact(p,sqfrflag) produces the complete factorization
        ++ of the polynomial p returning an internal data structure.
        ++ If argument sqfrflag is true, the polynomial is assumed square free.
      separateDegrees :         FP      -> List fact
        ++ separateDegrees(p) splits the square free polynomial p into
        ++ factors each of which is a product of irreducibles of the
        ++ same degree.
      separateFactors :  List fact  -> List FP
        ++ separateFactors(lfact) takes the list produced by separateDegrees
        ++ and produces the complete list of factors.
      exptMod         :   (FP,NNI,FP)   -> FP
        ++ exptMod(u,k,v) raises the polynomial u to the kth power
        ++ modulo the polynomial v.
      trace2PowMod     :   (FP,NNI,FP)   -> FP
        ++ trace2PowMod(u,k,v) produces the sum of u**(2**i) for i running
```

```
      ++ from 1 to k all computed modulo the polynomial v.
    tracePowMod     :   (FP,NNI,FP)   -> FP
      ++ tracePowMod(u,k,v) produces the sum of \spad{u**(q**i)}
      ++ for i running and q= size F
    irreducible?    :         FP      -> Boolean
      ++ irreducible?(p) tests whether the polynomial p is irreducible.


  T == add
    --declarations
    D:=ModMonic(F,FP)
    import UnivariatePolynomialSquareFree(F,FP)

    --local functions
    notSqFr : (FP,FP -> List(FP)) -> List(ParFact)
    ddffact : FP -> List(FP)
    ddffact1 : (FP,Boolean) -> List fact
    ranpol :         NNI       -> FP

    charF : Boolean := characteristic()$F = 2

    --construct a random polynomial of random degree < d
    ranpol(d:NNI):FP ==
      k1: NNI := 0
      while k1 = 0 repeat k1 := random d
      -- characteristic F = 2
      charF =>
         u:=0$FP
         for j in 1..k1 repeat u:=u+monomial(random()$F,j)
         u
      u := monomial(1,k1)
      for j in 0..k1-1 repeat u:=u+monomial(random()$F,j)
      u

    notSqFr(m:FP,appl: FP->List(FP)):List(ParFact) ==
      factlist : List(ParFact) :=empty()
      llf : List FFE
      fln :List(FP) := empty()
      if (lcm:=leadingCoefficient m)^=1 then m:=(inv lcm)*m
      llf:= factorList(squareFree(m))
      for lf in llf repeat
        d1:= lf.xpnt
        pol := lf.fctr
        if (lcp:=leadingCoefficient pol)^=1 then pol := (inv lcp)*pol
        degree pol=1 => factlist:=cons([pol,d1]$ParFact,factlist)
        fln := appl(pol)
        factlist :=append([[pf,d1]$ParFact for pf in fln],factlist)
      factlist

    -- compute u**k mod v (requires call to setPoly of multiple of v)
```

```
-- characteristic not equal 2
exptMod(u:FP,k:NNI,v:FP):FP == (reduce(u)$D**k):FP rem v

-- compute u**k mod v (requires call to setPoly of multiple of v)
-- characteristic equal 2
trace2PowMod(u:FP,k:NNI,v:FP):FP ==
  uu:=u
  for i in 1..k repeat uu:=(u+uu*uu) rem v
  uu

-- compute u+u**q+..+u**(q**k) mod v
-- (requires call to setPoly of multiple of v) where q=size< F
tracePowMod(u:FP,k:NNI,v:FP):FP ==
  u1 :D :=reduce(u)$D
  uu : D := u1
  for i in 1..k repeat uu:=(u1+frobenius uu)
  (lift uu) rem v

-- compute u**(1+q+..+q**k) rem v where q=#F
-- (requires call to setPoly of multiple of v)
-- frobenius map is used
normPowMod(u:FP,k:NNI,v:FP):FP ==
  u1 :D :=reduce(u)$D
  uu : D := u1
  for i in 1..k repeat uu:=(u1*frobenius uu)
  (lift uu) rem v

--find the factorization of m as product of factors each containing
--terms of equal degree .
-- if testirr=true the function returns the first factor found
ddffact1(m:FP,testirr:Boolean):List(fact) ==
  p:=size$F
  dg:NNI :=0
  ddfact:List(fact):=empty()
  --evaluation of x**p mod m
  k1:NNI
  u:= m
  du := degree u
  setPoly u
  mon: FP := monomial(1,1)
  v := mon
  for k1 in 1.. while k1 <= (du quo 2) repeat
      v := lift frobenius reduce(v)$D
      g := gcd(v-mon,u)
      dg := degree g
      dg =0  => "next k1"
      if leadingCoefficient g ^=1 then g := (inv leadingCoefficient g)*g
      ddfact := cons([k1,g]$fact,ddfact)
      testirr => return ddfact
      u := u quo g
```

```
      du := degree u
      du = 0 => return ddfact
      setPoly u
  cons([du,u]$fact,ddfact)

-- test irreducibility
irreducible?(m:FP):Boolean ==
  mf:fact:=first ddffact1(m,true)
  degree m = mf.deg

--export ddfact1
separateDegrees(m:FP):List(fact) == ddffact1(m,false)

--find the complete factorization of m, using the result of ddfact1
separateFactors(distf : List fact) :List FP ==
  ddfact := distf
  n1:Integer
  p1:=size()$F
  if charF then n1:=length(p1)-1
  newaux,aux,ris : List FP
  ris := empty()
  t,fprod : FP
  for ffprod in ddfact repeat
    fprod := ffprod.prod
    d := ffprod.deg
    degree fprod = d => ris := cons(fprod,ris)
    aux:=[fprod]
    setPoly fprod
    while ^(empty? aux) repeat
      t := ranpol(2*d)
      if charF then t:=trace2PowMod(t,(n1*d-1)::NNI,fprod)
      else t:=exptMod(tracePowMod(t,(d-1)::NNI,fprod),
                              (p1 quo 2)::NNI,fprod)-1$FP
      newaux:=empty()
      for u in aux repeat
          g := gcd(u,t)
          dg:= degree g
          dg=0 or dg = degree u => newaux:=cons(u,newaux)
          v := u quo g
          if dg=d then ris := cons(inv(leadingCoefficient g)*g,ris)
                  else newaux := cons(g,newaux)
          if degree v=d then ris := cons(inv(leadingCoefficient v)*v,ris)
                        else newaux := cons(v,newaux)
      aux:=newaux
  ris

--distinct degree algorithm for monic ,square-free polynomial
ddffact(m:FP):List(FP)==
  ddfact:=ddffact1(m,false)
  empty? ddfact => [m]
```

```
  separateFactors ddfact

--factorize a general polynomial with distinct degree algorithm
--if test=true no check is executed on square-free
distdfact(m:FP,test:Boolean):FinalFact ==
  factlist: List(ParFact):= empty()
  fln : List(FP) :=empty()

  --make m monic
  if (lcm := leadingCoefficient m) ^=1 then m := (inv lcm)*m

  --is x**d factor of m?
  if (d := minimumDegree m)>0 then
    m := (monicDivide (m,monomial(1,d))).quotient
    factlist := [[monomial(1,1),d]$ParFact]
  d:=degree m

  --is m constant?
  d=0 => [lcm,factlist]$FinalFact

  --is m linear?
  d=1 => [lcm,cons([m,d]$ParFact,factlist)]$FinalFact

  --m is square-free
  test =>
    fln := ddffact m
    factlist := append([[pol,1]$ParFact for pol in fln],factlist)
    [lcm,factlist]$FinalFact

  --factorize the monic,square-free terms
  factlist:= append(notSqFr(m,ddffact),factlist)
  [lcm,factlist]$FinalFact

--factorize the polynomial m
factor(m:FP) ==
  m = 0 => 0
  flist := distdfact(m,false)
  makeFR(flist.cont::FP,[["prime",u.irr,u.pow]$FFE
                          for u in flist.factors])


--factorize the square free polynomial m
factorSquareFree(m:FP) ==
  m = 0 => 0
  flist := distdfact(m,true)
  makeFR(flist.cont::FP,[["prime",u.irr,u.pow]$FFE
                          for u in flist.factors])
```

———-

— **DDFACT.dotabb** —

```
"DDFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DDFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DDFACT" -> "PFECAT"
```

———-

5.9 package DFSFUN DoubleFloatSpecialFunctions

The special functions in this section are developed as special cases but can all be expressed in terms of generalized hypergeomentric functions ${}_pF_q$ or its generalization, the Meijer G function. [**?**, **?**] The long term plan is to reimplement these functions using the generalized version.

— **DoubleFloatSpecialFunctions.input** —

```
)set break resume
)sys rm -f DoubleFloatSpecialFunctions.output
)spool DoubleFloatSpecialFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
)show DoubleFloatSpecialFunctions
--R
--R DoubleFloatSpecialFunctions  is a package constructor
--R Abbreviation for DoubleFloatSpecialFunctions is DFSFUN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DFSFUN
--R
--R------------------------------ Operations --------------------------------
--R Gamma : DoubleFloat -> DoubleFloat    fresnelC : Float -> Float
--R fresnelS : Float -> Float
--R Beta : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R Beta : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R E1 : DoubleFloat -> OnePointCompletion(DoubleFloat)
--R Ei : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei1 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei2 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei3 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei4 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei5 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei6 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R En : (Integer,DoubleFloat) -> OnePointCompletion(DoubleFloat)
```

```
--R Gamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyAi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyAi : DoubleFloat -> DoubleFloat
--R airyBi : DoubleFloat -> DoubleFloat
--R airyBi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R besselI : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselI : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselJ : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselJ : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselK : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselK : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselY : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselY : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R digamma : DoubleFloat -> DoubleFloat
--R digamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R hypergeometricOF1 : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R hypergeometricOF1 : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R logGamma : DoubleFloat -> DoubleFloat
--R logGamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R polygamma : (NonNegativeInteger,DoubleFloat) -> DoubleFloat
--R polygamma : (NonNegativeInteger,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R
--E 1

--S 2 of 5
pearceyC:=_
[ [0.00, 0.0000000], [0.25, 0.3964561], [0.50, 0.5502472], [0.75, 0.6531193],_
  [1.00, 0.7217059], [1.25, 0.762404],  [1.50, 0.779084],  [1.75, 0.774978],_
  [2.00, 0.753302],  [2.25, 0.717446],  [2.50, 0.670986],  [2.75, 0.617615],_
  [3.00, 0.561020],  [3.25, 0.504745],  [3.50, 0.452047],  [3.75, 0.405762],_
  [4.00, 0.368193],  [4.25, 0.341021],  [4.50, 0.325249],  [4.75, 0.321186],_
  [5.00, 0.328457],  [5.25, 0.346058],  [5.50, 0.372439],  [5.75, 0.405610],_
  [6.00, 0.443274],  [6.25, 0.482966],  [6.50, 0.522202],  [6.75, 0.558620],_
  [7.00, 0.590116],  [7.25, 0.614951],  [7.50, 0.631845],  [7.75, 0.640034],_
  [8.00, 0.639301],  [8.25, 0.629969],  [8.50, 0.612868],  [8.75, 0.589271],_
  [9.00, 0.560804],  [9.25, 0.529344],  [9.50, 0.496895],  [9.75, 0.465469],_
 [10.00, 0.436964], [10.25, 0.413053], [10.50, 0.395087], [10.75, 0.384027],_
 [11.00, 0.380390], [11.25, 0.384231], [11.50, 0.395149], [11.75, 0.412319],_
 [12.00, 0.434555], [12.25, 0.460384], [12.50, 0.488146], [12.75, 0.516096],_
 [13.00, 0.542511], [13.25, 0.565798], [13.50, 0.584583], [13.75, 0.597795],_
 [14.00, 0.604721], [14.25, 0.605048], [14.50, 0.598871], [14.75, 0.586682],_
 [15.00, 0.569335], [15.25, 0.547984], [15.50, 0.524009], [15.75, 0.498930],_
 [16.00, 0.474310], [16.25, 0.451659], [16.50, 0.432343], [16.75, 0.417502],_
 [17.00, 0.407985], [17.25, 0.404300], [17.50, 0.406589], [17.75, 0.414627],_
 [18.00, 0.427837], [18.25, 0.445331], [18.50, 0.465972], [18.75, 0.488443],_
 [19.00, 0.511332], [19.25, 0.533222], [19.50, 0.552774], [19.75, 0.568812],_
 [20.00, 0.580389], [20.25, 0.586847], [20.50, 0.587849], [20.75, 0.583401],_
 [21.00, 0.573842], [21.25, 0.559824], [21.50, 0.542266], [21.75, 0.522293],_
 [22.00, 0.501167], [22.25, 0.480207], [22.50, 0.460707], [22.75, 0.443854],_
 [23.00, 0.430662], [23.25, 0.421906], [23.50, 0.418080], [23.75, 0.419367],_
```

```
 [24.00, 0.425635], [24.25, 0.436444], [24.50, 0.451078], [24.75, 0.468594],_
 [25.00, 0.487880], [25.25, 0.507725], [25.50, 0.526896], [25.75, 0.544215],_
 [26.00, 0.558626], [26.25, 0.569272], [26.50, 0.575524], [26.75, 0.577038],_
 [27.00, 0.573766], [27.25, 0.565954], [27.50, 0.554127], [27.75, 0.539054],_
 [28.00, 0.521695], [28.25, 0.503146], [28.50, 0.484566], [28.75, 0.467104],_
 [29.00, 0.451832], [29.25, 0.439675], [29.50, 0.431359], [29.75, 0.427366],_
 [30.00, 0.427908], [30.25, 0.432913], [30.50, 0.442034], [30.75, 0.454673],_
 [31.00, 0.470019], [31.25, 0.487100], [31.50, 0.504844], [31.75, 0.522148],_
 [32.00, 0.537944], [32.25, 0.551266], [32.50, 0.561307], [32.75, 0.567471],_
 [33.00, 0.569407], [33.25, 0.567026], [33.50, 0.560508], [33.75, 0.550288],_
 [34.00, 0.537026], [34.25, 0.521566], [34.50, 0.504881], [34.75, 0.488015],_
 [35.00, 0.472012], [35.25, 0.457857], [35.50, 0.446415], [35.75, 0.438375],_
 [36.00, 0.434212], [36.25, 0.434156], [36.50, 0.438182], [36.75, 0.446014],_
 [37.00, 0.457140], [37.25, 0.470848], [37.50, 0.486272], [37.75, 0.502444],_
 [38.00, 0.518359], [38.25, 0.533031], [38.50, 0.545560], [38.75, 0.555182],_
 [39.00, 0.561321], [39.25, 0.563619], [39.50, 0.561957], [39.75, 0.556463],_
 [40.00, 0.547503], [40.25, 0.535653], [40.50, 0.521665], [40.75, 0.506420],_
 [41.00, 0.490870], [41.25, 0.475980], [41.50, 0.462670], [41.75, 0.451755],_
 [42.00, 0.443897], [42.25, 0.439565], [42.50, 0.439006], [42.75, 0.442234],_
 [43.00, 0.449025], [43.25, 0.458938], [43.50, 0.471341], [43.75, 0.485450],_
 [44.00, 0.500382], [44.25, 0.515205], [44.50, 0.529002], [44.75, 0.540923],_
 [45.00, 0.550239], [45.25, 0.556387], [45.50, 0.559004], [45.75, 0.557947],_
 [46.00, 0.553301], [46.25, 0.545374], [46.50, 0.534676], [46.75, 0.521883],_
 [47.00, 0.507802], [47.25, 0.493312], [47.50, 0.479313], [47.75, 0.466670],_
 [48.00, 0.456160], [48.25, 0.448425], [48.50, 0.443930], [48.75, 0.442936],_
 [49.00, 0.445486], [49.25, 0.451406], [49.50, 0.460311], [49.75, 0.471633],_
 [50.00, 0.484658]]
--R
--R
--R   (1)
--R   [[0.0,0.0], [0.25,0.3964561], [0.5,0.5502472], [0.75,0.6531193],
--R    [1.0,0.7217059], [1.25,0.762404], [1.5,0.779084], [1.75,0.774978],
--R    [2.0,0.753302], [2.25,0.717446], [2.5,0.670986], [2.75,0.617615],
--R    [3.0,0.56102], [3.25,0.504745], [3.5,0.452047], [3.75,0.405762],
--R    [4.0,0.368193], [4.25,0.341021], [4.5,0.325249], [4.75,0.321186],
--R    [5.0,0.328457], [5.25,0.346058], [5.5,0.372439], [5.75,0.40561],
--R    [6.0,0.443274], [6.25,0.482966], [6.5,0.522202], [6.75,0.55862],
--R    [7.0,0.590116], [7.25,0.614951], [7.5,0.631845], [7.75,0.640034],
--R    [8.0,0.639301], [8.25,0.629969], [8.5,0.612868], [8.75,0.589271],
--R    [9.0,0.560804], [9.25,0.529344], [9.5,0.496895], [9.75,0.465469],
--R    [10.0,0.436964], [10.25,0.413053], [10.5,0.395087], [10.75,0.384027],
--R    [11.0,0.38039], [11.25,0.384231], [11.5,0.395149], [11.75,0.412319],
--R    [12.0,0.434555], [12.25,0.460384], [12.5,0.488146], [12.75,0.516096],
--R    [13.0,0.542511], [13.25,0.565798], [13.5,0.584583], [13.75,0.597795],
--R    [14.0,0.604721], [14.25,0.605048], [14.5,0.598871], [14.75,0.586682],
--R    [15.0,0.569335], [15.25,0.547984], [15.5,0.524009], [15.75,0.49893],
--R    [16.0,0.47431], [16.25,0.451659], [16.5,0.432343], [16.75,0.417502],
--R    [17.0,0.407985], [17.25,0.4043], [17.5,0.406589], [17.75,0.414627],
--R    [18.0,0.427837], [18.25,0.445331], [18.5,0.465972], [18.75,0.488443],
--R    [19.0,0.511332], [19.25,0.533222], [19.5,0.552774], [19.75,0.568812],
```

```
--R    [20.0,0.580389], [20.25,0.586847], [20.5,0.587849], [20.75,0.583401],
--R    [21.0,0.573842], [21.25,0.559824], [21.5,0.542266], [21.75,0.522293],
--R    [22.0,0.501167], [22.25,0.480207], [22.5,0.460707], [22.75,0.443854],
--R    [23.0,0.430662], [23.25,0.421906], [23.5,0.41808], [23.75,0.419367],
--R    [24.0,0.425635], [24.25,0.436444], [24.5,0.451078], [24.75,0.468594],
--R    [25.0,0.48788], [25.25,0.507725], [25.5,0.526896], [25.75,0.544215],
--R    [26.0,0.558626], [26.25,0.569272], [26.5,0.575524], [26.75,0.577038],
--R    [27.0,0.573766], [27.25,0.565954], [27.5,0.554127], [27.75,0.539054],
--R    [28.0,0.521695], [28.25,0.503146], [28.5,0.484566], [28.75,0.467104],
--R    [29.0,0.451832], [29.25,0.439675], [29.5,0.431359], [29.75,0.427366],
--R    [30.0,0.427908], [30.25,0.432913], [30.5,0.442034], [30.75,0.454673],
--R    [31.0,0.470019], [31.25,0.4871], [31.5,0.504844], [31.75,0.522148],
--R    [32.0,0.537944], [32.25,0.551266], [32.5,0.561307], [32.75,0.567471],
--R    [33.0,0.569407], [33.25,0.567026], [33.5,0.560508], [33.75,0.550288],
--R    [34.0,0.537026], [34.25,0.521566], [34.5,0.504881], [34.75,0.488015],
--R    [35.0,0.472012], [35.25,0.457857], [35.5,0.446415], [35.75,0.438375],
--R    [36.0,0.434212], [36.25,0.434156], [36.5,0.438182], [36.75,0.446014],
--R    [37.0,0.45714], [37.25,0.470848], [37.5,0.486272], [37.75,0.502444],
--R    [38.0,0.518359], [38.25,0.533031], [38.5,0.54556], [38.75,0.555182],
--R    [39.0,0.561321], [39.25,0.563619], [39.5,0.561957], [39.75,0.556463],
--R    [40.0,0.547503], [40.25,0.535653], [40.5,0.521665], [40.75,0.50642],
--R    [41.0,0.49087], [41.25,0.47598], [41.5,0.46267], [41.75,0.451755],
--R    [42.0,0.443897], [42.25,0.439565], [42.5,0.439006], [42.75,0.442234],
--R    [43.0,0.449025], [43.25,0.458938], [43.5,0.471341], [43.75,0.48545],
--R    [44.0,0.500382], [44.25,0.515205], [44.5,0.529002], [44.75,0.540923],
--R    [45.0,0.550239], [45.25,0.556387], [45.5,0.559004], [45.75,0.557947],
--R    [46.0,0.553301], [46.25,0.545374], [46.5,0.534676], [46.75,0.521883],
--R    [47.0,0.507802], [47.25,0.493312], [47.5,0.479313], [47.75,0.46667],
--R    [48.0,0.45616], [48.25,0.448425], [48.5,0.44393], [48.75,0.442936],
--R    [49.0,0.445486], [49.25,0.451406], [49.5,0.460311], [49.75,0.471633],
--R    [50.0,0.484658]]
--R                                                        Type: List(List(Float))
--E 2

--S 3 of 5
[[x.1,x.2,fresnelC(x.1),fresnelC(x.1)-x.2] for x in pearceyC]
--R
--R
--R   (2)
--R   [[0.0,0.0,0.0,0.0],
--R    [0.25,0.3964561,0.3964560954 2000459941,- 0.4579995400 59 E -8],
--R    [0.5,0.5502472,0.5502471546 4500637208,- 0.4535499362 79 E -7],
--R    [0.75,0.6531193,0.6531193584 292607957,0.5842926079 57 E -7],
--R    [1.0,0.7217059,0.7217059242 9260508777,0.2429260508 78 E -7],
--R    [1.25,0.762404,0.7624042531 0695862575,0.2531069586 258 E -6],
--R    [1.5,0.779084,0.7790837385 0396370968,- 0.2614960362 903 E -6],
--R    [1.75,0.774978,0.7749781554 8647675573,0.1554864767 557 E -6],
--R    [2.0,0.753302,0.7533023754 6789116559,0.3754678911 656 E -6],
--R    [2.25,0.717446,0.7174457114 2671496912,- 0.2885732850 309 E -6],
--R    [2.5,0.670986,0.6709858725 0950347483,- 0.1274904965 252 E -6],
```

```
--R    [2.75,0.617615,0.6176149424 522644261,- 0.5754773557 39 E -7],
--R    [3.0,0.56102,0.5610203289 781386693,0.3289781386 693 E -6],
--R    [3.25,0.504745,0.5047454684 5758505252,0.4684575850 525 E -6],
--R    [3.5,0.452047,0.4520471473 7344948252,0.1473734494 825 E -6],
--R    [3.75,0.405762,0.4057621282 0111750857,0.1282011175 086 E -6],
--R    [4.0,0.368193,0.3681929762 8097479631,- 0.2371902520 37 E -7],
--R    [4.25,0.341021,0.3410206544 3025861592,- 0.3455697413 8408 E -6],
--R    [4.5,0.325249,0.3252492294 0997382931,0.2294099738 293 E -6],
--R    [4.75,0.321186,0.3211858108 1411496285,- 0.1891858850 372 E -6],
--R    [5.0,0.328457,0.3284566248 6755260618,- 0.3751324473 9382 E -6],
--R    [5.25,0.346058,0.3460579739 835289212,- 0.2601647107 88 E -7],
--R    [5.5,0.372439,0.3724388324 2286847464,- 0.1675771315 254 E -6],
--R    [5.75,0.40561,0.4056100692 8402241876,0.6928402241 876 E -7],
--R    [6.0,0.443274,0.4432738563 3762333739,- 0.1436623766 626 E -6],
--R    [6.25,0.482966,0.4829657793 6269790038,- 0.2206373020 996 E -6],
--R    [6.5,0.522202,0.5222015767 8062637928,- 0.4232193736 207 E -6],
--R    [6.75,0.55862,0.5586203035 9563381818,0.3035956338 182 E -6],
--R    [7.0,0.590116,0.5901160610 939772876,0.6109397728 76 E -7],
--R    [7.25,0.614951,0.6149512165 651359762,0.2165651359 762 E -6],
--R    [7.5,0.631845,0.6318452111 5510492853,0.2111551049 285 E -6],
--R    [7.75,0.640034,0.6400345450 8057441808,0.5450805744 1808 E -6],
--R    [8.0,0.639301,0.6393012479 3060490759,0.2479306049 076 E -6],
--R    [8.25,0.629969,0.6299689859 2595953795,- 0.1407404046 2 E -7],
--R    [8.5,0.612868,0.6128678201 6845088171,- 0.1798315491 183 E -6],
--R    [8.75,0.589271,0.5892704028 202327594,- 0.5971797672 406 E -6],
--R    [9.0,0.560804,0.5608039810 6395486433,- 0.1893604513 57 E -7],
--R    [9.25,0.529344,0.5293438831 4394301245,- 0.1168560569 876 E -6],
--R    [9.5,0.496895,0.4968951155 6828252077,0.1155682825 208 E -6],
--R    [9.75,0.465469,0.4654692556 4195264614,0.2556419526 4614 E -6],
--R    [10.0,0.436964,0.4369639527 2938203483,- 0.4727061796 517 E -7],
--R    [10.25,0.413053,0.4130520539 2945147154,- 0.9460705485 2846 E -6],
--R    [10.5,0.395087,0.3950866689 6445290526,- 0.3310355470 9474 E -6],
--R    [10.75,0.384027,0.3840274319 9186745464,0.4319918674 5464 E -6],
--R    [11.0,0.38039,0.3803918718 5818433242,0.0000018718 581843324],
--R    [11.25,0.384231,0.3842342501 3415159269,0.0000032501 3415159269],
--R    [11.5,0.395149,0.3951525621 4136633426,0.0000035621 4136633426],
--R    [11.75,0.412319,0.4123227194 2948890487,0.0000037194 2948890487],
--R    [12.0,0.434555,0.4345573415 1310106383,0.0000023415 1310106383],
--R    [12.25,0.460384,0.4603851724 4692111457,0.0000011724 469211146],
--R    [12.5,0.488146,0.4881459845 7100939501,- 0.1542899060 5 E -7],
--R    [12.75,0.516096,0.5160950016 9800402129,- 0.9983019959 7871 E -6],
--R    [13.0,0.542511,0.5425104114 0076790311,- 0.5885992320 9689 E -6],
--R    [13.25,0.565798,0.5657974628 3445804807,- 0.5371655419 5193 E -6],
--R    [13.5,0.584583,0.5845829612 9626646639,- 0.3870373353 36 E -7],
--R    [13.75,0.597795,0.5977946491 2254037734,- 0.3508774596 227 E -6],
--R    [14.0,0.604721,0.6047209589 3428343112,- 0.4106571656 89 E -7],
--R    [14.25,0.605048,0.6050478757 7470272898,- 0.1242252972 71 E -6],
--R    [14.5,0.598871,0.5988710711 7868251227,0.7117868251 227 E -7],
--R    [14.75,0.586682,0.5866829870 7071159647,0.9870707115 9647 E -6],
--R    [15.0,0.569335,0.5693360588 8342021462,0.0000010588 834202146],
```

```
--R    [15.25,0.547984,0.5479846850 9637199303,0.6850963719 9303 E -6],
--R    [15.5,0.524009,0.5240097909 4969920392,0.7909496992 0392 E -6],
--R    [15.75,0.49893,0.4989308254 9359679937,0.8254935967 9937 E -6],
--R    [16.0,0.47431,0.4743107173 2032792592,0.7173203279 2592 E -6],
--R    [16.25,0.451659,0.4516596582 0625475374,0.6582062547 5374 E -6],
--R    [16.5,0.432343,0.4323435693 667817725,0.5693667817 725 E -6],
--R    [16.75,0.417502,0.4175027376 9772555286,0.7376977255 5286 E -6],
--R    [17.0,0.407985,0.4079854159 5598154173,0.4159559815 4173 E -6],
--R    [17.25,0.4043,0.4043002072 9975704736,0.2072997570 474 E -6],
--R    [17.5,0.406589,0.4065898632 2726313915,0.8632272631 3915 E -6],
--R    [17.75,0.414627,0.4146277893 4285610292,0.7893428561 0292 E -6],
--R    [18.0,0.427837,0.4278371578 9257267748,0.1578925726 775 E -6],
--R    [18.25,0.445331,0.4453311546 869756159,0.1546869756 159 E -6],
--R    [18.5,0.465972,0.4659716234 4840774835,- 0.3765515922 5165 E -6],
--R    [18.75,0.488443,0.4884422879 5458921791,- 0.7120454107 8209 E -6],
--R    [19.0,0.511332,0.5113318949 159239085,- 0.1050840760 915 E -6],
--R    [19.25,0.533222,0.5332220760 9031239166,0.7609031239 166 E -7],
--R    [19.5,0.552774,0.5527745062 1484209042,0.5062148420 9042 E -6],
--R    [19.75,0.568812,0.5688120435 8652883009,0.4358652883 01 E -7],
--R    [20.0,0.580389,0.5803889720 0491079211,- 0.2799508920 79 E -7],
--R    [20.25,0.586847,0.5868461863 8595115362,- 0.8136140488 4638 E -6],
--R    [20.5,0.587849,0.5878481324 17475698,- 0.8675825243 0201 E -6],
--R    [20.75,0.583401,0.5833994615 8414205371,- 0.0000015384 158579463],
--R    [21.0,0.573842,0.5738406247 6208387835,- 0.0000013752 379161216],
--R    [21.25,0.559824,0.5598229224 1791771666,- 0.0000010775 820822833],
--R    [21.5,0.542266,0.5422647781 3626412855,- 0.0000012218 637358715],
--R    [21.75,0.522293,0.5222921292 4856805145,- 0.8707514319 4855 E -6],
--R    [22.0,0.501167,0.5011667664 6519292577,- 0.2335348070 742 E -6],
--R    [22.25,0.480207,0.4802071484 9285761974,0.1484928576 197 E -6],
--R    [22.5,0.460707,0.4607066279 5313545574,- 0.3720468645 4426 E -6],
--R    [22.75,0.443854,0.4438541294 6962922904,0.1294696292 29 E -6],
--R    [23.0,0.430662,0.4306621163 5307345639,0.1163530734 564 E -6],
--R    [23.25,0.421906,0.4219061845 5918877179,0.1845591887 718 E -6],
--R    [23.5,0.41808,0.4180798648 2406114921,- 0.1351759388 508 E -6],
--R    [23.75,0.419367,0.4193672450 5058364187,0.2450505836 4187 E -6],
--R    [24.0,0.425635,0.4256349063 1204612991,- 0.9368795387 009 E -7],
--R    [24.25,0.436444,0.4364434671 0955178019,- 0.5328904482 1981 E -6],
--R    [24.5,0.451078,0.4510778257 8097250046,- 0.1742190274 995 E -6],
--R    [24.75,0.468594,0.4685940538 2129131152,0.5382129131 152 E -7],
--R    [25.0,0.48788,0.4878798923 5075186185,- 0.1076492481 382 E -6],
--R    [25.25,0.507725,0.5077249997 4785550835,- 0.2521444917 E -9],
--R    [25.5,0.526896,0.5268965378 5056351834,0.5378505635 1834 E -6],
--R    [25.75,0.544215,0.5442153996 057056159,0.3996057056 159 E -6],
--R    [26.0,0.558626,0.5586283863 2269416639,0.0000023863 226941664],
--R    [26.25,0.569272,0.5692719365 6311174305,- 0.6343688825 695 E -7],
--R    [26.5,0.575524,0.5755235681 519286467,- 0.4318480713 533 E -6],
--R    [26.75,0.577038,0.5770379859 3635357755,- 0.1406364642 24 E -7],
--R    [27.0,0.573766,0.5737657770 124270981,- 0.2229875729 019 E -6],
--R    [27.25,0.565954,0.5659537051 6013269979,- 0.2948398673 002 E -6],
--R    [27.5,0.554127,0.5541267540 7959236651,- 0.2459204076 335 E -6],
```

```
--R    [27.75,0.539054,0.5390531912 4040124686,- 0.8087595987 5314 E -6],
--R    [28.0,0.521695,0.5216949544 6705388656,- 0.4553294611 34 E -7],
--R    [28.25,0.503146,0.5031465467 9677094812,0.5467967709 4812 E -6],
--R    [28.5,0.484566,0.4845663019 0920749659,0.3019092074 9659 E -6],
--R    [28.75,0.467104,0.4671043214 6483016106,0.3214648301 6106 E -6],
--R    [29.0,0.451832,0.4518315476 4628819554,- 0.4523537118 0446 E -6],
--R    [29.25,0.439675,0.4396743309 7300438172,- 0.6690269956 1828 E -6],
--R    [29.5,0.431359,0.4313584765 6101829235,- 0.5234389817 0765 E -6],
--R    [29.75,0.427366,0.4273661360 3768100434,0.1360376810 043 E -6],
--R    [30.0,0.427908,0.4279080905 152246846,0.9051522468 46 E -7],
--R    [30.25,0.432913,0.4329130066 7882319144,0.6678823191 44 E -8],
--R    [30.5,0.442034,0.4420341837 397595623,0.1837397595 623 E -6],
--R    [30.75,0.454673,0.4546732432 9642671639,0.2432964267 1639 E -6],
--R    [31.0,0.470019,0.4700191387 880518641,0.1387880518 641 E -6],
--R    [31.25,0.4871,0.4870999928 5423804547,- 0.7145761954 53 E -8],
--R    [31.5,0.504844,0.5048444286 1506750598,0.4286150675 06 E -6],
--R    [31.75,0.522148,0.5221485610 4009044596,0.5610400904 4596 E -6],
--R    [32.0,0.537944,0.5379444615 1644558148,0.4615164455 815 E -6],
--R    [32.25,0.551266,0.5512658647 1800830465,- 0.1352819916 953 E -6],
--R    [32.5,0.561307,0.5613070997 1630231457,0.9971630231 457 E -7],
--R    [32.75,0.567471,0.5674716241 0900084533,0.6241090008 4533 E -6],
--R    [33.0,0.569407,0.5694072875 9780709976,0.2875978070 998 E -6],
--R    [33.25,0.567026,0.5670262735 8602313224,0.2735860231 322 E -6],
--R    [33.5,0.560508,0.5605085017 9224639203,0.5017922463 9203 E -6],
--R    [33.75,0.550288,0.5502884623 6065133847,0.4623606513 385 E -6],
--R    [34.0,0.537026,0.5370265183 9150336492,0.5183915033 6492 E -6],
--R    [34.25,0.521566,0.5215664629 6901266906,0.4629690126 691 E -6],
--R    [34.5,0.504881,0.5048817837 1619742811,0.7837161974 2811 E -6],
--R    [34.75,0.488015,0.4880148426 3306102148,- 0.1573669389 785 E -6],
--R    [35.0,0.472012,0.4720115763 0531557892,- 0.4236946844 2108 E -6],
--R    [35.25,0.457857,0.4578571669 7762795948,0.1669776279 595 E -6],
--R    [35.5,0.446415,0.4464153233 6579743027,0.3233657974 3027 E -6],
--R    [35.75,0.438375,0.4383753574 7411594227,0.3574741159 4227 E -6],
--R    [36.0,0.434212,0.4342121246 9792940288,0.1246979294 029 E -6],
--R    [36.25,0.434156,0.4341557286 0605475409,- 0.2713939452 4591 E -6],
--R    [36.5,0.438182,0.4381823268 5646395843,0.3268564639 5843 E -6],
--R    [36.75,0.446014,0.4460136608 378042178,- 0.3391621957 8221 E -6],
--R    [37.0,0.45714,0.4571399045 2295784095,- 0.9547704215 905 E -7],
--R    [37.25,0.470848,0.4708481333 0533700846,0.1333053370 085 E -6],
--R    [37.5,0.486272,0.4862712468 1703660919,- 0.7531829633 9081 E -6],
--R    [37.75,0.502444,0.5024452759 1583009151,0.0000012759 158300915],
--R    [38.0,0.518359,0.5183578008 5448394353,- 0.0000011991 455160565],
--R    [38.25,0.533031,0.5330321320 7141016248,0.0000011320 714101625],
--R    [38.5,0.54556,0.5455618898 0934649103,0.0000018898 09346491],
--R    [38.75,0.555182,0.5551816153 550106069,- 0.3846449893 931 E -6],
--R    [39.0,0.561321,0.5613199426 5924299224,- 0.0000010573 407570078],
--R    [39.25,0.563619,0.5636202521 4551507916,0.0000012521 455150792],
--R    [39.5,0.561957,0.5619620534 8342820413,0.0000050534 8342820413],
--R    [39.75,0.556463,0.5564674768 4327383218,0.0000044768 4327383218],
--R    [40.0,0.547503,0.5475016168 5739915545,- 0.0000013831 426008446],
```

```
--R    [40.25,0.535653,0.5356620581 907903875,0.0000090581 907903875],
--R    [40.5,0.521665,0.5216761291 3655653692,0.0000111291 365565369],
--R    [40.75,0.50642,0.5064307953 651030568,0.0000107953 651030568],
--R    [41.0,0.49087,0.4908775154 973959802,0.0000075154 973959802],
--R    [41.25,0.47598,0.4759947400 5522621443,0.0000147400 552262144],
--R    [41.5,0.46267,0.4626713569 4418969324,0.0000013569 441896932],
--R    [41.75,0.451755,0.4517471829 4138239325,- 0.0000078170 5861760675],
--R    [42.0,0.443897,0.4438901875 6697015185,- 0.0000068124 3302984815],
--R    [42.25,0.439565,0.4396208010 3531908999,0.0000558010 3531908999],
--R    [42.5,0.439006,0.4390276694 2121953287,0.0000216694 212195329],
--R    [42.75,0.442234,0.4421897258 6637518416,- 0.0000442741 3362481584],
--R    [43.0,0.449025,0.4492050976 5650075833,0.0001800976 565007583],
--R    [43.25,0.458938,0.4590688249 4791286065,0.0001308249 479128606],
--R    [43.5,0.471341,0.4714100547 838144553,0.0000690547 838144553],
--R    [43.75,0.48545,0.4856358171 3210831443,0.0001858171 321083144],
--R    [44.0,0.500382,0.5006613400 4130067711,0.0002793400 413006771],
--R    [44.25,0.515205,0.5151342238 0563989679,- 0.0000707761 943601032],
--R    [44.5,0.529002,0.5292284441 2946417563,0.0002264441 294641756],
--R    [44.75,0.540923,0.5412892775 4773223337,0.0003662775 477322334],
--R    [45.0,0.550239,0.5511413346 3543889498,0.0009023346 3543889498],
--R    [45.25,0.556387,0.5576853265 4573892342,0.0012983265 457389234],
--R    [45.5,0.559004,0.5587593755 575499822,- 0.0002446244 424500178],
--R    [45.75,0.557947,0.5579019728 3302653036,- 0.0000450271 669734696],
--R    [46.0,0.553301,0.5514473569 0540180212,- 0.0018536430 945981979],
--R    [46.25,0.545374,0.5464394049 9280209722,0.0010654049 928020972],
--R    [46.5,0.534676,0.5343156292 8025671166,- 0.0003603707 197432883],
--R    [46.75,0.521883,0.5290838154 5620869748,0.0072008154 5620869748],
--R    [47.0,0.507802,0.5077713992 4578468391,- 0.0000306007 542153161],
--R    [47.25,0.493312,0.4896517184 8241185069,- 0.0036602815 1758814931],
--R    [47.5,0.479313,0.4811196308 9536307999,0.0018066308 9536308],
--R    [47.75,0.46667,0.4702336348 628440715,0.0035636348 628440715],
--R    [48.0,0.45616,0.4632451705 9779549321,0.0070851705 9779549321],
--R    [48.25,0.448425,0.4345971450 8343415472,- 0.0138278549 165658453],
--R    [48.5,0.44393,0.4374345410 0388858939,- 0.0064954589 9611141061],
--R    [48.75,0.442936,0.4642108712 1599523247,0.0212748712 159952325],
--R    [49.0,0.445486,0.4326668777 9106094573,- 0.0128191222 089390543],
--R    [49.25,0.451406,0.4916664060 5304481548,0.0402604060 5304481548],
--R    [49.5,0.460311,0.3803109193 0779099871,- 0.0800000806 9220900129],
--R    [49.75,0.471633,0.5230763977 8694224618,0.0514433977 8694224618],
--R    [50.0,0.484658,0.4157389501 9459370449,- 0.0689190498 0540629551]]
--R                                                    Type: List(List(Float))
--E 3

--S 4 of 4
pearceyS:=_
[ [0.00, 0.0000000], [0.25, 0.0330970], [0.50, 0.0923658], [0.75, 0.1659294],_
  [1.00, 0.2475583], [1.25, 0.332216],  [1.50, 0.415483],  [1.75, 0.493469],_
  [2.00, 0.562849],  [2.25, 0.620944],  [2.50, 0.665787],  [2.75, 0.696174],_
  [3.00, 0.711685],  [3.25, 0.712666],  [3.50, 0.700180],  [3.75, 0.675925],_
  [4.00, 0.642119],  [4.25, 0.601362],  [4.50, 0.556489],  [4.75, 0.510408],_
```

```
  [5.00, 0.465942],  [5.25, 0.425677],  [5.50, 0.391834],  [5.75, 0.366161],_
  [6.00, 0.349852],  [6.25, 0.343503],  [6.50, 0.347099],  [6.75, 0.360040],_
  [7.00, 0.381195],  [7.25, 0.408982],  [7.50, 0.441485],  [7.75, 0.476568],_
  [8.00, 0.512010],  [8.25, 0.545638],  [8.50, 0.575457],  [8.75, 0.599758],_
  [9.00, 0.617214],  [9.25, 0.626948],  [9.50, 0.628573],  [9.75, 0.622204],_
 [10.00, 0.608436], [10.25, 0.588297], [10.50, 0.563176], [10.75, 0.534731],_
 [11.00, 0.504784], [11.25, 0.475208], [11.50, 0.447809], [11.75, 0.424220],_
 [12.00, 0.405810], [12.25, 0.393601], [12.50, 0.388217], [12.75, 0.389852],_
 [13.00, 0.398268], [13.25, 0.412817], [13.50, 0.432489], [13.75, 0.455978],_
 [14.00, 0.481770], [14.25, 0.508236], [14.50, 0.533736], [14.75, 0.556716],_
 [15.00, 0.575803], [15.25, 0.589887], [15.50, 0.598183], [15.75, 0.600273],_
 [16.00, 0.596126], [16.25, 0.586095], [16.50, 0.570890], [16.75, 0.551526],_
 [17.00, 0.529259], [17.25, 0.505505], [17.50, 0.481750], [17.75, 0.459460],_
 [18.00, 0.439989], [18.25, 0.424500], [18.50, 0.413893], [18.75, 0.408757],_
 [19.00, 0.409336], [19.25, 0.415520], [19.50, 0.426853], [19.75, 0.442571],_
 [20.00, 0.461646], [20.25, 0.482860], [20.50, 0.504875], [20.75, 0.526323],_
 [21.00, 0.545885], [21.25, 0.562375], [21.50, 0.574811], [21.75, 0.582472],_
 [22.00, 0.584939], [22.25, 0.582119], [22.50, 0.574246], [22.75, 0.561862],_
 [23.00, 0.545782], [23.25, 0.527040], [23.50, 0.506824], [23.75, 0.486399],_
 [24.00, 0.467029], [24.25, 0.449901], [24.50, 0.436051], [24.75, 0.426303],_
 [25.00, 0.421217], [25.25, 0.421062], [25.50, 0.425797], [25.75, 0.435083],_
 [26.00, 0.448300], [26.25, 0.464594], [26.50, 0.482927], [26.75, 0.502146],_
 [27.00, 0.521054], [27.25, 0.538483], [27.50, 0.553369], [27.75, 0.564814],_
 [28.00, 0.572142], [28.25, 0.574935], [28.50, 0.573060], [28.75, 0.566674],_
 [29.00, 0.556212], [29.25, 0.542357], [29.50, 0.525995], [29.75, 0.508160],_
 [30.00, 0.489969], [30.25, 0.472549], [30.50, 0.456974], [30.75, 0.444193],_
 [31.00, 0.434973], [31.25, 0.429857], [31.50, 0.429129], [31.75, 0.432799],_
 [32.00, 0.440605], [32.25, 0.452031], [32.50, 0.466343], [32.75, 0.482632],_
 [33.00, 0.499873], [33.25, 0.516992], [33.50, 0.532930], [33.75, 0.546708],_
 [34.00, 0.557490], [34.25, 0.564629], [34.50, 0.567709], [34.75, 0.566570],_
 [35.00, 0.561313], [35.25, 0.552293], [35.50, 0.540094], [35.75, 0.525495],_
 [36.00, 0.509417], [36.25, 0.492866], [36.50, 0.476871], [36.75, 0.462420],_
 [37.00, 0.450396], [37.25, 0.441528], [37.50, 0.436345], [37.75, 0.435144],_
 [38.00, 0.437971], [38.25, 0.444626], [38.50, 0.454670], [38.75, 0.467461],_
 [39.00, 0.482187], [39.25, 0.497924], [39.50, 0.513690], [39.75, 0.528507],_
 [40.00, 0.541464], [40.25, 0.551768], [40.50, 0.558799], [40.75, 0.562140],_
 [41.00, 0.561608], [41.25, 0.557258], [41.50, 0.549384], [41.75, 0.538494],_
 [42.00, 0.525282], [42.25, 0.510580], [42.50, 0.495309], [42.75, 0.480418],_
 [43.00, 0.466829], [43.25, 0.455375], [43.50, 0.446755], [43.75, 0.441487],_
 [44.00, 0.439878], [44.25, 0.442007], [44.50, 0.447720], [44.75, 0.456645],_
 [45.00, 0.468209], [45.25, 0.481681], [45.50, 0.496215], [45.75, 0.510904],_
 [46.00, 0.524837], [46.25, 0.537153], [46.50, 0.547099], [46.75, 0.554070],_
 [47.00, 0.557650], [47.25, 0.557635], [47.50, 0.554044], [47.75, 0.547120],_
 [48.00, 0.537309], [48.25, 0.525234], [48.50, 0.511657], [48.75, 0.497426],_
 [49.00, 0.483428], [49.25, 0.470529], [49.50, 0.459523], [49.75, 0.451084],_
 [50.00, 0.445722]]
--R
--R
--R   (3)
--R   [[0.0,0.0], [0.25,0.033097], [0.5,0.0923658], [0.75,0.1659294],
```

```
--R    [1.0,0.2475583], [1.25,0.332216], [1.5,0.415483], [1.75,0.493469],
--R    [2.0,0.562849], [2.25,0.620944], [2.5,0.665787], [2.75,0.696174],
--R    [3.0,0.711685], [3.25,0.712666], [3.5,0.70018], [3.75,0.675925],
--R    [4.0,0.642119], [4.25,0.601362], [4.5,0.556489], [4.75,0.510408],
--R    [5.0,0.465942], [5.25,0.425677], [5.5,0.391834], [5.75,0.366161],
--R    [6.0,0.349852], [6.25,0.343503], [6.5,0.347099], [6.75,0.36004],
--R    [7.0,0.381195], [7.25,0.408982], [7.5,0.441485], [7.75,0.476568],
--R    [8.0,0.51201], [8.25,0.545638], [8.5,0.575457], [8.75,0.599758],
--R    [9.0,0.617214], [9.25,0.626948], [9.5,0.628573], [9.75,0.622204],
--R    [10.0,0.608436], [10.25,0.588297], [10.5,0.563176], [10.75,0.534731],
--R    [11.0,0.504784], [11.25,0.475208], [11.5,0.447809], [11.75,0.42422],
--R    [12.0,0.40581], [12.25,0.393601], [12.5,0.388217], [12.75,0.389852],
--R    [13.0,0.398268], [13.25,0.412817], [13.5,0.432489], [13.75,0.455978],
--R    [14.0,0.48177], [14.25,0.508236], [14.5,0.533736], [14.75,0.556716],
--R    [15.0,0.575803], [15.25,0.589887], [15.5,0.598183], [15.75,0.600273],
--R    [16.0,0.596126], [16.25,0.586095], [16.5,0.57089], [16.75,0.551526],
--R    [17.0,0.529259], [17.25,0.505505], [17.5,0.48175], [17.75,0.45946],
--R    [18.0,0.439989], [18.25,0.4245], [18.5,0.413893], [18.75,0.408757],
--R    [19.0,0.409336], [19.25,0.41552], [19.5,0.426853], [19.75,0.442571],
--R    [20.0,0.461646], [20.25,0.48286], [20.5,0.504875], [20.75,0.526323],
--R    [21.0,0.545885], [21.25,0.562375], [21.5,0.574811], [21.75,0.582472],
--R    [22.0,0.584939], [22.25,0.582119], [22.5,0.574246], [22.75,0.561862],
--R    [23.0,0.545782], [23.25,0.52704], [23.5,0.506824], [23.75,0.486399],
--R    [24.0,0.467029], [24.25,0.449901], [24.5,0.436051], [24.75,0.426303],
--R    [25.0,0.421217], [25.25,0.421062], [25.5,0.425797], [25.75,0.435083],
--R    [26.0,0.4483], [26.25,0.464594], [26.5,0.482927], [26.75,0.502146],
--R    [27.0,0.521054], [27.25,0.538483], [27.5,0.553369], [27.75,0.564814],
--R    [28.0,0.572142], [28.25,0.574935], [28.5,0.57306], [28.75,0.566674],
--R    [29.0,0.556212], [29.25,0.542357], [29.5,0.525995], [29.75,0.50816],
--R    [30.0,0.489969], [30.25,0.472549], [30.5,0.456974], [30.75,0.444193],
--R    [31.0,0.434973], [31.25,0.429857], [31.5,0.429129], [31.75,0.432799],
--R    [32.0,0.440605], [32.25,0.452031], [32.5,0.466343], [32.75,0.482632],
--R    [33.0,0.499873], [33.25,0.516992], [33.5,0.53293], [33.75,0.546708],
--R    [34.0,0.55749], [34.25,0.564629], [34.5,0.567709], [34.75,0.56657],
--R    [35.0,0.561313], [35.25,0.552293], [35.5,0.540094], [35.75,0.525495],
--R    [36.0,0.509417], [36.25,0.492866], [36.5,0.476871], [36.75,0.46242],
--R    [37.0,0.450396], [37.25,0.441528], [37.5,0.436345], [37.75,0.435144],
--R    [38.0,0.437971], [38.25,0.444626], [38.5,0.45467], [38.75,0.467461],
--R    [39.0,0.482187], [39.25,0.497924], [39.5,0.51369], [39.75,0.528507],
--R    [40.0,0.541464], [40.25,0.551768], [40.5,0.558799], [40.75,0.56214],
--R    [41.0,0.561608], [41.25,0.557258], [41.5,0.549384], [41.75,0.538494],
--R    [42.0,0.525282], [42.25,0.51058], [42.5,0.495309], [42.75,0.480418],
--R    [43.0,0.466829], [43.25,0.455375], [43.5,0.446755], [43.75,0.441487],
--R    [44.0,0.439878], [44.25,0.442007], [44.5,0.44772], [44.75,0.456645],
--R    [45.0,0.468209], [45.25,0.481681], [45.5,0.496215], [45.75,0.510904],
--R    [46.0,0.524837], [46.25,0.537153], [46.5,0.547099], [46.75,0.55407],
--R    [47.0,0.55765], [47.25,0.557635], [47.5,0.554044], [47.75,0.54712],
--R    [48.0,0.537309], [48.25,0.525234], [48.5,0.511657], [48.75,0.497426],
--R    [49.0,0.483428], [49.25,0.470529], [49.5,0.459523], [49.75,0.451084],
--R    [50.0,0.445722]]
```

```
--R                                                    Type: List(List(Float))
--E 4

--S 5 of 5
[[x.1,x.2,fresnelS(x.1),fresnelS(x.1)-x.2] for x in pearceyS]
--R
--R
--R   (4)
--R   [[0.0,0.0,0.0,0.0],
--R    [0.25,0.033097,0.0330970688 3016300650 6,0.6883016300 6506 E -7],
--R    [0.5,0.0923658,0.0923657602 0981415623 9,- 0.3979018584 376 E -7],
--R    [0.75,0.1659294,0.1659293548 0210796268,- 0.4519789203 732 E -7],
--R    [1.0,0.2475583,0.2475582876 5161084261,- 0.1234838915 74 E -7],
--R    [1.25,0.332216,0.3322160625 9944205945,0.6259944205 945 E -7],
--R    [1.5,0.415483,0.4154833182 6565542581,0.3182656554 2581 E -6],
--R    [1.75,0.493469,0.4934686007 9623515999,- 0.3992037648 4001 E -6],
--R    [2.0,0.562849,0.5628489062 300564793,- 0.9376994352 07 E -7],
--R    [2.25,0.620944,0.6209439286 8952425621,- 0.7131047574 378 E -7],
--R    [2.5,0.665787,0.6657868895 7817255522,- 0.1104218274 448 E -6],
--R    [2.75,0.696174,0.6961742312 5801822896,0.2312580182 29 E -6],
--R    [3.0,0.711685,0.7116850216 0753003252,0.2160753003 25 E -7],
--R    [3.25,0.712666,0.7126658336 0193196899,- 0.1663980680 31 E -6],
--R    [3.5,0.70018,0.7001803262 1710404986,0.3262171040 499 E -6],
--R    [3.75,0.675925,0.6759254869 8611906826,0.4869861190 6826 E -6],
--R    [4.0,0.642119,0.6421187357 4451469534,- 0.2642554853 047 E -6],
--R    [4.25,0.601362,0.6013618861 0406611871,- 0.1138959338 813 E -6],
--R    [4.5,0.556489,0.5564893045 0127589586,0.3045012758 959 E -6],
--R    [4.75,0.510408,0.5104084694 9752509967,0.4694975250 997 E -6],
--R    [5.0,0.465942,0.4659414967 6625853241,- 0.5032337414 6759 E -6],
--R    [5.25,0.425677,0.4256760562 5372584118,- 0.9437462741 5882 E -6],
--R    [5.5,0.391834,0.3918334884 2241092549,- 0.5115775890 7451 E -6],
--R    [5.75,0.366161,0.3661608711 7846843575,- 0.1288215315 642 E -6],
--R    [6.0,0.349852,0.3498523653 5397811419,0.3653539781 1419 E -6],
--R    [6.25,0.343503,0.3435034609 0379470942,0.4609037947 0942 E -6],
--R    [6.5,0.347099,0.3470998591 9509107808,0.8591950910 7808 E -6],
--R    [6.75,0.36004,0.3600407686 5353610359,0.7686535361 0359 E -6],
--R    [7.0,0.381195,0.3811944739 4496760982,- 0.5260550323 9018 E -6],
--R    [7.25,0.408982,0.4089822714 33853148,0.2714338531 48 E -6],
--R    [7.5,0.441485,0.4414853446 1004457156,0.3446100445 7156 E -6],
--R    [7.75,0.476568,0.4765679658 3912270914,- 0.3416087729 086 E -7],
--R    [8.0,0.51201,0.5120096184 674641167,- 0.3815325358 833 E -6],
--R    [8.25,0.545638,0.5456382758 4836770255,0.2758483677 026 E -6],
--R    [8.5,0.575457,0.5754571656 0424747441,0.1656042474 744 E -6],
--R    [8.75,0.599758,0.5997578767 1218929607,- 0.1232878107 039 E -6],
--R    [9.0,0.617214,0.6172135970 241896115,- 0.4029758103 885 E -6],
--R    [9.25,0.626948,0.6269475401 6193698812,- 0.4598380630 119 E -6],
--R    [9.5,0.628573,0.6285731549 3626283028,0.1549362628 303 E -6],
--R    [9.75,0.622204,0.6222044149 0114838121,0.4149011483 812 E -6],
--R    [10.0,0.608436,0.6084362590 651108963,0.2590651108 963 E -6],
--R    [10.25,0.588297,0.5882969931 8226702044,- 0.6817732979 56 E -8],
```

```
--R    [10.5,0.563176,0.5631760638 822475396,0.6388224753 96 E -7],
--R    [10.75,0.534731,0.5347319938 6555282657,0.9938655528 2656 E -6],
--R    [11.0,0.504784,0.5047863386 4734203894,0.0000023386 473420389],
--R    [11.25,0.475208,0.4752102358 2255398909,0.0000022358 2255398909],
--R    [11.5,0.447809,0.4478104304 5274307166,0.0000014304 527430717],
--R    [11.75,0.42422,0.4242215626 5460748949,0.0000015626 546074895],
--R    [12.0,0.40581,0.4058110077 5914323067,0.0000010077 591432307],
--R    [12.25,0.393601,0.3936017007 1076950741,0.7007107695 0741 E -6],
--R    [12.5,0.388217,0.3882172107 7941192612,0.2107794119 261 E -6],
--R    [12.75,0.389852,0.3898519335 8673393733,- 0.6641326606 266 E -7],
--R    [13.0,0.398268,0.3982677211 0844849665,- 0.2788915515 0335 E -6],
--R    [13.25,0.412817,0.4128166704 5498856625,- 0.3295450114 3375 E -6],
--R    [13.5,0.432489,0.4324882345 041217611,- 0.7654958782 389 E -6],
--R    [13.75,0.455978,0.4559773937 3230061439,- 0.6062676993 8561 E -6],
--R    [14.0,0.48177,0.4817694215 5974436016,- 0.5784402556 3984 E -6],
--R    [14.25,0.508236,0.5082358539 0961301684,- 0.1460903869 832 E -6],
--R    [14.5,0.533736,0.5337356883 2480717037,- 0.3116751928 296 E -6],
--R    [14.75,0.556716,0.5567156190 4121537424,- 0.3809587846 258 E -6],
--R    [15.0,0.575803,0.5758032698 0780549342,0.2698078054 934 E -6],
--R    [15.25,0.589887,0.5898879014 6212257458,0.9014621225 7458 E -6],
--R    [15.5,0.598183,0.5981839105 0565071631,0.9105056507 1631 E -6],
--R    [15.75,0.600273,0.6002735434 6065043602,0.5434606504 3602 E -6],
--R    [16.0,0.596126,0.5961265594 980177631,0.5594980177 631 E -6],
--R    [16.25,0.586095,0.5860959995 3489677741,0.9995348967 7741 E -6],
--R    [16.5,0.57089,0.5708906764 8964155137,0.6764896415 5137 E -6],
--R    [16.75,0.551526,0.5515264006 7065271525,0.4006706527 152 E -6],
--R    [17.0,0.529259,0.5292592129 0892379155,0.2129089237 915 E -6],
--R    [17.25,0.505505,0.5055049420 4294755784,- 0.5795705244 22 E -7],
--R    [17.5,0.48175,0.4817501725 5916779413,0.1725591677 941 E -6],
--R    [17.75,0.45946,0.4594601596 7650279535,0.1596765027 953 E -6],
--R    [18.0,0.439989,0.4399893396 8288159635,0.3396828815 9635 E -6],
--R    [18.25,0.4245,0.4244998504 5434107811,- 0.1495456589 219 E -6],
--R    [18.5,0.413893,0.4138929191 1930108708,- 0.8088069891 292 E -7],
--R    [18.75,0.408757,0.4087571284 8337051586,0.1284833705 159 E -6],
--R    [19.0,0.409336,0.4093364957 3057632165,0.4957305763 2165 E -6],
--R    [19.25,0.41552,0.4155200541 2770796534,0.5412770796 534 E -7],
--R    [19.5,0.426853,0.4268532982 2330761689,0.2982233076 1689 E -6],
--R    [19.75,0.442571,0.4425705170 1115000023,- 0.4829888499 9977 E -6],
--R    [20.0,0.461646,0.4616457788 1596354143,- 0.2211840364 586 E -6],
--R    [20.25,0.48286,0.4828592218 8905385632,- 0.7781109461 4368 E -6],
--R    [20.5,0.504875,0.5048744115 2503595117,- 0.5884749640 4883 E -6],
--R    [20.75,0.526323,0.5263218997 3686238137,- 0.0000011002 631376186],
--R    [21.0,0.545885,0.5458838021 1302594691,- 0.0000011978 869740531],
--R    [21.25,0.562375,0.5623742046 1345743628,- 0.7953865425 6372 E -6],
--R    [21.5,0.574811,0.5748105273 3873582708,- 0.4726612641 729 E -6],
--R    [21.75,0.582472,0.5824715801 7267885681,- 0.4198273211 432 E -6],
--R    [22.0,0.584939,0.5849389064 8781662671,- 0.9351218337 329 E -7],
--R    [22.25,0.582119,0.5821190706 5525943405,0.7065525943 405 E -7],
--R    [22.5,0.574246,0.5742457362 6788616758,- 0.2637321138 324 E -6],
--R    [22.75,0.561862,0.5618616308 0580392102,- 0.3691941960 79 E -6],
```

```
--R    [23.0,0.545782,0.5457817221 8835794711,- 0.2778116420 529 E -6],
--R    [23.25,0.52704,0.5270400684 8894301738,0.6848894301 738 E -7],
--R    [23.5,0.506824,0.5068237756 6322281921,- 0.2243367771 808 E -6],
--R    [23.75,0.486399,0.4863982517 4424787908,- 0.7482557521 2092 E -6],
--R    [24.0,0.467029,0.4670284356 6225373159,- 0.5643377462 6841 E -6],
--R    [24.25,0.449901,0.4499008775 2768569685,- 0.1224723143 032 E -6],
--R    [24.5,0.436051,0.4360514454 5938610932,0.4454593861 0932 E -6],
--R    [24.75,0.426303,0.4263030410 2444838947,0.4102444838 947 E -7],
--R    [25.0,0.421217,0.4212170480 2256113561,0.4802256113 561 E -7],
--R    [25.25,0.421062,0.4210613602 4178097468,- 0.6397582190 2532 E -6],
--R    [25.5,0.425797,0.4257967889 3146047209,- 0.2110685395 279 E -6],
--R    [25.75,0.435083,0.4350825056 1623586834,- 0.4943837641 3166 E -6],
--R    [26.0,0.4483,0.4483000011 9239677468,0.1192396774 7 E -8],
--R    [26.25,0.464594,0.4645939109 1118988398,- 0.8908881011 602 E -7],
--R    [26.5,0.482927,0.4829270344 0426767399,0.3440426767 4 E -7],
--R    [26.75,0.502146,0.5021460338 2247443232,0.3382247443 23 E -7],
--R    [27.0,0.521054,0.5210536692 2170336971,- 0.3307782966 303 E -6],
--R    [27.25,0.538483,0.5384830674 2690379321,0.6742690379 321 E -7],
--R    [27.5,0.553369,0.5533694357 5487427564,0.4357548742 756 E -6],
--R    [27.75,0.564814,0.5648148314 9166491027,0.8314916649 1027 E -6],
--R    [28.0,0.572142,0.5721420631 6790330366,0.6316790330 366 E -7],
--R    [28.25,0.574935,0.5749345026 8645175971,- 0.4973135482 4029 E -6],
--R    [28.5,0.57306,0.5730594822 652482619,- 0.5177347517 3809 E -6],
--R    [28.75,0.566674,0.5666739801 1092060771,- 0.1988907939 23 E -7],
--R    [29.0,0.556212,0.5562123974 5318039664,0.3974531803 966 E -6],
--R    [29.25,0.542357,0.5423573357 3057143592,0.3357305714 359 E -6],
--R    [29.5,0.525995,0.5259953183 8591850851,0.3183859185 085 E -6],
--R    [29.75,0.50816,0.5081603114 6691447077,0.3114669144 708 E -6],
--R    [30.0,0.489969,0.4899686293 6294561315,- 0.3706370543 8685 E -6],
--R    [30.25,0.472549,0.4725493003 70056332,0.3003700563 32 E -6],
--R    [30.5,0.456974,0.4569742329 8007908723,0.2329800790 872 E -6],
--R    [30.75,0.444193,0.4441924732 1202200062,- 0.5267879779 9938 E -6],
--R    [31.0,0.434973,0.4349725872 2359219467,- 0.4127764078 0533 E -6],
--R    [31.25,0.429857,0.4298566440 1538454345,- 0.3559846154 5655 E -6],
--R    [31.5,0.429129,0.4291285439 2130018393,- 0.4560786998 1607 E -6],
--R    [31.75,0.432799,0.4327985404 7868362044,- 0.4595213163 7956 E -6],
--R    [32.0,0.440605,0.4406047705 8303021964,- 0.2294169697 804 E -6],
--R    [32.25,0.452031,0.4520315818 7779157328,0.5818777915 7328 E -6],
--R    [32.5,0.466343,0.4663433562 2850339586,0.3562285033 9586 E -6],
--R    [32.75,0.482632,0.4826316757 5316315801,- 0.3242468368 4199 E -6],
--R    [33.0,0.499873,0.4998728498 5761360978,- 0.1501423863 902 E -6],
--R    [33.25,0.516992,0.5169919568 4608062411,- 0.4315391937 59 E -7],
--R    [33.5,0.53293,0.5329297341 0588518507,- 0.2658941148 149 E -6],
--R    [33.75,0.546708,0.5467080330 374204341,0.3303742043 41 E -7],
--R    [34.0,0.55749,0.5574894980 70161353,- 0.5019298386 47 E -6],
--R    [34.25,0.564629,0.5646284739 1529463715,- 0.5260847053 6285 E -6],
--R    [34.5,0.567709,0.5677093003 3772777525,0.3003377277 753 E -6],
--R    [34.75,0.56657,0.5665706285 1241575866,0.6285124157 5866 E -6],
--R    [35.0,0.561313,0.5613133551 8174616414,0.3551817461 641 E -6],
--R    [35.25,0.552293,0.5522925303 0361142379,- 0.4696963885 762 E -6],
```

```
--R    [35.5,0.540094,0.5400936428 6312004459,- 0.3571368799 554 E -6],
--R    [35.75,0.525495,0.5254947452 6478477127,- 0.2547352152 287 E -6],
--R    [36.0,0.509417,0.5094168513 7281277168,- 0.1486271872 283 E -6],
--R    [36.25,0.492866,0.4928661334 9354600731,0.1334935460 073 E -6],
--R    [36.5,0.476871,0.4768716484 1410150512,0.6484141015 0512 E -6],
--R    [36.75,0.46242,0.4624199779 4599876237,- 0.2205400123 76 E -7],
--R    [37.0,0.450396,0.4503960869 296054767,0.8692960547 67 E -7],
--R    [37.25,0.441528,0.4415290525 3642844985,0.0000010525 364284499],
--R    [37.5,0.436345,0.4363459415 0326090466,0.9415032609 0466 E -6],
--R    [37.75,0.435144,0.4351444622 142641974,0.4622142641 974 E -6],
--R    [38.0,0.437971,0.4379709919 0155134798,- 0.8098448652 02 E -8],
--R    [38.25,0.444626,0.4446262686 7031444849,0.2686703144 4849 E -6],
--R    [38.5,0.45467,0.4546708406 3699119293,0.8406369911 9293 E -6],
--R    [38.75,0.467461,0.4674625380 8725087486,0.0000015380 872508749],
--R    [39.0,0.482187,0.4821902969 1952370995,0.0000032969 1952370995],
--R    [39.25,0.497924,0.4979286483 1116955857,0.0000046483 1116955857],
--R    [39.5,0.51369,0.5136882217 6583353419,- 0.0000017782 341664658],
--R    [39.75,0.528507,0.5285123707 6397702394,0.0000053707 6397702393],
--R    [40.0,0.541464,0.5414672429 1598081817,0.0000032429 159808182],
--R    [40.25,0.551768,0.5517745240 9094539537,0.0000065240 9094539537],
--R    [40.5,0.558799,0.5588050506 5912486677,0.0000060506 5912486677],
--R    [40.75,0.56214,0.5621369791 6412056122,- 0.0000030208 358794388],
--R    [41.0,0.561608,0.5616247711 0814605692,0.0000167711 081460569],
--R    [41.25,0.557258,0.5572675955 430330709,0.0000095955 430330709],
--R    [41.5,0.549384,0.5493773200 0894645567,- 0.0000066799 9105354433],
--R    [41.75,0.538494,0.5385616342 9485623221,0.0000676342 9485623221],
--R    [42.0,0.525282,0.5252752111 2342898796,- 0.0000067888 7657101203],
--R    [42.25,0.51058,0.5104939314 7702373041,- 0.0000860685 2297626959],
--R    [42.5,0.495309,0.4953417684 038968965,0.0000327684 038968965],
--R    [42.75,0.480418,0.4806241406 5752533599,0.0002061406 57525336],
--R    [43.0,0.466829,0.4667676898 4497320465,- 0.0000613101 5502679535],
--R    [43.25,0.455375,0.4553638694 6250373139,- 0.0000111305 374962686],
--R    [43.5,0.446755,0.4466829634 3044419959,- 0.0000720365 6955580042],
--R    [43.75,0.441487,0.4413450709 5947820527,- 0.0001419290 405217947],
--R    [44.0,0.439878,0.4398575523 27691464,- 0.0000204476 72308536],
--R    [44.25,0.442007,0.4420222062 7982284284,0.0000152062 798228428],
--R    [44.5,0.44772,0.4478984397 6769945092,0.0001784397 676994509],
--R    [44.75,0.456645,0.4559273692 7109515615,- 0.0007176307 2890484385],
--R    [45.0,0.468209,0.4676487917 6051833044,- 0.0005602082 3948166956],
--R    [45.25,0.481681,0.4819138773 9121176935,0.0002328773 912117694],
--R    [45.5,0.496215,0.4954833490 8753370091,- 0.0007316509 1246629909],
--R    [45.75,0.510904,0.5106859986 5658467469,- 0.0002180013 434153253],
--R    [46.0,0.524837,0.5242725127 6971944478,- 0.0005644872 3028055522],
--R    [46.25,0.537153,0.5375126919 1563903102,0.0003596919 15639031],
--R    [46.5,0.547099,0.5460962647 9105883281,- 0.0010027352 089411672],
--R    [46.75,0.55407,0.5524834800 5502643628,- 0.0015865199 449735637],
--R    [47.0,0.55765,0.5580597398 2343233094,0.0004097398 234323309],
--R    [47.25,0.557635,0.5590882082 4170920981,0.0014532082 417092098],
--R    [47.5,0.554044,0.5475186455 874275914,- 0.0065253544 125724086],
--R    [47.75,0.54712,0.5455984940 601030917,- 0.0015215059 398969083],
```

```
--R    [48.0,0.537309,0.5196318344 7879605961,- 0.0176771655 212039404],
--R    [48.25,0.525234,0.5131703830 5664910816,- 0.0120636169 433508918],
--R    [48.5,0.511657,0.4873016398 0175736065,- 0.0243553601 982426393],
--R    [48.75,0.497426,0.5181479895 3691666632,0.0207219895 369166663],
--R    [49.0,0.483428,0.4598623177 3318010769,- 0.0235656822 668198923],
--R    [49.25,0.470529,0.4397312463 9609993647,- 0.0307977536 039000635],
--R    [49.5,0.459523,0.4156435749 1719610661,- 0.0438794250 8280389339],
--R    [49.75,0.451084,0.4230292350 336881258,- 0.0280547649 663118742],
--R    [50.0,0.445722,0.3044252284 9276788618,- 0.1412967715 072321138]]
--R                                                    Type: List(List(Float))
--E 5

)spool
)lisp (bye)
```

— DoubleFloatSpecialFunctions.help —

```
=====================================================================
DoubleFloatSpecialFunctions examples
=====================================================================

The formula used will agree with the Table of the Fresnel Integral
by Pearcey (1959) to 6 decimal places up to an argument of about 35.0.
After that the summation gets slowly worse, agreeing to only 2 digits
at about 45.0.

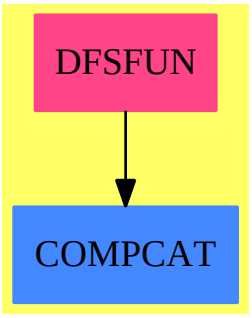
fresnelC(1.5)
   0.7790837385 0396370968

fresnelS(1.5)
   0.4154833182 6565542581



See Also:
o )show DoubleFloatSpecialFunctions
```

5.9.1 DoubleFloatSpecialFunctions (DFSFUN)



Exports:

airyAi	airyBi	besselI	besselJ	besselK
besselY	Beta	digamma	E1	Ei
Ei1	Ei2	Ei3	Ei4	Ei5
Ei6	En	fresnelC	fresnelS	Gamma
hypergeometric0F1	logGamma	polygamma		

— package DFSFUN DoubleFloatSpecialFunctions —

```
)abbrev package DFSFUN DoubleFloatSpecialFunctions
++ Author: Bruce W. Char, Timothy Daly, Stephen M. Watt
++ Date Created:  1990
++ Date Last Updated: Jan 19, 2008
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides special functions for double precision
++ real and complex floating point.

DoubleFloatSpecialFunctions(): Exports == Impl where
    NNI  ==> NonNegativeInteger
    PI   ==> Integer
    R    ==> DoubleFloat
    C    ==> Complex DoubleFloat
    OPR  ==> OnePointCompletion R
    F    ==> Float
    LF   ==> List Float

    Exports ==> with
        Gamma: R -> R
```

```
 ++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
 ++   \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.
Gamma: C -> C
 ++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
 ++   \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.

E1: R -> OPR
 ++ E1(x) is the Exponential Integral function
 ++ The current implementation is a piecewise approximation
 ++ involving one poly from -4..4 and a second poly for x > 4

En: (PI,R) -> OPR
 ++ En(n,x) is the nth Exponential Integral Function

Ei: (OPR) -> OPR
 ++ Ei is the Exponential Integral function
 ++ This is computed using a 6 part piecewise approximation.
 ++ DoubleFloat can only preserve about 16 digits but the
 ++ Chebyshev approximation used can give 30 digits.

Ei1: (OPR) -> OPR
 ++ Ei1 is the first approximation of Ei where the result is
 ++ x*%e^-x*Ei(x) from -infinity to -10 (preserves digits)

Ei2: (OPR) -> OPR
 ++ Ei2 is the first approximation of Ei where the result is
 ++ x*%e^-x*Ei(x) from -10 to -4 (preserves digits)

Ei3: (OPR) -> OPR
 ++ Ei3 is the first approximation of Ei where the result is
 ++ (Ei(x)-log |x| - gamma)/x from -4 to 4 (preserves digits)

Ei4: (OPR) -> OPR
 ++ Ei4 is the first approximation of Ei where the result is
 ++ x*%e^-x*Ei(x) from 4 to 12 (preserves digits)

Ei5: (OPR) -> OPR
 ++ Ei5 is the first approximation of Ei where the result is
 ++ x*%e^-x*Ei(x) from 12 to 32 (preserves digits)

Ei6: (OPR) -> OPR
 ++ Ei6 is the first approximation of Ei where the result is
 ++ x*%e^-x*Ei(x) from 32 to infinity (preserves digits)

Beta: (R, R) -> R
 ++ Beta(x, y) is the Euler beta function, \spad{B(x,y)}, defined by
 ++   \spad{Beta(x,y) = integrate(t^(x-1)*(1-t)^(y-1), t=0..1)}.
 ++ This is related to \spad{Gamma(x)} by
 ++   \spad{Beta(x,y) = Gamma(x)*Gamma(y) / Gamma(x + y)}.
Beta: (C, C) -> C
```

```
 ++ Beta(x, y) is the Euler beta function, \spad{B(x,y)}, defined by
 ++   \spad{Beta(x,y) = integrate(t^(x-1)*(1-t)^(y-1), t=0..1)}.
 ++ This is related to \spad{Gamma(x)} by
 ++   \spad{Beta(x,y) = Gamma(x)*Gamma(y) / Gamma(x + y)}.

logGamma: R -> R
 ++ logGamma(x) is the natural log of \spad{Gamma(x)}.
 ++ This can often be computed even if \spad{Gamma(x)} cannot.
logGamma: C -> C
 ++ logGamma(x) is the natural log of \spad{Gamma(x)}.
 ++ This can often be computed even if \spad{Gamma(x)} cannot.

digamma: R -> R
 ++ digamma(x) is the function, \spad{psi(x)}, defined by
 ++   \spad{psi(x) = Gamma'(x)/Gamma(x)}.
digamma: C -> C
 ++ digamma(x) is the function, \spad{psi(x)}, defined by
 ++   \spad{psi(x) = Gamma'(x)/Gamma(x)}.

polygamma: (NNI, R) -> R
 ++ polygamma(n, x) is the n-th derivative of \spad{digamma(x)}.
polygamma: (NNI, C) -> C
 ++ polygamma(n, x) is the n-th derivative of \spad{digamma(x)}.

besselJ: (R,R) -> R
 ++ besselJ(v,x) is the Bessel function of the first kind,
 ++ \spad{J(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
besselJ: (C,C) -> C
 ++ besselJ(v,x) is the Bessel function of the first kind,
 ++ \spad{J(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.

besselY: (R, R) -> R
 ++ besselY(v,x) is the Bessel function of the second kind,
 ++ \spad{Y(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
 ++ Note that the default implementation uses the relation
 ++   \spad{Y(v,x) = (J(v,x) cos(v*%pi) - J(-v,x))/sin(v*%pi)}
 ++ so is not valid for integer values of v.
besselY: (C, C) -> C
 ++ besselY(v,x) is the Bessel function of the second kind,
 ++ \spad{Y(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
 ++ Note that the default implementation uses the relation
 ++   \spad{Y(v,x) = (J(v,x) cos(v*%pi) - J(-v,x))/sin(v*%pi)}
```

```
 ++ so is not valid for integer values of v.

besselI: (R,R) -> R
 ++ besselI(v,x) is the modified Bessel function of the first kind,
 ++ \spad{I(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
besselI: (C,C) -> C
 ++ besselI(v,x) is the modified Bessel function of the first kind,
 ++ \spad{I(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.

besselK: (R, R) -> R
 ++ besselK(v,x) is the modified Bessel function of the second kind,
 ++ \spad{K(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
 ++ Note that the default implementation uses the relation
 ++   \spad{K(v,x) = %pi/2*(I(-v,x) - I(v,x))/sin(v*%pi)}.
 ++ so is not valid for integer values of v.
besselK: (C, C) -> C
 ++ besselK(v,x) is the modified Bessel function of the second kind,
 ++ \spad{K(v,x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
 ++ Note that the default implementation uses the relation
 ++   \spad{K(v,x) = %pi/2*(I(-v,x) - I(v,x))/sin(v*%pi)}
 ++ so is not valid for integer values of v.

airyAi:   C -> C
 ++ airyAi(x) is the Airy function \spad{Ai(x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{Ai''(x) - x * Ai(x) = 0}.
airyAi:   R -> R
 ++ airyAi(x) is the Airy function \spad{Ai(x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{Ai''(x) - x * Ai(x) = 0}.

airyBi:   R -> R
 ++ airyBi(x) is the Airy function \spad{Bi(x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{Bi''(x) - x * Bi(x) = 0}.
airyBi:   C -> C
 ++ airyBi(x) is the Airy function \spad{Bi(x)}.
 ++ This function satisfies the differential equation:
 ++   \spad{Bi''(x) - x * Bi(x) = 0}.

hypergeometric0F1: (R, R) -> R
 ++ hypergeometric0F1(c,z) is the hypergeometric function
```

```
        ++ \spad{0F1(; c; z)}.
       hypergeometric0F1: (C, C) -> C
        ++ hypergeometric0F1(c,z) is the hypergeometric function
        ++ \spad{0F1(; c; z)}.

       fresnelS : F -> F
         ++ fresnelS(f) denotes the Fresnel integral S
         ++
         ++X fresnelS(1.5)

       fresnelC : F -> F
         ++ fresnelC(f) denotes the Fresnel integral C
         ++
         ++X fresnelC(1.5)


   Impl ==> add
       a, v, w, z: C
       n, x, y: R

       Gamma z         == CGAMMA(z)$Lisp
       Gamma x         == RGAMMA(x)$Lisp
```

———-

5.9.2 The Exponential Integral

The E1 function

(Quoted from Segletes[**?**]):

A number of useful integrals exist for which no exact solutions have been found. In other cases, an exact solution, if found, may be impractical to utilize over the complete domain of the function because of precision limitations associated with what usually ends up as a series solution to the challenging integral. For many of these integrals, tabulated values may be published in various mathematical handbooks and articles. In some handbooks, fits (usually piecewise) also are offered. In some cases, an application may be forced to resort to numerical integration in order to acquire the integrated function. In this context, compact (*i.e.* not piecewise) analytical fits to some of these problematic integrals, accurate to within a small fraction of the numerically integrated value, serve as a useful tool to applications requiring the results of the integration, especially when the integration is required numerous times throughout the course of the application. Furthermore, the ability and methodology to develop intelligent fits, in contract to the more traditional "brute force" fits, provide the means to minimize parameters and maximize accuracy when tackling some of these difficult functions. The exponential integral will be used as an opportunity to both demonstrate a methodology for intelligent fitting as well as for providing an accurate, compact, analytical fit to the exponential integral.

The exponential integral is a useful class of functions that arise in a variety of applications [...]. The real branch of the family of exponential integrals may be defined as

$$E_n(x) = x^{n-1} \int_x^\infty \frac{e^{-t}}{t^n}\, dt \tag{5.1}$$

where n, a positive integer, denotes the specific member of the exponential integral family. The argument of the exponential integral, rather than expressing a lower limit of integration as in (1), may be thought of as describing the exponential decay constant, as given in this equivalent (and perhaps more popular) definition of the integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n}\, dt \tag{5.2}$$

Integration by parts permits any member of the exponential integral family to be converted to an adjacent member of the family, by way of

$$\int_x^\infty \frac{e^{-t}}{t^{n+1}}\, dt = \frac{1}{n}\left(\frac{e^{-x}}{x^n} - \int_x^\infty \frac{e^{-t}}{t^n}\, dt\right) \tag{5.3}$$

expressable in terms of E_n as

$$E_{n+1}(x) = \frac{1}{n}\left[e^{-x} - xE_n(x)\right] \quad (n = 1, 2, 3) \tag{5.4}$$

Through recursive employment of this equation, all members of the exponential integral family may be analytically related. However, this technique only allows for the transformation of one integral into another. There remains the problem of evaluating $E_1(x)$. There is an exact solution to the integral of (e^{-t}/t), appearing in a number of mathematical references [**?**, **?**] which is obtainable by expanding the exponential into a power series and integrating term by term. That exact solution, which is convergent, may be used to specify $E_1(x)$ as

$$E_1(x) = -\gamma - ln(x) + \frac{x}{1!} - \frac{x^2}{2 \cdot 2!} + \frac{x^3}{3 \cdot 3!} - \ldots \tag{5.5}$$

Euler's constant, γ, equal to $0.57721\ldots$, arises when the power series expansion for (e^{-t}/t) is integrated and evaluated at its upper limit, as $x \to \infty$[**?**].

Employing eqn (5), however, to evaluate $E_1(x)$ is problematic for finite x significantly larger than unity. One may well ask of the need to evaluate the exponential integral for large x, since the function to be integrated drops off so rapidly that the integral is surely a very flat function. Such reasoning is true when comparing the integrand at large x to that at small x. However, the definition of eqn (1) has as its upper limit not a small value of x, but rather that of ∞. Therefore, the actual values for $E_n(x)$ are extremely small numbers for large values of x. Thus, it is not sufficient merely to select enough terms of eqn (5) to evaluate the integral to within a value of, for example ± 0.0001 because the actual integral value for large x would be smaller than this arbitrary tolerance. To draw an analogy, it would be like saying that it is good enough to approximate e^{-x} as 0.0 for $x > 10$, since its actual value

is within 0.0001 of zero. For some applications, such an approximation may be warranted. In general, though, such an approximation is mathematically unacceptable. Worse yet, as seen from eqns (1) and (2), the need to evaluate the exponential integral for large arguments can arise in real-world problems from either a large integraion limit or a large value of an exponential decay constant. Thus, the need to evaluate exponential integrals for large values of the argument is established. It is here that the practical problems with the evaluation of eqn (5) become manifest.

First, the number of terms, N, required to achieve convergence rises rapidly with increasing x, making the summation an inefficient tool, even when expressed as a recursion relation (for three digits of accuracy, N is observed to vary roughly as $9 + 1.6x$, for $1 < x < 7$). More important, however, is the fact that, for calculations of finite precision, the accuracy of the complete summation will be governed by the individual term of greatest magnitude. The source of the problem is that as x is increased, the total summation decreases in magnitude more rapidly than a decaying exponential, while at the same time, the largest individual term in the series is observed to grow rapidly with increasing x ($\urcorner 10^1$ for $x = 7$, $\urcorner 10^2$ for $x = 10$, $\urcorner 10^3$ for $x = 13$, *etc.*). The magnitude of this largest individual term consumes the available precision and, as a result, leaves little or none left for the ever-diminishing net sum that constitutes the desired integral.

Literally, the use of eqn (5), even with (32-bit) double precision, does not permit the exponential integral to be evaluated to three places for $x > 14$ in any case, and with the situation worsening for lesser precision. For these reasons, the use of eqn (5) to evaluate the exponential integral numerically for large x is wholly unsuitable.

$$E_1(x) = e^{-x} \cdot \cfrac{1}{x + \cfrac{1}{1 + \cfrac{1}{x + \cfrac{2}{1 + \cfrac{2}{x + \ldots}}}}} \tag{5.6}$$

But as x becomes smaller, the number of terms required for convergence rises quickly. Similar arguments apply for the use of an asymptotic expansion for E_1, which also converges for large x. As such, the more typical approach employed by handbooks is that of a fit. While some steps are taken to make the fits intelligent (*e.g.*, transformation of variables), the fits are all piecewise over the domain of the integral.

Cody and Thatcher [**?**] performed what is perhaps the definitive work, with the use of Chebyshev[**?**, **?**] approximations to the exponential integral E_1. Like others, they fit the integral over a piecewise series of subdomains (three in their case) and provide the fitting parameters necessary to evaluate the function to various required precisions, down to relative errors of 10^-20. One of the problems with piecewise fitting over two or more subdomains is that functional value and derivatives of the spliced fits will not, in general, match at the domain transition point, unless special accomodations are made. This sort of discontinuity in functional value and/or slope, curvature, *etc.*, may cause difficulties for some numerical algorithms operating upon the fitted function. Numerical splicing/smoothing algorithms aimed at eliminating discontinuities in the value and/or derivatives of a piecewise fit are not,

in general, computationally insignificant. Problems associated with piecewise splicing of fits may also be obviated by obtaining an accurate enough fit, such that the error is on the order of magnitude of the limiting machine precision. This alternative, however, requires the use of additional fitting parameters to acquire the improved precision. Thus, regardless of approach, the desire to eliminate discontinuities in the function and its derivatives, between piecewise splices, requires extra computational effort. One final benefit to be had by avoiding the use of piecewise fits is the concomitant avoidance of conditional (*i.e.*, IF...THEN) programming statements in the coding of the routine. The use of conditional statements can preclude maximum computing efficiency on certain parallel computing architectures.

Segletes constructs an analytic, non-piecewise fit to the Exponential Integral but the precision is on the order of 4 decimal places and is not sufficient to compare against the Abramowitz and Stegun Handbook.

Instead we have chosen to use a two piece fitting function based on the Chebyshev polynomial for computing E_1. This agrees with the handbook values to almost the last published digit. See the `e1.input` pamphlet for regression testing against the handbook tables.

E1:R→OPR

The special function E1 below was originally derived from a function written by T.Haavie as the `expint.c` function in the Numlibc library by Lars Erik Lund. Haavie approximates the E1 function by two Chebyshev polynomials. For the range $-4 < x < 4$ the Chebyshev coefficients are:

```
 7.8737715392882774, -8.0314874286705335,  3.8797325768522250,
-1.6042971072992259,  0.5630905453891458, -0.1704423017433357,
 0.0452099390015415, -0.0106538986439085,  0.0022562638123478,
-0.0004335700473221,  0.0000762166811878, -0.0000123417443064,
 0.0000018519745698, -0.0000002588698662,  0.0000000338604319,
-0.0000000041611418,  0.0000000004821606, -0.0000000000528465,
 0.0000000000054945, -0.0000000000005433,  0.0000000000000512,
-0.0000000000000046,  0.0000000000000004
```

and for the range $x > 4$ the Chebyshev coefficients are:

```
 0.2155283776715125,  0.1028106215227030, -0.0045526707131788,
 0.0003571613122851, -0.0000379341616932,  0.0000049143944914,
-0.0000007355024922,  0.0000001230603606, -0.0000000225236907,
 0.0000000044412375, -0.0000000009328509,  0.0000000002069297,
-0.0000000000481502,  0.0000000000116891, -0.0000000000029474,
 0.0000000000007691, -0.0000000000002070,  0.0000000000000573,
-0.0000000000000163,  0.0000000000000047, -0.0000000000000014,
 0.0000000000000004, -0.0000000000000001
```

I've rewritten the polynomial to use precomputed coefficients that take into account the scaling used by Haavie. I've also rewritten the polynomial using Horner's method so the large powers of x are only computed once.

The result can be either a double float or, or if the argument is zero, infinity. Thus we need to extend the result to be a one-point completion to include infinity.

— package DFSFUN DoubleFloatSpecialFunctions —

```
E1(x:R):OPR ==
 x = 0.0::R => infinity()
 x > 4.0::R =>
  t1:R:=0.14999948967737774608E-15::R
  t2:R:=0.9999999999993112::R
  ta:R:=(t1*x+t2)
  t3:R:=0.99999999953685760001::R
  tb:R:=(ta*x-t3)
  t4:R:=1.9999998808293376::R
  tc:R:=(tb*x+t4)
  t5:R:=5.999983407661056::R
  td:R:=(tc*x-t5)
  t6:R:=23.9985380938481664::R
  te:R:=(td*x+t6)
  t7:R:=119.9108830382784512::R
  tf:R:=(te*x-t7)
  t8:R:=716.01351020920176641::R
  tg:R:=(tf*x+t8)
  t9:R:=4903.3466623370985473::R
  th:R:=(tg*x-t9)
  t10:R:=36601.25841454446674::R
  ti:R:=(th*x+t10)
  t11:R:=279913.28608482691646::R
  tj:R:=(ti*x-t11)
  t12:R:=2060518.7020296525186::R
  tk:R:=(tj*x+t12)
  t13:R:=13859772.093039815059::R
  tl:R:=(tk*x-t13)
  t14:R:=81945572.630072918857::R
  tm:R:=(tl*x+t14)
  t15:R:=413965714.82128317479::R
  tn:R:=(tm*x-t15)
  t16:R:=1747209536.2595547568::R
  to:R:=(tn*x+t16)
  t17:R:=6036182333.96179427::R
  tp:R:=(to*x-t17)
  t18:R:=16693683576.106267572::R
  tq:R:=(tp*x+t18)
  t19:R:=35938625644.58286097::R
  tr:R:=(tq*x-t19)
  t20:R:=57888657293.609258888::R
  ts:R:=(tr*x+t20)
  t21:R:=65523779423.11290127::R
  tt:R:=(ts*x-t21)
```

```
 t22:R:=46422751473.201760309::R
 tu:R:=(tt*x+t22)
 t23:R:=15474250491.067253436::R
 tv:R:=(tu*x-t23)
 tw:R:=(-1.0::R*x)
 tx:R:=exp(tw)
 ty:R:=tv*tx
 tz:R:=x**22
 taz:R:=ty/tz
 taz::OPR
x > -4.0::R =>
 a1:R:=0.476837158203125E-22::R
 a2:R:=0.10967254638671875E-20::R
 aa:R:=(-a1*x+a2)
 a3:R:=0.2021789550781250001E-19::R
 ab:R:=(aa*x-a3)
 a4:R:=0.4260063171386718750 1E-18::R
 ac:R:=(ab*x+a4)
 a5:R:=0.868625640869140625E-17::R
 ad:R:=(ac*x-a5)
 a6:R:=0.16553192138671875E-15::R
 ae:R:=(ad*x+a6)
 a7:R:=0.29870208740234375E-14::R
 af:R:=(ae*x-a7)
 a8:R:=0.5097890777587890625E-13::R
 ag:R:=(af*x+a8)
 a9:R:=0.8193406921386718749 9E-12::R
 ah:R:=(ag*x-a9)
 a10:R:=0.1235313123779296875E-10::R
 ai:R:=(ah*x+a10)
 a11:R:=0.1739729620849609375E-9::R
 aj:R:=(ai*x-a11)
 a12:R:=0.22774642697021484375E-8::R
 ak:R:=(aj*x+a12)
 a13:R:=0.275573192853515625E-7::R
 al:R:=(ak*x-a13)
 a14:R:=0.3061924363508789062 5E-6::R
 am:R:=(al*x+a14)
 a15:R:=0.0000031001984125191406 25::R
 an:R:=(am*x-a15)
 a16:R:=0.0000283446712004554687 5::R
 ao:R:=(an*x+a16)
 a17:R:=0.0002314814814817695312 5::R
 ap:R:=(ao*x-a17)
 a18:R:=0.0016666666666686609375::R
 aq:R:=(ap*x+a18)
 a19:R:=0.01041666666666646875::R
 ar:R:=(aq*x-a19)
 a20:R:=0.055555555555554168751::R
 as:R:=(ar*x+a20)
```

```
 a21:R:=0.2500000000000000375::R
 at:R:=(as*x-a21)
 a22:R:=1.000000000000000325::R
 au:R:=(at*x+a22)
 a23:R:=0.5772156649015328::R
 av:R:=au*x-a23
 aw:R:=- 1.0::R*log(abs(x)) + av
 aw::OPR
error "E1: no approximation available"
```

———-

5.9.3 En:(PI,R)→OPR

The E_n function is computed using the recurrence relation:

$$E_{n+1}(z) = \frac{1}{n}\left(e^{-z} - zE_n(z)\right) \quad (n = 1, 2, 3, \ldots)$$

The base case of the recursion depends on E1 above.

The formula is 5.1.14 in Abramowitz and Stegun, 1965, p229[**?**].

— **package DFSFUN DoubleFloatSpecialFunctions** —

```
En(n:PI,x:R):OPR ==
  n=1 => E1(x)
  v:R:=retract(En((n-1)::PI,x))
  w:R:=1/(n-1)*(exp(-x)-x*v)
  w::OPR
```

———-

5.9.4 The Ei Function

This function is based on Kin L. Lee's work[**?**]. See also [**?**].

Abstract

The exponential integral Ei(x) is evaluated via Chebyshev series expansion of its associated functions to achieve high relative accuracy throughout the entire real line. The Chebyshev coefficients for these functions are given to 30 significant digits. Clenshaw's[**?**] method is modified to furnish an efficient procedure for the accurate solution of linear systems having near-triangular coefficient matrices.

Introduction

The evaulation of the exponential integral

$$Ei(x) = \int_{-\infty}^{X} \frac{e^u}{u}\, du = -E_1(-x), x \neq 0 \tag{5.7}$$

is usually based on the value of its associated functions, for example, $xe^{-x}Ei(x)$. High accuracy tabulations of integral (1) by means of Taylor series techniques are given by Harris [**?**] and Miller and Hurst [**?**]. The evaluation of $Ei(x)$ for $-4 \leq x \leq \infty$ by means of Chebyshev series is provided by Clenshaw [**?**] to have the absolute accuracy of 20 decimal places. The evaluation of the same integral (1) by rational approximation of its associated functions is furnished by Cody and Thacher [**?**, **?**] for $-\infty < x < \infty$, and has the relative accuracy of 17 significant figures.

The approximation of Cody and Thacher from the point of view of efficient function evaluation are preferable to those of Clenshaw. However, the accuracy of the latter's procedure, unlike those of the former, is not limited by the accuracy or the availability of a master function, which is a means of explicitly evaluating the function in question.

In this paper $Ei(x)$ (or equivalently $-E_1(-x)$) for the entire real line is evaluted via Chebyshev series expansion of its associated functions that are accurate to 30 significant figures by a modification of Clenshaw's procedure. To verify the accuracy of the several Chebyshev series, values of the associated functions were checked against those computed by Taylor series and those of Murnaghan and Wrench [**?**] (see Remarks on Convergence and Accuracy).

Although for most purposes fewer than 30 figures of accuracy are required, such high accuracy is desirable for the following reasons. In order to further reduce the number of arithmetical operations in the evaluation of a function, the Chebyshev series in question can either be converted into a rational function or rearranged into an ordinary polynomial. Since several figures may be lost in either of these procedures, it is necessary to provide the Chebyshev series with a sufficient number of figures to achieve the desired accuracy. Furthermore, general function approximation routines, such as those used for minimax rational function approximations, require the explicit evaluation of the function to be approximated. To take account of the errors commited by these routines, the function values must have an accuracy higher than the approximation to be determined. Consequently, high-precision results are useful as a master function for finding approximations for (or involving) $Ei(x)$ (e.g. [**?**, **?**]) where prescribed accuracy is less than 30 figures.

Discussion

It is proposed here to provide for the evaluation of $Ei(x)$ by obtaining Chebyshev coefficients for the associated functions given by table 1.

Table 1: Associated Functions of $Ei(x)$ and their ranges of Chebyshev Series Expansions

	Associated function	Range of expansion
Ei1	$xe^{-x}Ei(x)$	$-\infty < x \leq -10$
Ei2	$xe^{-x}Ei(x)$	$-10 \leq x \leq -4$
Ei3	$\frac{Ei(x)-log\|x\|-\gamma}{x}$	$-4 \leq x \leq d42$
Ei4	$xe^{-x}Ei(x)$	$4 \leq x \leq 12$
Ei5	$xe^{-x}Ei(x)$	$12 \leq x \leq 32$
Ei6	$xe^{-x}Ei(x)$	$32 \leq x < \infty$

($\gamma = 0.5772156649...$ is Euler's constant.)

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei(y:OPR):OPR ==
  infinite? y => 1
  x:R:=retract(y)
  x < -10.0::R =>
    ei:R:=retract(Ei1(y))
    (ei/(x*exp(-x)))::OPR
  x <  -4.0::R =>
    ei:R:=retract(Ei2(y))
    (ei/(x*exp(-x)))::OPR
  x <   4.0::R =>
    ei3:R:=retract(Ei3(y))
    gamma:R:=0.57721566490153286060651209008 2::R
    (ei3*x+log(abs(x))+gamma)::OPR
  x <  12.0::R =>
    ei:R:=retract(Ei4(y))
    (ei/(x*exp(-x)))::OPR
  x <  32.0::R =>
    ei:R:=retract(Ei5(y))
    (ei/(x*exp(-x)))::OPR
  ei:R:=retract(Ei6(y))
  (ei/(x*exp(-x)))::OPR
```

———-

Note that the functions $[Ei(x)-log|x|-\gamma]/x$ and $xe^{-x}Ei(x)$ have the limiting values of unity at the origin and at infinity, respectively, and that the range of the associated function values is close to unity (see table 4). This makes for the evaluation of the associated functions over the indicated ranges in table 1 (and thus $Ei(x)$ over the entire real line) with high relative accuracy by means of the Chebyshev series. The reason for this will become apparent later.

Some remarks about the choice of the intervals of expansion for the serveral Chebyshev series are in order here. The partition of the real line indicated by table 1 is chosen to allow for the approximation of the associated functions with a maximum error of 0.5×10^{-30} by polynomials of degress < 50. The real line has also been partitioned with the objective of providing the interval about zero with the lowest degree of polynomial approximation of the six intervals. This should compensate for the computation of $log|x|$ required in the evaluation of $Ei(x)$ over that interval. The ranges $-\infty < x \leq -4$ and $4 \leq x < \infty$ are partitioned into

2 and 3 intervals, respectively, to provide approximations to $xe^{-x}Ei(x)$ by polynomials of about the same degree.

Expansions in Chebyshev Series

Let $\phi(t)$ be a differentiable function defined on [-1,1]. To facilitate discussion, denote its Chebyshev series and that of its derivative by

$$\phi(t) = \sum_{k=0}^{\infty}{}' A_k^{(0)} T_k(t) \quad \phi'(t) = \sum_{k=0}^{\infty}{}' A_k^{(1)} T_k(t) \tag{5.8}$$

where $T_k(t)$ are Chebyshev polynomials defined by

$$T_k(t) = cos(k \ \arccos \ t), \quad -1 \leq t \leq 1 \tag{5.9}$$

(A prime over a summation sign indicates that the first term is to be halved.)

If $\phi(t)$ and $\phi'(t)$ are continuous, the Chebyshev coefficients $A_k^{(0)}$ and $A_k^{(1)}$ can be obtained analytically (if possible) or by numerical quadrature. However, since each function in table 1 satisfies a linear differential equation with polynomial coefficients, the Chebyshev coefficients can be more readily evaluated by the method of Clenshaw [**?**].

There are several variations of Clenshaw's procedure (see, e.g. [**?**]), but for high-precision computation, where multiple precision arithmetic is employed, we find his original procedure easiest to implement. However, straightforward application of it may result in a loss of accuracy if the trial solutions selected are not sufficiently independent. How the difficulty is overcome will be pointed out subsequently.

The function $xe^{-x}Ei(x)$ on the Finite Interval

We consider first the Chebyshev series expansion of

$$f(x) = xe^{-x}Ei(x), \quad (a \leq x \leq b) \tag{5.10}$$

with $x \neq 0$. One can easily verify that after the change of variables

$$x = [(b-a)T + a + b]/2, \quad (-1 \leq t \leq 1) \tag{5.11}$$

the function

$$\phi(t) = f\left[\frac{(b-a)t + a + b}{2}\right] = f(x) \tag{5.12}$$

satisfies the differential equation

$$2(pt+q)\phi'(t) + p(pt+q-2)\phi(t) = p(pt+q) \tag{5.13}$$

with[1]

$$\phi(-1) = ae^{-a}Ei(a) \tag{5.14}$$

[1]The value of $Ei(a)$ may be evaluated by means of the Taylor series. In this report $Ei(a)$ is computed by first finding the Chebyshev series approximation to $[Ei(x) - log|x| - \gamma]/x$ to get $Ei(a)$. The quantities e^a and $\log|a|$ for integral values of a may be found in existing tables

where $p = b - a$ and $q = b + a$. Replacing $\phi(t)$ and $\phi^{'}(t)$ in equations 7 by their Chebyshev series, we obtain

$$\sum_{k=0}^{\infty}{}' (-1)^k A_k^{(0)} = \phi(-1) \tag{5.15}$$

$$2\sum_{k=0}^{\infty}{}' A_k^{(1)}(pt+q)T_k(t) + p\sum_{k=0}^{\infty}{}' A_k^{(0)}(pt+q-2)T_k(t) = p(pt+q) \tag{5.16}$$

It can be demonstrated that if B_k are the Chebyshev coefficients of a function $\Psi(t)$, then C_k, the Chebyshev coefficients of $t^r\Psi(t)$ for positive integers r, are given by [**?**]

$$C_k = 2^{-r}\sum_{i=0}^{r}\left(\begin{array}{c} r \\ i \end{array}\right) B_{|k-r+2i|} \tag{5.17}$$

Consequently, the left member of equation 15 can be rearranged into a single series involving $T_k(t)$. The comparison of the coefficients of $T_k(t)$ that yields the infinite system of equations

$$\left.\begin{array}{c} \displaystyle\sum_{k=0}^{\infty}{}' (-1)^k A_k^{(0)} = \phi(-1) \\ 2pA_{|k-1|}^{(1)} + 4qA_k^{(1)} + 2pA_{k+1}^{(1)} + p^2A_{|k-1|}^{(0)} + 2p(q-2)A_k^{(0)} + p^2A_{k+1}^{(0)} \\ = \left\{\begin{array}{rcl} 4pq & , & k=0 \\ 2p^2 & , & k=1 \\ 0 & , & k=2,3,\ldots \end{array}\right. \end{array}\right\} \tag{5.18}$$

The relation [**?**]

$$2kA_k^{(0)} = A_{k-1}^{(1)} - A_{k+1}^{(1)} \tag{5.19}$$

can be used to reduce equation 18 to a system of equations involving only $A_k^{(0)}$. Thus, replacing k of equations 18 by $k+2$ and subtracting the resulting equation from equations 18, we have, by means of equation 19, the system of equations

$$\left.\begin{array}{c} \displaystyle\sum_{k=0}^{\infty}{}' (-1)^k A_k^{(0)} = \phi(-1) \\ 2p(q-2)A_0 + (8q+p^2)A_1 + 2p(6-q)A_2 - p^2A_3 = 4pq \\ p^2A_{k-1} + 2p(2k+q-2)A_k + 8q(k+1)A_{k+1} + 2p(2k-q+6)A_{k+2} - p^2A_{k+3} \\ = \left\{\begin{array}{rcl} 2p^2 & , & k=1 \\ 0 & , & k=2,3,\ldots \end{array}\right. \end{array}\right\} \tag{5.20}$$

The superscript of $A_k^{(0)}$ is dropped for simplicity. In order to solve the infinite system 20, Clenshaw [**?**] essentially considered the required solution as the limiting solution of the

sequence of truncated systems consisting of the first $M+1$ equations of the same system, that is, the solution of the system

$$\sum_{k=0}^{M}{}'(-1)^k A_k = \phi(-1) \tag{5.21}$$

$$2p(q-2)A_0 + (8q+p^2)A_1 + 2p(q-6)A_2 - p^2 A_3 = 4pq \tag{5.22}$$

$$\left.\begin{array}{c} p^2 A_{k-1} + 2p(2k+q-2)A_k + 8q(k+1)A_{k+1} + 2p(2k-q+6)A_{k+2} - p^2 A_{k+3} \\ = \left\{\begin{array}{rcl} 2p^2 & , & k=1 \\ 0 & , & k=2,3,\ldots,M-3 \end{array}\right. \\ p^2 A_{M-3} + 2p(2M+q-6)A_{M-2} + 8q(M-1)A_{M-1} + 2p(2M+4-q)A_M = 0 \\ p^2 A_{M-2} + 2p(2M+q-4)A_{M-1} + 8qMA_M = 0 \end{array}\right\} \tag{5.23}$$

where A_k is assumed to vanish for $K \geq M+1$. To solve system (21,22,23) consider first the subsystem 23 consisting of $M-2$ equations in M unknowns. Here use is made of the fact that the subsystem 23 is satisfied by

$$A_k = c_1\alpha_k + c_2\beta_k + \gamma_k \quad (k=0,1,2,\ldots) \tag{5.24}$$

for arbitrary constants c_1 and c_2, where γ_k is a particular solution of 23 and where α_k and β_k are two independent solutions of the homogeneous equations (23 with $2p^2$ deleted) of the same subsystem. Hence, if α_k, β_k, and γ_k are available, the solution of system (21,22,23) reduces to the determinant of c_1 and c_2 from equations 21 and 22.

To solve equations (21,22,23), we note that

$$\gamma_0 = 2, \quad \gamma_k = 0, \quad for\ k = 1(1)M \tag{5.25}$$

is obviously a particular solution of equation 23. The two independent solutions γ_k and β_k of the homogeneous equations of the same subsystem can be generated in turn by backward recurrence if we set

$$\left.\begin{array}{lll} & \alpha_{M-1} = 0, \quad \alpha_M = 1 \\ \text{and} & \\ & \beta_{M-1} = 1, \quad \beta_M = 0 \end{array}\right\} \tag{5.26}$$

or choose any $\alpha M-1$, α_M, and β_{M-1}, β_M for which $\alpha_{M-1}\beta_M - \alpha_M\beta_{M-1} \neq 0$. The arbitrary constants c_1 and c_2 are determined, and consequently the solution of equations (21,22,23) if equation 24 is substituted into equation 21 and 22 and the resulting equations

$$c_1 R(\alpha) + c_2 R(\beta) = \phi(-1) - 1 \tag{5.27}$$

$$c_1 S(\alpha) + c_2 S(\beta) = 8p \tag{5.28}$$

are solved as two equations in two unknowns. The terms $R(\alpha)$ and $S(\alpha)$ are equal, respectively, to the left members of equations 21 and 22 corresponding to solution α_k. (The identical designation holds for $R(\beta)$ and $S(\beta)$.)

The quantities α_k and β_k are known as trial solutions in reference [**?**]. Clenshaw has pointed out that if α_k and β_k are not sufficiently independent, loss of significance will occur in the formation of the linear combination 24, with consequent loss of accuracy. Clenshaw suggested the Gauss-Seidel iteration procedure to improve the accuracy of the solution. However, this requires the application of an additional computing procedure and may prove to be extremely slow. A simpler procedure which does not alter the basic computing scheme given above is proposed here. The loss of accuracy can effectively be regained if we first generate a third trial solution δ_k (k=0,1,...,M), where δ_{M-1} and δ_M are equal to $c_1\alpha_{M-1} + c_2\beta_{M-1}$ and $c_1\alpha_M + c_2\beta_M$, respectively, and where δ_k (k=M-2,M-3,...,0) is determined using backward recurrence as before by means of equation 23. Then either α_k or β_k is replaced by δ_k and a new set of c_1 and c_2 is determined by equations 27 and 28. Such a procedure can be repeated until the required accuracy is reached. However, only one application of it was necessary in the computation of the coefficients of this report.

As an example, consider the case for $4 \leq x \leq 12$ with $M = 15$. The right member of equation 27 and of equation 28 assume, respectively, the values of 0.43820800 and 64. The trial solutions α_k and β_k generated with $\alpha_{14} = 8$, $\alpha_{15} = 9$ and $\beta_{14} = 7$, $\beta_{15} = 8$ are certainly independent, since $\alpha_{14}\beta_{15} - \alpha_{15}\beta_{14} = 1 \neq 0$. A check of table 2 shows that equations 27 and 28 have, respectively, the residuals of -0.137×10^-4 and -0.976×10^{-3}. The same table also shows that $c_1\alpha_k$ is opposite in sign but nearly equal in magnitude to $c_2\beta_k$. Cancellations in the formation of the linear combination 24 causes a loss of significance of 2 to 6 figures in the computed A_k. In the second iteration, where a new set of β_k is generated replacing β_{14} and β_{15}, respectively, by $c_1\alpha_{14} + c_2\beta_{14}$ and $c_1\alpha_{15} + c_2\beta_{15}$ of the first iteration, the new $c_1\alpha_k$ and $c_2\beta_k$ differed from 2 to 5 orders of magnitude. Consequently, no cancellation of significant figures in the computation of A_k occurred. Notice that equations 27 and 28 are now satisfied exactly. Further note that the new c_1 and c_2 are near zero and unity, respectively, for the reason that if equations 21, 22, and 23 are satisfied by equation 24 exactly in the first iteration, the new c_1 and c_2 should have the precise values zero and 1, respectively. The results of the third iteration show that the A_k of the second iteration are already accurate to eight decimal places, since the A_k in the two iterations differ in less that 0.5×10^{-8}. Notice that for the third iteration, equations 27 and 28 are also satisfied exactly and that $c_1 = 1$ and $c_2 = 0$ (relative to 8 places of accuracy).

Table 2: Computation of Chebyshev Coefficients for $xe^{-x}Ei(x)$

First iteration: $\alpha_{14} = 8$, $\alpha_{15} = 9$; $\beta_{14} = 7$, $\beta_{15} = 8$

k	$c_1\alpha_k$	$c_2\beta_k$	A_k
0	0.71690285E 03	-0.71644773E 03	0.24551200E 01
1	-0.33302683E 03	0.33286440E 03	-0.16243000E 00
2	0.13469341E 03	-0.13464845E 03	0.44960000E-01
3	-0.43211869E 02	0.43205127E 02	-0.67420000E-02
4	0.99929173E 01	-0.99942238E 01	-0.13065000E-02
5	-0.11670764E 01	0.11684574E 01	0.13810000E-02
6	-0.25552137E 00	0.25493635E 00	-0.58502000E-02
7	0.20617247E 00	-0.20599754E 00	0.17493000E-03
8	-0.75797238E-01	0.75756767E-01	-0.40471000E-04
9	0.20550680E-01	-0.20543463E-01	0.72170000E-05
10	-0.45192333E-02	0.45183721E-02	-0.86120000E-06
11	0.82656562E-03	-0.82656589E-03	-0.27000000E-09
12	-0.12333571E-03	0.12337366E-03	0.37950000E-07
13	0.13300910E-04	-0.13315328E-04	-0.14418000E-07
14	-0.29699001E-06	0.30091136E-06	0.39213500E-08
15	-0.33941716E-06	0.33852528E-06	-0.89188000E-09

$$c_1 = 0.37613920E - 07$$
$$c_2 = -0.42427144E - 07$$
$$c_1R(\alpha) + c_2R(\beta) - 0.43820800E\ 00 = -0.13700000E - 04$$
$$c_1S(\alpha) + c_2S(\beta) - 0.64000000E\ 00 = -0.97600000E - 03$$

Second iteration: $\alpha_{14} = 8$, $\alpha_{15} = 9$;
$\beta_{14} = 0.39213500E - 08$, $\beta_{15} = -0.89188000E - 09$

k	$c_1\alpha_k$	$c_2\beta_k$	A_k
0	0.36701576E-05	0.45512986E 00	0.24551335E 01
1	-0.17051695E-05	-0.16243666E 00	-0.16243837E 00
2	0.68976566E-06	0.44956834E-01	0.44957523E-01
3	-0.22132756E-06	-0.67413538E-02	-0.67415751E-02
4	0.51197561E-07	-0.13067496E-02	-0.13066984E-02
5	-0.59856744E-08	0.13810895E-02	0.13810835E-02
6	-0.13059663E-08	-0.58502164E-03	-0.58502294E-03
7	0.10552667E-08	0.17492889E-03	0.17492994E-03
8	-0.38808033E-09	-0.40472426E-04	-0.40472814E-04
9	0.10523831E-09	0.72169965E-05	0.72171017E-05
10	-0.23146333E-10	-0.86125438E-06	-0.86127752E-06
11	0.42342615E-11	-0.25542252E-09	-0.25118825E-09
12	-0.63200810E-12	0.37946968E-07	0.37946336E-07
13	0.68210630E-13	-0.14417584E-07	-0.14417516E-07
14	-0.15414832E-14	0.39212981E-08	0.39212965E-08
15	-0.17341686E-14	-0.89186818E-09	-0.89186991E-09

$$c_1 = -0.19268540E - 15$$
$$c_2 = 0.99998675E\ 00$$
$$c_1R(\alpha) + c_2R(\beta) - 0.43820800E\ 00 = 0.0$$

$c_1S(\alpha) + c_2S(\beta) - 0.64000000E\ 00 = 0.0$

Table 2: Computation of Chebyshev Coefficients for $xe^{-x}Ei(x)$ - Concluded
$[4 \leq x \leq 12 with M = 15; \gamma_0 = 2, \gamma_k = 0 \text{for } k = 1(1)15]$

Third iteration: $\alpha_{14} = 8$, $\alpha_{15} = 9$;
$\beta_{14} = 0.39212965E - 08$, $\beta_{15} = -0.89186991E - 09$

k	$c_1\alpha_k$	$c_2\beta_k$	A_k
0	-0.23083059E-07	0.45513355E 00	0.24551335E 01
1	0.10724479E-07	-0.16243838E 00	-0.16243837E 00
2	-0.43382065E-08	0.44957526E-01	0.44957522E-01
3	0.13920157E-08	-0.67415759E-02	-0.67415745E-02
4	-0.32200152E-09	-0.13066983E-02	-0.13066986E-02
5	0.37646251E-10	0.13810835E-02	0.13810836E-02
6	0.82137336E-11	-0.58502297E-03	-0.58502296E-03
7	-0.66369857E-11	0.17492995E-03	0.17492994E-03
8	0.24407892E-11	-0.40472817E-04	-0.40472814E-04
9	-0.66188494E-12	0.72171023E-05	0.72171017E-05
10	0.14557636E-12	-0.86127766E-06	-0.86127751E-06
11	-0.26630930E-13	-0.25116620E-09	-0.25119283E-09
12	0.39749465E-14	0.37946334E-07	0.37946337E-07
13	-0.42900337E-15	-0.14417516E-07	-0.14417516E-07
14	0.96949915E-17	0.39212966E-08	0.39212966E-08
15	0.10906865E-16	-0.89186992E-09	-0.89186990E-09

$c_1 = 0.12118739E - 17$
$c_2 = 0.10000000E\ 01$
$c_1R(\alpha) + c_2R(\beta) - 0.43820800E\ 00 = 0.0$
$c_1S(\alpha) + c_2S(\beta) - 0.64000000E\ 00 = 0.0$

It is worth noting that the coefficient matrix of system (21,22,23) yields an upper triangular matrix of order $M - 1$ after the deletion of the first two rows and the last two columns. Consequently, the procedure of this section is applicable to any linear system having this property. As a matter of fact, the same procedure can be generalized to solve linear systems having coefficient matrices of order N, the deletion of whose first r $(r < N)$ rows and last r columns yields upper triangular matrices of order $N - r$.

The Function $(1/x)[Ei(x) - log|x| - \gamma]$

Let

$$f(x) = (1/x)[Ei(x) - log|x| - \gamma], \quad g(x) = e^x, \quad |x| \leq b \tag{5.29}$$

These functions, with the change of variable $x = bt$, simultaneously satisfy the differential equations

$$bt^2\phi^{'}(t) + bt\phi(t) - \psi(t) = -1 \tag{5.30}$$

$$\psi^{'}(t) - b\psi(t) = 0, \quad -1 \leq t \leq 1 \tag{5.31}$$

Conversely,[2] any solution of equations 30 and 31 is equal to the functions given by equations 29 for the change of variable $x = bt$. Therefore, boundary conditions need not be imposed for the solution of the differential equations.

A procedure similar to that of the previous section gives the coupled infinite recurrence relations

$$bA_1 + bA_3 - B_0 + B_2 = -2 \tag{5.32}$$

$$\left.\begin{aligned} kbA_{k-1} + 2(k+1)bA_{k+1} + (k+2)bA_{k+3} - 2B_k + 2B_{k+2} = 0 \\ bB_{k-1} - 2kB_k - bB_{k+1} = 0, \quad k = 1, 2, \ldots \end{aligned}\right\} \tag{5.33}$$

where A_k and B_k are the Chebyshev coefficients of $\phi(t)$ and $\psi(t)$, respectively.

Consider first the subsystem 33. If $A_k = \alpha_k$ and $B_k = \beta_k$ are a simultaneous solution of the system, which is homogeneous, then

$$\text{and} \qquad \left.\begin{aligned} A_k = c\alpha_k \\ B_k = c\beta_k \end{aligned}\right\} \tag{5.34}$$

are also a solution for an arbitrary constant c. Thus based on considerations analogous to the solution of equations 21, 22, and 23, one can initiate an approximate solution of equations 32 and 33 by setting

$$\left.\begin{aligned} \alpha_M = 0, \quad \alpha_k = 0 \quad \text{for } k \geq M+1 \\ \beta_M = 1, \quad \beta_k = 0 \quad \text{for } k \geq M+1 \end{aligned}\right\} \tag{5.35}$$

and then determining α_k and β_k ($k = M-1, M-2, \ldots, 0$) by backward recurrence by means of equation 33. The arbitrary constant c is determined by substituting 34 into 32.

The Function $xe^{-x}Ei(x)$ on the Infinite Interval

Let

$$f(x) = xe^{-x}Ei(x), \quad -\infty < x \leq b < 0, \quad or 0 < b \leq x < \infty \tag{5.36}$$

By making the change of variables,

$$x = 2b/(t+1) \tag{5.37}$$

we can easily demonstrate that

$$f(x) = f[2b/(t+1)] = \phi(t) \tag{5.38}$$

[2]The general solution of the differential equations has the form

$$\phi(t) = (c_1/t) + [Ei(bt) - log|bt| - \gamma]/bt$$

$$\psi(t) = c_2 e^{bt}$$

where the first and second terms of $\phi(t)$ are, respectively, the complementary solution and a particular integral of equation 30. The requirement that $\phi(t)$ is bounded makes the constant $c_1 = 0$. The fact that $\psi(0) = 1$ is implicit in equation 30.

satisfies the differential equation

$$(t+1)^2\phi'(t) + (t+1-2b)\phi(t) = -2b \tag{5.39}$$

with

$$\phi(1) = be^{-b}Ei(b) \tag{5.40}$$

An infinite system of equations involving the Chebyshev coefficients A_k of $\phi(t)$ is deducible from equations 39 and 40 by the same procedure as applied to equations 13 and 14 to obtain the infinite system 20; it is given as follows.

$$\sum_{k=0}^{\infty}{}' A_k = \phi(1) = be^{-b}Ei(b) \tag{5.41}$$

$$(1-2b)A_0 + 3A_1 + (3+2b)A_2 + A_3 = -4b \tag{5.42}$$

$$\begin{aligned} kA_{k-1} + 2[(2k+1)-2b]A_k + 6(k+1)A_{k+1} + 2(2k+3+2b)A_{k+2} \\ + (k+2)A_{k+3} = 0, \quad k = 1, 2, \ldots \end{aligned} \tag{5.43}$$

As in the case of equations 21, 22 and 23, the solution of 41, 42 and 43 can be assumed to be

$$A_k = c_1\alpha_k + c_2\beta_k \tag{5.44}$$

with A_k vanishing for a $k \geq M$. Thus, we can set, say

$$\left.\begin{array}{lll} \alpha_{M-1} = 0 & , & \alpha_M = 1 \\ \beta_{M-1} = 1 & , & \beta_M = 0 \end{array}\right\} \tag{5.45}$$

and determine the trial solutions α_k and β_k (k=M-1,M-2,...,0) by means of equation 43 by backward recurrence. The required solution of equations 41,42,adn 43 is then determined by substituting equation 44 in equations 41 and 42 and solving the resulting equations for c_1 and c_2.

Loss of accuracy in the computation of A_k can also occur here, as in the solution of equations 21, 22 and 23, if the trial solutions are not sufficiently independent. The process used to improve the accuracy of A_k of the system 21, 22 and 23 can also be applied here.

For efficiency in computation, it is worth noting that for $b < 0$ ($-\infty < x \leq b < 0$) the boundary condition 40 is not required for the solution of equation 39 and 40. This follows from the fact that any solution[3] of the differential equation 39 is equal to $xe^{-x}Ei(x)$ ($x = 2b/(t+1)$). Hence the A_k of $xe^{-x}Ei(x)$ for $-\infty < x \leq b < 0$ can be obtained without the use of equation 39 and can be assumed to have the form

$$A_k = c\alpha_k, \quad (k = 0, 1, \ldots, M) \tag{5.46}$$

The M+1 values of α_k can be generated by setting $\alpha_M = 1$ and computing α_k (k=0,1,...,M-1) by means of equation 43 by backward recurrence. The substitution of equation 46 into 42 then enables one to determine c from the resulting equation.

[3]The general solution of the differential equation 39. Since equation 39 has no bounded complementary solution for $-\infty < x \leq b < 0$, every solution of it is equal to the particular integral $xe^{-x}Ei(x)$. On the other hand, a solution of equation 39 for $0 < x \leq b < \infty$ would, in general, involve the complementary function. Hence, boundary condition 40 is required to guarantee that the solution of equation 39 is equal to $xe^{-x}Ei(x)$.

Remarks on Convergence and Accuracy

The Chebyshev coefficients of table 3 were computed on the IBM 7094 with 50-digit normalized floating-point arithmetic. In order to assure that the sequence of approximate solutions (see Discusion) converged to the limiting solution of the differential equation in question, a trial M was incremented by 4 until the approximate Chebyshev coefficients showed no change greter than or equal to 0.5×10^{-35}. Hence the maximum error is bounded by

$$0.5(M+1) \times 10^{-35} + \sum_{M+1}^{\infty} |A_k| \tag{5.47}$$

where the first term is the maximum error of the M+1 approximate Chebyshev coefficients, and the sum is the maximum error of the truncated Chebyshev series of M+1 terms. If the Chebyshev series is rapidly convergent, the maximum error of the approximate Chebyshev series should be of the order of 10^{-30}. The coefficients of table 3 have been rounded to 30 digits, and higher terms for $k > N$ giving the maximum residual

$$\sum_{k=N+1}^{M} |A_k| < 0.5 \times 10^{-30} \tag{5.48}$$

have been dropped. This should allow for evaluation of the relevant function that is accurate to 30 decimal places. Since the range of values of each function is bounded between 2/5 and 5, the evaluated function should be good to 30 significant digits. Taylor series evaluation also checks with that of the function values of table 4 (computed with 30-digit floating-point arithmetic using the coefficients of table 3) for at least 28-1/2 significant digits. Evaluation of Ei(x) using the coefficients of table 3 also checked with Murnaghan and Wrench [**?**] for 28-1/2 significant figures.

Table 3: Chebyshev Coefficients (a)

$$xe^{-x}Ei(x) = \sum_{k-0}^{40}{}' A_k T_k(t), \quad t = (-20/x) - 1, \quad (-\infty < x \leq -10)$$

k	A_k
0	0.1912173225 8605534539 1519326510E 01
1	-0.4208355052 8684843755 0974986680E-01
2	0.1722819627 2843267833 7118157835E-02
3	-0.9915782173 4445636455 9842322973E-04
4	0.7176093168 0227750526 5590665592E-05
5	-0.6152733145 0951269682 7956791331E-06
6	0.6024857106 5627583129 3999701610E-07
7	-0.6573848845 2883048229 5894189637E-08
8	0.7853167541 8323998199 4810079871E-09
9	-0.1013730288 0038789855 4202774257E-09
10	0.1399770413 2267686027 7823488623E-10
11	-0.2051008376 7838189961 8962318711E-11
12	0.3168388726 0024778181 4907985818E-12
13	-0.5132760082 8391806541 5984751899E-13
14	0.8680933040 7665493418 7433687383E-14
15	-0.1527015040 9030849719 8572355351E-14
16	0.2784686251 6493573965 0105251453E-15
17	-0.5249890437 4217669680 8472933696E-16
18	0.1020717991 2485612924 7455787226E-16
19	-0.2042264679 8997184130 8462421876E-17
20	0.4197064172 7264847440 8827228562E-18
21	-0.8844508176 1728105081 6483737536E-19
22	0.1908272629 5947174199 5060168262E-19
23	-0.4209746222 9351995033 6450865676E-20
24	0.9483904058 1983732764 1500214512E-21
25	-0.2179467860 1366743199 4032574014E-21
26	0.5103936869 0714509499 3452562741E-22
27	-0.1216883113 3344150908 9746779693E-22
28	0.2951289166 4478751929 4773757144E-23
29	-0.7275353763 7728468971 4438950920E-24
30	0.1821639048 6230739612 1667115976E-24
31	-0.4629629963 1633171661 2753482064E-25
32	0.1193539790 9715779152 3052371292E-25
33	-0.3119493285 2201424493 1062147473E-26
34	0.8261419734 5334664228 4170028518E-27
35	-0.2215803373 6609829830 2591177697E-27
36	0.6016031671 6542638904 5303124429E-28
37	-0.1652725098 3821265964 9744302314E-28
38	0.4592230358 7730270279 5636377166E-29
39	-0.1290062767 2132638473 7453212670E-29
40	0.3662718481 0320025908 1177078922E-30

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei1(y:OPR):OPR ==
  infinite? y => 1
  x:R:=retract(y)
  t:R:=acos((-20.0::R/x)-1.0::R)::R
  t01:=    0.19121732258605534539151932651OE1::R*cos(0.0::R)/2.0::R
  t02:=t01-0.42083550528684843755097498668OE-01::R*cos(t::R)::R
  t03:=t02+0.17228196272843267833711815783 5E-02::R*cos( 2.0::R*t)
  t04:=t03-0.99157821734445636455984232297 3E-04::R*cos( 3.0::R*t)
  t05:=t04+0.71760931680227750526559066559 2E-05::R*cos( 4.0::R*t)
  t06:=t05-0.61527331450951269682795679133 1E-06::R*cos( 5.0::R*t)
  t07:=t06+0.60248571065627583129399970161 0E-07::R*cos( 6.0::R*t)
  t08:=t07-0.65738488452883048229589418963 7E-08::R*cos( 7.0::R*t)
  t09:=t08+0.78531675418323998199481007987 1E-09::R*cos( 8.0::R*t)
  t10:=t09-0.10137302880038789855420277425 7E-09::R*cos( 9.0::R*t)
  t11:=t10+0.13997704132267686027782348862 3E-10::R*cos(10.0::R*t)
  t12:=t11-0.20510083767838189961896231871 1E-11::R*cos(11.0::R*t)
  t13:=t12+0.31683887260024778181490798581 8E-12::R*cos(12.0::R*t)
  t14:=t13-0.51327600828391806541598475189 9E-13::R*cos(13.0::R*t)
  t15:=t14+0.86809330407665493418743368738 3E-14::R*cos(14.0::R*t)
  t16:=t15-0.15270150409030849719857235535 1E-14::R*cos(15.0::R*t)
  t17:=t16+0.27846862516493573965010525145 3E-15::R*cos(16.0::R*t)
  t18:=t17-0.52498904374217669680847293369 6E-16::R*cos(17.0::R*t)
  t19:=t18+0.10207179912485612924745578722 6E-16::R*cos(18.0::R*t)
```

```
t20:=t19-0.20422646798997184130846242187 6E-17::R*cos(19.0::R*t)
t21:=t20+0.41970641727264847440882722856 2E-18::R*cos(20.0::R*t)
t22:=t21-0.88445081761728105081648373753 6E-19::R*cos(21.0::R*t)
t23:=t22+0.19082726295947174199506016826 2E-19::R*cos(22.0::R*t)
t24:=t23-0.42097462229351995033645086567 6E-20::R*cos(23.0::R*t)
t25:=t24+0.94839040581983732764150021451 2E-21::R*cos(24.0::R*t)
t26:=t25-0.21794678601366743199403257401 4E-21::R*cos(25.0::R*t)
t27:=t26+0.51039368690714509499345256274 1E-22::R*cos(26.0::R*t)
t28:=t27-0.12168831133344150908974677969 3E-22::R*cos(27.0::R*t)
t29:=t28+0.29512891664478751929477375714 4E-23::R*cos(28.0::R*t)
t30:=t29-0.72753537637728468971443895092 0E-24::R*cos(29.0::R*t)
t31:=t30+0.18216390486230739612166711597 6E-24::R*cos(30.0::R*t)
t32:=t31-0.46296299631633171661275348206 4E-25::R*cos(31.0::R*t)
t33:=t32+0.11935397909715779152305237129 2E-25::R*cos(32.0::R*t)
t34:=t33-0.31194932852201424493106214747 3E-26::R*cos(33.0::R*t)
t35:=t34+0.82614197345334664228417002851 8E-27::R*cos(34.0::R*t)
t36:=t35-0.22158033736609829830259117769 7E-27::R*cos(35.0::R*t)
t37:=t36+0.60160316716542638904530312442 9E-28::R*cos(36.0::R*t)
t38:=t37-0.16527250983821265964974430231 4E-28::R*cos(37.0::R*t)
t39:=t38+0.45922303587730270279563637716 6E-29::R*cos(38.0::R*t)
t40:=t39-0.12900627672132638473745321267 0E-29::R*cos(39.0::R*t)
t41:=t40+0.36627184810320025908117707892 2E-30::R*cos(40.0::R*t)
t41::OPR
```

Table 3: Chebyshev Coefficients - Continued (b)

$$xe^{-x}Ei(x) = \sum_{k-0}^{40}{}' A_k T_k(t), \quad t = (x+7)/3, \quad (-10 \le x \le -4)$$

k	A_k	21	-0.4322776783 3833850564 5764394579E-15
		22	-0.9063014799 6650172551 4905603356E-16
0	0.1757556496 0612937384 8762834691E 011	23	-0.1904669979 5816613974 4015963342E-16
1	-0.4358541517 7361661170 5001867964E-01	24	-0.4011792326 3502786634 6744227520E-17
2	-0.7979507139 5584254013 3217027492E-02	25	-0.8467772130 0168322313 4166334685E-18
3	-0.1484372327 3037121385 0970210001E-02	26	-0.1790842733 6586966555 5826492204E-18
4	-0.2800301984 3775145748 6203954948E-03	27	-0.3794490638 1714782440 1106175166E-19
5	-0.5348648512 8657932303 9177361553E-04	28	-0.8053999236 7982798526 0999654058E-20
6	-0.1032867243 5735548661 0233266460E-04	29	-0.1712339011 2362012974 3228671244E-20
7	-0.2014083313 0055368773 2226198639E-05	30	-0.3646274058 7749686208 6576562816E-21
8	-0.3961758434 2738664582 2338443500E-06	31	-0.7775969638 8939479435 3098157647E-22
9	-0.7853872767 0966316306 7607656069E-07	32	-0.1660628498 4484020566 2531950966E-22
10	-0.1567925981 0074698262 4616270279E-07	33	-0.3551178625 7882509300 5927145352E-23
11	-0.3150055939 3763998825 0007372851E-08	34	-0.7603722685 9413580929 5734653294E-24
12	-0.6365096822 5242037304 0380263972E-09	35	-0.1630074137 2584900288 9638374755E-24
13	-0.1292888113 2805631835 6593121259E-09	36	-0.3498575202 7286322350 7538497255E-25
14	-0.2638690999 6592557613 2149942808E-10	37	-0.7517179627 8900988246 0645145143E-26
15	-0.5408958287 0450687349 1922207896E-11	38	-0.1616877440 0527227629 8777317918E-26
16	-0.1113222784 6010898999 7676692708E-11	39	-0.3481270085 7247569174 8202271565E-27
17	-0.2299624726 0744624618 4338864145E-12	40	-0.7502707775 5024654701 0642233720E-28
18	-0.4766682389 4951902622 3913482091E-13	41	-0.1618454364 4959102680 7612330206E-28
19	-0.9911756747 3352709450 6246643371E-14	42	-0.3494366771 7051616674 9482836452E-29
20	-0.2067103580 4957072400 0900805021E-14	43	-0.7551036906 1261678585 6037026797E-30

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei2(y:OPR):OPR ==
  x:R:=retract(y)
  t:R:=acos((x+7.0::R)/3.0::R)::R
  t01:=    0.17575564960612937384876283469 1E1::R*cos(0.0::R)/2.0::R
  t02:=t01-0.43585415177361661170500186796 4E-01::R*cos(t)
  t03:=t02-0.79795071395584254013321702749 2E-02::R*cos( 2.0::R*t)
  t04:=t03-0.14843723273037121385097021000 1E-02::R*cos( 3.0::R*t)
  t05:=t04-0.28003019843775145748620395494 8E-03::R*cos( 4.0::R*t)
  t06:=t05-0.53486485128657932303917736155 3E-04::R*cos( 5.0::R*t)
  t07:=t06-0.10328672435735548661023326646 0E-04::R*cos( 6.0::R*t)
  t08:=t07-0.20140833130055368773222619863 9E-05::R*cos( 7.0::R*t)
  t09:=t08-0.39617584342738664582233844350 0E-06::R*cos( 8.0::R*t)
  t10:=t09-0.78538727670966316306760765606 9E-07::R*cos( 9.0::R*t)
  t11:=t10-0.15679259810074698262461627027 9E-07::R*cos(10.0::R*t)
  t12:=t11-0.31500559393763998825000737285 1E-08::R*cos(11.0::R*t)
  t13:=t12-0.63650968225242037304038026397 2E-09::R*cos(12.0::R*t)
  t14:=t13-0.12928881132805631835659312125 9E-09::R*cos(13.0::R*t)
  t15:=t14-0.26386909996592557613214994280 8E-10::R*cos(14.0::R*t)
  t16:=t15-0.54089582870450687349192220789 6E-11::R*cos(15.0::R*t)
  t17:=t16-0.11132227846010898999767669270 8E-11::R*cos(16.0::R*t)
  t18:=t17-0.22996247260744624618433886414 5E-12::R*cos(17.0::R*t)
  t19:=t18-0.47666823894951902622391348209 1E-13::R*cos(18.0::R*t)
```

```
t20:=t19-0.991175674733527094506246643371E-14::R*cos(19.0::R*t)
t21:=t20-0.206710358049570724000900805021E-14::R*cos(20.0::R*t)
t22:=t21-0.432277678338338505645764394579E-15::R*cos(21.0::R*t)
t23:=t22-0.906301479966501725514905603356E-16::R*cos(22.0::R*t)
t24:=t23-0.190466997958166139744015963342E-16::R*cos(23.0::R*t)
t25:=t24-0.401179232635027866346744227520E-17::R*cos(24.0::R*t)
t26:=t25-0.846777213001683223134166334685E-18::R*cos(25.0::R*t)
t27:=t26-0.179084273365869665555826492204E-18::R*cos(26.0::R*t)
t28:=t27-0.379449063817147824401106175166E-19::R*cos(27.0::R*t)
t29:=t28-0.805399923679827985260999654058E-20::R*cos(28.0::R*t)
t30:=t29-0.171233901123620129743228671244E-20::R*cos(29.0::R*t)
t31:=t30-0.364627405877496862086576562816E-21::R*cos(30.0::R*t)
t32:=t31-0.777596963889394794353098157647E-22::R*cos(31.0::R*t)
t33:=t32-0.166062849844840205662531950966E-22::R*cos(32.0::R*t)
t34:=t33-0.355117862578825093005927145352E-23::R*cos(33.0::R*t)
t35:=t34-0.760372268594135809295734653294E-24::R*cos(34.0::R*t)
t36:=t35-0.163007413725849002889638374755E-24::R*cos(35.0::R*t)
t37:=t36-0.349857520272863223507538497255E-25::R*cos(36.0::R*t)
t38:=t37-0.751717962789009882460645145143E-26::R*cos(37.0::R*t)
t39:=t38-0.161687744005272276298777317918E-26::R*cos(38.0::R*t)
t40:=t39-0.348127008572475691748202271565E-27::R*cos(39.0::R*t)
t41:=t40-0.750270777550246547010642233720E-28::R*cos(40.0::R*t)
t42:=t41-0.161845436449591026807612330206E-28::R*cos(41.0::R*t)
t43:=t42-0.349436677170516166749482836452E-29::R*cos(42.0::R*t)
t44:=t43-0.755103690612616785856037026797E-30::R*cos(43.0::R*t)
t44::OPR
```

Table 3: Chebyshev Coefficients - Continued (c)

$$[Ei - log|x| - \gamma]/x = \sum_{k-0}^{33} {}' A_k T_k(t), \quad t = x/4, \quad (-4 \leq x \leq 4)$$

k	A_k		
0	0.3293700103 7673912939 3905231421E 01	16	0.2615386378 8854429666 9068664148E-10
1	0.1679835052 3713029156 5505796064E 01	17	0.2721858622 8541670644 6550268995E-11
2	0.7220436105 6787543524 0299679644E 00	18	0.2693750031 9835792992 5326427442E-12
3	0.2600312360 5480956171 3740181192E 00	19	0.2541220946 7072635546 7884089307E-13
4	0.8010494308 1737502239 4742889237E-01	20	0.2290130406 8650370941 8510620516E-14
5	0.2151403663 9763337548 0552483005E-01	21	0.1975465739 0746229940 1057650412E-15
6	0.5116207789 9303312062 1968910894E-02	22	0.1634024551 9289317406 8635419984E-16
7	0.1090932861 0073913560 5066199014E-02	23	0.1298235437 0796376099 1961293204E-17
8	0.2107415320 2393891631 8348675226E-03	24	0.9922587925 0737105964 4632581302E-19
9	0.3719904516 6518885709 5940815956E-04	25	0.7306252806 7221032944 7230880087E-20
10	0.6043491637 1238787570 4767032866E-05	26	0.5189676834 6043451272 0780080019E-21
11	0.9092954273 9626095264 9596541772E-06	27	0.3560409454 0997068112 8043162227E-22
12	0.1273805160 6592647886 5567184969E-06	28	0.2361979432 5793864237 0187203948E-23
13	0.1669185748 4109890739 0896143814E-07	29	0.1516837767 7214529754 9624516819E-24
14	0.2054417026 4010479254 7612484551E-08	30	0.9439089722 2448744292 5310405245E-26
15	0.2383584444 4668176591 4052321417E-09	31	0.5697227559 5036921198 9581737831E-27
		32	0.3338333627 7954330315 6597939562E-28
		33	0.1900626012 8161914852 6680482237E-29

(γ=0.5772156649 0153286060 6512090082 E 00)

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei3(y:OPR):OPR ==
  x:R:=retract(y)
  x = 0.0::R => 1
  t:R:=acos(x/4.0::R)::R
  t01:=    0.32937001037673912939390523142 1E1::R*cos(0.0::R)/2.0::R
  t02:=t01+0.16798350523713029156550579606 4E1::R*cos(t)
  t03:=t02+0.72204361056787543524029967964 4E0::R*cos( 2.0::R*t)
  t04:=t03+0.26003123605480956171374018119 2E0::R*cos( 3.0::R*t)
  t05:=t04+0.80104943081737502239474288923 7E-01::R*cos( 4.0::R*t)
  t06:=t05+0.21514036639763337548055248300 5E-01::R*cos( 5.0::R*t)
  t07:=t06+0.51162077899303312062196891089 4E-02::R*cos( 6.0::R*t)
  t08:=t07+0.10909328610073913560506619901 4E-02::R*cos( 7.0::R*t)
  t09:=t08+0.21074153202393891631834867522 6E-03::R*cos( 8.0::R*t)
  t10:=t09+0.37199045166518885709594081595 6E-04::R*cos( 9.0::R*t)
  t11:=t10+0.60434916371238787570476703286 6E-05::R*cos(10.0::R*t)
  t12:=t11+0.90929542739626095264959654177 2E-06::R*cos(11.0::R*t)
  t13:=t12+0.12738051606592647886556718496 9E-06::R*cos(12.0::R*t)
  t14:=t13+0.16691857484109890739089614381 4E-07::R*cos(13.0::R*t)
  t15:=t14+0.20544170264010479254761248455 1E-08::R*cos(14.0::R*t)
  t16:=t15+0.23835844444668176591405232141 7E-09::R*cos(15.0::R*t)
  t17:=t16+0.26153863788854429666906866414 8E-10::R*cos(16.0::R*t)
  t18:=t17+0.27218586228541670644655026899 5E-11::R*cos(17.0::R*t)
  t19:=t18+0.26937500319835792992532642744 2E-12::R*cos(18.0::R*t)
  t20:=t19+0.25412209467072635546788408930 7E-13::R*cos(19.0::R*t)
  t21:=t20+0.22901304068650370941851062051 6E-14::R*cos(20.0::R*t)
```

```
t22:=t21+0.19754657390746229940105765041 2E-15::R*cos(21.0::R*t)
t23:=t22+0.16340245519289317406863541998 4E-16::R*cos(22.0::R*t)
t24:=t23+0.12982354370796376099196129320 4E-17::R*cos(23.0::R*t)
t25:=t24+0.99225879250737105964463258130 2E-19::R*cos(24.0::R*t)
t26:=t25+0.73062528067221032944723088008 7E-20::R*cos(25.0::R*t)
t27:=t26+0.51896768346043451272078008001 9E-21::R*cos(26.0::R*t)
t28:=t27+0.35604094540997068112804316222 7E-22::R*cos(27.0::R*t)
t29:=t28+0.23619794325793864237018720394 8E-23::R*cos(28.0::R*t)
t30:=t29+0.15168377677214529754962451681 9E-24::R*cos(29.0::R*t)
t31:=t30+0.94390897222448744292531040524 5E-26::R*cos(30.0::R*t)
t32:=t31+0.56972275595036921198958173783 1E-27::R*cos(31.0::R*t)
t33:=t32+0.33383336277954330315659793956 2E-28::R*cos(32.0::R*t)
t34:=t33+0.19006260128161914852668048223 7E-29::R*cos(33.0::R*t)
t34::OPR
```

Table 3: Chebyshev Coefficients - Continued (d)

$$xe^{-x}Ei(x) = \sum_{k-0}^{49}{}' A_k T_k(t), \quad t = (x-8)/4, \quad (4 \le x \le 12)$$

k	A_k	24	0.1069023072 9386369566 8857256409E-14
		25	-0.2507030070 5700729569 2572254042E-15
0	0.2455133538 7812952867 3420457043E 01	26	0.5937322503 7915516070 6073763509E-16
1	-0.1624383791 3037652439 6002276856E 00	27	-0.1417734582 4376625234 4732005648E-16
2	0.4495753080 9357264148 0785417193E-01	28	0.3409203754 3608089342 6806402093E-17
3	-0.6741578679 9892299884 8718835050E-02	29	-0.8248290269 5054937928 8702529656E-18
4	-0.1306697142 8032942805 1599341387E-02	30	0.2006369712 6214423139 8824095937E-18
5	0.1381083146 0007257602 0202089820E-02	31	-0.4903851667 9674222440 3498152027E-19
6	-0.5850228790 1596579868 7368242394E-03	32	0.1203734482 3483321716 6664609324E-19
7	0.1749299341 0789197003 8740976432E-03	33	-0.2966282447 1413682538 1453572575E-20
8	-0.4047281499 0529303552 2869333800E-04	34	0.7335512384 2880759924 2142328436E-21
9	0.7217102412 1709975003 5752600049E-05	35	-0.1819924142 9085112734 4263485604E-21
10	-0.8612776970 1986775241 4815450193E-06	36	0.4528629374 2957606021 7359526404E-22
11	-0.2514475296 5322559777 9084739054E-09	37	-0.1129980043 7506096133 8906717853E-22
12	0.3794747138 2014951081 4074505574E-07	38	0.2826681251 2901165692 3764408445E-23
13	-0.1442117969 5211980616 0265640172E-07	39	-0.7087717977 1690496166 6732640699E-24
14	0.3935049295 9761013108 7190848042E-08	40	0.1781104524 0187095153 4401530034E-24
15	-0.9284689401 0633175304 7289210353E-09	41	-0.4485004076 6189635731 2006142358E-25
16	0.2031789568 0065461336 6090995698E-09	42	0.1131540292 5754766224 5053090840E-25
17	-0.4292498504 9923683142 7918026902E-10	43	-0.2859957899 7793216379 0414326136E-26
18	0.8992647177 7812393526 8001544182E-11	44	0.7240775806 9226736175 8172726753E-27
19	-0.1900869118 4121097524 2396635722E-11	45	-0.1836132234 1257789805 0666710105E-27
20	0.4092198912 2237383452 6121178338E-12	46	0.4663128735 2273048658 2600122073E-28
21	-0.8999253437 2931901982 5435824585E-13	47	-0.1185959588 9190288794 6724005478E-28
22	0.2019654670 8242638335 4948543451E-13	48	0.3020290590 5567131073 1137614875E-29
23	-0.4612930261 3830820719 4950531726E-14	49	-0.7701650548 1663660609 8827057102E-30

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei4(y:OPR):OPR ==
  x:R:=retract(y)
  t:R:=acos((x-8.0::R)/4.0::R)::R
  t01:=    0.24551335387812952867342045704 3E1::R*cos(0.0::R)/2.0::R
  t02:=t01-0.16243837913037652439600227685 6E0::R*cos(t)
  t03:=t02+0.44957530809357264148078541719 3E-01::R*cos( 2.0::R*t)
  t04:=t03-0.67415786799892299884871883505 0E-02::R*cos( 3.0::R*t)
  t05:=t04-0.13066971428032942805159934138 7E-02::R*cos( 4.0::R*t)
  t06:=t05+0.13810831460007257602020208982 0E-02::R*cos( 5.0::R*t)
  t07:=t06-0.58502287901596579868736824239 4E-03::R*cos( 6.0::R*t)
  t08:=t07+0.17492993410789197003874097643 2E-03::R*cos( 7.0::R*t)
  t09:=t08-0.40472814990529303552286933380 0E-04::R*cos( 8.0::R*t)
  t10:=t09+0.72171024121709975003575260004 9E-05::R*cos( 9.0::R*t)
  t11:=t10-0.86127769701986775241481545019 3E-06::R*cos(10.0::R*t)
  t12:=t11-0.25144752965322559779790847390 54E-09::R*cos(11.0::R*t)
  t13:=t12+0.37947471382014951081407450557 4E-07::R*cos(12.0::R*t)
  t14:=t13-0.14421179695211980616026564017 2E-07::R*cos(13.0::R*t)
  t15:=t14+0.39350492959761013108719084804 2E-08::R*cos(14.0::R*t)
  t16:=t15-0.92846894010633175304728921035 3E-09::R*cos(15.0::R*t)
```

```
t17:=t16+0.20317895680065461336609095698E-09::R*cos(16.0::R*t)
t18:=t17-0.42924985049923683142791802690 2E-10::R*cos(17.0::R*t)
t19:=t18+0.89926471777812393526800154418 2E-11::R*cos(18.0::R*t)
t20:=t19-0.19008691184121097524239663572 2E-11::R*cos(19.0::R*t)
t21:=t20+0.40921989122237383452612117833 8E-12::R*cos(20.0::R*t)
t22:=t21-0.89992534372931901982543582458 5E-13::R*cos(21.0::R*t)
t23:=t22+0.20196546708242638335494854345 1E-13::R*cos(22.0::R*t)
t24:=t23-0.46129302613830820719495053172 6E-14::R*cos(23.0::R*t)
t25:=t24+0.10690230729386369566885725640 9E-14::R*cos(24.0::R*t)
t26:=t25-0.25070300705700729569257225404 2E-15::R*cos(25.0::R*t)
t27:=t26+0.59373225037915516070607376350 9E-16::R*cos(26.0::R*t)
t28:=t27-0.14177345824376625234473200564 8E-16::R*cos(27.0::R*t)
t29:=t28+0.34092037543608089342680640209 3E-17::R*cos(28.0::R*t)
t30:=t29-0.82482902695054937928870252965 6E-18::R*cos(29.0::R*t)
t31:=t30+0.20063697126214423139882409593 7E-18::R*cos(30.0::R*t)
t32:=t31-0.49038516679674222440349815202 7E-19::R*cos(31.0::R*t)
t33:=t32+0.12037344823483321716666460932 4E-19::R*cos(32.0::R*t)
t34:=t33-0.29662824471413682538145357257 5E-20::R*cos(33.0::R*t)
t35:=t34+0.73355123842880759924214232843 6E-21::R*cos(34.0::R*t)
t36:=t35-0.18199241429085112734426348560 4E-21::R*cos(35.0::R*t)
t37:=t36+0.45286293742957606021735952640 4E-22::R*cos(36.0::R*t)
t38:=t37-0.11299800437506096133890671785 3E-22::R*cos(37.0::R*t)
t39:=t38+0.28266812512901165692376440844 5E-23::R*cos(38.0::R*t)
t40:=t39-0.70877179771690496166673264069 9E-24::R*cos(39.0::R*t)
t41:=t40+0.17811045240187095153440153003 4E-24::R*cos(40.0::R*t)
t42:=t41-0.44850040766189635731200614235 8E-25::R*cos(41.0::R*t)
t43:=t42+0.11315402925754766224505309084 0E-25::R*cos(42.0::R*t)
t44:=t43-0.28599578997793216379041432613 6E-26::R*cos(43.0::R*t)
t45:=t44+0.72407758069226736175817272675 3E-27::R*cos(44.0::R*t)
t46:=t45-0.18361322341257789805066671010 5E-27::R*cos(45.0::R*t)
t47:=t46+0.46631287352273048658260012207 3E-28::R*cos(46.0::R*t)
t48:=t47-0.11859595889190288794672400547 8E-28::R*cos(47.0::R*t)
t49:=t48+0.30202905905567131073113761487 5E-29::R*cos(48.0::R*t)
t50:=t49-0.77016505481663660609882705710 2E-30::R*cos(49.0::R*t)
t50::OPR
```

Table 3: Chebyshev Coefficients - Continued (e)

$$xe^{-x}Ei(x) = \sum_{k-0}^{47}{}' A_k T_k(t), \quad t = (x-22)/10, \quad (12 \le x \le 32)$$

k	A_k	23	0.5898114347 0713196171 1164283918E-14
		24	-0.9099707635 9564920464 3554720718E-15
0	0.2117028640 4369866832 9789991614E 01	25	0.1040752382 6695538658 5405697541E-15
1	-0.3204237273 7548579499 0618303177E-01	26	-0.1809815426 0592279322 7163355935E-17
2	0.8891732077 3531683589 0182400335E-02	27	-0.3777098842 5639477336 9593494417E-17
3	-0.2507952805 1892993708 8352442063E-02	28	0.1580332901 0284795713 6759888420E-17
4	0.7202789465 9598754887 5760902487E-03	29	-0.4684291758 8088273064 8433752957E-18
5	-0.2103490058 5011305342 3531441256E-03	30	0.1199516852 5919809370 7533478542E-18
6	0.6205732318 2769321658 8857730842E-04	31	-0.2823594749 8418651767 9349931117E-19
7	-0.1826566749 8167026544 9155689733E-04	32	0.6293738065 6446352262 7520190349E-20
8	0.5270651575 2893637580 7788296811E-05	33	-0.1352410249 5047975630 5343973177E-20
9	-0.1459666547 6199457532 3066719367E-05	34	0.2837106053 8552914159 0980426210E-21
10	0.3781719973 5896367198 0484193981E-06	35	-0.5867007420 2463832353 1936371015E-22
11	-0.8842581282 8407192007 7971589012E-07	36	0.1205247636 0954731111 2449686917E-22
12	0.1741749198 5383936137 7350309156E-07	37	-0.2474446616 9988486972 8416011246E-23
13	-0.2313517747 0436906350 6474480152E-08	38	0.5099962585 8378500814 2986465688E-24
14	-0.1228609819 1808623883 2104835230E-09	39	-0.1058382578 7754224088 7093294733E-24
15	0.2349966236 3228637047 8311381926E-09	40	0.2215276245 0704827856 6429387155E-25
16	-0.1100719401 0272628769 0738963049E-09	41	-0.4679278754 7569625867 1852546231E-26
17	0.3848275157 8612071114 9705563369E-10	42	0.9972872990 6020770482 4269828079E-27
18	-0.1148440967 4900158965 8439301603E-10	43	-0.2143267945 2167880459 1907805844E-27
19	0.3056876293 0885208263 0893626200E-11	44	0.4640656908 8381811433 8414829515E-28
20	-0.7388278729 2847356645 4163131431E-12	45	-0.1011447349 2115139094 8461800780E-28
21	0.1630933094 1659411056 4148013749E-12	46	0.2217211522 7100771109 3046878345E-29
22	-0.3276989373 3127124965 7111774748E-13	47	-0.4884890469 2437855322 4914645512E-30

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei5(y:OPR):OPR ==
  x:R:=retract(y)
  t:R:=acos((x-22.0::R)/10.0::R)::R
  t01:=    0.21170286404369866832929789991614E1::R*cos(0.0::R)::R/2.0::R
  t02:=t01-0.32042372737548579499061830317 7E-01::R*cos(t)
  t03:=t02+0.88917320773531683589018240033 5E-02::R*cos( 2.0::R*t)
  t04:=t03-0.25079528051892993708835244206 3E-02::R*cos( 3.0::R*t)
  t05:=t04+0.72027894659598754887576090248 7E-03::R*cos( 4.0::R*t)
  t06:=t05-0.21034900585011305342353144125 6E-03::R*cos( 5.0::R*t)
  t07:=t06+0.62057323182769321658885773084 2E-04::R*cos( 6.0::R*t)
  t08:=t07-0.18265667498167026544915568973 3E-04::R*cos( 7.0::R*t)
  t09:=t08+0.52706515752893637580778829681 1E-05::R*cos( 8.0::R*t)
  t10:=t09-0.14596665476199457532306671936 7E-05::R*cos( 9.0::R*t)
  t11:=t10+0.37817199735896367198048419398 1E-06::R*cos(10.0::R*t)
  t12:=t11-0.88425812828407192007797158901 2E-07::R*cos(11.0::R*t)
  t13:=t12+0.17417491985383936137735030915 6E-07::R*cos(12.0::R*t)
  t14:=t13-0.23135177470436906350647448015 2E-08::R*cos(13.0::R*t)
  t15:=t14-0.12286098191808623883210483523 0E-09::R*cos(14.0::R*t)
  t16:=t15+0.23499662363228637047831138192 6E-09::R*cos(15.0::R*t)
  t17:=t16-0.11007194010272628769073896304 9E-09::R*cos(16.0::R*t)
```

```
t18:=t17+0.384827515786120711149705563369E-10::R*cos(17.0::R*t)
t19:=t18-0.114844096749001589658439301603E-10::R*cos(18.0::R*t)
t20:=t19+0.305687629308852082630893626200E-11::R*cos(19.0::R*t)
t21:=t20-0.738827872928473566454163131431E-12::R*cos(20.0::R*t)
t22:=t21+0.163093309416594110564148013749E-12::R*cos(21.0::R*t)
t23:=t22-0.327698937331271249657111774748E-13::R*cos(22.0::R*t)
t24:=t23+0.589811434707131961711164283918E-14::R*cos(23.0::R*t)
t25:=t24-0.909970763595649204643554720718E-15::R*cos(24.0::R*t)
t26:=t25+0.104075238266955386585405697541E-15::R*cos(25.0::R*t)
t27:=t26-0.180981542605922793227163355935E-17::R*cos(26.0::R*t)
t28:=t27-0.377709884256394773369593494417E-17::R*cos(27.0::R*t)
t29:=t28+0.158033290102847957136759888420E-17::R*cos(28.0::R*t)
t30:=t29-0.468429175880882730648433752957E-18::R*cos(29.0::R*t)
t31:=t30+0.119951685259198093707533478542E-18::R*cos(30.0::R*t)
t32:=t31-0.282359474984186517679349931117E-19::R*cos(31.0::R*t)
t33:=t32+0.629373806564463522627520190349E-20::R*cos(32.0::R*t)
t34:=t33-0.135241024950479756305343973177E-20::R*cos(33.0::R*t)
t35:=t34+0.283710605385529141590980426210E-21::R*cos(34.0::R*t)
t36:=t35-0.586700742024638323531936371015E-22::R*cos(35.0::R*t)
t37:=t36+0.120524763609547311112449686917E-22::R*cos(36.0::R*t)
t38:=t37-0.247444661699884869728416011246E-23::R*cos(37.0::R*t)
t39:=t38+0.509996258583785008142986465688E-24::R*cos(38.0::R*t)
t40:=t39-0.105838257877542240887093294733E-24::R*cos(39.0::R*t)
t41:=t40+0.221527624507048278566429387155E-25::R*cos(40.0::R*t)
t42:=t41-0.467927875475696258671852546231E-26::R*cos(41.0::R*t)
t43:=t42+0.997287299060207704824269828079E-27::R*cos(42.0::R*t)
t44:=t42-0.214326794521678804591907805844E-27::R*cos(43.0::R*t)
t45:=t42+0.464065690883818114338414829515E-28::R*cos(44.0::R*t)
t46:=t42-0.101144734921151390948461800780E-28::R*cos(45.0::R*t)
t47:=t42+0.221721152271007711093046878345E-29::R*cos(46.0::R*t)
t48:=t42-0.488489046924378553224914645512E-30::R*cos(47.0::R*t)
t48::OPR
```

Table 3: Chebyshev Coefficients - Continued (f)

$$xe^{-x}Ei(x) = \sum_{k-0}^{46}{}' A_k T_k(t), \quad t = (64/x) - 1, \quad (32 \leq x < \infty)$$

k	A_k
0	0.2032843945 7961669908 7873844202E 01
1	0.1669920452 0313628514 7618434339E-01
2	0.2845284724 3613468074 2489985325E-03
3	0.7563944358 5162064894 8786693854E-05
4	0.2798971289 4508591575 0484318090E-06
5	0.1357901828 5345310695 2556392593E-07
6	0.8343596202 0404692558 5610289412E-09
7	0.6370971727 6402484382 7524337306E-10
8	0.6007247608 8118612357 6083084850E-11
9	0.7022876174 6797735907 5059216588E-12
10	0.1018302673 7036876930 9667322152E-12
11	0.1761812903 4308800404 0656741554E-13
12	0.3250828614 2353606942 4072007647E-14
13	0.5071770025 5058186788 1479300685E-15
14	0.1665177387 0432942985 3520036957E-16
15	-0.3166753890 7975144007 2410018963E-16
16	-0.1588403763 6641415154 8423134074E-16
17	-0.4175513256 1380188308 9626455063E-17
18	-0.2892347749 7071418820 2868862358E-18
19	0.2800625903 3966080728 9978777339E-18
20	0.1322938639 5392708914 0532005364E-18
21	0.1804447444 1773019958 5334811191E-19
22	-0.7905384086 5226165620 2021080364E-20
23	-0.4435711366 3695734471 8167314045E-20
24	-0.4264103994 9781026176 0579779746E-21
25	0.3920101766 9371439072 5625388636E-21
26	0.1527378051 3439636447 2804486402E-21
27	-0.1024849527 0494906078 6953149788E-22
28	-0.2134907874 7710893794 8904287231E-22
29	-0.3239139475 1602368761 4279789345E-23
30	0.2142183762 2964597029 6249355934E-23
31	0.8234609419 6189955316 9207838151E-24
32	-0.1524652829 6206721081 1495038147E-24
33	-0.1378208282 4882440129 0438126477E-24
34	0.2131311201 4287370679 1513005998E-26
35	0.2012649651 8713266585 9213006507E-25
36	0.1995535662 0563740232 0607178286E-26
37	-0.2798995812 2017971142 6020884464E-26
38	-0.5534511830 5070025094 9784942560E-27
39	0.3884995422 6845525312 9749000696E-27
40	0.1121304407 2330701254 0043264712E-27
41	-0.5566568286 7445948805 7823816866E-28
42	-0.2045482612 4651357628 8865878722E-28
43	0.8453814064 4893808943 7361193598E-29
44	0.3565755151 2015152659 0791715785E-29
45	-0.1383652423 4779775181 0195772006E-29
46	-0.6062142653 2093450576 7865286306E-30

— package DFSFUN DoubleFloatSpecialFunctions —

```
Ei6(y:OPR):OPR ==
  infinite? y => 1
  x:R:=retract(y)
  m:R:=64.0::R/x-1.0::R
  t:R:=acos(m::R)::R
  t01:=    0.20328439457961669908787384420 2E1::R*cos(0.0::R)::R/2.0::R
  t02:=t01+0.16699204520313628514761843433 9E-01::R*cos(t)
  t03:=t02+0.28452847243613468074248998532 5E-03::R*cos( 2.0::R*t)
  t04:=t03+0.75639443585162064894878669385 4E-05::R*cos( 3.0::R*t)
  t05:=t04+0.27989712894508591575048431809 0E-06::R*cos( 4.0::R*t)
  t06:=t05+0.13579018285345310695255639259 3E-07::R*cos( 5.0::R*t)
  t07:=t06+0.83435962020404692558561028941 2E-09::R*cos( 6.0::R*t)
  t08:=t07+0.63709717276402484382752433730 6E-10::R*cos( 7.0::R*t)
  t09:=t08+0.60072476088118612357608308485 0E-11::R*cos( 8.0::R*t)
  t10:=t09+0.70228761746797735907505921658 8E-12::R*cos( 9.0::R*t)
  t11:=t10+0.10183026737036876930966732215 2E-12::R*cos(10.0::R*t)
  t12:=t11+0.17618129034308800404065674155 4E-13::R*cos(11.0::R*t)
  t13:=t12+0.32508286142353606942407200764 7E-14::R*cos(12.0::R*t)
  t14:=t13+0.50717700255058186788147930068 5E-15::R*cos(13.0::R*t)
```

```
t15:=t14+0.16651773870432942985352003695 7E-16::R*cos(14.0::R*t)
t16:=t15-0.31667538907975144007241001896 3E-16::R*cos(15.0::R*t)
t17:=t16-0.15884037636641415154842313407 4E-16::R*cos(16.0::R*t)
t18:=t17-0.41755132561380188308962645506 3E-17::R*cos(17.0::R*t)
t19:=t18-0.28923477497071418820286886235 8E-18::R*cos(18.0::R*t)
t20:=t19+0.28006259033966080728997877733 9E-18::R*cos(19.0::R*t)
t21:=t20+0.13229386395392708914053200536 4E-18::R*cos(20.0::R*t)
t22:=t21+0.18044474441773019959585334811 191E-19::R*cos(21.0::R*t)
t23:=t22-0.79053840865226165620202108036 4E-20::R*cos(22.0::R*t)
t24:=t23-0.44357113663695734471816731404 5E-20::R*cos(23.0::R*t)
t25:=t24-0.42641039949781026176057977974 6E-21::R*cos(24.0::R*t)
t26:=t25+0.39201017669371439072562538863 6E-21::R*cos(25.0::R*t)
t27:=t26+0.15273780513439636447280448640 2E-21::R*cos(26.0::R*t)
t28:=t27-0.10248495270494906078695314978 8E-22::R*cos(27.0::R*t)
t29:=t28-0.21349078747710893794890428723 1E-22::R*cos(28.0::R*t)
t30:=t29-0.32391394751602368761427978934 5E-23::R*cos(29.0::R*t)
t31:=t30+0.21421837622964597029624935593 4E-23::R*cos(30.0::R*t)
t32:=t31+0.82346094196189955316920783815 1E-24::R*cos(31.0::R*t)
t33:=t32-0.15246528296206721081149503814 7E-24::R*cos(32.0::R*t)
t34:=t33-0.13782082824882440129043812647 7E-24::R*cos(33.0::R*t)
t35:=t34+0.21313112014287370679151300599 8E-26::R*cos(34.0::R*t)
t36:=t35+0.20126496518713266585921300650 7E-25::R*cos(35.0::R*t)
t37:=t36+0.19955356620563740232060717828 6E-26::R*cos(36.0::R*t)
t38:=t37-0.27989958122017971142602088446 4E-26::R*cos(37.0::R*t)
t39:=t38-0.55345118305070025094978494256 0E-27::R*cos(38.0::R*t)
t40:=t39+0.38849954226845525312974900069 6E-27::R*cos(39.0::R*t)
t41:=t40+0.11213044072330701254004326471 2E-27::R*cos(40.0::R*t)
t42:=t41-0.55665682867445948805782381686 6E-28::R*cos(41.0::R*t)
t43:=t42-0.20454826124651357628886587872 2E-28::R*cos(42.0::R*t)
t44:=t43+0.84538140644893808943736119359 8E-29::R*cos(43.0::R*t)
t45:=t44+0.35657551512015152659079171578 5E-29::R*cos(44.0::R*t)
t46:=t45-0.13836524234779775181019577200 6E-29::R*cos(45.0::R*t)
t47:=t46-0.60621426532093450576786528630 6E-30::R*cos(46.0::R*t)
t47::OPR
```

Table 4: Function Values of the Associated Functions

x	$t = -(20/x) - 1$	$xe^{-x}Ei(x)$
$-\infty$	-1.000	0.1000000000 0000000000 0000000000 E 01
-160	-0.875	0.9938266956 7406127387 8797850088 E 00
-80	-0.750	0.9878013330 9428877356 4522608410 E 00
-53 1/3	-0.625	0.9819162901 4319443961 7735426105 E 00
-40	-0.500	0.9761646031 8514305080 8000604060 E 00
-32	-0.375	0.9705398840 7466392046 2584664361 E 00
-26 2/3	-0.250	0.9650362511 2337703576 3536593528 E 00
-22 6/7	-0.125	0.9596482710 7936727616 5478970820 E 00
-20	-0.000	0.9543709099 1921683397 5195829433 E 00
-17 7/9	0.125	0.9491994907 7974574460 6445346803 E 00
-16	0.250	0.9441296577 3690297898 4149471583 E 00
-14 6/11	0.375	0.9391573444 1928424124 0422409988 E 00
-13 1/3	0.500	0.9342787466 5341046480 9375801650 E 00
-12 4/13	0.625	0.9294902984 9721403772 5319679042 E 00
-11 3/7	0.750	0.9247886511 4084169605 5993585492 E 00
-10 2/3	0.875	0.9201706542 4944567620 2148012149 E 00
-10	1.000	0.9156333393 9788081876 0698157666 E 00

x	$t = -(x + 7)/3$	$xe^{-x}Ei(x)$
-10.000	-1.000	0.9156333393 9788081876 0698157661 E 01
-9.625	-0.875	0.9128444614 6799341885 6575662217 E 00
-9.250	-0.750	0.9098627515 2542413937 8954274597 E 00
-8.875	-0.625	0.9066672706 5475388033 4995756418 E 00
-8.500	-0.500	0.9032339019 7320784414 4682926135 E 00
-8.125	-0.375	0.8995347176 8847383630 1415777697 E 00
-7.750	-0.250	0.8955371870 8753915717 9475513219 E 00
-7.375	-0.125	0.8912031763 2125431626 7087476258 E 00
-7.000	-0.000	0.8864876725 3642935289 3993846569 E 00
-6.625	0.125	0.8813371384 6821020039 4305706270 E 00
-6.250	0.250	0.8756873647 8846593227 6462155532 E 00
-5.875	0.375	0.8694606294 5411341030 2047153364 E 00
-5.500	0.500	0.8625618846 9070142209 0918986586 E 00
-5.125	0.625	0.8548735538 9019954239 2425567234 E 00
-4.750	0.750	0.8462482991 0358736117 1665798810 E 00
-4.375	0.875	0.8364987545 5629874174 2152267582 E 00
-4.000	1.000	0.8253825996 0422333240 8183035504 E 00

x	$t = x/4$	$[Ei(x) - log\|x\| - \gamma]/x$
-4.0	-1.000	0.4918223446 0781809647 9962798267 E 00
-3.5	-0.875	0.5248425066 4412835691 8258753311 E 00
-3.0	-0.750	0.5629587782 2127986313 8086024270 E 00
-2.5	-0.625	0.6073685258 5838306451 4266925640 E 00
-2.0	-0.500	0.6596316780 8476964479 5492023380 E 00
-1.5	-0.375	0.7218002369 4421992965 7623030310 E 00
-1.0	-0.250	0.7965995992 9705313428 3675865540 E 00
-0.5	-0.125	0.8876841582 3549672587 2151815870 E 00
0.0	-0.000	0.1000000000 0000000000 0000000000 E 01
0.5	0.125	0.1140302841 0431720574 6248768807 E 01
1.0	0.250	0.1317902151 4544038948 6000884424 E 01
1.5	0.375	0.1545736450 7467337302 4859074039 E 01
2.0	0.500	0.1841935755 2702059966 7788045934 E 01
2.5	0.625	0.2232103799 1211651144 5340506423 E 01
3.0	0.750	0.2752668205 6852580020 0219289740 E 01
3.5	0.875	0.3455821531 9301241243 7300898811 E 01
4.0	1.000	0.4416841111 0086991358 0118598668 E 01

x	$t = (x - 8)/4$	$xe^{-x}Ei(x)$
4.0	-1.000	0.1438208031 4544827847 0968670330 E 01
4.5	-0.875	0.1396419029 6297460710 0674523183 E 01
5.0	-0.750	0.1353831277 4552859779 0189174047 E 01
5.5	-0.625	0.1314143565 7421192454 1219816991 E 01
6.0	-0.500	0.1278883860 4895616189 2314099578 E 01
6.5	-0.375	0.1248391155 0017014864 0741941387 E 01
7.0	-0.250	0.1222408052 3605310590 3656846622 E 01
7.5	-0.125	0.1200421499 5996307864 3879158950 E 01
8.0	-0.000	0.1181847986 9872079731 7739362644 E 01
8.5	0.125	0.1166126525 8117484943 9918142965 E 01
9.0	0.250	0.1152759208 7089248132 2396814952 E 01
9.5	0.375	0.1141323475 9526242015 5338560641 E 01
10.0	0.500	0.1131470204 7341077803 4051681355 E 01
10.5	0.625	0.1122915570 0177606064 2888630755 E 01
11.0	0.750	0.1115430938 9980384416 4779434229 E 01
11.5	0.875	0.1108832926 3050773058 6855234934 E 01
12.0	1.000	0.1102974544 9067590726 7241234953 E 01

x	$t = (x - 22)/10$	$xe^{-x}Ei(x)$
12.00	-1.000	0.1102974544 9067590726 7241234952 E 01
13.25	-0.875	0.1090844898 2154756926 6468614954 E 01
14.50	-0.750	0.1081351395 7351912850 6346643795 E 01
15.75	-0.625	0.1073701384 1997572371 2157900374 E 01
17.00	-0.500	0.1067393691 9585378312 9572196197 E 01
18.25	-0.375	0.1062096608 6221502426 8372647556 E 01
19.50	-0.250	0.1057581342 1587250319 5393949410 E 01
20.75	-0.125	0.1053684451 2894094408 2102194964 E 01
22.00	-0.000	0.1050285719 6851897941 1780664532 E 01
23.25	0.125	0.1047294551 7053248581 1492365591 E 01
24.50	0.250	0.1044641267 9046436368 9761075289 E 01
25.75	0.375	0.1042271337 2023202388 5710928048 E 01
27.00	0.500	0.1040141438 3230104381 3713899754 E 01
28.25	0.625	0.1038216700 3601458768 0056548394 E 01
29.50	0.750	0.1036468726 2924118457 5154685419 E 01
30.75	0.875	0.1034874149 8964796947 2990938990 E 01
32.00	1.000	0.1033413564 2162410494 3493552567 E 01

x	$t = (64/x) - 1$	$xe^{-x}Ei(x)$
∞	-1.000	0.100000000 0000000000 00000000001 E 01
512	-0.875	0.100196079 9450711925 31337468473 E 01
256	-0.750	0.100393713 0905698627 88009078297 E 01
170 2/3	-0.625	0.100592927 5692929112 94663030932 E 01
128	-0.500	0.100793752 4408140182 81776821694 E 01
102 2/5	-0.375	0.100996217 7406449755 74367545570 E 01
85 1/3	-0.250	0.101200354 5332988482 01864466702 E 01
73 1/7	-0.125	0.101406194 9696971331 45942329335 E 01
64	-0.000	0.101613772 3494325321 70357100831 E 01
56 8/9	0.125	0.101823121 1884832696 82337017143 E 01
51 1/5	0.250	0.102034277 2930783774 87217829808 E 01
46 6/11	0.375	0.102247277 8405420595 91275364791 E 01
42 2/3	0.500	0.102462161 4681078391 01187804247 E 01
39 5/13	0.625	0.102678968 3709028524 50984510823 E 01
36 4/7	0.750	0.102897740 4105808008 63378435059 E 01
34 2/15	0.875	0.103118521 2364659263 55875784663 E 01
32	1.000	0.103341356 4216241049 43493552567 E 01

5.9.5 The Fresnel Integral[?, ?]

The Fresnel function is

$$C(x) - iS(x) = \int_0^x i^{-t^2}\ dt = \int_0^x \exp(-i\pi t^2/2)\ dt$$

We compare Axiom's results to Pearcey's tables which show the fresnel results to 6 decimal places. Computation of these values requires floats as the range quickly exceeds DoubleFloat. In each decade of the range we increase the number of terms by a factor of 10. So we compute with 10 terms in the range 0.0-10.0, 100 terms in 10.0-20.0, etc.

fresnelC

The fresnelC is the real portion of the Fresnel integral, C(u), is defined as:

$$C(\sqrt{2x/\pi}) = \frac{1}{2}\int_0^x J_{-\frac{1}{2}}(t)\ dt = \frac{1}{\sqrt{(2\pi)}}\int_0^x \frac{\cos(t)}{\sqrt{t}}\ dt$$

where $J_{-\frac{1}{2}}(t)$ is the Bessel function of the first kind of order $-\frac{1}{2}$.

This is related to the better known definition of C(u), namely:

$$C(u) = \int_0^u \cos\frac{\pi t^2}{2}\ dt$$

where $x = \pi u^2/2$, or $u = (2x/\pi)^{1/2}$

fresnelC is an analytic function of z with z=0 as a two-sheeted branch point. Along the positive real axis the real definition gives:

$$C(0) = 0$$

$$\lim_{x\to+\infty} C(x) = \frac{1}{2}$$

The asymptotic behavior of the function in the corner |arc z| $\leq \pi - \epsilon$, ($\epsilon > 0$), for $|z| \gg 1$ is given by

$$C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}}\left(1 - \frac{1\cdot 3}{(2z)^2} + \frac{1\cdot 3\cdot 5\cdot 7}{(2z)^4} - \cdots\right) - \frac{\cos z}{\sqrt{2\pi z}}\left(\frac{1}{(2z)} - \frac{1\cdot 3\cdot 5}{(2z)^3} + \cdots\right)$$

(Note: Pearcey has a sign error for the second term ([?],p7)

The first approximation is

$$C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}}$$

Axiom uses the power series at the zero point:

$$C(z) = \sqrt{\frac{2z}{\pi}}\sum_{k=0}^{n}(-1)^k\frac{z^{2k}}{(4k+1)(2k)!}$$

— package DFSFUN DoubleFloatSpecialFunctions —

```
fresnelC(z:F):F ==
  z < 0 => error "fresnelC not defined for negative argument"
  z = 0 => 0
  n:PI:= 100
  sqrt((2.0/pi()$F)*z)*_
   reduce(_+,[(-1)**k*z**(2*k)/(factorial(2*k)*(4*k+1))_
     for k in 0..n])$LF
```

———-

fresnelS

The fresnelS is the complex portion of the Fresnel integral, S(u), is defined as:

$$S(\sqrt{2x/\pi}) = \frac{1}{2}\int_0^x J_{\frac{1}{2}}(t)\ dt = \frac{1}{\sqrt{(2\pi)}}\int_0^x \frac{\sin(t)}{\sqrt{t}}\ dt$$

where $J_{\frac{1}{2}}(t)$ is the Bessel function of the first kind of order $\frac{1}{2}$.

This is related to the better known definition of S(u), namely:

$$S(u) = \int_0^u \sin\frac{\pi t^2}{2}\ dt$$

where $x = \pi u^2/2$, or $u = (2x/\pi)^{1/2}$

fresnelS is an analytic function of z with z=0 as a two-sheeted branch point. Along the positive real axis the real definition gives:

$$S(0) = 0$$

$$\lim_{x\to+\infty} S(x) = \frac{1}{2}$$

The asymptotic behavior of the function in the corner |arc z| $\leq \pi - \epsilon$, ($\epsilon > 0$), for $|z| \gg 1$ is given by

$$S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}}\left(1 - \frac{1\cdot 3}{(2z)^2} + \frac{1\cdot 3\cdot 5\cdot 7}{(2z)^4} - \cdots\right) - \frac{\sin z}{\sqrt{2\pi z}}\left(\frac{1}{(2z)} - \frac{1\cdot 3\cdot 5}{(2z)^3} + \cdots\right)$$

The first approximation is

$$S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}}$$

Axiom uses the power series at the zero point:

$$S(z) = \sqrt{\frac{2z}{\pi}} \sum_{k=0}^{n} (-1)^k \frac{z^{2k+1}}{(4k+3)(2k+1)!}$$

— **package DFSFUN DoubleFloatSpecialFunctions** —

```
fresnelS(z:F):F ==
  z < 0 => error "fresnelS not defined for negative argument"
  z = 0 => 0
  n:PI:= 100
  sqrt((2.0/pi()$F)*z)*_
   reduce(_+,
    [(-1)**k*(z**(2*k+1))/(factorial(2*k+1)*(4*k+3)) _
     for k in 0..n])$LF
```

———-

— **package DFSFUN DoubleFloatSpecialFunctions** —

```
polygamma(k,z)  == CPSI(k, z)$Lisp
polygamma(k,x)  == RPSI(k, x)$Lisp

logGamma z      == CLNGAMMA(z)$Lisp
logGamma x      == RLNGAMMA(x)$Lisp

besselJ(v,z)    == CBESSELJ(v,z)$Lisp
besselJ(n,x)    == RBESSELJ(n,x)$Lisp

besselI(v,z)    == CBESSELI(v,z)$Lisp
besselI(n,x)    == RBESSELI(n,x)$Lisp

hypergeometric0F1(a,z) == CHYPER0F1(a, z)$Lisp
hypergeometric0F1(n,x) == retract hypergeometric0F1(n::C, x::C)


-- All others are defined in terms of these.
digamma x == polygamma(0, x)
digamma z == polygamma(0, z)

Beta(x,y) == Gamma(x)*Gamma(y)/Gamma(x+y)
Beta(w,z) == Gamma(w)*Gamma(z)/Gamma(w+z)

fuzz := (10::R)**(-7)

import IntegerRetractions(R)
import IntegerRetractions(C)
```

```
besselY(n,x) ==
    if integer? n then n := n + fuzz
    vp := n * pi()$R
    (cos(vp) * besselJ(n,x) - besselJ(-n,x) )/sin(vp)
besselY(v,z) ==
    if integer? v then v := v + fuzz::C
    vp := v * pi()$C
    (cos(vp) * besselJ(v,z) - besselJ(-v,z) )/sin(vp)

besselK(n,x) ==
    if integer? n then n := n + fuzz
    p    := pi()$R
    vp   := n*p
    ahalf:= 1/(2::R)
    p * ahalf * ( besselI(-n,x) - besselI(n,x) )/sin(vp)
besselK(v,z) ==
    if integer? v then v := v + fuzz::C
    p    := pi()$C
    vp   := v*p
    ahalf:= 1/(2::C)
    p * ahalf * ( besselI(-v,z) - besselI(v,z) )/sin(vp)

airyAi x ==
    ahalf  := recip(2::R)::R
    athird := recip(3::R)::R
    eta := 2 * athird * (-x) ** (3*ahalf)
    (-x)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))
airyAi z ==
    ahalf  := recip(2::C)::C
    athird := recip(3::C)::C
    eta := 2 * athird * (-z) ** (3*ahalf)
    (-z)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))

airyBi x ==
    ahalf  := recip(2::R)::R
    athird := recip(3::R)::R
    eta := 2 * athird * (-x) ** (3*ahalf)
    (-x*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )

airyBi z ==
    ahalf  := recip(2::C)::C
    athird := recip(3::C)::C
    eta := 2 * athird * (-z) ** (3*ahalf)
    (-z*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )


    ———————-
```

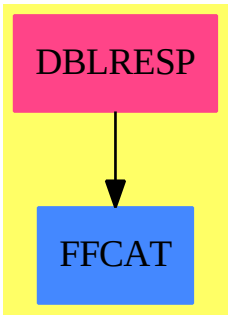
— **DFSFUN.dotabb** —

```
"DFSFUN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DFSFUN"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"DFSFUN" -> "COMPCAT"
```

———-

5.10 package DBLRESP DoubleResultantPackage

5.10.1 DoubleResultantPackage (DBLRESP)



Exports:
doubleResultant

— **package DBLRESP DoubleResultantPackage** —

```
)abbrev package DBLRESP DoubleResultantPackage
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 July 1990
++ Description:
++ This package provides functions for computing the residues
++ of a function on an algebraic curve.

DoubleResultantPackage(F, UP, UPUP, R): Exports == Implementation where
  F   : Field
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R   : FunctionFieldCategory(F, UP, UPUP)

  RF  ==> Fraction UP
  UP2 ==> SparseUnivariatePolynomial UP
  UP3 ==> SparseUnivariatePolynomial UP2
```

```
  Exports ==> with
    doubleResultant: (R, UP -> UP) -> UP
      ++ doubleResultant(f, ') returns p(x) whose roots are
      ++ rational multiples of the residues of f at all its
      ++ finite poles. Argument ' is the derivation to use.

  Implementation ==> add
    import CommuteUnivariatePolynomialCategory(F, UP, UP2)
    import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

    UP22   : UP   -> UP2
    UP23   : UPUP -> UP3
    remove0: UP   -> UP             -- removes the power of x dividing p

    remove0 p ==
      primitivePart((p exquo monomial(1, minimumDegree p))::UP)

    UP22 p ==
      map(x+->x::UP, p)$UnivariatePolynomialCategoryFunctions2(F,UP,UP,UP2)

    UP23 p ==
      map(x+->UP22(retract(x)@UP),p)_
         $UnivariatePolynomialCategoryFunctions2(RF, UPUP, UP2, UP3)

    doubleResultant(h, derivation) ==
      cd := splitDenominator lift h
      d  := (cd.den exquo (g := gcd(cd.den, derivation(cd.den))))::UP
      r  := swap primitivePart swap resultant(UP23(cd.num)
          - ((monomial(1, 1)$UP :: UP2) * UP22(g * derivation d))::UP3,
                                              UP23 definingPolynomial())
      remove0 resultant(r, UP22 d)
```

———-

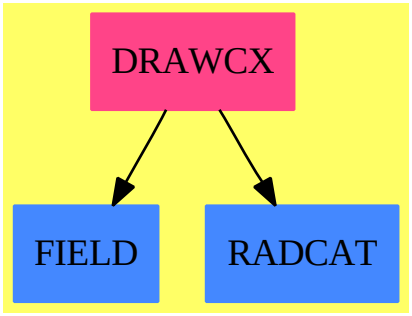
— DBLRESP.dotabb —

```
"DBLRESP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DBLRESP"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"DBLRESP" -> "FFCAT"
```

———-

5.11 package DRAWCX DrawComplex

5.11.1 DrawComplex (DRAWCX)



Exports:
setImagSteps setRealSteps drawComplex drawComplexVectorField setClipValue

— package DRAWCX DrawComplex —

```
)abbrev package DRAWCX DrawComplex
++ Description:
++ \axiomType{DrawComplex} provides some facilities
++ for drawing complex functions.

DrawComplex(): Exports == Implementation where
  C ==> Complex DoubleFloat
  S ==> Segment DoubleFloat
  PC ==> Record(rr:SF, th:SF)
  INT ==> Integer
  SF ==> DoubleFloat
  NNI ==> NonNegativeInteger
  VIEW3D ==> ThreeDimensionalViewport
  ARRAY2 ==> TwoDimensionalArray
  Exports == with
    drawComplex: (C -> C,S,S,Boolean) -> VIEW3D
      ++ drawComplex(f,rRange,iRange,arrows?)
      ++ draws a complex function as a height field.
      ++ It uses the complex norm as the height and the complex
      ++ argument as the color.
      ++ It will optionally draw arrows on the surface indicating the direction
      ++ of the complex value.
      ++ Sample call:
      ++   \spad{f z == exp(1/z)}
      ++   \spad{drawComplex(f, 0.3..3, 0..2*%pi, false)}
      ++ Parameter descriptions:
      ++   f:  the function to draw
```

```
    ++   rRange : the range of the real values
    ++   iRange : the range of imaginary values
    ++   arrows? : a flag indicating whether to draw the phase arrows for f
    ++ Call the functions \axiomFunFrom{setRealSteps}{DrawComplex} and
    ++ \axiomFunFrom{setImagSteps}{DrawComplex} to change the
    ++ number of steps used in each direction.
  drawComplexVectorField: (C -> C,S,S) -> VIEW3D
    ++ drawComplexVectorField(f,rRange,iRange)
    ++ draws a complex vector field using arrows on the \spad{x--y} plane.
    ++ These vector fields should be viewed from the top by pressing the
    ++ "XY" translate button on the 3-d viewport control panel.
    ++ Sample call:
    ++    \spad{f z == sin z}
    ++    \spad{drawComplexVectorField(f, -2..2, -2..2)}
    ++ Parameter descriptions:
    ++   f : the function to draw
    ++   rRange : the range of the real values
    ++   iRange : the range of the imaginary values
    ++ Call the functions \axiomFunFrom{setRealSteps}{DrawComplex} and
    ++ \axiomFunFrom{setImagSteps}{DrawComplex} to change the
    ++ number of steps used in each direction.
  setRealSteps: INT -> INT
    ++ setRealSteps(i)
    ++ sets to i the number of steps to use in the real direction
    ++ when drawing complex functions. Returns i.
  setImagSteps: INT -> INT
    ++ setImagSteps(i)
    ++ sets to i  the number of steps to use in the imaginary direction
    ++ when drawing complex functions. Returns i.
  setClipValue: SF-> SF
    ++ setClipValue(x)
    ++ sets to x the maximum value to plot when drawing complex functions. Returns x.
Implementation == add
  -- relative size of the arrow head compared to the length of the arrow
  arrowScale : SF := (0.125)::SF
  arrowAngle: SF := pi()-pi()/(20::SF)    -- angle of the arrow head
  realSteps: INT  := 11     -- the number of steps in the real direction
  imagSteps: INT  := 11     -- the number of steps in the imaginary direction
  clipValue: SF  := 10::SF -- the maximum length of a vector to draw


  -- Add an arrow head to a line segment, which starts at 'p1', ends at 'p2',
  -- has length 'len', and and angle 'arg'.  We pass 'len' and 'arg' as
  -- arguments since thet were already computed by the calling program
  makeArrow(p1:Point SF, p2:Point SF, len: SF, arg:SF):List List Point SF ==
    c1 := cos(arg + arrowAngle)
    s1 := sin(arg + arrowAngle)
    c2 := cos(arg - arrowAngle)
    s2 := sin(arg - arrowAngle)
    p3 := point [p2.1 + c1*arrowScale*len, p2.2 + s1*arrowScale*len,
```

```
              p2.3, p2.4]
  p4 := point [p2.1 + c2*arrowScale*len, p2.2 + s2*arrowScale*len,
              p2.3, p2.4]
  [[p1, p2, p3], [p2, p4]]

-- clip a value in the interval (-clip...clip)
clipFun(x:SF):SF ==
  min(max(x, -clipValue), clipValue)

drawComplex(f, realRange, imagRange, arrows?) ==
  delReal := (hi(realRange) - lo(realRange))/realSteps::SF
  delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
  funTable: ARRAY2(PC) :=
     new((realSteps::NNI)+1, (imagSteps::NNI)+1, [0,0]$PC)
  real := lo(realRange)
  for i in 1..realSteps+1 repeat
    imag := lo(imagRange)
    for j in 1..imagSteps+1 repeat
      z := f complex(real, imag)
      funTable(i,j) := [clipFun(sqrt norm z), argument(z)]$PC
      imag := imag + delImag
    real := real + delReal
  llp := empty()$(List List Point SF)
  real := lo(realRange)
  for i in 1..realSteps+1 repeat
    imag := lo(imagRange)
    lp := empty()$(List Point SF)
    for j in 1..imagSteps+1 repeat
      p := point [real, imag, funTable(i,j).rr, funTable(i,j).th]
      lp := cons(p, lp)
      imag := imag + delImag
    real := real + delReal
    llp := cons(lp, llp)
  space := mesh(llp)$(ThreeSpace SF)
  if arrows? then
    real := lo(realRange)
    for i in 1..realSteps+1 repeat
      imag := lo(imagRange)
      for j in 1..imagSteps+1 repeat
        arg := funTable(i,j).th
        p1 := point [real,imag, funTable(i,j).rr, arg]
        len := delReal*2.0::SF
        p2 := point [p1.1 + len*cos(arg), p1.2 + len*sin(arg),
                     p1.3, p1.4]
        arrow := makeArrow(p1, p2, len, arg)
        for a in arrow repeat curve(space, a)$(ThreeSpace SF)
        imag := imag + delImag
      real := real + delReal
  makeViewport3D(space, "Complex Function")$VIEW3D
```

```
drawComplexVectorField(f, realRange, imagRange): VIEW3D ==
  -- compute the steps size of the grid
  delReal := (hi(realRange) - lo(realRange))/realSteps::SF
  delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
  -- create the space to hold the arrows
  space := create3Space()$(ThreeSpace SF)
  real := lo(realRange)
  for i in 1..realSteps+1 repeat
    imag := lo(imagRange)
    for j in 1..imagSteps+1 repeat
      -- compute the function
      z := f complex(real, imag)
      -- get the direction of the arrow
      arg := argument z
      -- get the length of the arrow
      len := clipFun(sqrt norm z)
      -- create point at the base of the arrow
      p1 :=  point [real, imag, 0::SF, arg]
      -- scale the arrow length so it isn't too long
      scaleLen := delReal * len
      -- create the point at the top of the arrow
      p2 := point [p1.1 + scaleLen*cos(arg), p1.2 + scaleLen*sin(arg),
                   0::SF, arg]
      -- make the pointer at the top of the arrow
      arrow := makeArrow(p1, p2, scaleLen, arg)
      -- add the line segments in the arrow to the space
      for a in arrow repeat curve(space, a)$(ThreeSpace SF)
      imag := imag + delImag
    real := real + delReal
  -- draw the vector feild
  makeViewport3D(space, "Complex Vector Field")$VIEW3D

-- set the number of steps to use in the real direction
setRealSteps(n) ==
  realSteps := n

-- set the number of steps to use in the imaginary direction
setImagSteps(n) ==
  imagSteps := n

-- set the maximum value to plot
setClipValue clip ==
  clipValue := clip
```

———-

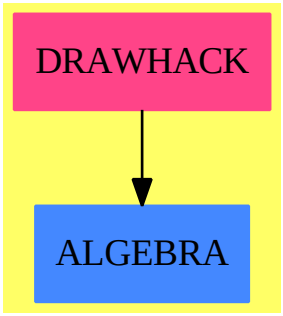
— DRAWCX.dotabb —

```
"DRAWCX" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWCX"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"DRAWCX" -> "FIELD"
"DRAWCX" -> "RADCAT"
```

———-

5.12 package DRAWHACK DrawNumericHack

5.12.1 DrawNumericHack (DRAWHACK)



Exports:
coerce

— package DRAWHACK DrawNumericHack —

```
)abbrev package DRAWHACK DrawNumericHack
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 21 Feb 1990
++ Basic Operations: coerce
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Hack for the draw interface. DrawNumericHack provides
++ a "coercion" from something of the form \spad{x = a..b} where \spad{a}
++ and b are
++ formal expressions to a binding of the form \spad{x = c..d} where c and d
++ are the numerical values of \spad{a} and b. This "coercion" fails if
++ \spad{a} and b contains symbolic variables, but is meant for expressions
```

```
++ involving %pi.
++ Note that this package is meant for internal use only.

DrawNumericHack(R:Join(OrderedSet,IntegralDomain,ConvertibleTo Float)):
 with coerce: SegmentBinding Expression R -> SegmentBinding Float
        ++ coerce(x = a..b) returns \spad{x = c..d} where c and d are the
        ++ numerical values of \spad{a} and b.
  == add
   coerce s ==
     map(numeric$Numeric(R),s)$SegmentBindingFunctions2(Expression R, Float)
```

———-

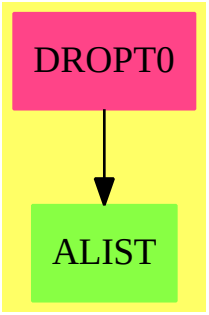
— **DRAWHACK.dotabb** —

```
"DRAWHACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWHACK"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"DRAWHACK" -> "ALGEBRA"
```

———-

5.13 package DROPT0 DrawOptionFunctions0

5.13.1 DrawOptionFunctions0 (DROPT0)



Exports:

adaptive	clipBoolean	coord	curveColorPalette	pointColorPalette
ranges	space	style	title	toScale
tubePoints	tubeRadius	units	var1Steps	var2Steps
viewpoint				

— **package DROPT0 DrawOptionFunctions0** —

```
)abbrev package DROPT0 DrawOptionFunctions0
++ Description:
++ This package has no description
-- The functions here are not in DrawOptions since they are not
-- visible to the interpreter.

DrawOptionFunctions0(): Exports == Implementation where
 RANGE ==> List Segment Float
 UNIT  ==> List Float
 PAL   ==> Palette
 POINT ==> Point(DoubleFloat)
 SEG   ==> Segment Float
 SF     ==> DoubleFloat
 SPACE3 ==> ThreeSpace(DoubleFloat)
 VIEWPT ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, delt

 Exports ==> with
  adaptive: (List DrawOption, Boolean) -> Boolean
    ++ adaptive(l,b) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{adaptive}.
    ++ If the option does not exist the value, b is returned.
  clipBoolean: (List DrawOption, Boolean) -> Boolean
    ++ clipBoolean(l,b) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{clipBoolean}.
    ++ If the option does not exist the value, b is returned.
  viewpoint: (List DrawOption, VIEWPT) -> VIEWPT
    ++ viewpoint(l,ls) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{viewpoint}.
    ++ IF the option does not exist, the value ls is returned.
  title: (List DrawOption, String) -> String
    ++ title(l,s) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{title}.
    ++ If the option does not exist the value, s is returned.
  style: (List DrawOption, String) -> String
    ++ style(l,s) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{style}.
    ++ If the option does not exist the value, s is returned.
  toScale: (List DrawOption, Boolean) -> Boolean
    ++ toScale(l,b) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{toScale}.
    ++ If the option does not exist the value, b is returned.

  pointColorPalette: (List DrawOption,PAL) -> PAL
    ++ pointColorPalette(l,p) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{pointColorPalette}.
    ++ If the option does not exist the value, p is returned.
  curveColorPalette: (List DrawOption,PAL) -> PAL
    ++ curveColorPalette(l,p) takes the list of draw options, l, and checks
    ++ the list to see if it contains the option \spad{curveColorPalette}.
    ++ If the option does not exist the value, p is returned.
```

```
 ranges: (List DrawOption, RANGE) -> RANGE
   ++ ranges(l,r) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{ranges}.
   ++ If the option does not exist the value, r is returned.
 var1Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
   ++ var1Steps(l,n) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{var1Steps}.
   ++ If the option does not exist the value, n is returned.
 var2Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
   ++ var2Steps(l,n) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{var2Steps}.
   ++ If the option does not exist the value, n is returned.
 space: (List DrawOption) -> SPACE3
   ++ space(l) takes a list of draw options, l, and checks to see
   ++ if it contains the option \spad{space}.  If the the option
   ++ doesn't exist, then an empty space is returned.
 tubePoints : (List DrawOption, PositiveInteger) -> PositiveInteger
   ++ tubePoints(l,n) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{tubePoints}.
   ++ If the option does not exist the value, n is returned.
 tubeRadius : (List DrawOption, Float) -> Float
   ++ tubeRadius(l,n) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{tubeRadius}.
   ++ If the option does not exist the value, n is returned.
 coord: (List DrawOption, (POINT->POINT)) -> (POINT->POINT)
   ++ coord(l,p) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{coord}.
   ++ If the option does not exist the value, p is returned.
 units: (List DrawOption, UNIT) -> UNIT
   ++ units(l,u) takes the list of draw options, l, and checks
   ++ the list to see if it contains the option \spad{unit}.
   ++ If the option does not exist the value, u is returned.

Implementation ==> add
 adaptive(l,s) ==
   (u := option(l, "adaptive"::Symbol)$DrawOptionFunctions1(Boolean))
     case "failed" => s
   u::Boolean

 clipBoolean(l,s) ==
   (u := option(l, "clipBoolean"::Symbol)$DrawOptionFunctions1(Boolean))
     case "failed" => s
   u::Boolean

 title(l, s) ==
   (u := option(l, "title"::Symbol)$DrawOptionFunctions1(String))
     case "failed" => s
   u::String
```

```
viewpoint(l, vp) ==
  (u := option(l, "viewpoint"::Symbol)$DrawOptionFunctions1(VIEWPT))
    case "failed" => vp
  u::VIEWPT

style(l, s) ==
  (u := option(l, "style"::Symbol)$DrawOptionFunctions1(String))
    case "failed" => s
  u::String

toScale(l,s) ==
  (u := option(l, "toScale"::Symbol)$DrawOptionFunctions1(Boolean))
    case "failed" => s
  u::Boolean

pointColorPalette(l,s) ==
  (u := option(l, "pointColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
    case "failed" => s
  u::PAL

curveColorPalette(l,s) ==
  (u := option(l, "curveColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
    case "failed" => s
  u::PAL



ranges(l, s) ==
  (u := option(l, "ranges"::Symbol)$DrawOptionFunctions1(RANGE))
    case "failed" => s
  u::RANGE

space(l) ==
  (u := option(l, "space"::Symbol)$DrawOptionFunctions1(SPACE3))
    case "failed" => create3Space()$SPACE3
  u::SPACE3

var1Steps(l,s) ==
  (u := option(l, "var1Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
    case "failed" => s
  u::PositiveInteger

var2Steps(l,s) ==
  (u := option(l, "var2Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
    case "failed" => s
  u::PositiveInteger

tubePoints(l,s) ==
  (u := option(l, "tubePoints"::Symbol)$DrawOptionFunctions1(PositiveInteger))
    case "failed" => s
```

```
    u::PositiveInteger

  tubeRadius(l,s) ==
    (u := option(l, "tubeRadius"::Symbol)$DrawOptionFunctions1(Float))
      case "failed" => s
    u::Float

  coord(l,s) ==
    (u := option(l, "coord"::Symbol)$DrawOptionFunctions1(POINT->POINT))
      case "failed" => s
    u::(POINT->POINT)

  units(l,s) ==
    (u := option(l, "unit"::Symbol)$DrawOptionFunctions1(UNIT))
      case "failed" => s
    u::UNIT
```

———-

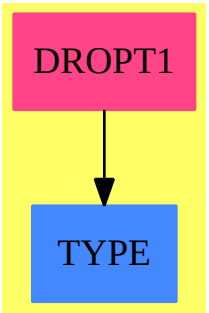
— **DROPT0.dotabb** —

```
"DROPT0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT0"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DROPT0" -> "ALIST"
```

———-

5.14 package DROPT1 DrawOptionFunctions1

5.14.1 DrawOptionFunctions1 (DROPT1)



Exports:
option

— package DROPT1 DrawOptionFunctions1 —

```
)abbrev package DROPT1 DrawOptionFunctions1
++ Description:
++ This package has no description

DrawOptionFunctions1(S:Type): Exports == Implementation where
 RANGE ==> List Segment Float
 UNIT  ==> List Float
 PAL   ==> Palette
 POINT ==> Point(DoubleFloat)
 SEG   ==> Segment Float
 SF     ==> DoubleFloat
 SPACE3 ==> ThreeSpace(DoubleFloat)
 VIEWPT ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, delt

 Exports ==> with
  option: (List DrawOption, Symbol) -> Union(S, "failed")
    ++ option(l,s) determines whether the indicated drawing option, s,
    ++ is contained in the list of drawing options, l, which is defined
    ++ by the draw command.
 Implementation ==> add
  option(l, s) ==
    (u := option(l, s)@Union(Any, "failed")) case "failed" => "failed"
    retract(u::Any)$AnyFunctions1(S)
```

———-

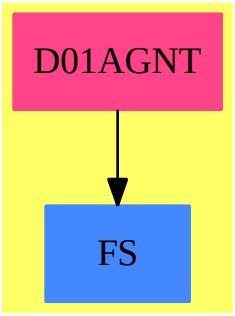
— DROPT1.dotabb —

```
"DROPT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"DROPT1" -> "TYPE"
```

———-

5.15 package D01AGNT d01AgentsPackage

5.15.1 d01AgentsPackage (D01AGNT)



Exports:

changeName	commaSeparate	df2st	functionIsContinuousAtEndPoints	functionIsOscillatory
gethi	getlo	ldf2lst	problemPoints	rangeIsFinite
sdf2lst	singularitiesOf			

— **package D01AGNT d01AgentsPackage** —

```
)abbrev package D01AGNT d01AgentsPackage
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Basic Operations: rangeIsFinite, functionIsContinuousAtEndPoints,
++ functionIsOscillatory
++ Description:
++ \axiomType{d01AgentsPackage} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \axiomFun{measure} of an appropriate numerical integration routine.
++ It contains functions \axiomFun{rangeIsFinite} to test the input range and
++ \axiomFun{functionIsContinuousAtEndPoints} to check for continuity at
++ the end points of the range.

d01AgentsPackage(): E == I where
  EF2   ==> ExpressionFunctions2
  EFI  ==> Expression Fraction Integer
  FI  ==> Fraction Integer
  LEDF  ==> List Expression DoubleFloat
  KEDF  ==> Kernel Expression DoubleFloat
  EEDF  ==> Equation Expression DoubleFloat
  EDF  ==> Expression DoubleFloat
  PDF  ==> Polynomial DoubleFloat
  LDF  ==> List DoubleFloat
  SDF  ==> Stream DoubleFloat
  DF  ==> DoubleFloat
```

```
F  ==> Float
ST  ==> String
LST  ==> List String
SI  ==> SingleInteger
SOCDF  ==> Segment OrderedCompletion DoubleFloat
OCDF  ==> OrderedCompletion DoubleFloat
OCEDF  ==> OrderedCompletion Expression DoubleFloat
EOCEFI  ==> Equation OrderedCompletion Expression Fraction Integer
OCEFI   ==> OrderedCompletion Expression Fraction Integer
OCFI    ==> OrderedCompletion Fraction Integer
NIA  ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
INT  ==> Integer
CTYPE  ==> Union(continuous: "Continuous at the end points",
           lowerSingular: "There is a singularity at the lower end point",
            upperSingular: "There is a singularity at the upper end point",
             bothSingular: "There are singularities at both end points",
              notEvaluated: "End point continuity not yet evaluated")
RTYPE  ==> Union(finite: "The range is finite",
            lowerInfinite: "The bottom of range is infinite",
              upperInfinite: "The top of range is infinite",
                bothInfinite: "Both top and bottom points are infinite",
                  notEvaluated: "Range not yet evaluated")
STYPE  ==> Union(str:SDF,
                 notEvaluated:"Internal singularities not yet evaluated")
ATT  ==> Record(endPointContinuity:CTYPE,
                 singularitiesStream:STYPE,range:RTYPE)
ROA  ==> Record(key:NIA,entry:ATT)

E ==> with

  rangeIsFinite : NIA -> RTYPE
    ++ rangeIsFinite(args) tests the endpoints of \spad{args.range} for
    ++ infinite end points.
  functionIsContinuousAtEndPoints: NIA -> CTYPE
    ++ functionIsContinuousAtEndPoints(args) uses power series limits
    ++ to check for problems at the end points of the range of \spad{args}.
  getlo : SOCDF -> DF
    ++ getlo(x) gets the \axiomType{DoubleFloat} equivalent of
    ++ the first endpoint of the range \axiom{x}
  gethi : SOCDF -> DF
    ++ gethi(x) gets the \axiomType{DoubleFloat} equivalent of
    ++ the second endpoint of the range \axiom{x}
  functionIsOscillatory:NIA -> F
    ++ functionIsOscillatory(a) tests whether the function \spad{a.fn}
    ++ has many zeros of its derivative.
  problemPoints: (EDF, Symbol, SOCDF) -> List DF
    ++ problemPoints(f,var,range) returns a list of possible problem points
    ++ by looking at the zeros of the denominator of the function if it
    ++ can be retracted to \axiomType{Polynomial DoubleFloat}.
  singularitiesOf:NIA -> SDF
```

```
      ++ singularitiesOf(args) returns a list of potential
      ++ singularities of the function within the given range
    df2st:DF -> String
      ++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
    ldf2lst:LDF -> LST
      ++ ldf2lst(ln) coerces a List of \axiomType{DoubleFloat} to
      ++ \axiomType{List String}
    sdf2lst:SDF -> LST
      ++ sdf2lst(ln) coerces a Stream of \axiomType{DoubleFloat} to
      ++ \axiomType{List String}
    commaSeparate:LST -> ST
      ++ commaSeparate(l) produces a comma separated string from a
      ++ list of strings.
    changeName:(Symbol,Symbol,Result) -> Result
      ++ changeName(s,t,r) changes the name of item \axiom{s} in \axiom{r}
      ++ to \axiom{t}.

  I ==> ExpertSystemContinuityPackage add

    import ExpertSystemToolsPackage
    import ExpertSystemContinuityPackage

    -- local functions
    ocdf2ocefi : OCDF -> OCEFI
    rangeOfArgument : (KEDF, NIA) -> DF
    continuousAtPoint? : (EFI,EOCEFI) -> Boolean
    rand:(SOCDF,INT) -> LDF
    eval:(EDF,Symbol,LDF) -> LDF
    numberOfSignChanges:LDF -> INT
    rangeIsFiniteFunction:NIA -> RTYPE
    functionIsContinuousAtEndPointsFunction:NIA -> CTYPE

    changeName(s:Symbol,t:Symbol,r:Result):Result ==
      a := remove!(s,r)$Result
      a case Any =>
        insert!([t,a],r)$Result
        r
      r

    commaSeparate(l:LST):ST ==
      empty?(l)$LST => ""
--      one?(#(l)) => concat(l)$ST
      (#(l) = 1) => concat(l)$ST
      f := first(l)$LST
      t := [concat([", ",l.i])$ST for i in 2..#(l)]
      concat(f,concat(t)$ST)$ST

    rand(seg:SOCDF,n:INT):LDF ==
      -- produced a sorted list of random numbers in the given range
      l:DF := getlo seg
```

```
    s:DF := (gethi seg) - l
    seed:INT := random()$INT
    dseed:DF := seed :: DF
    r:LDF := [(((random(seed)$INT) :: DF)*s/dseed + l) for i in 1..n]
    sort(r)$LDF

  eval(f:EDF,var:Symbol,l:LDF):LDF ==
    empty?(l)$LDF => [0$DF]
    ve := var::EDF
    [retract(eval(f,equation(ve,u::EDF)$EEDF)$EDF)@DF for u in l]

  numberOfSignChanges(l:LDF):INT ==
    -- calculates the number of sign changes in a list
    a := 0$INT
    empty?(l)$LDF => 0
    for i in 2..# l repeat
      if negative?(l.i*l.(i-1))  then
        a := a + 1
    a

  rangeOfArgument(k: KEDF, args:NIA): DF ==
    Args := copy args
    Args.fn := arg := first(argument(k)$KEDF)$LEDF
    functionIsContinuousAtEndPoints(Args) case continuous =>
      r:SOCDF := args.range
      low:EDF := (getlo r) :: EDF
      high:EDF := (gethi r) :: EDF
      eql := equation(a := args.var :: EDF, low)$EEDF
      eqh := equation(a, high)$EEDF
      e1 := (numeric(eval(arg,eql)$EDF)$Numeric(DF)) :: DF
      e2 := (numeric(eval(arg,eqh)$EDF)$Numeric(DF)) :: DF
      e2-e1
    0$DF

  ocdf2ocefi(r:OCDF):OCEFI ==
    finite?(r)$OCDF => (edf2efi(((retract(r)@DF)$OCDF)::EDF))::OCEFI
    r pretend OCEFI

  continuousAtPoint?(f:EFI,e:EOCEFI):Boolean ==
    (l := limit(f,e)$PowerSeriesLimitPackage(FI,EFI)) case OCEFI =>
                    finite?(l :: OCEFI)
    -- if the left hand limit equals the right hand limit, or if neither
    -- side has a limit at this point, the return type of  limit() is
    -- Union(Ordered Completion Expression Fraction Integer,"failed")
    false

  -- exported functions

  rangeIsFiniteFunction(args:NIA): RTYPE ==
    -- rangeIsFinite(x) tests the endpoints of x.range for infinite
```

```
  -- end points.
  --             [-inf,  inf]  =>  4
  --             [ x  ,  inf]  =>  3
  --             [-inf,  x  ]  =>  1
  --             [ x  ,  y  ]  =>  0
  fr:SI := (3::SI * whatInfinity(hi(args.range))$OCDF
                  - whatInfinity(lo(args.range))$OCDF)
  fr = 0 => ["The range is finite"]
  fr = 1 => ["The bottom of range is infinite"]
  fr = 3 => ["The top of range is infinite"]
  fr = 4 => ["Both top and bottom points are infinite"]
  error("rangeIsFinite",["this is not a valid range"])$ErrorFunctions

rangeIsFinite(args:NIA): RTYPE ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s := coerce(t)@ATT
    s.range case notEvaluated =>
      s.range := rangeIsFiniteFunction(nia)
      r:ROA := [nia,s]
      insert!(r)$IntegrationFunctionsTable
      s.range
    s.range
  a:ATT := [["End point continuity not yet evaluated"],
              ["Internal singularities not yet evaluated"],
                  e:=rangeIsFiniteFunction(nia)]
  r:ROA := [nia,a]
  insert!(r)$IntegrationFunctionsTable
  e

functionIsContinuousAtEndPointsFunction(args:NIA):CTYPE ==

  v := args.var :: EFI :: OCEFI
  high:OCEFI := ocdf2ocefi(hi(args.range))
  low:OCEFI := ocdf2ocefi(lo(args.range))
  f := edf2efi(args.fn)
  l:Boolean := continuousAtPoint?(f,equation(v,low)$EOCEFI)
  h:Boolean := continuousAtPoint?(f,equation(v,high)$EOCEFI)
  l and h => ["Continuous at the end points"]
  l => ["There is a singularity at the upper end point"]
  h => ["There is a singularity at the lower end point"]
  ["There are singularities at both end points"]

functionIsContinuousAtEndPoints(args:NIA): CTYPE ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s := coerce(t)@ATT
    s.endPointContinuity case notEvaluated =>
      s.endPointContinuity := functionIsContinuousAtEndPointsFunction(nia)
      r:ROA := [nia,s]
```

```
      insert!(r)$IntegrationFunctionsTable
      s.endPointContinuity
    s.endPointContinuity
  a:ATT := [e:=functionIsContinuousAtEndPointsFunction(nia),
             ["Internal singularities not yet evaluated"],
               ["Range not yet evaluated"]]
  r:ROA := [nia,a]
  insert!(r)$IntegrationFunctionsTable
  e

functionIsOscillatory(a:NIA):F ==

  args := copy a
  k := tower(numerator args.fn)$EDF
  p:F := pi()$F
  for i in 1..# k repeat
    is?(ker := k.i, sin :: Symbol) =>
      ra := convert(rangeOfArgument(ker,args))@F
      ra > 2*p => return (ra/p)
    is?(ker, cos :: Symbol) =>
      ra := convert(rangeOfArgument(ker,args))@F
      ra > 2*p => return (ra/p)
  l:LDF := rand(args.range,30)
  l := eval(args.fn,args.var,l)
  numberOfSignChanges(l) :: F

singularitiesOf(args:NIA):SDF ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s:ATT := coerce(t)@ATT
    p:STYPE := s.singularitiesStream
    p case str => p.str
    e:SDF := singularitiesOf(nia.fn,[nia.var],nia.range)
    if not empty?(e) then
      if less?(e,10)$SDF then extend(e,10)$SDF
    s.singularitiesStream := [e]
    r:ROA := [nia,s]
    insert!(r)$IntegrationFunctionsTable
    e
  e:=singularitiesOf(nia.fn,[nia.var],nia.range)
  if not empty?(e) then
    if less?(e,10)$SDF then extend(e,10)$SDF
  a:ATT := [["End point continuity not yet evaluated"],[e],
                      ["Range not yet evaluated"]]
  r:ROA := [nia,a]
  insert!(r)$IntegrationFunctionsTable
  e
```

———-

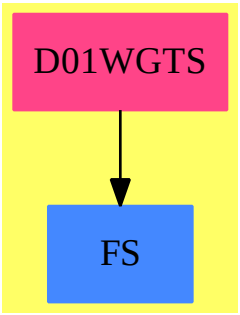
— D01AGNT.dotabb —

```
"D01AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D01AGNT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D01AGNT" -> "FS"
```

———-

5.16 package D01WGTS d01WeightsPackage

5.16.1 d01WeightsPackage (D01WGTS)



Exports:
exprHasAlgebraicWeight exprHasLogarithmicWeights exprHasWeightCosWXorSinWX

— package D01WGTS d01WeightsPackage —

```
)abbrev package D01WGTS d01WeightsPackage
++ Author: Brian Dupee
++ Date Created: July 1994
++ Date Last Updated: January 1998 (Bug fix - exprHasListOfWeightsCosWXorSinWX)
++ Basic Operations: exprHasWeightCosWXorSinWX, exprHasAlgebraicWeight,
++ exprHasLogarithmicWeights
++ Description:
++ \axiom{d01WeightsPackage} is a package for functions used to investigate
++ whether a function can be divided into a simpler function and a weight
++ function.  The types of weights investigated are those giving rise to
++ end-point singularities of the algebraico-logarithmic type, and
++ trigonometric weights.

d01WeightsPackage(): E == I where
  LEDF  ==> List Expression DoubleFloat
```

```
KEDF  ==> Kernel Expression DoubleFloat
LKEDF  ==> List Kernel Expression DoubleFloat
EDF  ==> Expression DoubleFloat
PDF  ==> Polynomial DoubleFloat
FI  ==> Fraction Integer
LDF  ==> List DoubleFloat
DF  ==> DoubleFloat
SOCDF  ==> Segment OrderedCompletion DoubleFloat
OCDF  ==> OrderedCompletion DoubleFloat
NIA  ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
INT  ==> Integer
BOP  ==> BasicOperator
URBODF  ==> Union(Record(op:BasicOperator,w:DF),"failed")
LURBODF  ==> List(Union(Record(op:BasicOperator,w:DF), "failed"))

E ==> with
  exprHasWeightCosWXorSinWX:NIA -> URBODF
    ++ \axiom{exprHasWeightCosWXorSinWX} looks for trigonometric
    ++ weights in an expression of the form \axiom{cos \omega x} or
    ++ \axiom{sin \omega x}, returning the value of \omega
    ++ (\notequal 1) and the operator.
  exprHasAlgebraicWeight:NIA -> Union(LDF,"failed")
    ++ \axiom{exprHasAlgebraicWeight} looks for algebraic weights
    ++ giving rise to singularities of the function at the end-points.
  exprHasLogarithmicWeights:NIA -> INT
    ++ \axiom{exprHasLogarithmicWeights} looks for logarithmic weights
    ++ giving rise to singularities of the function at the end-points.



I ==> add
  score:(EDF,EDF) -> FI
  kernelIsLog:KEDF -> Boolean
  functionIsPolynomial?:EDF -> Boolean
  functionIsNthRoot?:(EDF,EDF) -> Boolean
  functionIsQuotient:EDF -> Union(EDF,"failed")
  findCommonFactor:LEDF -> Union(LEDF,"failed")
  findAlgebraicWeight:(NIA,EDF) -> Union(DF,"failed")
  exprHasListOfWeightsCosWXorSinWX:(EDF,Symbol) -> LURBODF
  exprOfFormCosWXorSinWX:(EDF,Symbol) -> URBODF
  bestWeight:LURBODF -> URBODF
  weightIn?:(URBODF,LURBODF) -> Boolean
  inRest?:(EDF,LEDF)->Boolean
  factorIn?:(EDF,LEDF)->Boolean
  voo?:(EDF,EDF)->Boolean

  kernelIsLog(k:KEDF):Boolean ==
    (name k = (log :: Symbol))@Boolean

  factorIn?(a:EDF,l:LEDF):Boolean ==
```

```
      for i in 1..# l repeat
        (a = l.i)@Boolean => return true
      false

    voo?(b:EDF,a:EDF):Boolean ==
       (voo:=isTimes(b)) case LEDF and factorIn?(a,voo)

    inRest?(a:EDF,l:LEDF):Boolean ==
      every?(x+->voo?(x,a) ,l)

    findCommonFactor(l:LEDF):Union(LEDF,"failed") ==
      empty?(l)$LEDF => "failed"
      f := first(l)$LEDF
      r := rest(l)$LEDF
      (t := isTimes(f)$EDF) case LEDF =>
        pos:=select(x+->inRest?(x,r),t)
        empty?(pos) => "failed"
        pos
      "failed"

    exprIsLogarithmicWeight(f:EDF,Var:EDF,a:EDF,b:EDF):INT ==
      ans := 0$INT
      k := tower(f)$EDF
      lf := select(kernelIsLog,k)$LKEDF
      empty?(lf)$LKEDF => ans
      for i in 1..# lf repeat
        arg := argument lf.i
        if (arg.1 = (Var - a)) then
          ans := ans + 1
        else if (arg.1 = (b - Var)) then
          ans := ans + 2
      ans

    exprHasLogarithmicWeights(args:NIA):INT ==
      ans := 1$INT
      a := getlo(args.range)$d01AgentsPackage :: EDF
      b := gethi(args.range)$d01AgentsPackage :: EDF
      Var := args.var :: EDF
      (l := isPlus numerator args.fn) case LEDF =>
        (cf := findCommonFactor l) case LEDF =>
          for j in 1..# cf repeat
            ans := ans + exprIsLogarithmicWeight(cf.j,Var,a,b)
          ans
        ans
      ans := ans + exprIsLogarithmicWeight(args.fn,Var,a,b)

    functionIsQuotient(expr:EDF):Union(EDF,"failed") ==
      (k := mainKernel expr) case KEDF =>
        expr = inv(f := k :: KEDF :: EDF)$EDF => f
--        one?(numerator expr) => denominator expr
```

```
        (numerator expr = 1) => denominator expr
        "failed"
      "failed"

    functionIsPolynomial?(f:EDF):Boolean ==
      (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

    functionIsNthRoot?(f:EDF,e:EDF):Boolean ==
      (m := mainKernel f) case "failed" => false
--      (one?(# (kernels f)))
      ((# (kernels f)) = 1)
        and (name operator m = (nthRoot :: Symbol))@Boolean
          and (((argument m).1 = e)@Boolean)

    score(f:EDF,e:EDF):FI ==
      ans := 0$FI
      (t := isTimes f) case LEDF =>
        for i in 1..# t repeat
          ans := ans + score(t.i,e)
        ans
      (q := functionIsQuotient f) case EDF =>
        ans := ans - score(q,e)
      functionIsPolynomial? f =>
        g:EDF := f/e
        if functionIsPolynomial? g then
          ans := 1+score(g,e)
        else
          ans
      (l := isPlus f) case LEDF =>
        (cf := findCommonFactor l) case LEDF =>
          factor := 1$EDF
          for i in 1..# cf repeat
            factor := factor*cf.i
          ans := ans + score(f/factor,e) + score(factor,e)
        ans
      functionIsNthRoot?(f,e) =>
        (p := isPower f) case "failed" => ans
        exp := p.exponent
        m := mainKernel f
        m case KEDF =>
          arg := argument m
          a:INT := (retract(arg.2)@INT)$EDF
          exp / a
        ans
      ans

    findAlgebraicWeight(args:NIA,e:EDF):Union(DF,"failed") ==
      zero?(s := score(args.fn,e)) => "failed"
      s :: DF
```

```
    exprHasAlgebraicWeight(args:NIA):Union(LDF,"failed") ==
      (f := functionIsContinuousAtEndPoints(args)$d01AgentsPackage)
                                          case continuous =>"failed"
      Var := args.var :: EDF
      a := getlo(args.range)$d01AgentsPackage :: EDF
      b := gethi(args.range)$d01AgentsPackage :: EDF
      A := Var - a
      B := b - Var
      f case lowerSingular =>
        (h := findAlgebraicWeight(args,A)) case "failed" => "failed"
        [h,0]
      f case upperSingular =>
        (g := findAlgebraicWeight(args,B)) case "failed" => "failed"
        [0,g]
      h := findAlgebraicWeight(args,A)
      g := findAlgebraicWeight(args,B)
      r := (h case "failed")
      s := (g case "failed")
      (r) and (s) => "failed"
      r => [0,coerce(g)@DF]
      s => [coerce(h)@DF,0]
      [coerce(h)@DF,coerce(g)@DF]

    exprOfFormCosWXorSinWX(f:EDF,var:Symbol): URBODF ==
      l:LKEDF := kernels(f)$EDF
--      one?((# l)$LKEDF)$INT =>
      # l = 1 =>
        a:LEDF := argument(e:KEDF := first(l)$LKEDF)$KEDF
        empty?(a) => "failed"
        m:Union(LEDF,"failed") := isTimes(first(a)$LEDF)$EDF
        m case LEDF => -- if it is a list, it will have at least two elements
          is?(second(m)$LEDF,var)$EDF =>
            omega:DF := retract(first(m)$LEDF)@DF
            o:BOP := operator(n:Symbol:=name(e)$KEDF)$BOP
            (n = cos@Symbol)@Boolean => [o,omega]
            (n = sin@Symbol)@Boolean => [o,omega]
            "failed"
          "failed"
        "failed"
      "failed"

    exprHasListOfWeightsCosWXorSinWX(f:EDF,var:Symbol): LURBODF ==
      (e := isTimes(f)$EDF) case LEDF =>
        [exprOfFormCosWXorSinWX(u,var) for u in e]
      empty?(k := kernels f) => ["failed"]
      ((first(k)::EDF) = f) =>
        [exprOfFormCosWXorSinWX(f,var)]
      ["failed"]

    bestWeight(l:LURBODF): URBODF ==
```

```
      empty?(l)$LURBODF => "failed"
      best := first(l)$LURBODF        --  best is first in list
      empty?(rest(l)$LURBODF) => best
      for i in 2..# l repeat          --  unless next is better
        r:URBODF := l.i
        if r case "failed" then leave
        else if best case "failed" then
          best := r
        else if r.w > best.w then
          best := r
      best

    weightIn?(weight:URBODF,listOfWeights:LURBODF):Boolean ==
      n := # listOfWeights
      for i in 1..n repeat                                -- cycle through list
        (weight = listOfWeights.i)@Boolean => return true -- return when found
      false

    exprHasWeightCosWXorSinWX(args:NIA):URBODF ==
      ans := empty()$LURBODF
      f:EDF := numerator(args.fn)$EDF
      (t:Union(LEDF,"failed") := isPlus(f)) case "failed" =>
        bestWeight(exprHasListOfWeightsCosWXorSinWX(f,args.var))
      if t case LEDF then
        e1 := first(t)$LEDF
        le1:LURBODF := exprHasListOfWeightsCosWXorSinWX(e1,args.var)
        le1 := [u for u in le1 | (not (u case "failed"))]
        empty?(le1)$LURBODF => "failed"
        test := true
        for i in 1..# le1 repeat
          le1i:URBODF := le1.i
          for j in 2..# t repeat
            if test then
              tj:LURBODF := exprHasListOfWeightsCosWXorSinWX(t.j,args.var)
              test := weightIn?(le1i,tj)
          if test then
            ans := concat([le1i],ans)
        bestWeight ans
      else "failed"
```

———-

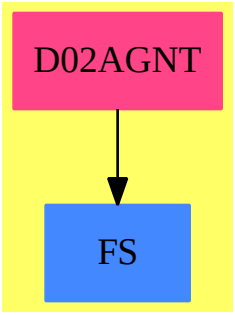
— D01WGTS.dotabb —

```
"D01WGTS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D01WGTS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D01WGTS" -> "FS"
```

———-

5.17 package D02AGNT d02AgentsPackage

5.17.1 d02AgentsPackage (D02AGNT)



Exports:

accuracyIF	combineFeatureCompatibility	eval	expenseOfEvaluationIF	intern
jacobian	sparsityIF	stiffnessAndStabilityFactor	stiffnessAndStabilityOfODEIF	syster

— package D02AGNT d02AgentsPackage —

```
)abbrev package D02AGNT d02AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: January 1997
++ Basic Operations: stiffnessFactor, jacobian
++ Description:
++ \axiom{d02AgentsPackage} contains a set of computational agents
++ for use with Ordinary Differential Equation solvers.

d02AgentsPackage(): E == I where
  LEDF  ==> List Expression DoubleFloat
  LEEDF  ==> List Equation Expression DoubleFloat
  EEDF  ==> Equation Expression DoubleFloat
  VEDF  ==> Vector Expression DoubleFloat
  MEDF  ==> Matrix Expression DoubleFloat
  MDF  ==> Matrix DoubleFloat
  EDF  ==> Expression DoubleFloat
  DF  ==> DoubleFloat
  F  ==> Float
  INT  ==> Integer
  CDF  ==> Complex DoubleFloat
  LDF  ==> List DoubleFloat
  LF  ==> List Float
```

```
S  ==> Symbol
LS  ==> List Symbol
MFI  ==> Matrix Fraction Integer
LFI  ==> List Fraction Integer
FI  ==> Fraction Integer
ODEA  ==> Record(xinit:DF,xend:DF,fn:VEDF,yinit:LDF,intvals:LDF,g:EDF,abserr:DF,relerr:DF)
ON  ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
RVE   ==> Record(val:EDF,exponent:INT)
RSS  ==> Record(stiffnessFactor:F,stabilityFactor:F)
ATT  ==> Record(stiffness:F,stability:F,expense:F,accuracy:F,intermediateResults:F)
ROA  ==> Record(key:ODEA,entry:ATT)

E ==>  with
  combineFeatureCompatibility: (F,F) -> F
    ++ combineFeatureCompatibility(C1,C2) is for interacting attributes
  combineFeatureCompatibility: (F,LF) -> F
    ++ combineFeatureCompatibility(C1,L) is for interacting attributes
  sparsityIF: MEDF -> F
    ++ sparsityIF(m) calculates the sparsity of a jacobian matrix
  jacobian: (VEDF,LS) -> MEDF
    ++ jacobian(v,w) is a local function to make a jacobian matrix
  eval: (MEDF,LS,VEDF) -> MEDF
    ++ eval(mat,symbols,values) evaluates a multivariable matrix at given values
    ++ for each of a list of variables
  stiffnessAndStabilityFactor: MEDF -> RSS
    ++ stiffnessAndStabilityFactor(me) calculates the stability and
    ++ stiffness factor of a system of first-order differential equations
    ++ (by evaluating the maximum difference in the real parts of the
    ++ negative eigenvalues of the jacobian of the system for which O(10)
    ++ equates to mildly stiff wheras stiffness ratios of O(10^6) are not
    ++ uncommon) and whether the system is likely to show any oscillations
    ++ (identified by the closeness to the imaginary axis of the complex
    ++ eigenvalues of the jacobian).
  stiffnessAndStabilityOfODEIF:ODEA -> RSS
    ++ stiffnessAndStabilityOfODEIF(ode) calculates the intensity values
    ++ of stiffness of a system of first-order differential equations
    ++ (by evaluating the maximum difference in the real parts of the
    ++ negative eigenvalues of the jacobian of the system for which O(10)
    ++ equates to mildly stiff wheras stiffness ratios of O(10^6) are not
    ++ uncommon) and whether the system is likely to show any oscillations
    ++ (identified by the closeness to the imaginary axis of the complex
    ++ eigenvalues of the jacobian).
    ++
    ++ It returns two values in the range [0,1].
  systemSizeIF:ODEA -> F
    ++ systemSizeIF(ode) returns the intensity value of the size of
    ++ the system of ODEs.  20 equations corresponds to the neutral
    ++ value.  It returns a value in the range [0,1].
  expenseOfEvaluationIF:ODEA -> F
    ++ expenseOfEvaluationIF(o) returns the intensity value of the
```

```
    ++ cost of evaluating the input ODE.  This is in terms of the number
    ++ of ‘‘operational units’’.  It returns a value in the range
    ++ [0,1].\indent{20}
    ++ 400 ‘‘operation units’’ -> 0.75
    ++ 200 ‘‘operation units’’ -> 0.5
    ++ 83 ‘‘operation units’’ -> 0.25 \indent{15}
    ++ exponentiation = 4 units , function calls = 10 units.
  accuracyIF:ODEA -> F
    ++ accuracyIF(o) returns the intensity value of the accuracy
    ++ requirements of the input ODE.  A request of accuracy of 10^-6
    ++ corresponds to the neutral intensity.  It returns a value
    ++ in the range [0,1].
  intermediateResultsIF:ODEA -> F
    ++ intermediateResultsIF(o) returns a value corresponding to the
    ++ required number of intermediate results required and, therefore,
    ++ an indication of how much this would affect the step-length of the
    ++ calculation.  It returns a value in the range [0,1].

I ==> add

  import ExpertSystemToolsPackage

  accuracyFactor:ODEA -> F
  expenseOfEvaluation:ODEA -> F
  eval1:(LEDF,LEEDF) -> LEDF
  stiffnessAndStabilityOfODE:ODEA -> RSS
  intermediateResultsFactor:ODEA -> F
  leastStabilityAngle:(LDF,LDF) -> F

  intermediateResultsFactor(ode:ODEA):F ==
    resultsRequirement := #(ode.intvals)
    (1.0-exp(-(resultsRequirement::F)/50.0)$F)

  intermediateResultsIF(o:ODEA):F ==
    ode := copy o
    (t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
      s := coerce(t)@ATT
      negative?(s.intermediateResults)$F =>
        s.intermediateResults := intermediateResultsFactor(ode)
        r:ROA := [ode,s]
        insert!(r)$ODEIntensityFunctionsTable
        s.intermediateResults
      s.intermediateResults
    a:ATT := [-1.0,-1.0,-1.0,-1.0,e:=intermediateResultsFactor(ode)]
    r:ROA := [ode,a]
    insert!(r)$ODEIntensityFunctionsTable
    e

  accuracyFactor(ode:ODEA):F ==
    accuracyRequirements := convert(ode.abserr)@F
```

```
    if zero?(accuracyRequirements) then
      accuracyRequirements := convert(ode.relerr)@F
    val := inv(accuracyRequirements)$F
    n := log10(val)$F
    (1.0-exp(-(n/(2.0))**2/(15.0))$F)

  accuracyIF(o:ODEA):F ==
    ode := copy o
    (t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
      s := coerce(t)@ATT
      negative?(s.accuracy)$F =>
        s.accuracy := accuracyFactor(ode)
        r:ROA := [ode,s]
        insert!(r)$ODEIntensityFunctionsTable
        s.accuracy
      s.accuracy
    a:ATT := [-1.0,-1.0,-1.0,e:=accuracyFactor(ode),-1.0]
    r:ROA := [ode,a]
    insert!(r)$ODEIntensityFunctionsTable
    e

  systemSizeIF(ode:ODEA):F ==
    n := #(ode.fn)
    (1.0-exp((-n::F/75.0))$F)

  expenseOfEvaluation(o:ODEA):F ==
    -- expense of evaluation of an ODE -- <0.3 inexpensive - 0.5 neutral - >0.7 very expensive
    -- 400 'operation units' -> 0.75
    -- 200 'operation units' -> 0.5
    -- 83 'operation units' -> 0.25
    -- ** = 4 units , function calls = 10 units.
    ode := copy o.fn
    expenseOfEvaluation(ode)

  expenseOfEvaluationIF(o:ODEA):F ==
    ode := copy o
    (t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
      s := coerce(t)@ATT
      negative?(s.expense)$F =>
        s.expense := expenseOfEvaluation(ode)
        r:ROA := [ode,s]
        insert!(r)$ODEIntensityFunctionsTable
        s.expense
      s.expense
    a:ATT := [-1.0,-1.0,e:=expenseOfEvaluation(ode),-1.0,-1.0]
    r:ROA := [ode,a]
    insert!(r)$ODEIntensityFunctionsTable
    e

  leastStabilityAngle(realPartsList:LDF,imagPartsList:LDF):F ==
```

```
    complexList := [complex(u,v)$CDF for u in realPartsList for v in imagPartsList]
    argumentList := [abs((abs(argument(u)$CDF)$DF)-(pi()$DF)/2)$DF for u in complexList]
    sortedArgumentList := sort(argumentList)$LDF
    list := [u for u in sortedArgumentList | not zero?(u) ]
    empty?(list)$LDF => 0$F
    convert(first(list)$LDF)@F

  stiffnessAndStabilityFactor(me:MEDF):RSS ==

    -- search first for real eigenvalues of the jacobian (symbolically)
    -- if the system isn't too big
    r:INT := ncols(me)$MEDF
    b:Boolean := ((# me) < 150)
    if b then
      mc:MFI := map(edf2fi,me)$ExpertSystemToolsPackage2(EDF,FI)
      e:LFI := realEigenvalues(mc,1/100)$NumericRealEigenPackage(FI)
      b := ((# e) >= r-1)@Boolean
    b =>
      -- if all the eigenvalues are real, find negative ones
      e := sort(neglist(e)$ExpertSystemToolsPackage1(FI))
      -- if there are two or more, calculate stiffness ratio
      ((n:=#e)>1)@Boolean => [coerce(e.1/e.n)@F,0$F]
      -- otherwise stiffness not present
      [0$F,0$F]

    md:MDF := map(edf2df,me)$ExpertSystemToolsPackage2(EDF,DF)

    -- otherwise calculate numerically the complex eigenvalues
    -- using NAG routine f02aff.

    res:Result := f02aff(r,r,md,-1)$NagEigenPackage
    realParts:Union(Any,"failed") := search(rr::Symbol,res)$Result
    realParts case "failed" => [0$F,0$F]
    realPartsMatrix:MDF := retract(realParts)$AnyFunctions1(MDF) -- array === matrix
    imagParts:Union(Any,"failed") := search(ri::Symbol,res)$Result
    imagParts case "failed" => [0$F,0$F]
    imagPartsMatrix:MDF := retract(imagParts)$AnyFunctions1(MDF) -- array === matrix
    imagPartsList:LDF := members(imagPartsMatrix)$MDF
    realPartsList:LDF := members(realPartsMatrix)$MDF
    stabilityAngle := leastStabilityAngle(realPartsList,imagPartsList)
    negRealPartsList := sort(neglist(realPartsList)$ExpertSystemToolsPackage1(DF))
    empty?(negRealPartsList)$LDF => [0$F,stabilityAngle]
    ((n:=#negRealPartsList)>1)@Boolean =>
      out := convert(negRealPartsList.1/negRealPartsList.n)@F
      [out,stabilityAngle]    -- calculate stiffness ratio
    [-convert(negRealPartsList.1)@F,stabilityAngle]

  eval1(l:LEDF,e:LEEDF):LEDF ==
    [eval(u,e)$EDF for u in l]
```

```
eval(mat:MEDF,symbols:LS,values:VEDF):MEDF ==
  l := listOfLists(mat)
  ledf := entries(values)$VEDF
  e := [equation(u::EDF,v)$EEDF for u in symbols for v in ledf]
  l := [eval1(w,e) for w in l]
  matrix l

combineFeatureCompatibility(C1:F,C2:F):F ==

  --                          C1 C2
  --    s(C1,C2) = -----------------------
  --               C1 C2 + (1 - C1)(1 - C2)

  C1*C2/((C1*C2)+(1$F-C1)*(1$F-C2))

combineFeatureCompatibility(C1:F,L:LF):F ==

  empty?(L)$LF => C1
  C2 := combineFeatureCompatibility(C1,first(L)$LF)
  combineFeatureCompatibility(C2,rest(L)$LF)

jacobian(v:VEDF,w:LS):Matrix EDF ==
  jacobian(v,w)$MultiVariableCalculusFunctions(S,EDF,VEDF,LS)

sparsityIF(m:Matrix EDF):F ==
  l:LEDF :=parts m
  z:LEDF := [u for u in l | zero?(u)$EDF]
  ((#z)::F/(#l)::F)

sum(a:EDF,b:EDF):EDF == a+b

stiffnessAndStabilityOfODE(ode:ODEA):RSS ==
  odefns := copy ode.fn
  ls:LS := [subscript(Y,[coerce(n)])$Symbol for n in 1..# odefns]
  yvals := copy ode.yinit
  for i in 1..#yvals repeat
    zero?(yvals.i) => yvals.i := 0.1::DF
  yexpr := [coerce(v)@EDF for v in yvals]
  yv:VEDF := vector(yexpr)
  j1:MEDF := jacobian(odefns,ls)
  ej1:MEDF := eval(j1,ls,yv)
  ej1 := eval(ej1,variables(reduce(sum,members(ej1)$MEDF)),vector([(ode.xinit)::EDF]))
  ssf := stiffnessAndStabilityFactor(ej1)
  stability := 1.0-sqrt((ssf.stabilityFactor)*(2.0)/(pi()$F))
  stiffness := (1.0)-exp(-(ssf.stiffnessFactor)/(500.0))
  [stiffness,stability]

stiffnessAndStabilityOfODEIF(ode:ODEA):RSS ==
  odefn := copy ode
  (t := showIntensityFunctions(odefn)$ODEIntensityFunctionsTable) case ATT =>
```

```
      s:ATT := coerce(t)@ATT
      negative?(s.stiffness)$F =>
        ssf:RSS := stiffnessAndStabilityOfODE(odefn)
        s := [ssf.stiffnessFactor,ssf.stabilityFactor,s.expense,
                s.accuracy,s.intermediateResults]
        r:ROA := [odefn,s]
        insert!(r)$ODEIntensityFunctionsTable
        ssf
      [s.stiffness,s.stability]
    ssf:RSS := stiffnessAndStabilityOfODE(odefn)
    s:ATT := [ssf.stiffnessFactor,ssf.stabilityFactor,-1.0,-1.0,-1.0]
    r:ROA := [odefn,s]
    insert!(r)$ODEIntensityFunctionsTable
    ssf
```

———-

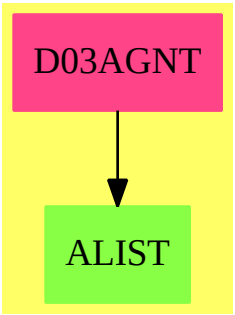
— **D02AGNT.dotabb** —

```
"D02AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D02AGNT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D02AGNT" -> "FS"
```

———-

5.18 package D03AGNT d03AgentsPackage

5.18.1 d03AgentsPackage (D03AGNT)



Exports:

central?	elliptic?	subscriptedVariables	varList

— **package D03AGNT d03AgentsPackage** —

```
)abbrev package D03AGNT d03AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: December 1997
++ Basic Operations:
++ Description:
++ \axiom{d03AgentsPackage} contains a set of computational agents
++ for use with Partial Differential Equation solvers.

d03AgentsPackage(): E == I where
  LEDF  ==> List Expression DoubleFloat
  EDF  ==> Expression DoubleFloat
  MDF  ==> Matrix DoubleFloat
  DF  ==> DoubleFloat
  F  ==> Float
  INT  ==> Integer
  NNI  ==> NonNegativeInteger
  EEDF  ==> Equation Expression DoubleFloat
  LEEDF  ==> List Equation Expression DoubleFloat
  LDF  ==> List DoubleFloat
  LOCDF  ==> List OrderedCompletion DoubleFloat
  OCDF  ==> OrderedCompletion DoubleFloat
  LS  ==> List Symbol
  PDEC  ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
                     dStart:MDF, dFinish:MDF)
  PDEB  ==> Record(pde:LEDF, constraints:List PDEC,
                   f:List LEDF, st:String, tol:DF)
  NOA  ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)

  E ==>  with
    varList:(Symbol,NonNegativeInteger) -> LS
      ++ varList(s,n) \undocumented{}
    subscriptedVariables:EDF -> EDF
      ++ subscriptedVariables(e) \undocumented{}
    central?:(DF,DF,LEDF) -> Boolean
      ++ central?(f,g,l) \undocumented{}
    elliptic?:PDEB -> Boolean
      ++ elliptic?(r) \undocumented{}

  I ==> add

    import ExpertSystemToolsPackage

    sum(a:EDF,b:EDF):EDF == a+b

    varList(s:Symbol,n:NonNegativeInteger):LS ==
      [subscript(s,[t::OutputForm]) for t in expand([1..n])$Segment(Integer)]

    subscriptedVariables(e:EDF):EDF ==
      oldVars:List Symbol := variables(e)
```

```
      o := [a :: EDF for a in oldVars]
      newVars := varList(X::Symbol,# oldVars)
      n := [b :: EDF for b in newVars]
      subst(e,[a=b for a in o for b in n])

    central?(x:DF,y:DF,p:LEDF):Boolean ==
      ls := variables(reduce(sum,p))
      le := [equation(u::EDF,v)$EEDF for u in ls for v in [x::EDF,y::EDF]]
      l := [eval(u,le)$EDF for u in p]
      max(l.4,l.5) < 20 * max(l.1,max(l.2,l.3))

    elliptic?(args:PDEB):Boolean ==
      (args.st)="elliptic" => true
      p := args.pde
      xcon:PDEC := first(args.constraints)
      ycon:PDEC := second(args.constraints)
      xs := xcon.start
      ys := ycon.start
      xf := xcon.finish
      yf := ycon.finish
      xstart:DF := ((xf-xs)/2)$DF
      ystart:DF := ((yf-ys)/2)$DF
      optStart:LDF := [xstart,ystart]
      lower:LOCDF := [xs::OCDF,ys::OCDF]
      upper:LOCDF := [xf::OCDF,yf::OCDF]
      v := variables(e := 4*first(p)*third(p)-(second(p))**2)
      eq := subscriptedVariables(e)
      noa:NOA :=
--        one?(# v) =>
        (# v) = 1 =>
          ((first v) = X@Symbol) =>
            [eq,[xstart],[xs::OCDF],empty()$LEDF,[xf::OCDF]]
          [eq,[ystart],[ys::OCDF],empty()$LEDF,[yf::OCDF]]
        [eq,optStart,lower,empty()$LEDF,upper]
      ell := optimize(noa::NumericalOptimizationProblem)$AnnaNumericalOptimizationPackage
      o:Union(Any,"failed") := search(objf::Symbol,ell)$Result
      o case "failed" => false
      ob := o :: Any
      obj:DF := retract(ob)$AnyFunctions1(DF)
      positive?(obj)
```

———-

— D03AGNT.dotabb —

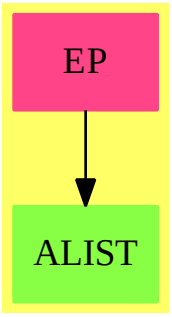
```
"D03AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D03AGNT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"D03AGNT" -> "ALIST"
```

Chapter 6

Chapter E

6.1 package EP EigenPackage

6.1.1 EigenPackage (EP)



Exports:
characteristicPolynomial eigenvalues eigenvector eigenvectors generalizedEigenvector generalizedEigenvec

— package EP EigenPackage —

```
)abbrev package EP EigenPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Date Last Updated: October 1992
++ Basic Functions:
++ Related Constructors: NumericRealEigenPackage,  NumericComplexEigenPackage,
++ RadicalEigenPackage
++ Also See:
++ AMS Classifications:
```

```
++ Keywords:
++ References:
++ Description:
++ This is a package for the exact computation of eigenvalues and eigenvectors.
++ This package can be made to work for matrices with coefficients which are
++ rational functions over a ring where we can factor polynomials.
++ Rational eigenvalues are always explicitly computed while the
++ non-rational ones are expressed in terms of their minimal
++ polynomial.
-- Functions for the numeric computation of eigenvalues and eigenvectors
-- are in numeigen spad.

EigenPackage(R) : C == T
 where
   R     : GcdDomain
   P     ==> Polynomial R
   F     ==> Fraction P
   SE    ==> Symbol()
   SUP   ==> SparseUnivariatePolynomial(P)
   SUF   ==> SparseUnivariatePolynomial(F)
   M     ==> Matrix(F)
   NNI   ==> NonNegativeInteger
   ST    ==> SuchThat(SE,P)

   Eigenvalue  ==> Union(F,ST)
   EigenForm   ==> Record(eigval:Eigenvalue,eigmult:NNI,eigvec : List M)
   GenEigen    ==> Record(eigval:Eigenvalue,geneigvec:List M)

   C == with
     characteristicPolynomial :  (M,Symbol)  ->  P
       ++ characteristicPolynomial(m,var) returns the
       ++ characteristicPolynomial of the matrix m using
       ++ the symbol var as the main variable.

     characteristicPolynomial :      M       ->  P
       ++ characteristicPolynomial(m) returns the
       ++ characteristicPolynomial of the matrix m using
       ++ a new generated symbol symbol as the main variable.

     eigenvalues       :    M        ->  List Eigenvalue
       ++ eigenvalues(m) returns the
       ++ eigenvalues of the matrix m which are expressible
       ++ as rational functions over the rational numbers.

     eigenvector       :   (Eigenvalue,M)  ->  List M
       ++ eigenvector(eigval,m) returns the
       ++ eigenvectors belonging to the eigenvalue eigval
       ++ for the matrix m.

     generalizedEigenvector  : (Eigenvalue,M,NNI,NNI) -> List M
```

```
    ++ generalizedEigenvector(alpha,m,k,g)
    ++ returns the generalized eigenvectors
    ++ of the matrix relative to the eigenvalue alpha.
    ++ The integers k and g are respectively the algebraic and the
    ++ geometric multiplicity of tye eigenvalue alpha.
    ++ alpha can be either rational or not.
    ++ In the seconda case apha is the minimal polynomial of the
    ++ eigenvalue.

  generalizedEigenvector  : (EigenForm,M) -> List M
    ++ generalizedEigenvector(eigen,m)
    ++ returns the generalized eigenvectors
    ++ of the matrix relative to the eigenvalue eigen, as
    ++ returned by the function eigenvectors.

  generalizedEigenvectors  : M -> List GenEigen
    ++ generalizedEigenvectors(m)
    ++ returns the generalized eigenvectors
    ++ of the matrix m.

  eigenvectors      :    M        ->  List(EigenForm)
    ++ eigenvectors(m) returns the eigenvalues and eigenvectors
    ++ for the matrix m.
    ++ The rational eigenvalues and the correspondent eigenvectors
    ++ are explicitely computed, while the non rational ones
    ++ are given via their minimal polynomial and the corresponding
    ++ eigenvectors are expressed in terms of a "generic" root of
    ++ such a polynomial.

T == add
  PI       ==> PositiveInteger


  MF  := GeneralizedMultivariateFactorize(SE,IndexedExponents SE,R,R,P)
  UPCF2:= UnivariatePolynomialCategoryFunctions2(P,SUP,F,SUF)


            ----  Local  Functions  ----
  tff              :  (SUF,SE)      ->  F
  fft              :  (SUF,SE)      ->  F
  charpol          :   (M,SE)       ->   F
  intRatEig        :  (F,M,NNI)    ->   List M
  intAlgEig        :  (ST,M,NNI)    ->   List M
  genEigForm       : (EigenForm,M)  ->   GenEigen

 ---- next functions needed for defining  ModularField ----
  reduction(u:SUF,p:SUF):SUF == u rem p

  merge(p:SUF,q:SUF):Union(SUF,"failed") ==
      p = q => p
```

```
    p = 0 => q
    q = 0 => p
    "failed"

exactquo(u:SUF,v:SUF,p:SUF):Union(SUF,"failed") ==
   val:=extendedEuclidean(v,p,u)
   val case "failed" => "failed"
   val.coef1

          ----  functions for conversions  ----
fft(t:SUF,x:SE):F ==
  n:=degree(t)
  cf:=monomial(1,x,n)$P :: F
  cf * leadingCoefficient t

tff(p:SUF,x:SE) : F ==
  degree p=0 => leadingCoefficient p
  r:F:=0$F
  while p^=0 repeat
    r:=r+fft(p,x)
    p := reductum p
  r

 ---- generalized eigenvectors associated to a given eigenvalue ---
genEigForm(eigen : EigenForm,A:M) : GenEigen ==
  alpha:=eigen.eigval
  k:=eigen.eigmult
  g:=#(eigen.eigvec)
  k = g  => [alpha,eigen.eigvec]
  [alpha,generalizedEigenvector(alpha,A,k,g)]

      ---- characteristic polynomial  ----
charpol(A:M,x:SE) : F ==
  dimA :PI := (nrows A):PI
  dimA ^= ncols A => error " The matrix is not square"
  B:M:=zero(dimA,dimA)
  for i in 1..dimA repeat
    for j in 1..dimA repeat  B(i,j):=A(i,j)
    B(i,i) := B(i,i) - monomial(1$P,x,1)::F
  determinant B

     --------  EXPORTED  FUNCTIONS  --------

       ----  characteristic polynomial of a matrix A ----
characteristicPolynomial(A:M):P ==
  x:SE:=new()$SE
  numer charpol(A,x)

       ----  characteristic polynomial of a matrix A ----
characteristicPolynomial(A:M,x:SE) : P == numer charpol(A,x)
```

```
            ----  Eigenvalues of the matrix A  ----
  eigenvalues(A:M): List Eigenvalue  ==
    x:=new()$SE
    pol:= charpol(A,x)
    lrat:List F :=empty()
    lsym:List ST :=empty()
    for eq in solve(pol,x)$SystemSolvePackage(R) repeat
      alg:=numer lhs eq
      degree(alg, x)=1 => lrat:=cons(rhs eq,lrat)
      lsym:=cons([x,alg],lsym)
    append([lr::Eigenvalue for lr in lrat],
           [ls::Eigenvalue for ls in lsym])

       ----  Eigenvectors belonging to a given eigenvalue  ----
             ----  the eigenvalue must be exact  ----
  eigenvector(alpha:Eigenvalue,A:M) : List M  ==
    alpha case F => intRatEig(alpha::F,A,1$NNI)
    intAlgEig(alpha::ST,A,1$NNI)

----  Eigenvectors belonging to a given rational eigenvalue  ----
             ---- Internal function -----
  intRatEig(alpha:F,A:M,m:NNI) : List M  ==
    n:=nrows A
    B:M := zero(n,n)$M
    for i in 1..n repeat
      for j in 1..n repeat B(i,j):=A(i,j)
      B(i,i):= B(i,i) - alpha
    [v::M for v in nullSpace(B**m)]

----  Eigenvectors belonging to a given algebraic eigenvalue  ----
      ------   Internal  Function  -----
  intAlgEig(alpha:ST,A:M,m:NNI) : List M  ==
    n:=nrows A
    MM := ModularField(SUF,SUF,reduction,merge,exactquo)
    AM:=Matrix MM
    x:SE:=lhs alpha
    pol:SUF:=unitCanonical map(coerce,univariate(rhs alpha,x))$UPCF2
    alg:MM:=reduce(monomial(1,1),pol)
    B:AM := zero(n,n)
    for i in 1..n repeat
      for j in 1..n repeat B(i,j):=reduce(A(i,j)::SUF,pol)
      B(i,i):= B(i,i) - alg
    sol: List M :=empty()
    for vec in nullSpace(B**m) repeat
      w:M:=zero(n,1)
      for i in 1..n repeat w(i,1):=tff((vec.i)::SUF,x)
      sol:=cons(w,sol)
    sol
```

```
---- Generalized Eigenvectors belonging to a given eigenvalue ----
generalizedEigenvector(alpha:Eigenvalue,A:M,k:NNI,g:NNI) : List M ==
  alpha case F => intRatEig(alpha::F,A,(1+k-g)::NNI)
  intAlgEig(alpha::ST,A,(1+k-g)::NNI)

---- Generalized Eigenvectors belonging to a given eigenvalue ----
generalizedEigenvector(eigen :EigenForm,A:M) : List M ==
  generalizedEigenvector(eigen.eigval,A,eigen.eigmult,# eigen.eigvec)

    ---- Generalized Eigenvectors -----
generalizedEigenvectors(A:M) : List GenEigen ==
  n:= nrows A
  leig:=eigenvectors A
  [genEigForm(leg,A) for leg in leig]

          ---- eigenvectors and eigenvalues ----
eigenvectors(A:M):List(EigenForm) ==
  n:=nrows A
  x:=new()$SE
  p:=numer charpol(A,x)
  MM := ModularField(SUF,SUF,reduction,merge,exactquo)
  AM:=Matrix(MM)
  ratSol : List EigenForm := empty()
  algSol : List EigenForm := empty()
  lff:=factors factor p
  for fact in lff repeat
    pol:=fact.factor
    degree(pol,x)=1 =>
      vec:F :=-coefficient(pol,x,0)/coefficient(pol,x,degree(pol,x))
      ratSol:=cons([vec,fact.exponent :: NNI,
                    intRatEig(vec,A,1$NNI)]$EigenForm,ratSol)
    alpha:ST:=[x,pol]
    algSol:=cons([alpha,fact.exponent :: NNI,
                  intAlgEig(alpha,A,1$NNI)]$EigenForm,algSol)
  append(ratSol,algSol)
```

———-

— EP.dotabb —

```
"EP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"EP" -> "ALIST"
```

———-

6.2 package EF ElementaryFunction

— ElementaryFunction.input —

```
)set break resume
)sys rm -f ElementaryFunction.output
)spool ElementaryFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 32
)trace EF
--R
--R
--R   Parameterized constructors traced:
--R      EF
--E 1

--S 2 of 32
D(cos(3*x+6*y),x)
--I
--I1<enter ElementaryFunction.cos,64 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3)))
--I 1<enter ElementaryFunction.iicos,154 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3
--I  1<enter ElementaryFunction.iisqrt2,58 :
--I  1>exit  ElementaryFunction.iisqrt2,58 : ((1 #<vector 0917aab8> (1 0 . 1)) 0 . 1)
--I  1<enter ElementaryFunction.iisqrt3,59 :
--I  1>exit  ElementaryFunction.iisqrt3,59 : ((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 1)
--I  1<enter ElementaryFunction.specialTrigs,116 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0>
--I   1<enter ElementaryFunction.pi,46 :
--I   1>exit  ElementaryFunction.pi,46 : ((1 #<vector 090c3a64> (1 0 . 1)) 0 . 1)
--I  1>exit  ElementaryFunction.specialTrigs,116 : (1 . "failed")
--I 1>exit  ElementaryFunction.iicos,154 : ((1 #<vector 0941ed74> (1 0 . 1)) 0 . 1)
--I1>exit  ElementaryFunction.cos,64 : ((1 #<vector 0941ed74> (1 0 . 1)) 0 . 1)
--I1<enter ElementaryFunction.sin,63 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3)))
--I 1<enter ElementaryFunction.iisin,152 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3
--I  1<enter ElementaryFunction.iisqrt2,58 :
--I  1>exit  ElementaryFunction.iisqrt2,58 : ((1 #<vector 0917aab8> (1 0 . 1)) 0 . 1)
--I  1<enter ElementaryFunction.iisqrt3,59 :
--I  1>exit  ElementaryFunction.iisqrt3,59 : ((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 1)
--I  1<enter ElementaryFunction.specialTrigs,116 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0>
--I   1<enter ElementaryFunction.pi,46 :
--I   1>exit  ElementaryFunction.pi,46 : ((1 #<vector 090c3a64> (1 0 . 1)) 0 . 1)
--I  1>exit  ElementaryFunction.specialTrigs,116 : (1 . "failed")
--I 1>exit  ElementaryFunction.iisin,152 : ((1 #<vector 0941eb60> (1 0 . 1)) 0 . 1)
--I1>exit  ElementaryFunction.sin,63 : ((1 #<vector 0941eb60> (1 0 . 1)) 0 . 1)
--R
--R   (1)  - 3sin(6y + 3x)
```

```
--R                                                    Type: Expression(Integer)
--E 2

--S 3 of 32
)trace )off
--R
--R
--R   Nothing is traced now.
--R
--E 3


--S 4 of 32
D(sin(3*x+6*y),x)
--R
--R
--R   (2)  3cos(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 4

--S 5 of 32
D(cos(3*x+6*y),x)
--R
--R
--R   (3)  - 3sin(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 5
```

———-

Note that both Mathematica and Maxima return this result as $3sec(3*x+6*y)^2$ but Maple returns the same result as Axiom. They are equivalent results.

— **ElementaryFunction.input** —

```
--S 6 of 32
D(tan(3*x+6*y),x)
--R
--R
--R                     2
--R   (4)  3tan(6y + 3x)  + 3
--R                                                    Type: Expression(Integer)
--E 6

--S 7 of 32
simplify ((3*tan(6*y+3*x)^2+3) - (3*sec(3*x+6*y)^2))
--R
--R
--R   (5)  0
--R                                                    Type: Expression(Integer)
```

```
--E 7
```

———-

Note that Mathematica and Maxima return $-3csc(3x+6y)^2$ and Maple returns the same form as Axiom. They are equivalent.

— **ElementaryFunction.input** —

```
--S 8 of 32
D(cot(3*x+6*y),x)
--R
--R
--R                        2
--R   (6)  - 3cot(6y + 3x)  - 3
--R                                                    Type: Expression(Integer)
--E 8

--S 9 of 32
simplify ((-3*cot(6*y+3*x)^2-3) -(-3*csc(3*x+6*y)^2))
--R
--R
--R   (7)  0
--R                                                    Type: Expression(Integer)
--E 9

--S 10 of 32
D(sec(3*x+6*y),x)
--R
--R
--R   (8)  3sec(6y + 3x)tan(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 10

--S 11 of 32
D(csc(3*x+6*y),x)
--R
--R
--R   (9)  - 3cot(6y + 3x)csc(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 11

--S 12 of 32
D(asin(3*x+6*y),x)
--R
--R
--R                      3
--R   (10)  --------------------------
--R          +----------------------+
--R          |     2             2
```

```
--R         \|- 36y  - 36x y - 9x  + 1
--R                                                    Type: Expression(Integer)
--E 12

--S 13 of 32
D(acos(3*x+6*y),x)
--R
--R
--R                          3
--R   (11)  - --------------------------
--R             +-----------------------+
--R             |     2             2
--R           \|- 36y  - 36x y - 9x  + 1
--R                                                    Type: Expression(Integer)
--E 13

--S 14 of 32
D(atan(3*x+6*y),x)
--R
--R
--R                     3
--R   (12)  ---------------------
--R             2             2
--R         36y  + 36x y + 9x  + 1
--R                                                    Type: Expression(Integer)
--E 14

--S 15 of 32
D(acot(3*x+6*y),x)
--R
--R
--R                       3
--R   (13)  - ---------------------
--R              2             2
--R           36y  + 36x y + 9x  + 1
--R                                                    Type: Expression(Integer)
--E 15
```

———-

Mathematica, Maple, and Maxima give:

$$\frac{3}{(3*x+6*y)^2\sqrt{1-\frac{1}{(3*x+6*y)^2}}}$$

which proceeds directly from the formula for the derivative of asec:

$$\frac{d}{dx}arcsec(x) = \frac{1}{x\sqrt{x^2-1}}$$

— **ElementaryFunction.input** —

```
--S 16 of 32
D(asec(3*x+6*y),x)
--R
--R
--R                          1
--R   (14)  --------------------------------
--R                 +----------------------+
--R                 |   2             2
--R         (2y + x)\|36y  + 36x y + 9x  - 1
--R                                                  Type: Expression(Integer)
--E 16
```

———-

If we use the same formula for this example:

$$\frac{1}{(3*x+6*y)\sqrt{(3*x+6*y)^2-1}}d(3*x+6*y)/dx$$

— **ElementaryFunction.input** —

```
--S 17 of 32
3/((3*x+6*y)*sqrt((3*x+6*y)^2-1))
--R
--R
--R                          1
--R   (15)  --------------------------------
--R                 +----------------------+
--R                 |   2             2
--R         (2y + x)\|36y  + 36x y + 9x  - 1
--R                                                  Type: Expression(Integer)
--E 17
```

———-

Mathematica, Maple, and Maxima give

$$-\frac{3}{(3*x+6*y)^2\sqrt{1-\frac{1}{(3*x+6*y)^2}}}$$

which is just the negative of the above result and we can see that the same analysis applies to explain the results.

— **ElementaryFunction.input** —

```
--S 18 of 32
D(acsc(3*x+6*y),x)
--R
--R
--R                              1
--R   (16)  - ---------------------------------
--R                     +----------------------+
--R                     |   2             2
--R            (2y + x)\|36y  + 36x y + 9x  - 1
--R                                                    Type: Expression(Integer)
--E 18

--S 19 of 32
D(sinh(3*x+6*y),x)
--R
--R
--R   (17)  3cosh(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 19

--S 20 of 32
D(cosh(3*x+6*y),x)
--R
--R
--R   (18)  3sinh(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 20
```

———-

Mathematica and Maxima return

$$3sech(3x+6y)^2$$

Maple returns Axiom's answer. Both are equivalent.

— **ElementaryFunction.input** —

```
--S 21 of 32
D(tanh(3*x+6*y),x)
--R
--R
--R                           2
--R   (19)  - 3tanh(6y + 3x)  + 3
--R                                                    Type: Expression(Integer)
--E 21

--S 22 of 32
simplify ((-3*tanh(6*y+3*x)^2+3)-(3*sech(3*x+6*y)^2))
--R
```

```
--R
--R   (20)  0
--R                                                    Type: Expression(Integer)
--E 22
```

———-

Mathematica and Maxima return

$$-3csch(3*x+6*y)^2$$

Maple returns Axiom's answer. Both are equivalent.

— **ElementaryFunction.input** —

```
--S 23 of 32
D(coth(3*x+6*y),x)
--R
--R
--R                          2
--R   (21)  - 3coth(6y + 3x)  + 3
--R                                                    Type: Expression(Integer)
--E 23

--S 24 of 32
simplify ((-3*coth(6*y+3*x)^2+3) - (-3*csch(3*x+6*y)^2))
--R
--R
--R   (22)  0
--R                                                    Type: Expression(Integer)
--E 24

--S 25 of 32
D(sech(3*x+6*y),x)
--R
--R
--R   (23)  - 3sech(6y + 3x)tanh(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 25

--S 26 of 32
D(csch(3*x+6*y),x)
--R
--R
--R   (24)  - 3coth(6y + 3x)csch(6y + 3x)
--R                                                    Type: Expression(Integer)
--E 26

--S 27 of 32
D(asinh(3*x+6*y),x)
```

```
--R
--R
--R                      3
--R   (25)  ------------------------
--R          +----------------------+
--R          |   2             2
--R         \|36y  + 36x y + 9x  + 1
--R                                                  Type: Expression(Integer)
--E 27
```

———-

Mathematica and Maple show

$$\frac{3}{\sqrt{-1+3*x+6*y}\sqrt{1+3*x+6*y}}$$

Maxima gives Axiom's answer. Both are equivalent, just factored forms.

— **ElementaryFunction.input** —

```
--S 28 of 32
D(acosh(3*x+6*y),x)
--R
--R
--R                      3
--R   (26)  ------------------------
--R          +----------------------+
--R          |   2             2
--R         \|36y  + 36x y + 9x  - 1
--R                                                  Type: Expression(Integer)
--E 28

--S 29 of 32
D(atanh(3*x+6*y),x)
--R
--R
--R                       3
--R   (27)  - ----------------------
--R               2             2
--R            36y  + 36x y + 9x  - 1
--R                                                  Type: Expression(Integer)
--E 29

--S 30 of 32
D(acoth(3*x+6*y),x)
--R
--R
--R                       3
--R   (28)  - ----------------------
--R               2             2
```

```
--R           36y  + 36x y + 9x  - 1
--R                                                    Type: Expression(Integer)
--E 30
```

———-

Mathematica gives

$$-\frac{3}{(3x+6y)\sqrt{\frac{1-3x-6y}{1_3x+6y}}(1+3x+6y)}$$

Maxima gives

$$-\frac{3}{(6y+3x)^2\sqrt{\frac{1}{(6y+3x)^2}-1}}$$

Maple gives

$$-\frac{3}{(3x+6y)^2\sqrt{\frac{1}{3x+6y}-1}\sqrt{\frac{1}{3x+6y}+1}}$$

Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.

— **ElementaryFunction.input** —

```
--S 31 of 32
D(asech(3*x+6*y),x)
--R
--R
--R                              1
--R   (29)  - -----------------------------------
--R                      +-----------------------+
--R                      |     2             2
--R            (2y + x)\|- 36y  - 36x y - 9x  + 1
--R                                                    Type: Expression(Integer)
--E 31
```

———-

Mathematica, Maple, and Maxima all generate the answer

$$-\frac{3}{(3x+6y)^2\sqrt{1+\frac{1}{(3x+6y)^2}}}$$

Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.

— **ElementaryFunction.input** —

```
--S 32 of 32
D(acsch(3*x+6*y),x)
--R
--R
--R                              1
--R   (30)  - ---------------------------------
--R                     +----------------------+
--R                     |   2             2
--R           (2y + x)\|36y  + 36x y + 9x  + 1
--R                                                    Type: Expression(Integer)
--E 32

)spool
)lisp (bye)
```

———-

— ElementaryFunction.help —

```
========================================================================
ElementaryFunction examples
========================================================================

D(sin(3*x+6*y),x)
   3cos(6y + 3x)

D(cos(3*x+6*y),x)
   - 3sin(6y + 3x)

D(tan(3*x+6*y),x)
                2
   3tan(6y + 3x)  + 3

D(cot(3*x+6*y),x)
                 2
   - 3cot(6y + 3x)  - 3

D(sec(3*x+6*y),x)
   3sec(6y + 3x)tan(6y + 3x)

D(csc(3*x+6*y),x)
   - 3cot(6y + 3x)csc(6y + 3x)

D(asin(3*x+6*y),x)
                3
   --------------------------
    +-----------------------+
    |     2             2
   \|- 36y  - 36x y - 9x  + 1
```

```
D(acos(3*x+6*y),x)
                  3
   - --------------------------
      +-----------------------+
      |     2             2
     \|- 36y  - 36x y - 9x  + 1

D(atan(3*x+6*y),x)
               3
    ----------------------
       2             2
    36y  + 36x y + 9x  + 1

D(acot(3*x+6*y),x)
                3
    - ----------------------
         2             2
      36y  + 36x y + 9x  + 1

D(asec(3*x+6*y),x)
                   1
    ---------------------------------
             +----------------------+
             |   2             2
    (2y + x)\|36y  + 36x y + 9x  - 1

D(acsc(3*x+6*y),x)
                     1
    - ---------------------------------
               +----------------------+
               |   2             2
      (2y + x)\|36y  + 36x y + 9x  - 1


D(sinh(3*x+6*y),x)
    3cosh(6y + 3x)

D(cosh(3*x+6*y),x)
    3sinh(6y + 3x)

D(tanh(3*x+6*y),x)
                   2
    - 3tanh(6y + 3x)  + 3

D(coth(3*x+6*y),x)
                   2
    - 3coth(6y + 3x)  + 3

D(sech(3*x+6*y),x)
```

```
    - 3sech(6y + 3x)tanh(6y + 3x)

D(csch(3*x+6*y),x)
    - 3coth(6y + 3x)csch(6y + 3x)

D(asinh(3*x+6*y),x)
                3
    -------------------------
     +----------------------+
     |   2             2
    \|36y  + 36x y + 9x  + 1

D(acosh(3*x+6*y),x)
                3
    -------------------------
     +----------------------+
     |   2             2
    \|36y  + 36x y + 9x  - 1

D(atanh(3*x+6*y),x)
                 3
    - ----------------------
         2             2
      36y  + 36x y + 9x  - 1

D(acoth(3*x+6*y),x)
                 3
    - ----------------------
         2             2
      36y  + 36x y + 9x  - 1

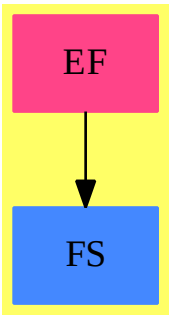
D(asech(3*x+6*y),x)
                      1
    - ----------------------------------
              +------------------------+
              |     2             2
      (2y + x)\|- 36y  - 36x y - 9x  + 1

D(acsch(3*x+6*y),x)
                     1
    - ---------------------------------
              +----------------------+
              |   2             2
      (2y + x)\|36y  + 36x y + 9x  + 1

See Also:
o )show ElementaryFunction
```

———-

6.2.1 ElementaryFunction (EF)



Exports:

acos	acosh	acot	acoth	acsc
acsch	asec	asech	asin	asinh
atan	atanh	belong?	cos	cosh
cot	coth	csc	csch	exp
iiacos	iiacosh	iiacot	iiacoth	iiacsc
iiacsch	iiasec	iiasech	iiasin	iiasinh
iiatan	iiatanh	iicos	iicosh	iicot
iicoth	iicsc	iicsch	iiexp	iilog
iisec	iisech	iisin	iisinh	iisqrt2
iisqrt3	iitan	iitanh	localReal?	log
operator	pi	sec	sech	sin
sinh	specialTrigs	tan	tanh	

— package EF ElementaryFunction —

```
)abbrev package EF ElementaryFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 April 1995
++ Keywords: elementary, function, logarithm, exponential.
++ Examples:  )r EF INPUT
++ Description:
++ Provides elementary functions over an integral domain.

ElementaryFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: Join(FunctionSpace R, RadicalCategory)

  B   ==> Boolean
  L   ==> List
  Z   ==> Integer
```

```
OP  ==> BasicOperator
K   ==> Kernel F
INV ==> error "Invalid argument"

Exports ==> with
  exp     : F -> F
    ++ exp(x) applies the exponential operator to x
  log     : F -> F
    ++ log(x) applies the logarithm operator to x
  sin     : F -> F
    ++ sin(x) applies the sine operator to x
  cos     : F -> F
    ++ cos(x) applies the cosine operator to x
  tan     : F -> F
    ++ tan(x) applies the tangent operator to x
  cot     : F -> F
    ++ cot(x) applies the cotangent operator to x
  sec     : F -> F
    ++ sec(x) applies the secant operator to x
  csc     : F -> F
    ++ csc(x) applies the cosecant operator to x
  asin    : F -> F
    ++ asin(x) applies the inverse sine operator to x
  acos    : F -> F
    ++ acos(x) applies the inverse cosine operator to x
  atan    : F -> F
    ++ atan(x) applies the inverse tangent operator to x
  acot    : F -> F
    ++ acot(x) applies the inverse cotangent operator to x
  asec    : F -> F
    ++ asec(x) applies the inverse secant operator to x
  acsc    : F -> F
    ++ acsc(x) applies the inverse cosecant operator to x
  sinh    : F -> F
    ++ sinh(x) applies the hyperbolic sine operator to x
  cosh    : F -> F
    ++ cosh(x) applies the hyperbolic cosine operator to x
  tanh    : F -> F
    ++ tanh(x) applies the hyperbolic tangent operator to x
  coth    : F -> F
    ++ coth(x) applies the hyperbolic cotangent operator to x
  sech    : F -> F
    ++ sech(x) applies the hyperbolic secant operator to x
  csch    : F -> F
    ++ csch(x) applies the hyperbolic cosecant operator to x
  asinh   : F -> F
    ++ asinh(x) applies the inverse hyperbolic sine operator to x
  acosh   : F -> F
    ++ acosh(x) applies the inverse hyperbolic cosine operator to x
  atanh   : F -> F
```

```
  ++ atanh(x) applies the inverse hyperbolic tangent operator to x
acoth   : F -> F
  ++ acoth(x) applies the inverse hyperbolic cotangent operator to x
asech   : F -> F
  ++ asech(x) applies the      inverse hyperbolic secant operator to x
acsch   : F -> F
  ++ acsch(x) applies the inverse hyperbolic cosecant operator to x
pi      : () -> F
  ++ pi() returns the pi operator
belong? : OP -> Boolean
  ++ belong?(p) returns true if operator p is elementary
operator: OP -> OP
  ++ operator(p) returns an elementary operator with the same symbol as p
-- the following should be local, but are conditional
iisqrt2   : () -> F
  ++ iisqrt2() should be local but conditional
iisqrt3   : () -> F
  ++ iisqrt3() should be local but conditional
iiexp     : F -> F
  ++ iiexp(x) should be local but conditional
iilog     : F -> F
  ++ iilog(x) should be local but conditional
iisin     : F -> F
  ++ iisin(x) should be local but conditional
iicos     : F -> F
  ++ iicos(x) should be local but conditional
iitan     : F -> F
  ++ iitan(x) should be local but conditional
iicot     : F -> F
  ++ iicot(x) should be local but conditional
iisec     : F -> F
  ++ iisec(x) should be local but conditional
iicsc     : F -> F
  ++ iicsc(x) should be local but conditional
iiasin    : F -> F
  ++ iiasin(x) should be local but conditional
iiacos    : F -> F
  ++ iiacos(x) should be local but conditional
iiatan    : F -> F
  ++ iiatan(x) should be local but conditional
iiacot    : F -> F
  ++ iiacot(x) should be local but conditional
iiasec    : F -> F
  ++ iiasec(x) should be local but conditional
iiacsc    : F -> F
  ++ iiacsc(x) should be local but conditional
iisinh    : F -> F
  ++ iisinh(x) should be local but conditional
iicosh    : F -> F
  ++ iicosh(x) should be local but conditional
```

```
    iitanh    : F -> F
      ++ iitanh(x) should be local but conditional
    iicoth    : F -> F
      ++ iicoth(x) should be local but conditional
    iisech    : F -> F
      ++ iisech(x) should be local but conditional
    iicsch    : F -> F
      ++ iicsch(x) should be local but conditional
    iiasinh   : F -> F
      ++ iiasinh(x) should be local but conditional
    iiacosh   : F -> F
      ++ iiacosh(x) should be local but conditional
    iiatanh   : F -> F
      ++ iiatanh(x) should be local but conditional
    iiacoth   : F -> F
      ++ iiacoth(x) should be local but conditional
    iiasech   : F -> F
      ++ iiasech(x) should be local but conditional
    iiacsch   : F -> F
      ++ iiacsch(x) should be local but conditional
    specialTrigs:(F, L Record(func:F,pole:B)) -> Union(F, "failed")
      ++ specialTrigs(x,l) should be local but conditional
    localReal?: F -> Boolean
      ++ localReal?(x) should be local but conditional

  Implementation ==> add
    ipi      : List F -> F
    iexp     : F -> F
    ilog     : F -> F
    iiilog   : F -> F
    isin     : F -> F
    icos     : F -> F
    itan     : F -> F
    icot     : F -> F
    isec     : F -> F
    icsc     : F -> F
    iasin    : F -> F
    iacos    : F -> F
    iatan    : F -> F
    iacot    : F -> F
    iasec    : F -> F
    iacsc    : F -> F
    isinh    : F -> F
    icosh    : F -> F
    itanh    : F -> F
    icoth    : F -> F
    isech    : F -> F
    icsch    : F -> F
    iasinh   : F -> F
    iacosh   : F -> F
```

```
iatanh   : F -> F
iacoth   : F -> F
iasech   : F -> F
iacsch   : F -> F
dropfun  : F -> F
kernel   : F -> K
posrem   :(Z, Z) -> Z
iisqrt1  : () -> F
valueOrPole : Record(func:F, pole:B) -> F

oppi  := operator("pi"::Symbol)$CommonOperators
oplog := operator("log"::Symbol)$CommonOperators
opexp := operator("exp"::Symbol)$CommonOperators
opsin := operator("sin"::Symbol)$CommonOperators
opcos := operator("cos"::Symbol)$CommonOperators
optan := operator("tan"::Symbol)$CommonOperators
opcot := operator("cot"::Symbol)$CommonOperators
opsec := operator("sec"::Symbol)$CommonOperators
opcsc := operator("csc"::Symbol)$CommonOperators
opasin := operator("asin"::Symbol)$CommonOperators
opacos := operator("acos"::Symbol)$CommonOperators
opatan := operator("atan"::Symbol)$CommonOperators
opacot := operator("acot"::Symbol)$CommonOperators
opasec := operator("asec"::Symbol)$CommonOperators
opacsc := operator("acsc"::Symbol)$CommonOperators
opsinh := operator("sinh"::Symbol)$CommonOperators
opcosh := operator("cosh"::Symbol)$CommonOperators
optanh := operator("tanh"::Symbol)$CommonOperators
opcoth := operator("coth"::Symbol)$CommonOperators
opsech := operator("sech"::Symbol)$CommonOperators
opcsch := operator("csch"::Symbol)$CommonOperators
opasinh := operator("asinh"::Symbol)$CommonOperators
opacosh := operator("acosh"::Symbol)$CommonOperators
opatanh := operator("atanh"::Symbol)$CommonOperators
opacoth := operator("acoth"::Symbol)$CommonOperators
opasech := operator("asech"::Symbol)$CommonOperators
opacsch := operator("acsch"::Symbol)$CommonOperators

-- Pi is a domain...
Pie, isqrt1, isqrt2, isqrt3: F

-- following code is conditionalized on arbitraryPrecesion to recompute in
-- case user changes the precision

if R has TranscendentalFunctionCategory then
  Pie := pi()$R :: F
else
  Pie := kernel(oppi, nil()$List(F))

if R has TranscendentalFunctionCategory and R has arbitraryPrecision then
```

```
  pi() == pi()$R :: F
else
  pi() == Pie

if R has imaginary: () -> R then
  isqrt1 := imaginary()$R :: F
else isqrt1 := sqrt(-1::F)

if R has RadicalCategory then
  isqrt2 := sqrt(2::R)::F
  isqrt3 := sqrt(3::R)::F
else
  isqrt2 := sqrt(2::F)
  isqrt3 := sqrt(3::F)

iisqrt1() == isqrt1
if R has RadicalCategory and R has arbitraryPrecision then
  iisqrt2() == sqrt(2::R)::F
  iisqrt3() == sqrt(3::R)::F
else
  iisqrt2() == isqrt2
  iisqrt3() == isqrt3

ipi l == pi()
log x == oplog x
exp x == opexp x
sin x == opsin x
cos x == opcos x
tan x == optan x
cot x == opcot x
sec x == opsec x
csc x == opcsc x
asin x == opasin x
acos x == opacos x
atan x == opatan x
acot x == opacot x
asec x == opasec x
acsc x == opacsc x
sinh x == opsinh x
cosh x == opcosh x
tanh x == optanh x
coth x == opcoth x
sech x == opsech x
csch x == opcsch x
asinh x == opasinh x
acosh x == opacosh x
atanh x == opatanh x
acoth x == opacoth x
asech x == opasech x
acsch x == opacsch x
```

```
kernel x == retract(x)@K

posrem(n, m)    == ((r := n rem m) < 0 => r + m; r)
valueOrPole rec == (rec.pole => INV; rec.func)
belong? op      == has?(op, "elem")

operator op ==
  is?(op, "pi"::Symbol)    => oppi
  is?(op, "log"::Symbol)   => oplog
  is?(op, "exp"::Symbol)   => opexp
  is?(op, "sin"::Symbol)   => opsin
  is?(op, "cos"::Symbol)   => opcos
  is?(op, "tan"::Symbol)   => optan
  is?(op, "cot"::Symbol)   => opcot
  is?(op, "sec"::Symbol)   => opsec
  is?(op, "csc"::Symbol)   => opcsc
  is?(op, "asin"::Symbol)  => opasin
  is?(op, "acos"::Symbol)  => opacos
  is?(op, "atan"::Symbol)  => opatan
  is?(op, "acot"::Symbol)  => opacot
  is?(op, "asec"::Symbol)  => opasec
  is?(op, "acsc"::Symbol)  => opacsc
  is?(op, "sinh"::Symbol)  => opsinh
  is?(op, "cosh"::Symbol)  => opcosh
  is?(op, "tanh"::Symbol)  => optanh
  is?(op, "coth"::Symbol)  => opcoth
  is?(op, "sech"::Symbol)  => opsech
  is?(op, "csch"::Symbol)  => opcsch
  is?(op, "asinh"::Symbol) => opasinh
  is?(op, "acosh"::Symbol) => opacosh
  is?(op, "atanh"::Symbol) => opatanh
  is?(op, "acoth"::Symbol) => opacoth
  is?(op, "asech"::Symbol) => opasech
  is?(op, "acsch"::Symbol) => opacsch
  error "Not an elementary operator"

dropfun x ==
  ((k := retractIfCan(x)@Union(K, "failed")) case "failed") or
    empty?(argument(k::K)) => 0
  first argument(k::K)

if R has RetractableTo Z then
  specialTrigs(x, values) ==
    (r := retractIfCan(y := x/pi())@Union(Fraction Z, "failed"))
      case "failed" => "failed"
    q := r::Fraction(Integer)
    m := minIndex values
    (n := retractIfCan(q)@Union(Z, "failed")) case Z =>
      even?(n::Z) => valueOrPole(values.m)
      valueOrPole(values.(m+1))
```

```
        (n := retractIfCan(2*q)@Union(Z, "failed")) case Z =>
--          one?(s := posrem(n::Z, 4)) => valueOrPole(values.(m+2))
          (s := posrem(n::Z, 4)) = 1 => valueOrPole(values.(m+2))
          valueOrPole(values.(m+3))
        (n := retractIfCan(3*q)@Union(Z, "failed")) case Z =>
--          one?(s := posrem(n::Z, 6)) => valueOrPole(values.(m+4))
          (s := posrem(n::Z, 6)) = 1 => valueOrPole(values.(m+4))
          s = 2 => valueOrPole(values.(m+5))
          s = 4 => valueOrPole(values.(m+6))
          valueOrPole(values.(m+7))
        (n := retractIfCan(4*q)@Union(Z, "failed")) case Z =>
--          one?(s := posrem(n::Z, 8)) => valueOrPole(values.(m+8))
          (s := posrem(n::Z, 8)) = 1 => valueOrPole(values.(m+8))
          s = 3 => valueOrPole(values.(m+9))
          s = 5 => valueOrPole(values.(m+10))
          valueOrPole(values.(m+11))
        (n := retractIfCan(6*q)@Union(Z, "failed")) case Z =>
--          one?(s := posrem(n::Z, 12)) => valueOrPole(values.(m+12))
          (s := posrem(n::Z, 12)) = 1 => valueOrPole(values.(m+12))
          s = 5 => valueOrPole(values.(m+13))
          s = 7 => valueOrPole(values.(m+14))
          valueOrPole(values.(m+15))
        "failed"

    else specialTrigs(x, values) == "failed"

    isin x ==
      zero? x => 0
      y := dropfun x
      is?(x, opasin) => y
      is?(x, opacos) => sqrt(1 - y**2)
      is?(x, opatan) => y / sqrt(1 + y**2)
      is?(x, opacot) => inv sqrt(1 + y**2)
      is?(x, opasec) => sqrt(y**2 - 1) / y
      is?(x, opacsc) => inv y
      h  := inv(2::F)
      s2 := h * iisqrt2()
      s3 := h * iisqrt3()
      u  := specialTrigs(x, [[0,false], [0,false], [1,false], [-1,false],
                          [s3,false], [s3,false], [-s3,false], [-s3,false],
                           [s2,false], [s2,false], [-s2,false], [-s2,false],
                            [h,false], [h,false], [-h,false], [-h,false]])
      u case F => u :: F
      kernel(opsin, x)

    icos x ==
      zero? x => 1
      y := dropfun x
      is?(x, opasin) => sqrt(1 - y**2)
      is?(x, opacos) => y
```

```
    is?(x, opatan) => inv sqrt(1 + y**2)
    is?(x, opacot) => y / sqrt(1 + y**2)
    is?(x, opasec) => inv y
    is?(x, opacsc) => sqrt(y**2 - 1) / y
    h  := inv(2::F)
    s2 := h * iisqrt2()
    s3 := h * iisqrt3()
    u  := specialTrigs(x, [[1,false],[-1,false], [0,false], [0,false],
                           [h,false],[-h,false],[-h,false],[h,false],
                            [s2,false],[-s2,false],[-s2,false],[s2,false],
                             [s3,false], [-s3,false],[-s3,false],[s3,false]])
    u case F => u :: F
    kernel(opcos, x)

  itan x ==
    zero? x => 0
    y := dropfun x
    is?(x, opasin) => y / sqrt(1 - y**2)
    is?(x, opacos) => sqrt(1 - y**2) / y
    is?(x, opatan) => y
    is?(x, opacot) => inv y
    is?(x, opasec) => sqrt(y**2 - 1)
    is?(x, opacsc) => inv sqrt(y**2 - 1)
    s33 := (s3 := iisqrt3()) / (3::F)
    u := specialTrigs(x, [[0,false], [0,false], [0,true], [0,true],
                    [s3,false], [-s3,false], [s3,false], [-s3,false],
                     [1,false], [-1,false], [1,false], [-1,false],
                      [s33,false], [-s33, false], [s33,false], [-s33, false]])
    u case F => u :: F
    kernel(optan, x)

  icot x ==
    zero? x => INV
    y := dropfun x
    is?(x, opasin) => sqrt(1 - y**2) / y
    is?(x, opacos) => y / sqrt(1 - y**2)
    is?(x, opatan) => inv y
    is?(x, opacot) => y
    is?(x, opasec) => inv sqrt(y**2 - 1)
    is?(x, opacsc) => sqrt(y**2 - 1)
    s33 := (s3 := iisqrt3()) / (3::F)
    u := specialTrigs(x, [[0,true], [0,true], [0,false], [0,false],
                       [s33,false], [-s33,false], [s33,false], [-s33,false],
                        [1,false], [-1,false], [1,false], [-1,false],
                         [s3,false], [-s3, false], [s3,false], [-s3, false]])
    u case F => u :: F
    kernel(opcot, x)

  isec x ==
    zero? x => 1
```

```
      y := dropfun x
      is?(x, opasin) => inv sqrt(1 - y**2)
      is?(x, opacos) => inv y
      is?(x, opatan) => sqrt(1 + y**2)
      is?(x, opacot) => sqrt(1 + y**2) / y
      is?(x, opasec) => y
      is?(x, opacsc) => y / sqrt(y**2 - 1)
      s2 := iisqrt2()
      s3 := 2 * iisqrt3() / (3::F)
      h  := 2::F
      u  := specialTrigs(x, [[1,false],[-1,false],[0,true],[0,true],
                           [h,false], [-h,false], [-h,false], [h,false],
                            [s2,false], [-s2,false], [-s2,false], [s2,false],
                             [s3,false], [-s3,false], [-s3,false], [s3,false]])
      u case F => u :: F
      kernel(opsec, x)

    icsc x ==
      zero? x => INV
      y := dropfun x
      is?(x, opasin) => inv y
      is?(x, opacos) => inv sqrt(1 - y**2)
      is?(x, opatan) => sqrt(1 + y**2) / y
      is?(x, opacot) => sqrt(1 + y**2)
      is?(x, opasec) => y / sqrt(y**2 - 1)
      is?(x, opacsc) => y
      s2 := iisqrt2()
      s3 := 2 * iisqrt3() / (3::F)
      h  := 2::F
      u  := specialTrigs(x, [[0,true], [0,true], [1,false], [-1,false],
                            [s3,false], [s3,false], [-s3,false], [-s3,false],
                              [s2,false], [s2,false], [-s2,false], [-s2,false],
                                 [h,false], [h,false], [-h,false], [-h,false]])
      u case F => u :: F
      kernel(opcsc, x)

    iasin x ==
      zero? x => 0
--      one? x =>   pi() / (2::F)
      (x = 1) =>   pi() / (2::F)
      x = -1 => - pi() / (2::F)
      y := dropfun x
      is?(x, opsin) => y
      is?(x, opcos) => pi() / (2::F) - y
      kernel(opasin, x)

    iacos x ==
      zero? x => pi() / (2::F)
--      one? x => 0
      (x = 1) => 0
```

```
      x = -1 => pi()
      y := dropfun x
      is?(x, opsin) => pi() / (2::F) - y
      is?(x, opcos) => y
      kernel(opacos, x)

    iatan x ==
      zero? x => 0
--      one? x =>   pi() / (4::F)
      (x = 1) =>   pi() / (4::F)
      x = -1 => - pi() / (4::F)
      x = (r3:=iisqrt3()) => pi() / (3::F)
--      one?(x*r3)          => pi() / (6::F)
      (x*r3) = 1          => pi() / (6::F)
      y := dropfun x
      is?(x, optan) => y
      is?(x, opcot) => pi() / (2::F) - y
      kernel(opatan, x)

    iacot x ==
      zero? x =>   pi() / (2::F)
--      one? x  =>   pi() / (4::F)
      (x = 1)  =>   pi() / (4::F)
      x = -1  =>   3 * pi() / (4::F)
      x = (r3:=iisqrt3())  =>  pi() / (6::F)
      x = -r3              =>  5 * pi() / (6::F)
--      one?(xx:=x*r3)       =>  pi() / (3::F)
      (xx:=x*r3) = 1      =>  pi() / (3::F)
      xx = -1           =>     2* pi() / (3::F)
      y := dropfun x
      is?(x, optan) => pi() / (2::F) - y
      is?(x, opcot) => y
      kernel(opacot, x)

    iasec x ==
      zero? x => INV
--      one? x => 0
      (x = 1) => 0
      x = -1 => pi()
      y := dropfun x
      is?(x, opsec) => y
      is?(x, opcsc) => pi() / (2::F) - y
      kernel(opasec, x)

    iacsc x ==
      zero? x => INV
--      one? x =>   pi() / (2::F)
      (x = 1) =>   pi() / (2::F)
      x = -1 => - pi() / (2::F)
      y := dropfun x
```

```
    is?(x, opsec) => pi() / (2::F) - y
    is?(x, opcsc) => y
    kernel(opacsc, x)

  isinh x ==
    zero? x => 0
    y := dropfun x
    is?(x, opasinh) => y
    is?(x, opacosh) => sqrt(y**2 - 1)
    is?(x, opatanh) => y / sqrt(1 - y**2)
    is?(x, opacoth) => - inv sqrt(y**2 - 1)
    is?(x, opasech) => sqrt(1 - y**2) / y
    is?(x, opacsch) => inv y
    kernel(opsinh, x)

  icosh x ==
    zero? x => 1
    y := dropfun x
    is?(x, opasinh) => sqrt(y**2 + 1)
    is?(x, opacosh) => y
    is?(x, opatanh) => inv sqrt(1 - y**2)
    is?(x, opacoth) => y / sqrt(y**2 - 1)
    is?(x, opasech) => inv y
    is?(x, opacsch) => sqrt(y**2 + 1) / y
    kernel(opcosh, x)

  itanh x ==
    zero? x => 0
    y := dropfun x
    is?(x, opasinh) => y / sqrt(y**2 + 1)
    is?(x, opacosh) => sqrt(y**2 - 1) / y
    is?(x, opatanh) => y
    is?(x, opacoth) => inv y
    is?(x, opasech) => sqrt(1 - y**2)
    is?(x, opacsch) => inv sqrt(y**2 + 1)
    kernel(optanh, x)

  icoth x ==
    zero? x => INV
    y := dropfun x
    is?(x, opasinh) => sqrt(y**2 + 1) / y
    is?(x, opacosh) => y / sqrt(y**2 - 1)
    is?(x, opatanh) => inv y
    is?(x, opacoth) => y
    is?(x, opasech) => inv sqrt(1 - y**2)
    is?(x, opacsch) => sqrt(y**2 + 1)
    kernel(opcoth, x)

  isech x ==
    zero? x => 1
```

```
    y := dropfun x
    is?(x, opasinh) => inv sqrt(y**2 + 1)
    is?(x, opacosh) => inv y
    is?(x, opatanh) => sqrt(1 - y**2)
    is?(x, opacoth) => sqrt(y**2 - 1) / y
    is?(x, opasech) => y
    is?(x, opacsch) => y / sqrt(y**2 + 1)
    kernel(opsech, x)

  icsch x ==
    zero? x => INV
    y := dropfun x
    is?(x, opasinh) => inv y
    is?(x, opacosh) => inv sqrt(y**2 - 1)
    is?(x, opatanh) => sqrt(1 - y**2) / y
    is?(x, opacoth) => - sqrt(y**2 - 1)
    is?(x, opasech) => y / sqrt(1 - y**2)
    is?(x, opacsch) => y
    kernel(opcsch, x)

  iasinh x ==
    is?(x, opsinh) => first argument kernel x
    kernel(opasinh, x)

  iacosh x ==
    is?(x, opcosh) => first argument kernel x
    kernel(opacosh, x)

  iatanh x ==
    is?(x, optanh) => first argument kernel x
    kernel(opatanh, x)

  iacoth x ==
    is?(x, opcoth) => first argument kernel x
    kernel(opacoth, x)

  iasech x ==
    is?(x, opsech) => first argument kernel x
    kernel(opasech, x)

  iacsch x ==
    is?(x, opcsch) => first argument kernel x
    kernel(opacsch, x)

  iexp x ==
    zero? x => 1
    is?(x, oplog) => first argument kernel x
    x < 0 and empty? variables x => inv iexp(-x)
    h  := inv(2::F)
    i  := iisqrt1()
```

```
      s2 := h * iisqrt2()
      s3 := h * iisqrt3()
      u  := specialTrigs(x / i, [[1,false],[-1,false], [i,false], [-i,false],
            [h + i * s3,false], [-h + i * s3, false], [-h - i * s3, false],
             [h - i * s3, false], [s2 + i * s2, false], [-s2 + i * s2, false],
              [-s2 - i * s2, false], [s2 - i * s2, false], [s3 + i * h, false],
               [-s3 + i * h, false], [-s3 - i * h, false], [s3 - i * h, false]])
      u case F => u :: F
      kernel(opexp, x)

-- THIS DETERMINES WHEN TO PERFORM THE log exp f -> f SIMPLIFICATION
-- CURRENT BEHAVIOR:
--     IF R IS COMPLEX(S) THEN ONLY ELEMENTS WHICH ARE RETRACTABLE TO R
--     AND EQUAL TO THEIR CONJUGATES ARE DEEMED REAL (OVERRESTRICTIVE FOR NOW)
--     OTHERWISE (e.g. R = INT OR FRAC INT), ALL THE ELEMENTS ARE DEEMED REAL

    if (R has imaginary:() -> R) and (R has conjugate: R -> R) then
         localReal? x ==
            (u := retractIfCan(x)@Union(R, "failed")) case R
               and (u::R) = conjugate(u::R)

    else localReal? x == true

    iiilog x ==
      zero? x => INV
--      one? x => 0
      (x = 1) => 0
      (u := isExpt(x, opexp)) case Record(var:K, exponent:Integer) =>
           rec := u::Record(var:K, exponent:Integer)
           arg := first argument(rec.var);
           localReal? arg => rec.exponent * first argument(rec.var);
           ilog x
      ilog x

    ilog x ==
--      ((num1 := one?(num := numer x)) or num = -1) and (den := denom x) ^= 1
      ((num1 := ((num := numer x) = 1)) or num = -1) and (den := denom x) ^= 1
        and empty? variables x => - kernel(oplog, (num1 => den; -den)::F)
      kernel(oplog, x)

    if R has ElementaryFunctionCategory then
      iilog x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iiilog x
        log(r::R)::F

      iiexp x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iexp x
        exp(r::R)::F

    else
```

```
    iilog x == iiilog x
    iiexp x == iexp x

  if R has TrigonometricFunctionCategory then
    iisin x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isin x
      sin(r::R)::F

    iicos x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icos x
      cos(r::R)::F

    iitan x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => itan x
      tan(r::R)::F

    iicot x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icot x
      cot(r::R)::F

    iisec x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isec x
      sec(r::R)::F

    iicsc x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icsc x
      csc(r::R)::F

  else
    iisin x == isin x
    iicos x == icos x
    iitan x == itan x
    iicot x == icot x
    iisec x == isec x
    iicsc x == icsc x

  if R has ArcTrigonometricFunctionCategory then
    iiasin x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasin x
      asin(r::R)::F

    iiacos x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacos x
      acos(r::R)::F

    iiatan x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iatan x
      atan(r::R)::F

    iiacot x ==
```

```
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacot x
      acot(r::R)::F

    iiasec x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasec x
      asec(r::R)::F

    iiacsc x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsc x
      acsc(r::R)::F

  else
    iiasin x == iasin x
    iiacos x == iacos x
    iiatan x == iatan x
    iiacot x == iacot x
    iiasec x == iasec x
    iiacsc x == iacsc x

  if R has HyperbolicFunctionCategory then
    iisinh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isinh x
      sinh(r::R)::F

    iicosh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icosh x
      cosh(r::R)::F

    iitanh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => itanh x
      tanh(r::R)::F

    iicoth x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icoth x
      coth(r::R)::F

    iisech x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isech x
      sech(r::R)::F

    iicsch x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icsch x
      csch(r::R)::F

  else
    iisinh x == isinh x
    iicosh x == icosh x
    iitanh x == itanh x
    iicoth x == icoth x
    iisech x == isech x
```

```
    iicsch x == icsch x

  if R has ArcHyperbolicFunctionCategory then
    iiasinh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasinh x
      asinh(r::R)::F

    iiacosh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacosh x
      acosh(r::R)::F

    iiatanh x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iatanh x
      atanh(r::R)::F

    iiacoth x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacoth x
      acoth(r::R)::F

    iiasech x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasech x
      asech(r::R)::F

    iiacsch x ==
      (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsch x
      acsch(r::R)::F

  else
    iiasinh x == iasinh x
    iiacosh x == iacosh x
    iiatanh x == iatanh x
    iiacoth x == iacoth x
    iiasech x == iasech x
    iiacsch x == iacsch x

  import BasicOperatorFunctions1(F)

  evaluate(oppi, ipi)
  evaluate(oplog, iilog)
  evaluate(opexp, iiexp)
  evaluate(opsin, iisin)
  evaluate(opcos, iicos)
  evaluate(optan, iitan)
  evaluate(opcot, iicot)
  evaluate(opsec, iisec)
  evaluate(opcsc, iicsc)
  evaluate(opasin, iiasin)
  evaluate(opacos, iiacos)
  evaluate(opatan, iiatan)
  evaluate(opacot, iiacot)
```

```
    evaluate(opasec, iiasec)
    evaluate(opacsc, iiacsc)
    evaluate(opsinh, iisinh)
    evaluate(opcosh, iicosh)
    evaluate(optanh, iitanh)
    evaluate(opcoth, iicoth)
    evaluate(opsech, iisech)
    evaluate(opcsch, iicsch)
    evaluate(opasinh, iiasinh)
    evaluate(opacosh, iiacosh)
    evaluate(opatanh, iiatanh)
    evaluate(opacoth, iiacoth)
    evaluate(opasech, iiasech)
    evaluate(opacsch, iiacsch)
    derivative(opexp, exp)
    derivative(oplog, inv)
    derivative(opsin, cos)
    derivative(opcos,(x:F):F +-> - sin x)
    derivative(optan,(x:F):F +-> 1 + tan(x)**2)
    derivative(opcot,(x:F):F +-> - 1 - cot(x)**2)
    derivative(opsec,(x:F):F +-> tan(x) * sec(x))
    derivative(opcsc,(x:F):F +-> - cot(x) * csc(x))
    derivative(opasin,(x:F):F +-> inv sqrt(1 - x**2))
    derivative(opacos,(x:F):F +-> - inv sqrt(1 - x**2))
    derivative(opatan,(x:F):F +-> inv(1 + x**2))
    derivative(opacot,(x:F):F +-> - inv(1 + x**2))
    derivative(opasec,(x:F):F +-> inv(x * sqrt(x**2 - 1)))
    derivative(opacsc,(x:F):F +-> - inv(x * sqrt(x**2 - 1)))
    derivative(opsinh, cosh)
    derivative(opcosh, sinh)
    derivative(optanh,(x:F):F +-> 1 - tanh(x)**2)
    derivative(opcoth,(x:F):F +-> 1 - coth(x)**2)
    derivative(opsech,(x:F):F +-> - tanh(x) * sech(x))
    derivative(opcsch,(x:F):F +-> - coth(x) * csch(x))
    derivative(opasinh,(x:F):F +-> inv sqrt(1 + x**2))
    derivative(opacosh,(x:F):F +-> inv sqrt(x**2 - 1))
    derivative(opatanh,(x:F):F +-> inv(1 - x**2))
    derivative(opacoth,(x:F):F +-> inv(1 - x**2))
    derivative(opasech,(x:F):F +-> - inv(x * sqrt(1 - x**2)))
    derivative(opacsch,(x:F):F +-> - inv(x * sqrt(1 + x**2)))
```

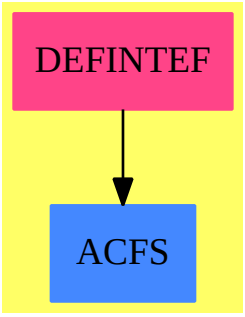
———-

— EF.dotabb —

```
"EF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EF" -> "FS"
```

6.3 package DEFINTEF ElementaryFunctionDefiniteIntegration

6.3.1 ElementaryFunctionDefiniteIntegration (DEFINTEF)



Exports:
innerint integrate integrate

— package DEFINTEF ElementaryFunctionDefiniteIntegration —

```
)abbrev package DEFINTEF ElementaryFunctionDefiniteIntegration
++ Author: Manuel Bronstein
++ Date Created: 14 April 1992
++ Date Last Updated: 2 February 1993
++ Description:
++ \spadtype{ElementaryFunctionDefiniteIntegration}
++ provides functions to compute definite
++ integrals of elementary functions.

ElementaryFunctionDefiniteIntegration(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
           AlgebraicallyClosedFunctionSpace R)

  B   ==> Boolean
  SE  ==> Symbol
  Z   ==> Integer
  P   ==> SparseMultivariatePolynomial(R, K)
  K   ==> Kernel F
```

```
UP  ==> SparseUnivariatePolynomial F
OFE ==> OrderedCompletion F
U   ==> Union(f1:OFE, f2:List OFE, fail:"failed", pole:"potentialPole")

Exports ==> with
  integrate: (F, SegmentBinding OFE) -> U
    ++ integrate(f, x = a..b) returns the integral of
    ++ \spad{f(x)dx} from a to b.
    ++ Error: if f has a pole for x between a and b.
  integrate: (F, SegmentBinding OFE, String) -> U
    ++ integrate(f, x = a..b, "noPole") returns the
    ++ integral of \spad{f(x)dx} from a to b.
    ++ If it is not possible to check whether f has a pole for x
    ++ between a and b (because of parameters), then this function
    ++ will assume that f has no such pole.
    ++ Error: if f has a pole for x between a and b or
    ++ if the last argument is not "noPole".
  innerint: (F, SE, OFE, OFE, B) -> U
    ++ innerint(f, x, a, b, ignore?) should be local but conditional

Implementation ==> add
  import ElementaryFunctionSign(R, F)
  import DefiniteIntegrationTools(R, F)
  import FunctionSpaceIntegration(R, F)

  polyIfCan   : (P, K) -> Union(UP, "failed")
  int         : (F, SE, OFE, OFE, B) -> U
  nopole      : (F, SE, K, OFE, OFE) -> U
  checkFor0   : (P, K, OFE, OFE) -> Union(B, "failed")
  checkSMP    : (P, SE, K, OFE, OFE) -> Union(B, "failed")
  checkForPole: (F, SE, K, OFE, OFE) -> Union(B, "failed")
  posit       : (F, SE, K, OFE, OFE) -> Union(B, "failed")
  negat       : (F, SE, K, OFE, OFE) -> Union(B, "failed")
  moreThan    : (OFE, Fraction Z) -> Union(B, "failed")

  if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
    and F has SpecialFunctionCategory then
      import PatternMatchIntegration(R, F)

      innerint(f, x, a, b, ignor?) ==
        ((u := int(f, x, a, b, ignor?)) case f1) or (u case f2)
          or ((v := pmintegrate(f, x, a, b)) case "failed") => u
        [v::F::OFE]

  else
    innerint(f, x, a, b, ignor?) == int(f, x, a, b, ignor?)

  integrate(f:F, s:SegmentBinding OFE) ==
    innerint(f, variable s, lo segment s, hi segment s, false)
```

```
    integrate(f:F, s:SegmentBinding OFE, str:String) ==
      innerint(f, variable s, lo segment s, hi segment s, ignore? str)

    int(f, x, a, b, ignor?) ==
      a = b => [0::OFE]
      k := kernel(x)@Kernel(F)
      (z := checkForPole(f, x, k, a, b)) case "failed" =>
        ignor? => nopole(f, x, k, a, b)
        ["potentialPole"]
      z::B => error "integrate: pole in path of integration"
      nopole(f, x, k, a, b)

    checkForPole(f, x, k, a, b) ==
      ((u := checkFor0(d := denom f, k, a, b)) case "failed") or (u::B) => u
      ((u := checkSMP(d, x, k, a, b)) case "failed") or (u::B) => u
      checkSMP(numer f, x, k, a, b)

-- true if p has a zero between a and b exclusive
    checkFor0(p, x, a, b) ==
      (u := polyIfCan(p, x)) case UP => checkForZero(u::UP, a, b, false)
      (v := isTimes p) case List(P) =>
         for t in v::List(P) repeat
           ((w := checkFor0(t, x, a, b)) case "failed") or (w::B) => return w
         false
      (r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"
      k := r::K
-- functions with no real zeros
      is?(k, "exp"::SE) or is?(k, "acot"::SE) or is?(k, "cosh"::SE) => false
-- special case for log
      is?(k, "log"::SE) =>
        (w := moreThan(b, 1)) case "failed" or not(w::B) => w
        moreThan(-a, -1)
      "failed"

-- returns true if a > b, false if a < b, "failed" if can't decide
    moreThan(a, b) ==
      (r := retractIfCan(a)@Union(F, "failed")) case "failed" =>  -- infinite
        whatInfinity(a) > 0
      (u := retractIfCan(r::F)@Union(Fraction Z, "failed")) case "failed" =>
        "failed"
      u::Fraction(Z) > b

-- true if p has a pole between a and b
    checkSMP(p, x, k, a, b) ==
      (u := polyIfCan(p, k)) case UP => false
      (v := isTimes p) case List(P) =>
         for t in v::List(P) repeat
           ((w := checkSMP(t, x, k, a, b)) case "failed") or (w::B) => return w
         false
      (v := isPlus p) case List(P) =>
```

```
        n := 0              -- number of summand having a pole
        for t in v::List(P) repeat
          (w := checkSMP(t, x, k, a, b)) case "failed" => return w
          if w::B then n := n + 1
        zero? n => false    -- no summand has a pole
--        one? n => true      -- only one summand has a pole
        (n = 1) => true      -- only one summand has a pole
        "failed"            -- at least 2 summands have a pole
      (r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"
      kk := r::K
      -- nullary operators have no poles
      nullary? operator kk => false
      f := first argument kk
      -- functions which are defined over all the reals:
      is?(kk, "exp"::SE) or is?(kk, "sin"::SE) or is?(kk, "cos"::SE)
        or is?(kk, "sinh"::SE) or is?(kk, "cosh"::SE) or is?(kk, "tanh"::SE)
          or is?(kk, "sech"::SE) or is?(kk, "atan"::SE) or is?(kk, "acot"::SE)
            or is?(kk, "asinh"::SE) => checkForPole(f, x, k, a, b)
      -- functions which are defined on (-1,+1):
      is?(kk, "asin"::SE) or is?(kk, "acos"::SE) or is?(kk, "atanh"::SE) =>
        ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
        ((w := posit(f - 1, x, k, a, b)) case "failed") or (w::B) => w
        negat(f + 1, x, k, a, b)
      -- functions which are defined on (+1, +infty):
      is?(kk, "acosh"::SE) =>
        ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
        negat(f - 1, x, k, a, b)
      -- functions which are defined on (0, +infty):
      is?(kk, "log"::SE) =>
        ((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w
        negat(f, x, k, a, b)
      "failed"

-- returns true if it is certain that f takes at least one strictly positive
-- value for x in (a,b), false if it is certain that f takes no strictly
-- positive value in (a,b), "failed" otherwise
-- f must be known to have no poles in (a,b)
    posit(f, x, k, a, b) ==
      z :=
        (r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
        sign(f, x, r::F, "right")
      (b1 := z case Z) and z::Z > 0 => true
      z :=
        (r := retractIfCan(b)@Union(F, "failed")) case "failed" => sign(f, x, b)
        sign(f, x, r::F, "left")
      (b2 := z case Z) and z::Z > 0 => true
      b1 and b2 =>
        ((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
        false
      "failed"
```

```
-- returns true if it is certain that f takes at least one strictly negative
-- value for x in (a,b), false if it is certain that f takes no strictly
-- negative value in (a,b), "failed" otherwise
-- f must be known to have no poles in (a,b)
    negat(f, x, k, a, b) ==
      z :=
        (r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
        sign(f, x, r::F, "right")
      (b1 := z case Z) and z::Z < 0 => true
      z :=
        (r := retractIfCan(b)@Union(F, "failed")) case "failed" => sign(f, x, b)
        sign(f, x, r::F, "left")
      (b2 := z case Z) and z::Z < 0 => true
      b1 and b2 =>
        ((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
        false
      "failed"

-- returns a UP if p is only a poly w.r.t. the kernel x
    polyIfCan(p, x) ==
      q := univariate(p, x)
      ans:UP := 0
      while q ^= 0 repeat
        member?(x, tower(c := leadingCoefficient(q)::F)) => return "failed"
        ans := ans + monomial(c, degree q)
        q := reductum q
      ans

-- integrate f for x between a and b assuming that f has no pole in between
    nopole(f, x, k, a, b) ==
      (u := integrate(f, x)) case F =>
        (v := computeInt(k, u::F, a, b, false)) case "failed" => ["failed"]
        [v::OFE]
      ans := empty()$List(OFE)
      for g in u::List(F) repeat
        (v := computeInt(k, g, a, b, false)) case "failed" => return ["failed"]
        ans := concat_!(ans, [v::OFE])
      [ans]
```

———-

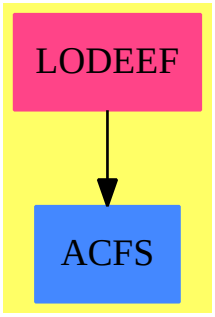
— DEFINTEF.dotabb —

```
"DEFINTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEFINTEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"DEFINTEF" -> "ACFS"
```

———-

6.4 package LODEEF ElementaryFunctionLODESolver

6.4.1 ElementaryFunctionLODESolver (LODEEF)



Exports:
solve

— package LODEEF ElementaryFunctionLODESolver —

```
)abbrev package LODEEF ElementaryFunctionLODESolver
++ Author: Manuel Bronstein
++ Date Created: 3 February 1994
++ Date Last Updated: 9 March 1994
++ Keywords: differential equation, ODE
++ Description:
++ \spad{ElementaryFunctionLODESolver} provides the top-level
++ functions for finding closed form solutions of linear ordinary
++ differential equations and initial value problems.

ElementaryFunctionLODESolver(R, F, L): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
          PrimitiveFunctionCategory)
  L: LinearOrdinaryDifferentialOperatorCategory F

  SY  ==> Symbol
  N   ==> NonNegativeInteger
  K   ==> Kernel F
  V   ==> Vector F
  M   ==> Matrix F
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
```

```
UPUP==> SparseUnivariatePolynomial RF
P   ==> SparseMultivariatePolynomial(R, K)
P2  ==> SparseMultivariatePolynomial(P, K)
LQ  ==> LinearOrdinaryDifferentialOperator1 RF
REC ==> Record(particular: F, basis: List F)
U   ==> Union(REC, "failed")
ALGOP  ==> "%alg"

Exports ==> with
  solve: (L, F, SY) -> U
    ++ solve(op, g, x) returns either a solution of the ordinary differential
    ++ equation \spad{op y = g} or "failed" if no non-trivial solution can be
    ++ found; When found, the solution is returned in the form
    ++ \spad{[h, [b1,...,bm]]} where \spad{h} is a particular solution and
    ++ and \spad{[b1,...bm]} are linearly independent solutions of the
    ++ associated homogenuous equation \spad{op y = 0}.
    ++ A full basis for the solutions of the homogenuous equation
    ++ is not always returned, only the solutions which were found;
    ++ \spad{x} is the dependent variable.
  solve: (L, F, SY, F, List F) -> Union(F, "failed")
    ++ solve(op, g, x, a, [y0,...,ym]) returns either the solution
    ++ of the initial value problem \spad{op y = g, y(a) = y0, y'(a) = y1,...}
    ++ or "failed" if the solution cannot be found;
    ++ \spad{x} is the dependent variable.

Implementation ==> add
  import Kovacic(F, UP)
  import ODETools(F, L)
  import RationalLODE(F, UP)
  import RationalRicDE(F, UP)
  import ODEIntegration(R, F)
  import ConstantLODE(R, F, L)
  import IntegrationTools(R, F)
  import ReductionOfOrder(F, L)
  import ReductionOfOrder(RF, LQ)
  import PureAlgebraicIntegration(R, F, L)
  import FunctionSpacePrimitiveElement(R, F)
  import LinearSystemMatrixPackage(F, V, V, M)
  import SparseUnivariatePolynomialFunctions2(RF, F)
  import FunctionSpaceUnivariatePolynomialFactor(R, F, UP)
  import LinearOrdinaryDifferentialOperatorFactorizer(F, UP)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                        K, R, P, F)

  upmp       : (P, List K) -> P2
  downmp     : (P2, List K, List P) -> P
  xpart      : (F, SY) -> F
  smpxpart   : (P, SY, List K, List P) -> P
  multint    : (F, List F, SY) -> F
  ulodo      : (L, K) -> LQ
```

```
    firstOrder : (F, F, F, SY) -> REC
    rfSolve    : (L, F, K, SY) -> U
    ratlogsol  : (LQ, List RF, K, SY) -> List F
    expsols    : (LQ, K, SY) -> List F
    homosolve  : (L, LQ, List RF, K, SY) -> List F
    homosolve1 : (L, List F, K, SY) -> List F
    norf1      : (L, K, SY, N) -> List F
    kovode     : (LQ, K, SY) -> List F
    doVarParams: (L, F, List F, SY) -> U
    localmap   : (F -> F, L) -> L
    algSolve   : (L, F, K, List K, SY) -> U
    palgSolve  : (L, F, K, K, SY) -> U
    lastChance : (L, F, SY) -> U

    diff := D()$L

    smpxpart(p, x, l, lp) == downnmp(primitivePart upmp(p, l), l, lp)
    downnmp(p, l, lp)      == ground eval(p, l, lp)
    homosolve(lf, op, sols, k, x) == homosolve1(lf, ratlogsol(op,sols,k,x),k,x)

-- left hand side has algebraic (not necessarily pure) coefficients
    algSolve(op, g, k, l, x) ==
      symbolIfCan(kx := ksec(k, l, x)) case SY => palgSolve(op, g, kx, k, x)
      has?(operator kx, ALGOP) =>
        rec := primitiveElement(kx::F, k::F)
        z   := rootOf(rec.prim)
        lk:List K := [kx, k]
        lv:List F := [(rec.pol1) z, (rec.pol2) z]
        (u := solve(localmap((f1:F):F +-> eval(f1, lk, lv), op), _
                              eval(g, lk, lv), x))
            case "failed" => "failed"
        rc := u::REC
        kz := retract(z)@K
        [eval(rc.particular, kz, rec.primelt),
            [eval(f, kz, rec.primelt) for f in rc.basis]]
      lastChance(op, g, x)

    doVarParams(eq, g, bas, x) ==
      (u := particularSolution(eq, g, bas, (f1:F):F +-> int(f1, x)))
         case "failed" => lastChance(eq, g, x)
      [u::F, bas]

    lastChance(op, g, x) ==
--      one? degree op => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
      (degree op) = 1 => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
      "failed"

-- solves a0 y + a1 y' = g
-- does not check whether there is a solution in the field generated by
-- a0, a1 and g
```

```
    firstOrder(a0, a1, g, x) ==
      h := xpart(expint(- a0 / a1, x), x)
      [h * int((g / h) / a1, x), [h]]

-- xpart(f,x) removes any constant not involving x from f
    xpart(f, x) ==
      l  := reverse_! varselect(tower f, x)
      lp := [k::P for k in l]
      smpxpart(numer f, x, l, lp) / smpxpart(denom f, x, l, lp)

    upmp(p, l) ==
      empty? l => p::P2
      up := univariate(p, k := first l)
      l := rest l
      ans:P2 := 0
      while up ^= 0 repeat
        ans := ans + monomial(upmp(leadingCoefficient up, l), k, degree up)
        up  := reductum up
      ans

-- multint(a, [g1,...,gk], x) returns gk \int(g(k-1) \int(....g1 \int(a))...)
    multint(a, l, x) ==
       for g in l repeat a := g * xpart(int(a, x), x)
       a

    expsols(op, k, x) ==
--      one? degree op =>
      (degree op) = 1 =>
          firstOrder(multivariate(coefficient(op, 0), k),
                     multivariate(leadingCoefficient op, k), 0, x).basis
      [xpart(expint(multivariate(h, k), x), x) for h in ricDsolve(op, ffactor)]

-- Finds solutions with rational logarithmic derivative
    ratlogsol(oper, sols, k, x) ==
      bas := [xpart(multivariate(h, k), x) for h in sols]
      degree(oper) = #bas => bas            -- all solutions are found already
      rec := ReduceOrder(oper, sols)
      le := expsols(rec.eq, k, x)
      int:List(F) := [xpart(multivariate(h, k), x) for h in rec.op]
      concat_!([xpart(multivariate(h, k), x) for h in sols],
               [multint(e, int, x) for e in le])

    homosolve1(oper, sols, k, x) ==
      zero?(n := (degree(oper) - #sols)::N) => sols   -- all solutions found
      rec := ReduceOrder(oper, sols)
      int:List(F) := [xpart(h, x) for h in rec.op]
      concat_!(sols, [multint(e, int, x) for e in norf1(rec.eq, k, x, n::N)])

-- if the coefficients are rational functions, then the equation does not
-- not have a proper 1st-order right factor over the rational functions
```

```
    norf1(op, k, x, n) ==
--      one? n => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
      (n = 1) => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
-- for order > 2, we check that the coeffs are still rational functions
      symbolIfCan(kmax vark(coefficients op, x)) case SY =>
        eq := ulodo(op, k)
        n = 2 => kovode(eq, k, x)
        eq := last factor1 eq        -- eq cannot have order 1
        degree(eq) = 2 =>
          empty?(bas := kovode(eq, k, x)) => empty()
          homosolve1(op, bas, k, x)
        empty()
      empty()

    kovode(op, k, x) ==
      b := coefficient(op, 1)
      a := coefficient(op, 2)
      (u := kovacic(coefficient(op, 0), b, a, ffactor)) case "failed" => empty()
      p := map(z1+->multivariate(z1, k), u::UPUP)
      ba := multivariate(- b / a, k)
-- if p has degree 2 (case 2), then it must be squarefree since the
-- ode is irreducible over the rational functions, so the 2 roots of p
-- are distinct and must yield 2 independent solutions.
      degree(p) = 2 => [xpart(expint(ba/(2::F) + e, x), x) for e in zerosOf p]
-- otherwise take 1 root of p and find the 2nd solution by reduction of order
      y1 := xpart(expint(ba / (2::F) + zeroOf p, x), x)
      [y1, y1 * xpart(int(expint(ba, x) / y1**2, x), x)]

    solve(op:L, g:F, x:SY) ==
      empty?(l := vark(coefficients op, x)) => constDsolve(op, g, x)
      symbolIfCan(k := kmax l) case SY => rfSolve(op, g, k, x)
      has?(operator k, ALGOP) => algSolve(op, g, k, l, x)
      lastChance(op, g, x)

    ulodo(eq, k) ==
        op:LQ := 0
        while eq ^= 0 repeat
            op := op + monomial(univariate(leadingCoefficient eq, k), degree eq)
            eq := reductum eq
        op

-- left hand side has rational coefficients
    rfSolve(eq, g, k, x) ==
      op := ulodo(eq, k)
      empty? remove_!(k, varselect(kernels g, x)) =>  -- i.e. rhs is rational
        rc := ratDsolve(op, univariate(g, k))
        rc.particular case "failed" =>                -- this implies g ^= 0
          doVarParams(eq, g, homosolve(eq, op, rc.basis, k, x), x)
        [multivariate(rc.particular::RF, k), homosolve(eq, op, rc.basis, k, x)]
      doVarParams(eq, g, homosolve(eq, op, ratDsolve(op, 0).basis, k, x), x)
```

```
    solve(op, g, x, a, y0) ==
      (u := solve(op, g, x)) case "failed" => "failed"
      hp := h := (u::REC).particular
      b := (u::REC).basis
      v:V := new(n := #y0, 0)
      kx:K := kernel x
      for i in minIndex v .. maxIndex v for yy in y0 repeat
        v.i := yy - eval(h, kx, a)
        h := diff h
      (sol := particularSolution(
         map_!((f1:F):F+->eval(f1,kx,a),wronskianMatrix(b,n)), v))
           case "failed" => "failed"
      for f in b for i in minIndex(s := sol::V) .. repeat
        hp := hp + s.i * f
      hp

    localmap(f, op) ==
        ans:L := 0
        while op ^= 0 repeat
            ans := ans + monomial(f leadingCoefficient op, degree op)
            op  := reductum op
        ans

-- left hand side has pure algebraic coefficients
    palgSolve(op, g, kx, k, x) ==
      rec := palgLODE(op, g, kx, k, x)   -- finds solutions in the coef. field
      rec.particular case "failed" =>
        doVarParams(op, g, homosolve1(op, rec.basis, k, x), x)
      [(rec.particular)::F, homosolve1(op, rec.basis, k, x)]
```

———-

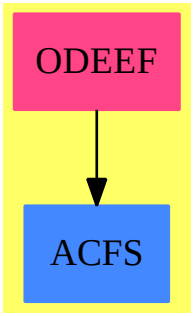
— LODEEF.dotabb —

```
"LODEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODEEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"LODEEF" -> "ACFS"
```

———-

6.5 package ODEEF ElementaryFunctionODESolver

6.5.1 ElementaryFunctionODESolver (ODEEF)



Exports:
solve

— package ODEEF ElementaryFunctionODESolver —

```
)abbrev package ODEEF ElementaryFunctionODESolver
++ Author: Manuel Bronstein
++ Date Created: 18 March 1991
++ Date Last Updated: 8 March 1994
++ Keywords: differential equation, ODE
++ Description:
++ \spad{ElementaryFunctionODESolver} provides the top-level
++ functions for finding closed form solutions of ordinary
++ differential equations and initial value problems.

ElementaryFunctionODESolver(R, F): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
          PrimitiveFunctionCategory)

  N   ==> NonNegativeInteger
  OP  ==> BasicOperator
  SY  ==> Symbol
  K   ==> Kernel F
  EQ  ==> Equation F
  V   ==> Vector F
  M   ==> Matrix F
  UP  ==> SparseUnivariatePolynomial F
  P   ==> SparseMultivariatePolynomial(R, K)
  LEQ ==> Record(left:UP, right:F)
  NLQ ==> Record(dx:F, dy:F)
```

```
REC ==> Record(particular: F, basis: List F)
VEC ==> Record(particular: V, basis: List V)
ROW ==> Record(index: Integer, row: V, rh: F)
SYS ==> Record(mat:M, vec: V)
U   ==> Union(REC, F, "failed")
UU  ==> Union(F, "failed")
OPDIFF ==> "%diff"::SY

Exports ==> with
  solve: (M, V, SY) -> Union(VEC, "failed")
    ++ solve(m, v, x) returns \spad{[v_p, [v_1,...,v_m]]} such that
    ++ the solutions of the system \spad{D y = m y + v} are
    ++ \spad{v_p + c_1 v_1 + ... + c_m v_m} where the \spad{c_i's} are
    ++ constants, and the \spad{v_i's} form a basis for the solutions of
    ++ \spad{D y = m y}.
    ++ \spad{x} is the dependent variable.
  solve: (M, SY) -> Union(List V, "failed")
    ++ solve(m, x) returns a basis for the solutions of \spad{D y = m y}.
    ++ \spad{x} is the dependent variable.
  solve: (List EQ, List OP, SY) -> Union(VEC, "failed")
    ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
    ++ or, if the equations form a fist order linear system, a solution
    ++ of the form \spad{[y_p, [b_1,...,b_n]]} where \spad{h_p} is a
    ++ particular solution and \spad{[b_1,...b_m]} are linearly independent
    ++ solutions of the associated homogenuous system.
    ++ error if the equations do not form a first order linear system
  solve: (List F, List OP, SY) -> Union(VEC, "failed")
    ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
    ++ or, if the equations form a fist order linear system, a solution
    ++ of the form \spad{[y_p, [b_1,...,b_n]]} where \spad{h_p} is a
    ++ particular solution and \spad{[b_1,...b_m]} are linearly independent
    ++ solutions of the associated homogenuous system.
    ++ error if the equations do not form a first order linear system
  solve: (EQ, OP, SY) -> U
    ++ solve(eq, y, x) returns either a solution of the ordinary differential
    ++ equation \spad{eq} or "failed" if no non-trivial solution can be found;
    ++ If the equation is linear ordinary, a solution is of the form
    ++ \spad{[h, [b1,...,bm]]} where \spad{h} is a particular solution
    ++ and \spad{[b1,...bm]} are linearly independent solutions of the
    ++ associated homogenuous equation \spad{f(x,y) = 0};
    ++ A full basis for the solutions of the homogenuous equation
    ++ is not always returned, only the solutions which were found;
    ++ If the equation is of the form {dy/dx = f(x,y)}, a solution is of
    ++ the form \spad{h(x,y)} where \spad{h(x,y) = c} is a first integral
    ++ of the equation for any constant \spad{c};
    ++ error if the equation is not one of those 2 forms;
  solve: (F, OP, SY) -> U
    ++ solve(eq, y, x) returns either a solution of the ordinary differential
    ++ equation \spad{eq} or "failed" if no non-trivial solution can be found;
    ++ If the equation is linear ordinary, a solution is of the form
```

```
      ++ \spad{[h, [b1,...,bm]]} where \spad{h} is a particular solution and
      ++ and \spad{[b1,...bm]} are linearly independent solutions of the
      ++ associated homogenuous equation \spad{f(x,y) = 0};
      ++ A full basis for the solutions of the homogenuous equation
      ++ is not always returned, only the solutions which were found;
      ++ If the equation is of the form {dy/dx = f(x,y)}, a solution is of
      ++ the form \spad{h(x,y)} where \spad{h(x,y) = c} is a first integral
      ++ of the equation for any constant \spad{c};
    solve: (EQ, OP, EQ, List F) -> UU
      ++ solve(eq, y, x = a, [y0,...,ym]) returns either the solution
      ++ of the initial value problem \spad{eq, y(a) = y0, y'(a) = y1,...}
      ++ or "failed" if the solution cannot be found;
      ++ error if the equation is not one linear ordinary or of the form
      ++ \spad{dy/dx = f(x,y)};
    solve: (F, OP, EQ, List F) -> UU
      ++ solve(eq, y, x = a, [y0,...,ym]) returns either the solution
      ++ of the initial value problem \spad{eq, y(a) = y0, y'(a) = y1,...}
      ++ or "failed" if the solution cannot be found;
      ++ error if the equation is not one linear ordinary or of the form
      ++ \spad{dy/dx = f(x,y)};

  Implementation ==> add
    import ODEIntegration(R, F)
    import IntegrationTools(R, F)
    import NonLinearFirstOrderODESolver(R, F)

    getfreelincoeff : (F, K, SY) -> F
    getfreelincoeff1: (F, K, List F) -> F
    getlincoeff     : (F, K) -> F
    getcoeff        : (F, K) -> UU
    parseODE        : (F, OP, SY) -> Union(LEQ, NLQ)
    parseLODE       : (F, List K, UP, SY) -> LEQ
    parseSYS        : (List F, List OP, SY) -> Union(SYS, "failed")
    parseSYSeq      : (F, List K, List K, List F, SY) -> Union(ROW, "failed")

    solve(diffeq:EQ, y:OP, x:SY) == solve(lhs diffeq - rhs diffeq, y, x)

    solve(leq: List EQ, lop: List OP, x:SY) ==
        solve([lhs eq - rhs eq for eq in leq], lop, x)

    solve(diffeq:EQ, y:OP, center:EQ, y0:List F) ==
      solve(lhs diffeq - rhs diffeq, y, center, y0)

    solve(m:M, x:SY) ==
        (u := solve(m, new(nrows m, 0), x)) case "failed" => "failed"
        u.basis

    solve(m:M, v:V, x:SY) ==
        Lx := LinearOrdinaryDifferentialOperator(F, diff x)
        uu := solve(m, v, (z1,z2) +-> solve(z1, z2, x)_
```

```
          $ElementaryFunctionLODESolver(R, F, Lx))$SystemODESolver(F, Lx)
        uu case "failed" => "failed"
        rec := uu::Record(particular: V, basis: M)
        [rec.particular, [column(rec.basis, i) for i in 1..ncols(rec.basis)]]

    solve(diffeq:F, y:OP, center:EQ, y0:List F) ==
      a := rhs center
      kx:K := kernel(x := retract(lhs(center))@SY)
      (ur := parseODE(diffeq, y, x)) case NLQ =>
--        not one?(#y0) => error "solve: more than one initial condition!"
        not ((#y0) = 1) => error "solve: more than one initial condition!"
        rc := ur::NLQ
        (u := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
        u::F - eval(u::F,  [kx, retract(y(x::F))@K], [a, first y0])
      rec := ur::LEQ
      p := rec.left
      Lx := LinearOrdinaryDifferentialOperator(F, diff x)
      op:Lx := 0
      while p ^= 0 repeat
        op := op + monomial(leadingCoefficient p, degree p)
        p  := reductum p
      solve(op, rec.right, x, a, y0)$ElementaryFunctionLODESolver(R, F, Lx)

    solve(leq: List F, lop: List OP, x:SY) ==
        (u := parseSYS(leq, lop, x)) case SYS =>
            rec := u::SYS
            solve(rec.mat, rec.vec, x)
        error "solve: not a first order linear system"

    solve(diffeq:F, y:OP, x:SY) ==
      (u := parseODE(diffeq, y, x)) case NLQ =>
        rc := u::NLQ
        (uu := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
        uu::F
      rec := u::LEQ
      p := rec.left
      Lx := LinearOrdinaryDifferentialOperator(F, diff x)
      op:Lx := 0
      while p ^= 0 repeat
        op := op + monomial(leadingCoefficient p, degree p)
        p  := reductum p
      (uuu := solve(op, rec.right, x)$ElementaryFunctionLODESolver(R, F, Lx))
         case "failed" => "failed"
      uuu::REC

-- returns [M, v] s.t. the equations are D x = M x + v
    parseSYS(eqs, ly, x) ==
      (n := #eqs) ^= #ly => "failed"
      m:M := new(n, n, 0)
      v:V := new(n, 0)
```

```
      xx := x::F
      lf := [y xx for y in ly]
      lk0:List(K) := [retract(f)@K for f in lf]
      lk1:List(K) := [retract(differentiate(f, x))@K for f in lf]
      for eq in eqs repeat
          (u := parseSYSeq(eq,lk0,lk1,lf,x)) case "failed" => return "failed"
          rec := u::ROW
          setRow_!(m, rec.index, rec.row)
          v(rec.index) := rec.rh
      [m, v]

    parseSYSeq(eq, l0, l1, lf, x) ==
      l := [k for k in varselect(kernels eq, x) | is?(k, OPDIFF)]
      empty? l or not empty? rest l or zero?(n := position(k := first l,l1)) =>
         "failed"
      c := getfreelincoeff1(eq, k, lf)
      eq := eq - c * k::F
      v:V := new(#l0, 0)
      for y in l0 for i in 1.. repeat
          ci := getfreelincoeff1(eq, y, lf)
          v.i := - ci / c
          eq := eq - ci * y::F
      [n, v, -eq]

-- returns either [p, g] where the equation (diffeq) is of the form p(D)(y) = g
-- or [p, q] such that the equation (diffeq) is of the form p dx + q dy = 0
    parseODE(diffeq, y, x) ==
      f := y(x::F)
      l:List(K) := [retract(f)@K]
      n:N := 2
      for k in varselect(kernels diffeq, x) | is?(k, OPDIFF) repeat
        if (m := height k) > n then n := m
      n := (n - 2)::N
-- build a list of kernels in the order [y^(n)(x),...,y''(x),y'(x),y(x)]
      for i in 1..n repeat
        l := concat(retract(f := differentiate(f, x))@K, l)
      k:K   -- #$^#& compiler requires this line and the next one too...
      c:F
      while not(empty? l) and zero?(c := getlincoeff(diffeq, k := first l))
        repeat l := rest l
      empty? l or empty? rest l => error "parseODE: equation has order 0"
      diffeq := diffeq - c * (k::F)
      ny := name y
      l := rest l
      height(k) > 3 => parseLODE(diffeq, l, monomial(c, #l), ny)
      (u := getcoeff(diffeq, k := first l)) case "failed" => [diffeq, c]
      eqrhs := (d := u::F) * (k::F) - diffeq
      freeOf?(eqrhs, ny) and freeOf?(c, ny) and freeOf?(d, ny) =>
        [monomial(c, 1) + d::UP, eqrhs]
      [diffeq, c]
```

```
-- returns [p, g] where the equation (diffeq) is of the form p(D)(y) = g
    parseLODE(diffeq, l, p, y) ==
      not freeOf?(leadingCoefficient p, y) =>
        error "parseLODE: not a linear ordinary differential equation"
      d := degree(p)::Integer - 1
      for k in l repeat
        p := p + monomial(c := getfreelincoeff(diffeq, k, y), d::N)
        d := d - 1
        diffeq := diffeq - c * (k::F)
      freeOf?(diffeq, y) => [p, - diffeq]
      error "parseLODE: not a linear ordinary differential equation"

    getfreelincoeff(f, k, y) ==
      freeOf?(c := getlincoeff(f, k), y) => c
      error "getfreelincoeff: not a linear ordinary differential equation"

    getfreelincoeff1(f, k, ly) ==
      c := getlincoeff(f, k)
      for y in ly repeat
         not freeOf?(c, y) =>
           error "getfreelincoeff: not a linear ordinary differential equation"
      c

    getlincoeff(f, k) ==
      (u := getcoeff(f, k)) case "failed" =>
        error "getlincoeff: not an appropriate ordinary differential equation"
      u::F

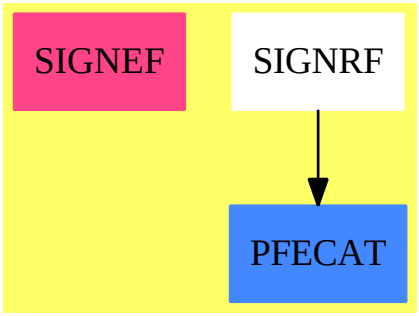
    getcoeff(f, k) ==
      (r := retractIfCan(univariate(denom f, k))@Union(P, "failed"))
        case "failed" or degree(p := univariate(numer f, k)) > 1 => "failed"
      coefficient(p, 1) / (r::P)
```

— ODEEF.dotabb —

```
"ODEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODEEF" -> "ACFS"
```

6.6 package SIGNEF ElementaryFunctionSign

6.6.1 ElementaryFunctionSign (SIGNEF)



Exports:
sign

— package SIGNEF ElementaryFunctionSign —

```
)abbrev package SIGNEF ElementaryFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1989
++ Date Last Updated: 4 May 1992
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: elementary function, sign
++ Examples:
++ References:
++ Description:
++ This package provides functions to determine the sign of an
++ elementary function around a point or infinity.

ElementaryFunctionSign(R,F): Exports == Implementation where
  R : Join(IntegralDomain,OrderedSet,RetractableTo Integer,_
           LinearlyExplicitRingOver Integer,GcdDomain)
  F : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
            FunctionSpace R)

  N  ==> NonNegativeInteger
  Z  ==> Integer
  SY ==> Symbol
  RF ==> Fraction Polynomial R
  ORF ==> OrderedCompletion RF
  OFE ==> OrderedCompletion F
```

```
K  ==> Kernel F
P  ==> SparseMultivariatePolynomial(R, K)
U  ==> Union(Z, "failed")
FS2 ==> FunctionSpaceFunctions2
POSIT ==> "positive"
NEGAT ==> "negative"

Exports ==> with
  sign: F -> U
    ++ sign(f) returns the sign of f if it is constant everywhere.
  sign: (F, SY, OFE) -> U
    ++ sign(f, x, a) returns the sign of f as x nears \spad{a}, from both
    ++ sides if \spad{a} is finite.
  sign: (F, SY, F, String) -> U
    ++ sign(f, x, a, s) returns the sign of f as x nears \spad{a} from below
    ++ if s is "left", or above if s is "right".

Implementation ==> add
  import ToolsForSign R
  import RationalFunctionSign(R)
  import PowerSeriesLimitPackage(R, F)
  import TrigonometricManipulations(R, F)

  smpsign : P -> U
  sqfrSign: P -> U
  termSign: P -> U
  kerSign : K -> U
  listSign: (List P,Z) -> U
  insign  : (F,SY,OFE, N) -> U
  psign   : (F,SY,F,String, N) -> U
  ofesign : OFE -> U
  overRF  : OFE -> Union(ORF, "failed")

  sign(f, x, a) ==
    not real? f => "failed"
    insign(f, x, a, 0)

  sign(f, x, a, st) ==
    not real? f => "failed"
    psign(f, x, a, st, 0)

  sign f ==
    not real? f => "failed"
    (u := retractIfCan(f)@Union(RF,"failed")) case RF => sign(u::RF)
    (un := smpsign numer f) case Z and (ud := smpsign denom f) case Z =>
      un::Z * ud::Z
    --abort if there are any variables
    not empty? variables f => "failed"
    -- abort in the presence of algebraic numbers
    member?(coerce("rootOf")::Symbol,
```

```
    map(name,operators f)$ListFunctions2(BasicOperator,Symbol)) => "failed"
  -- In the last resort try interval evaluation where feasible.
  if R has ConvertibleTo Float then
    import Interval(Float)
    import Expression(Interval Float)
    mapfun : (R -> Interval(Float)) := z +-> interval(convert(z)$R)
    f2 : Expression(Interval Float) :=
        map(mapfun,f)$FS2(R,F,Interval(Float),Expression(Interval Float))
    r : Union(Interval(Float),"failed") := retractIfCan f2
    if r case "failed" then  return "failed"
    negative? r => return(-1)
    positive? r => return 1
    zero? r => return 0
    "failed"
  "failed"

overRF a ==
  (n := whatInfinity a) = 0 =>
    (u := retractIfCan(retract(a)@F)@Union(RF,"failed")) _
           case "failed" => "failed"
    u::RF::ORF
  n * plusInfinity()$ORF

ofesign a ==
  (n := whatInfinity a) ^= 0 => convert(n)@Z
  sign(retract(a)@F)

insign(f, x, a, m) ==
  m > 10 => "failed"                -- avoid infinite loops for now
  (uf := retractIfCan(f)@Union(RF,"failed")) case RF and
              (ua := overRF a) case ORF => sign(uf::RF, x, ua::ORF)
  eq : Equation OFE := equation(x :: F :: OFE,a)
  (u := limit(f,eq)) case "failed" => "failed"
  u case OFE =>
    (n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
    (v := retract(u::OFE)@F) = 0 =>
      (s := insign(differentiate(f, x), x, a, m + 1)) case "failed"
                                                      => "failed"
      - s::Z * n
    sign v
  (u.leftHandLimit case "failed") or
     (u.rightHandLimit case "failed") => "failed"
  (ul := ofesign(u.leftHandLimit::OFE))  case "failed" => "failed"
  (ur := ofesign(u.rightHandLimit::OFE)) case "failed" => "failed"
  (ul::Z) = (ur::Z) => ul
  "failed"

psign(f, x, a, st, m) ==
  m > 10 => "failed"                -- avoid infinite loops for now
  f = 0 => 0
```

```
    (uf := retractIfCan(f)@Union(RF,"failed")) case RF and
         (ua := retractIfCan(a)@Union(RF,"failed")) case RF =>
          sign(uf::RF, x, ua::RF, st)
    eq : Equation F := equation(x :: F,a)
    (u := limit(f,eq,st)) case "failed" => "failed"
    u case OFE =>
      (n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
      (v := retract(u::OFE)@F) = 0 =>
        (s := psign(differentiate(f,x),x,a,st,m + 1)) case "failed"=>
          "failed"
        direction(st) * s::Z
      sign v

  smpsign p ==
    (r := retractIfCan(p)@Union(R,"failed")) case R => sign(r::R)
    (u := sign(retract(unit(s := squareFree p))@R)) case "failed" =>
      "failed"
    ans := u::Z
    for term in factorList s | odd?(term.xpnt) repeat
      (u := sqfrSign(term.fctr)) case "failed" => return "failed"
      ans := ans * u::Z
    ans

  sqfrSign p ==
    (u := termSign first(l := monomials p)) case "failed" => "failed"
    listSign(rest l, u::Z)

  listSign(l, s) ==
    for term in l repeat
      (u := termSign term) case "failed" => return "failed"
      not(s = u::Z) => return "failed"
    s

  termSign term ==
    (us := sign leadingCoefficient term) case "failed" => "failed"
    for var in (lv := variables term) repeat
      odd? degree(term, var) =>
        empty? rest lv and (vs := kerSign first lv) case Z =>
                                                 return(us::Z * vs::Z)
        return "failed"
    us::Z

  kerSign k ==
    has?(op := operator k, "NEGAT") => -1
    has?(op, "POSIT") or is?(op,  "pi"::SY) or is?(op,"exp"::SY) or
                         is?(op,"cosh"::SY) or is?(op,"sech"::SY) => 1
    empty?(arg := argument k) => "failed"
    (s := sign first arg) case "failed" =>
      is?(op,"nthRoot" :: SY) =>
        even?(retract(second arg)@Z) => 1
```

```
        "failed"
      "failed"
    is?(op,"log" :: SY) =>
      s::Z < 0 => "failed"
      sign(first arg - 1)
    is?(op,"tanh" :: SY) or is?(op,"sinh" :: SY) or
                   is?(op,"csch" :: SY) or is?(op,"coth" :: SY) => s
    is?(op,"nthRoot" :: SY) =>
      even?(retract(second arg)@Z) =>
        s::Z < 0 => "failed"
        s
      s
    "failed"
```

———-

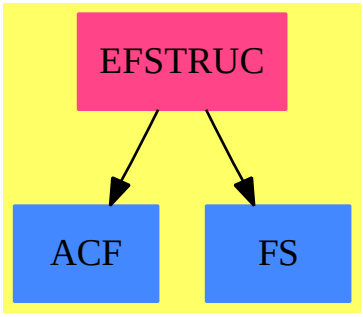
— **SIGNEF.dotabb** —

```
"SIGNEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIGNEF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SIGNRF" -> "PFECAT"
```

———-

6.7 package EFSTRUC ElementaryFunctionStructurePackage

6.7.1 ElementaryFunctionStructurePackage (EFSTRUC)



Exports:
normalize realElementary rootNormalize rischNormalize tanQ validExponential

— package EFSTRUC ElementaryFunctionStructurePackage —

```
)abbrev package EFSTRUC ElementaryFunctionStructurePackage
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 16 August 1995
++ Keywords: elementary, function, structure.
++ Description:
++ ElementaryFunctionStructurePackage provides functions to test the
++ algebraic independence of various elementary functions, using the
++ Risch structure theorem (real and complex versions).
++ It also provides transformations on elementary functions
++ which are not considered simplifications.

ElementaryFunctionStructurePackage(R,F): Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  B   ==> Boolean
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Z
  SY  ==> Symbol
  K   ==> Kernel F
  UP  ==> SparseUnivariatePolynomial F
  SMP ==> SparseMultivariatePolynomial(R, K)
  REC ==> Record(func:F, kers: List K, vals:List F)
  U   ==> Union(vec:Vector Q, func:F, fail: Boolean)
  POWER ==> "%power"::SY
  NTHR  ==> "nthRoot"::SY

  Exports ==> with
    normalize: F -> F
      ++ normalize(f) rewrites \spad{f} using the least possible number of
      ++ real algebraically independent kernels.
    normalize: (F, SY) -> F
      ++ normalize(f, x) rewrites \spad{f} using the least possible number of
      ++ real algebraically independent kernels involving \spad{x}.
    rischNormalize: (F, SY) -> REC
      ++ rischNormalize(f, x) returns \spad{[g, [k1,...,kn], [h1,...,hn]]}
      ++ such that \spad{g = normalize(f, x)} and each \spad{ki} was
      ++ rewritten as \spad{hi} during the normalization.
    realElementary: F -> F
      ++ realElementary(f) rewrites \spad{f} in terms of the 4 fundamental real
      ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
    realElementary: (F, SY) -> F
      ++ realElementary(f,x) rewrites the kernels of \spad{f} involving
```

```
    ++ \spad{x} in terms of the 4 fundamental real
    ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
  validExponential: (List K, F, SY) -> Union(F, "failed")
    ++ validExponential([k1,...,kn],f,x) returns \spad{g} if \spad{exp(f)=g}
    ++ and \spad{g} involves only \spad{k1...kn}, and "failed" otherwise.
  rootNormalize: (F, K) -> F
    ++ rootNormalize(f, k) returns \spad{f} rewriting either \spad{k} which
    ++ must be an nth-root in terms of radicals already in \spad{f}, or some
    ++ radicals in \spad{f} in terms of \spad{k}.
  tanQ: (Q, F) -> F
    ++ tanQ(q,a) is a local function with a conditional implementation.

Implementation ==> add
  import TangentExpansions F
  import IntegrationTools(R, F)
  import IntegerLinearDependence F
  import AlgebraicManipulations(R, F)
  import InnerCommonDenominator(Z, Q, Vector Z, Vector Q)

  k2Elem             : (K, List SY) -> F
  realElem           : (F, List SY) -> F
  smpElem            : (SMP, List SY) -> F
  deprel             : (List K, K, SY) -> U
  rootDep            : (List K, K)     -> U
  qdeprel            : (List F, F)     -> U
  factdeprel         : (List K, K)     -> U
  toR                : (List K, F) -> List K
  toY                : List K -> List F
  toZ                : List K -> List F
  toU                : List K -> List F
  toV                : List K -> List F
  ktoY               : K  -> F
  ktoZ               : K  -> F
  ktoU               : K  -> F
  ktoV               : K  -> F
  gdCoef?            : (Q, Vector Q) -> Boolean
  goodCoef           : (Vector Q, List K, SY) ->
                              Union(Record(index:Z, ker:K), "failed")
  tanRN              : (Q, K) -> F
  localnorm          : F -> F
  rooteval           : (F, List K, K, Q) -> REC
  logeval            : (F, List K, K, Vector Q) -> REC
  expeval            : (F, List K, K, Vector Q) -> REC
  taneval            : (F, List K, K, Vector Q) -> REC
  ataneval           : (F, List K, K, Vector Q) -> REC
  depeval            : (F, List K, K, Vector Q) -> REC
  expnosimp          : (F, List K, K, Vector Q, List F, F) -> REC
  tannosimp          : (F, List K, K, Vector Q, List F, F) -> REC
  rtNormalize        : F -> F
  rootNormalize0     : F -> REC
```

```
rootKernelNormalize: (F, List K, K) -> Union(REC, "failed")
tanSum             : (F, List F) -> F

comb?     := F has CombinatorialOpsCategory
mpiover2:F := pi()$F / (-2::F)

realElem(f, l)       == smpElem(numer f, l) / smpElem(denom f, l)
realElementary(f, x) == realElem(f, [x])
realElementary f     == realElem(f, variables f)
toY ker              == [func for k in ker | (func := ktoY k) ^= 0]
toZ ker              == [func for k in ker | (func := ktoZ k) ^= 0]
toU ker              == [func for k in ker | (func := ktoU k) ^= 0]
toV ker              == [func for k in ker | (func := ktoV k) ^= 0]
rtNormalize f        == rootNormalize0(f).func
toR(ker, x) == select(s+->is?(s, NTHR) and first argument(s) = x, ker)

if R has GcdDomain then
  tanQ(c, x) ==
    tanNa(rootSimp zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)
else
  tanQ(c, x) ==
    tanNa(zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)

-- tanSum(c, [a1,...,an]) returns f(c, a1,...,an) such that
-- if ai = tan(ui) then f(c, a1,...,an) = tan(c + u1 + ... + un).
-- MUST BE CAREFUL FOR WHEN c IS AN ODD MULTIPLE of pi/2
tanSum(c, l) ==
  k := c / mpiover2        -- k = - 2 c / pi, check for odd integer
                           -- tan((2n+1) pi/2 x) = - 1 / tan x
  (r := retractIfCan(k)@Union(Z, "failed")) case Z and odd?(r::Z) =>
       - inv tanSum l
  tanSum concat(tan c, l)

rootNormalize0 f ==
  ker := select_!(s+->is?(s, NTHR) and empty? variables first argument s,
                  tower f)$List(K)
  empty? ker => [f, empty(), empty()]
  (n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
  for i in 1..n for kk in rest ker repeat
    (u := rootKernelNormalize(f, first(ker, i), kk)) case REC =>
      rec := u::REC
      rn  := rootNormalize0(rec.func)
      return [rn.func, concat(rec.kers,rn.kers), concat(rec.vals, rn.vals)]
  [f, empty(), empty()]

deprel(ker, k, x) ==
  is?(k, "log"::SY) or is?(k, "exp"::SY) =>
    qdeprel([differentiate(g, x) for g in toY ker],
              differentiate(ktoY k, x))
  is?(k, "atan"::SY) or is?(k, "tan"::SY) =>
```

```
      qdeprel([differentiate(g, x) for g in toU ker],
              differentiate(ktoU k, x))
    is?(k, NTHR) => rootDep(ker, k)
    comb? and is?(k, "factorial"::SY) =>
      factdeprel([x for x in ker | is?(x,"factorial"::SY) and x^=k],k)
    [true]

  ktoY k ==
    is?(k, "log"::SY) => k::F
    is?(k, "exp"::SY) => first argument k
    0

  ktoZ k ==
    is?(k, "log"::SY) => first argument k
    is?(k, "exp"::SY) => k::F
    0

  ktoU k ==
    is?(k, "atan"::SY) => k::F
    is?(k,  "tan"::SY) => first argument k
    0

  ktoV k ==
    is?(k,  "tan"::SY) => k::F
    is?(k, "atan"::SY) => first argument k
    0

  smpElem(p, l) ==
    map(x+->k2Elem(x, l), y+->y::F, p)_
     $PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)

  k2Elem(k, l) ==
    ez, iez, tz2: F
    kf := k::F
    not(empty? l) and empty? [v for v in variables kf | member?(v, l)] => kf
    empty?(args :List F := [realElem(a, l) for a in argument k]) => kf
    z := first args
    is?(k, POWER)       => (zero? z => 0; exp(last(args) * log z))
    is?(k, "cot"::SY)   => inv tan z
    is?(k, "acot"::SY)  => atan inv z
    is?(k, "asin"::SY)  => atan(z / sqrt(1 - z**2))
    is?(k, "acos"::SY)  => atan(sqrt(1 - z**2) / z)
    is?(k, "asec"::SY)  => atan sqrt(1 - z**2)
    is?(k, "acsc"::SY)  => atan inv sqrt(1 - z**2)
    is?(k, "asinh"::SY) => log(sqrt(1 + z**2) + z)
    is?(k, "acosh"::SY) => log(sqrt(z**2 - 1) + z)
    is?(k, "atanh"::SY) => log((z + 1) / (1 - z)) / (2::F)
    is?(k, "acoth"::SY) => log((z + 1) / (z - 1)) / (2::F)
    is?(k, "asech"::SY) => log((inv z) + sqrt(inv(z**2) - 1))
    is?(k, "acsch"::SY) => log((inv z) + sqrt(1 + inv(z**2)))
```

```
      is?(k, "%paren"::SY) or is?(k, "%box"::SY) =>
        empty? rest args => z
        kf
      if has?(op := operator k, "htrig") then iez  := inv(ez  := exp z)
      is?(k, "sinh"::SY)  => (ez - iez) / (2::F)
      is?(k, "cosh"::SY)  => (ez + iez) / (2::F)
      is?(k, "tanh"::SY)  => (ez - iez) / (ez + iez)
      is?(k, "coth"::SY)  => (ez + iez) / (ez - iez)
      is?(k, "sech"::SY)  => 2 * inv(ez + iez)
      is?(k, "csch"::SY)  => 2 * inv(ez - iez)
      if has?(op, "trig") then tz2  := tan(z / (2::F))
      is?(k, "sin"::SY)   => 2 * tz2 / (1 + tz2**2)
      is?(k, "cos"::SY)   => (1 - tz2**2) / (1 + tz2**2)
      is?(k, "sec"::SY)   => (1 + tz2**2) / (1 - tz2**2)
      is?(k, "csc"::SY)   => (1 + tz2**2) / (2 * tz2)
      op args

--The next 5 functions are used by normalize, once a relation is found
    depeval(f, lk, k, v) ==
      is?(k, "log"::SY)  => logeval(f, lk, k, v)
      is?(k, "exp"::SY)  => expeval(f, lk, k, v)
      is?(k, "tan"::SY)  => taneval(f, lk, k, v)
      is?(k, "atan"::SY) => ataneval(f, lk, k, v)
      is?(k, NTHR) => rooteval(f, lk, k, v(minIndex v))
      [f, empty(), empty()]

    rooteval(f, lk, k, n) ==
      nv := nthRoot(x := first argument k, m := retract(n)@Z)
      l  := [r for r in concat(k, toR(lk, x)) |
             retract(second argument r)@Z ^= m]
      lv := [nv ** (n / (retract(second argument r)@Z::Q)) for r in l]
      [eval(f, l, lv), l, lv]

    ataneval(f, lk, k, v) ==
      w := first argument k
      s := tanSum [tanQ(qelt(v,i), x)
                   for i in minIndex v .. maxIndex v for x in toV lk]
      g := +/[qelt(v, i) * x for i in minIndex v .. maxIndex v for x in toU lk]
      h:F :=
        zero?(d := 1 + s * w) => mpiover2
        atan((w - s) / d)
      g := g + h
      [eval(f, [k], [g]), [k], [g]]

    gdCoef?(c, v) ==
      for i in minIndex v .. maxIndex v repeat
        retractIfCan(qelt(v, i) / c)@Union(Z, "failed") case "failed" =>
          return false
      true
```

```
goodCoef(v, l, s) ==
  for i in minIndex v .. maxIndex v for k in l repeat
    is?(k, s) and
       ((r:=recip(qelt(v,i))) case Q) and
        (retractIfCan(r::Q)@Union(Z, "failed") case Z)
          and gdCoef?(qelt(v, i), v) => return([i, k])
  "failed"

taneval(f, lk, k, v) ==
  u := first argument k
  fns := toU lk
  c := u - +/[qelt(v, i)*x for i in minIndex v .. maxIndex v for x in fns]
  (rec := goodCoef(v, lk, "tan"::SY)) case "failed" =>
      tannosimp(f, lk, k, v, fns, c)
  v0 := retract(inv qelt(v, rec.index))@Z
  lv := [qelt(v, i) for i in minIndex v .. maxIndex v |
                                          i ^= rec.index]$List(Q)
  l  := [kk for kk in lk | kk ^= rec.ker]
  g := tanSum(-v0 * c, concat(tanNa(k::F, v0),
       [tanNa(x, - retract(a * v0)@Z) for a in lv for x in toV l]))
  [eval(f, [rec.ker], [g]), [rec.ker], [g]]

tannosimp(f, lk, k, v, fns, c) ==
  every?(x+->is?(x, "tan"::SY), lk) =>
    dd := (d := (cd := splitDenominator v).den)::F
    newt := [tan(u / dd) for u in fns]$List(F)
    newtan := [tanNa(t, d) for t in newt]$List(F)
    h := tanSum(c, [tanNa(t, qelt(cd.num, i))
                    for i in minIndex v .. maxIndex v for t in newt])
    lk := concat(k, lk)
    newtan := concat(h, newtan)
    [eval(f, lk, newtan), lk, newtan]
  h := tanSum(c, [tanQ(qelt(v, i), x)
                  for i in minIndex v .. maxIndex v for x in toV lk])
  [eval(f, [k], [h]), [k], [h]]

expnosimp(f, lk, k, v, fns, g) ==
  every?(x+->is?(x, "exp"::SY), lk) =>
    dd := (d := (cd := splitDenominator v).den)::F
    newe := [exp(y / dd) for y in fns]$List(F)
    newexp := [e ** d for e in newe]$List(F)
    h := */[e ** qelt(cd.num, i)
            for i in minIndex v .. maxIndex v for e in newe] * g
    lk := concat(k, lk)
    newexp := concat(h, newexp)
    [eval(f, lk, newexp), lk, newexp]
  h := */[exp(y) ** qelt(v, i)
            for i in minIndex v .. maxIndex v for y in fns] * g
  [eval(f, [k], [h]), [k], [h]]
```

```
    logeval(f, lk, k, v) ==
      z := first argument k
      c := z / (*/[x**qelt(v, i)
                   for x in toZ lk for i in minIndex v .. maxIndex v])
-- CHANGED log ktoZ x TO ktoY x SINCE WE WANT log exp f TO BE REPLACED BY f.
      g := +/[qelt(v, i) * x
              for i in minIndex v .. maxIndex v for x in toY lk] + log c
      [eval(f, [k], [g]), [k], [g]]

    rischNormalize(f, v) ==
      empty?(ker := varselect(tower f, v)) => [f, empty(), empty()]
      first(ker) ^= kernel(v)@K => error "Cannot happen"
      ker := rest ker
      (n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
      for i in 1..n for kk in rest ker repeat
        klist := first(ker, i)
        -- NO EVALUATION ON AN EMPTY VECTOR, WILL CAUSE INFINITE LOOP
        (c := deprel(klist, kk, v)) case vec and not empty?(c.vec) =>
          rec := depeval(f, klist, kk, c.vec)
          rn  := rischNormalize(rec.func, v)
          return [rn.func,
                   concat(rec.kers, rn.kers), concat(rec.vals, rn.vals)]
        c case func =>
          rn := rischNormalize(eval(f, [kk], [c.func]), v)
          return [rn.func, concat(kk, rn.kers), concat(c.func, rn.vals)]
      [f, empty(), empty()]

    rootNormalize(f, k) ==
      (u := rootKernelNormalize(f, toR(tower f, first argument k), k))
         case "failed" => f
      (u::REC).func

    rootKernelNormalize(f, l, k) ==
      (c := rootDep(l, k)) case vec =>
        rooteval(f, l, k, (c.vec)(minIndex(c.vec)))
      "failed"

    localnorm f ==
      for x in variables f repeat
        f := rischNormalize(f, x).func
      f

    validExponential(twr, eta, x) ==
      (c := solveLinearlyOverQ(construct([differentiate(g, x)
         for g in (fns := toY twr)]$List(F))@Vector(F),
           differentiate(eta, x))) case "failed" => "failed"
      v := c::Vector(Q)
      g := eta - +/[qelt(v, i) * yy
                        for i in minIndex v .. maxIndex v for yy in fns]
      */[exp(yy) ** qelt(v, i)
```

```
            for i in minIndex v .. maxIndex v for yy in fns] * exp g

rootDep(ker, k) ==
  empty?(ker := toR(ker, first argument k)) => [true]
  [new(1,lcm(retract(second argument k)@Z,
   "lcm"/[retract(second argument r)@Z for r in ker])::Q)$Vector(Q)]

qdeprel(l, v) ==
  (u := solveLinearlyOverQ(construct(l)@Vector(F), v))
    case Vector(Q) => [u::Vector(Q)]
  [true]

expeval(f, lk, k, v) ==
  y   := first argument k
  fns := toY lk
  g := y - +/[qelt(v, i) * z for i in minIndex v .. maxIndex v for z in fns]
  (rec := goodCoef(v, lk, "exp"::SY)) case "failed" =>
    expnosimp(f, lk, k, v, fns, exp g)
  v0 := retract(inv qelt(v, rec.index))@Z
  lv := [qelt(v, i) for i in minIndex v .. maxIndex v |
                                          i ^= rec.index]$List(Q)
  l  := [kk for kk in lk | kk ^= rec.ker]
  h :F := */[exp(z) ** (- retract(a * v0)@Z) for a in lv for z in toY l]
  h := h * exp(-v0 * g) * (k::F) ** v0
  [eval(f, [rec.ker], [h]), [rec.ker], [h]]

if F has CombinatorialOpsCategory then
  normalize f == rtNormalize localnorm factorials realElementary f

  normalize(f, x) ==
    rtNormalize(rischNormalize(factorials(realElementary(f,x),x),x).func)

  factdeprel(l, k) ==
    ((r := retractIfCan(n := first argument k)@Union(Z, "failed"))
      case Z) and (r::Z > 0) => [factorial(r::Z)::F]
    for x in l repeat
      m := first argument x
      ((r := retractIfCan(n - m)@Union(Z, "failed")) case Z) and
        (r::Z > 0) => return([*/[(m + i::F) for i in 1..r] * x::F])
    [true]

else
  normalize f     == rtNormalize localnorm realElementary f
  normalize(f, x) == rtNormalize(rischNormalize(realElementary(f,x),x).func)
```

———-

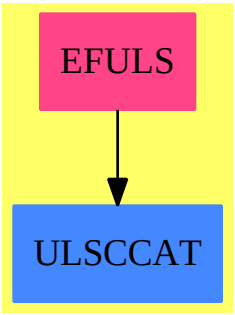
— EFSTRUC.dotabb —

```
"EFSTRUC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFSTRUC"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EFSTRUC" -> "ACF"
"EFSTRUC" -> "FS"
```

———-

6.8 package EFULS ElementaryFunctionsUnivariateLaurentSeries

6.8.1 ElementaryFunctionsUnivariateLaurentSeries (EFULS)



Exports:

acos	acosIfCan	acosh	acoshIfCan	acot
acotIfCan	acoth	acothIfCan	acsc	acscIfCan
acsch	acschIfCan	asec	asecIfCan	asech
asechIfCan	asin	asinIfCan	asinh	asinhIfCan
atan	atanIfCan	atanh	atanhIfCan	cos
cosIfCan	cosh	coshIfCan	cot	cotIfCan
coth	cothIfCan	csc	cscIfCan	csch
cschIfCan	exp	expIfCan	log	logIfCan
nthRootIfCan	sec	secIfCan	sech	sechIfCan
sin	sinIfCan	sinh	sinhIfCan	tan
tanIfCan	tanh	tanhIfCan	?**?	

— **package EFULS ElementaryFunctionsUnivariateLaurentSeries** —

```
)abbrev package EFULS ElementaryFunctionsUnivariateLaurentSeries
++ Author: Clifton J. Williamson
++ Date Created: 6 February 1990
++ Date Last Updated: 25 February 1990
++ Keywords: elementary function, Laurent series
```

```
++ Examples:
++ References:
++ Description:
++ This package provides elementary functions on any Laurent series
++ domain over a field which was constructed from a Taylor series
++ domain.  These functions are implemented by calling the
++ corresponding functions on the Taylor series domain.  We also
++ provide 'partial functions' which compute transcendental
++ functions of Laurent series when possible and return "failed"
++ when this is not possible.

ElementaryFunctionsUnivariateLaurentSeries(Coef,UTS,ULS):_
 Exports == Implementation where
  Coef   : Algebra Fraction Integer
  UTS    : UnivariateTaylorSeriesCategory Coef
  ULS    : UnivariateLaurentSeriesConstructorCategory(Coef,UTS)
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  RN   ==> Fraction Integer
  S    ==> String
  STTF ==> StreamTranscendentalFunctions(Coef)

  Exports ==> PartialTranscendentalFunctions(ULS) with

    if Coef has Field then
      "**": (ULS,RN) -> ULS
        ++ s ** r raises a Laurent series s to a rational power r

--% Exponentials and Logarithms

    exp: ULS -> ULS
      ++ exp(z) returns the exponential of Laurent series z.
    log: ULS -> ULS
      ++ log(z) returns the logarithm of Laurent series z.

--% TrigonometricFunctionCategory

    sin: ULS -> ULS
      ++ sin(z) returns the sine of Laurent series z.
    cos: ULS -> ULS
      ++ cos(z) returns the cosine of Laurent series z.
    tan: ULS -> ULS
      ++ tan(z) returns the tangent of Laurent series z.
    cot: ULS -> ULS
      ++ cot(z) returns the cotangent of Laurent series z.
    sec: ULS -> ULS
      ++ sec(z) returns the secant of Laurent series z.
    csc: ULS -> ULS
      ++ csc(z) returns the cosecant of Laurent series z.
```

```
--% ArcTrigonometricFunctionCategory

    asin: ULS -> ULS
      ++ asin(z) returns the arc-sine of Laurent series z.
    acos: ULS -> ULS
      ++ acos(z) returns the arc-cosine of Laurent series z.
    atan: ULS -> ULS
      ++ atan(z) returns the arc-tangent of Laurent series z.
    acot: ULS -> ULS
      ++ acot(z) returns the arc-cotangent of Laurent series z.
    asec: ULS -> ULS
      ++ asec(z) returns the arc-secant of Laurent series z.
    acsc: ULS -> ULS
      ++ acsc(z) returns the arc-cosecant of Laurent series z.

--% HyperbolicFunctionCategory

    sinh: ULS -> ULS
      ++ sinh(z) returns the hyperbolic sine of Laurent series z.
    cosh: ULS -> ULS
      ++ cosh(z) returns the hyperbolic cosine of Laurent series z.
    tanh: ULS -> ULS
      ++ tanh(z) returns the hyperbolic tangent of Laurent series z.
    coth: ULS -> ULS
      ++ coth(z) returns the hyperbolic cotangent of Laurent series z.
    sech: ULS -> ULS
      ++ sech(z) returns the hyperbolic secant of Laurent series z.
    csch: ULS -> ULS
      ++ csch(z) returns the hyperbolic cosecant of Laurent series z.

--% ArcHyperbolicFunctionCategory

    asinh: ULS -> ULS
      ++ asinh(z) returns the inverse hyperbolic sine of Laurent series z.
    acosh: ULS -> ULS
      ++ acosh(z) returns the inverse hyperbolic cosine of Laurent series z.
    atanh: ULS -> ULS
      ++ atanh(z) returns the inverse hyperbolic tangent of Laurent series z.
    acoth: ULS -> ULS
      ++ acoth(z) returns the inverse hyperbolic cotangent of Laurent series z.
    asech: ULS -> ULS
      ++ asech(z) returns the inverse hyperbolic secant of Laurent series z.
    acsch: ULS -> ULS
      ++ acsch(z) returns the inverse hyperbolic cosecant of Laurent series z.

  Implementation ==> add

--% roots

    RATPOWERS : Boolean := Coef has "**":(Coef,RN) -> Coef
```

```
    TRANSFCN  : Boolean := Coef has TranscendentalFunctionCategory
    RATS      : Boolean := Coef has retractIfCan: Coef -> Union(RN,"failed")

    nthRootUTS:(UTS,I) -> Union(UTS,"failed")
    nthRootUTS(uts,n) ==
      -- assumed: n > 1, uts has non-zero constant term
--      one? coefficient(uts,0) => uts ** inv(n::RN)
      coefficient(uts,0) = 1 => uts ** inv(n::RN)
      RATPOWERS => uts ** inv(n::RN)
      "failed"

    nthRootIfCan(uls,nn) ==
      (n := nn :: I) < 1 => error "nthRootIfCan: n must be positive"
      n = 1 => uls
      deg := degree uls
      if zero? (coef := coefficient(uls,deg)) then
        uls := removeZeroes(1000,uls); deg := degree uls
        zero? (coef := coefficient(uls,deg)) =>
          error "root of series with many leading zero coefficients"
      (k := deg exquo n) case "failed" => "failed"
      uts := taylor(uls * monomial(1,-deg))
      (root := nthRootUTS(uts,n)) case "failed" => "failed"
      monomial(1,k :: I) * (root :: UTS :: ULS)

    if Coef has Field then
       (uls:ULS) ** (r:RN) ==
         num := numer r; den := denom r
--         one? den => uls ** num
         den = 1 => uls ** num
         deg := degree uls
         if zero? (coef := coefficient(uls,deg)) then
           uls := removeZeroes(1000,uls); deg := degree uls
           zero? (coef := coefficient(uls,deg)) =>
             error "power of series with many leading zero coefficients"
         (k := deg exquo den) case "failed" =>
           error "**: rational power does not exist"
         uts := taylor(uls * monomial(1,-deg)) ** r
         monomial(1,(k :: I) * num) * (uts :: ULS)

--% transcendental functions

    applyIfCan: (UTS -> UTS,ULS) -> Union(ULS,"failed")
    applyIfCan(fcn,uls) ==
      uts := taylorIfCan uls
      uts case "failed" => "failed"
      fcn(uts :: UTS) :: ULS

    expIfCan   uls == applyIfCan(exp,uls)
    sinIfCan   uls == applyIfCan(sin,uls)
    cosIfCan   uls == applyIfCan(cos,uls)
```

```
asinIfCan  uls == applyIfCan(asin,uls)
acosIfCan  uls == applyIfCan(acos,uls)
asecIfCan  uls == applyIfCan(asec,uls)
acscIfCan  uls == applyIfCan(acsc,uls)
sinhIfCan  uls == applyIfCan(sinh,uls)
coshIfCan  uls == applyIfCan(cosh,uls)
asinhIfCan uls == applyIfCan(asinh,uls)
acoshIfCan uls == applyIfCan(acosh,uls)
atanhIfCan uls == applyIfCan(atanh,uls)
acothIfCan uls == applyIfCan(acoth,uls)
asechIfCan uls == applyIfCan(asech,uls)
acschIfCan uls == applyIfCan(acsch,uls)

logIfCan uls ==
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  zero? coefficient(ts := uts :: UTS,0) => "failed"
  log(ts) :: ULS

tanIfCan uls ==
  -- don't call 'tan' on a UTS (tan(uls) may have a singularity)
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  sc := sincos(coefficients(uts :: UTS))$STTF
  (cosInv := recip(series(sc.cos) :: ULS)) case "failed" => "failed"
  (series(sc.sin) :: ULS) * (cosInv :: ULS)

cotIfCan uls ==
  -- don't call 'cot' on a UTS (cot(uls) may have a singularity)
  uts := taylorIfCan uls
  uts case "failed" => "failed"
  sc := sincos(coefficients(uts :: UTS))$STTF
  (sinInv := recip(series(sc.sin) :: ULS)) case "failed" => "failed"
  (series(sc.cos) :: ULS) * (sinInv :: ULS)

secIfCan uls ==
  cos := cosIfCan uls
  cos case "failed" => "failed"
  (cosInv := recip(cos :: ULS)) case "failed" => "failed"
  cosInv :: ULS

cscIfCan uls ==
  sin := sinIfCan uls
  sin case "failed" => "failed"
  (sinInv := recip(sin :: ULS)) case "failed" => "failed"
  sinInv :: ULS

atanIfCan uls ==
  coef := coefficient(uls,0)
  (ord := order(uls,0)) = 0 and coef * coef = -1 => "failed"
```

```
    cc : Coef :=
      ord < 0 =>
        TRANSFCN =>
          RATS =>
            lc := coefficient(uls,ord)
            (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" =>
              (1/2) * pi()
            (rat :: RN) > 0 => (1/2) * pi()
            (-1/2) * pi()
          (1/2) * pi()
        return "failed"
      coef = 0 => 0
      TRANSFCN => atan coef
      return "failed"
    (z := recip(1 + uls*uls)) case "failed" => "failed"
    (cc :: ULS) + integrate(differentiate(uls) * (z :: ULS))

  acotIfCan uls ==
    coef := coefficient(uls,0)
    (ord := order(uls,0)) = 0 and coef * coef = -1 => "failed"
    cc : Coef :=
      ord < 0 =>
        RATS =>
          lc := coefficient(uls,ord)
          (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" => 0
          (rat :: RN) > 0 => 0
          TRANSFCN => pi()
          return "failed"
        0
      TRANSFCN => acot coef
      return "failed"
    (z := recip(1 + uls*uls)) case "failed" => "failed"
    (cc :: ULS) - integrate(differentiate(uls) * (z :: ULS))

  tanhIfCan uls ==
    -- don't call 'tanh' on a UTS (tanh(uls) may have a singularity)
    uts := taylorIfCan uls
    uts case "failed" => "failed"
    sc := sinhcosh(coefficients(uts :: UTS))$STTF
    (coshInv := recip(series(sc.cosh) :: ULS)) case "failed" =>
      "failed"
    (series(sc.sinh) :: ULS) * (coshInv :: ULS)

  cothIfCan uls ==
    -- don't call 'coth' on a UTS (coth(uls) may have a singularity)
    uts := taylorIfCan uls
    uts case "failed" => "failed"
    sc := sinhcosh(coefficients(uts :: UTS))$STTF
    (sinhInv := recip(series(sc.sinh) :: ULS)) case "failed" =>
      "failed"
```

```
    (series(sc.cosh) :: ULS) * (sinhInv :: ULS)

  sechIfCan uls ==
    cosh := coshIfCan uls
    cosh case "failed" => "failed"
    (coshInv := recip(cosh :: ULS)) case "failed" => "failed"
    coshInv :: ULS

  cschIfCan uls ==
    sinh := sinhIfCan uls
    sinh case "failed" => "failed"
    (sinhInv := recip(sinh :: ULS)) case "failed" => "failed"
    sinhInv :: ULS

  applyOrError:(ULS -> Union(ULS,"failed"),S,ULS) -> ULS
  applyOrError(fcn,name,uls) ==
    ans := fcn uls
    ans case "failed" =>
      error concat(name," of function with singularity")
    ans :: ULS

  exp uls   == applyOrError(expIfCan,"exp",uls)
  log uls   == applyOrError(logIfCan,"log",uls)
  sin uls   == applyOrError(sinIfCan,"sin",uls)
  cos uls   == applyOrError(cosIfCan,"cos",uls)
  tan uls   == applyOrError(tanIfCan,"tan",uls)
  cot uls   == applyOrError(cotIfCan,"cot",uls)
  sec uls   == applyOrError(secIfCan,"sec",uls)
  csc uls   == applyOrError(cscIfCan,"csc",uls)
  asin uls  == applyOrError(asinIfCan,"asin",uls)
  acos uls  == applyOrError(acosIfCan,"acos",uls)
  asec uls  == applyOrError(asecIfCan,"asec",uls)
  acsc uls  == applyOrError(acscIfCan,"acsc",uls)
  sinh uls  == applyOrError(sinhIfCan,"sinh",uls)
  cosh uls  == applyOrError(coshIfCan,"cosh",uls)
  tanh uls  == applyOrError(tanhIfCan,"tanh",uls)
  coth uls  == applyOrError(cothIfCan,"coth",uls)
  sech uls  == applyOrError(sechIfCan,"sech",uls)
  csch uls  == applyOrError(cschIfCan,"csch",uls)
  asinh uls == applyOrError(asinhIfCan,"asinh",uls)
  acosh uls == applyOrError(acoshIfCan,"acosh",uls)
  atanh uls == applyOrError(atanhIfCan,"atanh",uls)
  acoth uls == applyOrError(acothIfCan,"acoth",uls)
  asech uls == applyOrError(asechIfCan,"asech",uls)
  acsch uls == applyOrError(acschIfCan,"acsch",uls)

  atan uls ==
  -- code is duplicated so that correct error messages will be returned
    coef := coefficient(uls,0)
    (ord := order(uls,0)) = 0 and coef * coef = -1 =>
```

```
        error "atan: series expansion has logarithmic term"
      cc : Coef :=
        ord < 0 =>
          TRANSFCN =>
            RATS =>
              lc := coefficient(uls,ord)
              (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" =>
                (1/2) * pi()
              (rat :: RN) > 0 => (1/2) * pi()
              (-1/2) * pi()
            (1/2) * pi()
          error "atan: series expansion involves transcendental constants"
        coef = 0 => 0
        TRANSFCN => atan coef
        error "atan: series expansion involves transcendental constants"
      (z := recip(1 + uls*uls)) case "failed" =>
        error "atan: leading coefficient not invertible"
      (cc :: ULS) + integrate(differentiate(uls) * (z :: ULS))

    acot uls ==
    -- code is duplicated so that correct error messages will be returned
      coef := coefficient(uls,0)
      (ord := order(uls,0)) = 0 and coef * coef = -1 =>
        error "acot: series expansion has logarithmic term"
      cc : Coef :=
        ord < 0 =>
          RATS =>
            lc := coefficient(uls,ord)
            (rat := retractIfCan(lc)@Union(RN,"failed")) case "failed" => 0
            (rat :: RN) > 0 => 0
            TRANSFCN => pi()
            error "acot: series expansion involves transcendental constants"
          0
        TRANSFCN => acot coef
        error "acot: series expansion involves transcendental constants"
      (z := recip(1 + uls*uls)) case "failed" =>
        error "acot: leading coefficient not invertible"
      (cc :: ULS) - integrate(differentiate(uls) * (z :: ULS))
```

———-

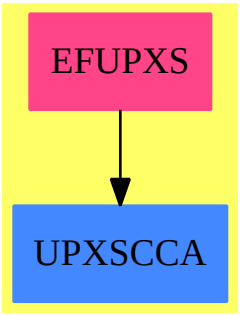
— EFULS.dotabb —

```
"EFULS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFULS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"EFULS" -> "ULSCCAT"
```

———-

6.9 package EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries

6.9.1 ElementaryFunctionsUnivariatePuiseuxSeries (EFUPXS)



Exports:

acos	acosIfCan	acosh	acoshIfCan	acot
acotIfCan	acoth	acothIfCan	acsc	acscIfCan
acsch	acschIfCan	asec	asecIfCan	asech
asechIfCan	asin	asinIfCan	asinh	asinhIfCan
atan	atanIfCan	atanh	atanhIfCan	cos
cosIfCan	cosh	coshIfCan	cot	cotIfCan
coth	cothIfCan	csc	cscIfCan	csch
cschIfCan	exp	expIfCan	log	logIfCan
nthRootIfCan	sec	secIfCan	sech	sechIfCan
sin	sinIfCan	sinh	sinhIfCan	tan
tanIfCan	tanh	tanhIfCan	?**?	

— **package EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries** —

```
)abbrev package EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries
++ Author: Clifton J. Williamson
++ Date Created: 20 February 1990
++ Date Last Updated: 20 February 1990
++ References:
++ Description:
++ This package provides elementary functions on any Laurent series
++ domain over a field which was constructed from a Taylor series
++ domain.  These functions are implemented by calling the
++ corresponding functions on the Taylor series domain.  We also
++ provide 'partial functions' which compute transcendental
++ functions of Laurent series when possible and return "failed"
++ when this is not possible.

ElementaryFunctionsUnivariatePuiseuxSeries(Coef,ULS,UPXS,EFULS):_
 Exports == Implementation where
```

```
  Coef   : Algebra Fraction Integer
  ULS    : UnivariateLaurentSeriesCategory Coef
  UPXS   : UnivariatePuiseuxSeriesConstructorCategory(Coef,ULS)
  EFULS  : PartialTranscendentalFunctions(ULS)
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  RN   ==> Fraction Integer

  Exports ==> PartialTranscendentalFunctions(UPXS) with

    if Coef has Field then
      "**": (UPXS,RN) -> UPXS
        ++ z ** r raises a Puiseaux series z to a rational power r

--% Exponentials and Logarithms

    exp: UPXS -> UPXS
      ++ exp(z) returns the exponential of a Puiseux series z.
    log: UPXS -> UPXS
      ++ log(z) returns the logarithm of a Puiseux series z.

--% TrigonometricFunctionCategory

    sin: UPXS -> UPXS
      ++ sin(z) returns the sine of a Puiseux series z.
    cos: UPXS -> UPXS
      ++ cos(z) returns the cosine of a Puiseux series z.
    tan: UPXS -> UPXS
      ++ tan(z) returns the tangent of a Puiseux series z.
    cot: UPXS -> UPXS
      ++ cot(z) returns the cotangent of a Puiseux series z.
    sec: UPXS -> UPXS
      ++ sec(z) returns the secant of a Puiseux series z.
    csc: UPXS -> UPXS
      ++ csc(z) returns the cosecant of a Puiseux series z.

--% ArcTrigonometricFunctionCategory

    asin: UPXS -> UPXS
      ++ asin(z) returns the arc-sine of a Puiseux series z.
    acos: UPXS -> UPXS
      ++ acos(z) returns the arc-cosine of a Puiseux series z.
    atan: UPXS -> UPXS
      ++ atan(z) returns the arc-tangent of a Puiseux series z.
    acot: UPXS -> UPXS
      ++ acot(z) returns the arc-cotangent of a Puiseux series z.
    asec: UPXS -> UPXS
      ++ asec(z) returns the arc-secant of a Puiseux series z.
    acsc: UPXS -> UPXS
      ++ acsc(z) returns the arc-cosecant of a Puiseux series z.
```

```
--% HyperbolicFunctionCategory

    sinh: UPXS -> UPXS
      ++ sinh(z) returns the hyperbolic sine of a Puiseux series z.
    cosh: UPXS -> UPXS
      ++ cosh(z) returns the hyperbolic cosine of a Puiseux series z.
    tanh: UPXS -> UPXS
      ++ tanh(z) returns the hyperbolic tangent of a Puiseux series z.
    coth: UPXS -> UPXS
      ++ coth(z) returns the hyperbolic cotangent of a Puiseux series z.
    sech: UPXS -> UPXS
      ++ sech(z) returns the hyperbolic secant of a Puiseux series z.
    csch: UPXS -> UPXS
      ++ csch(z) returns the hyperbolic cosecant of a Puiseux series z.

--% ArcHyperbolicFunctionCategory

    asinh: UPXS -> UPXS
      ++ asinh(z) returns the inverse hyperbolic sine of a Puiseux series z.
    acosh: UPXS -> UPXS
      ++ acosh(z) returns the inverse hyperbolic cosine of a Puiseux series z.
    atanh: UPXS -> UPXS
      ++ atanh(z) returns the inverse hyperbolic tangent of a Puiseux series z.
    acoth: UPXS -> UPXS
      ++ acoth(z) returns the inverse hyperbolic cotangent
      ++ of a Puiseux series z.
    asech: UPXS -> UPXS
      ++ asech(z) returns the inverse hyperbolic secant of a Puiseux series z.
    acsch: UPXS -> UPXS
      ++ acsch(z) returns the inverse hyperbolic cosecant
      ++ of a Puiseux series z.

  Implementation ==> add

    TRANSFCN : Boolean := Coef has TranscendentalFunctionCategory

--% roots

    nthRootIfCan(upxs,n) ==
--      one? n => upxs
      n = 1 => upxs
      r := rationalPower upxs; uls := laurentRep upxs
      deg := degree uls
      if zero?(coef := coefficient(uls,deg)) then
        deg := order(uls,deg + 1000)
        zero?(coef := coefficient(uls,deg)) =>
          error "root of series with many leading zero coefficients"
      uls := uls * monomial(1,-deg)$ULS
      (ulsRoot := nthRootIfCan(uls,n)) case "failed" => "failed"
```

```
      puiseux(r,ulsRoot :: ULS) * monomial(1,deg * r * inv(n :: RN))

    if Coef has Field then
       (upxs:UPXS) ** (q:RN) ==
         num := numer q; den := denom q
--         one? den => upxs ** num
         den = 1 => upxs ** num
         r := rationalPower upxs; uls := laurentRep upxs
         deg := degree uls
         if zero?(coef := coefficient(uls,deg)) then
           deg := order(uls,deg + 1000)
           zero?(coef := coefficient(uls,deg)) =>
             error "power of series with many leading zero coefficients"
         ulsPow := (uls * monomial(1,-deg)$ULS) ** q
         puiseux(r,ulsPow) * monomial(1,deg*q*r)

--% transcendental functions

    applyIfCan: (ULS -> Union(ULS,"failed"),UPXS) -> Union(UPXS,"failed")
    applyIfCan(fcn,upxs) ==
      uls := fcn laurentRep upxs
      uls case "failed" => "failed"
      puiseux(rationalPower upxs,uls :: ULS)

    expIfCan   upxs == applyIfCan(expIfCan,upxs)
    logIfCan   upxs == applyIfCan(logIfCan,upxs)
    sinIfCan   upxs == applyIfCan(sinIfCan,upxs)
    cosIfCan   upxs == applyIfCan(cosIfCan,upxs)
    tanIfCan   upxs == applyIfCan(tanIfCan,upxs)
    cotIfCan   upxs == applyIfCan(cotIfCan,upxs)
    secIfCan   upxs == applyIfCan(secIfCan,upxs)
    cscIfCan   upxs == applyIfCan(cscIfCan,upxs)
    atanIfCan  upxs == applyIfCan(atanIfCan,upxs)
    acotIfCan  upxs == applyIfCan(acotIfCan,upxs)
    sinhIfCan  upxs == applyIfCan(sinhIfCan,upxs)
    coshIfCan  upxs == applyIfCan(coshIfCan,upxs)
    tanhIfCan  upxs == applyIfCan(tanhIfCan,upxs)
    cothIfCan  upxs == applyIfCan(cothIfCan,upxs)
    sechIfCan  upxs == applyIfCan(sechIfCan,upxs)
    cschIfCan  upxs == applyIfCan(cschIfCan,upxs)
    asinhIfCan upxs == applyIfCan(asinhIfCan,upxs)
    acoshIfCan upxs == applyIfCan(acoshIfCan,upxs)
    atanhIfCan upxs == applyIfCan(atanhIfCan,upxs)
    acothIfCan upxs == applyIfCan(acothIfCan,upxs)
    asechIfCan upxs == applyIfCan(asechIfCan,upxs)
    acschIfCan upxs == applyIfCan(acschIfCan,upxs)

    asinIfCan upxs ==
      order(upxs,0) < 0 => "failed"
      (coef := coefficient(upxs,0)) = 0 =>
```

```
      integrate((1 - upxs*upxs)**(-1/2) * (differentiate upxs))
    TRANSFCN =>
      cc := asin(coef) :: UPXS
      cc + integrate((1 - upxs*upxs)**(-1/2) * (differentiate upxs))
    "failed"

  acosIfCan upxs ==
    order(upxs,0) < 0 => "failed"
    TRANSFCN =>
      cc := acos(coefficient(upxs,0)) :: UPXS
      cc + integrate(-(1 - upxs*upxs)**(-1/2) * (differentiate upxs))
    "failed"

  asecIfCan upxs ==
    order(upxs,0) < 0 => "failed"
    TRANSFCN =>
      cc := asec(coefficient(upxs,0)) :: UPXS
      f := (upxs*upxs - 1)**(-1/2) * (differentiate upxs)
      (rec := recip upxs) case "failed" => "failed"
      cc + integrate(f * (rec :: UPXS))
    "failed"

  acscIfCan upxs ==
    order(upxs,0) < 0 => "failed"
    TRANSFCN =>
      cc := acsc(coefficient(upxs,0)) :: UPXS
      f := -(upxs*upxs - 1)**(-1/2) * (differentiate upxs)
      (rec := recip upxs) case "failed" => "failed"
      cc + integrate(f * (rec :: UPXS))
    "failed"

  asinhIfCan upxs ==
    order(upxs,0) < 0 => "failed"
    TRANSFCN or (coefficient(upxs,0) = 0) =>
      log(upxs + (1 + upxs*upxs)**(1/2))
    "failed"

  acoshIfCan upxs ==
    TRANSFCN =>
      order(upxs,0) < 0 => "failed"
      log(upxs + (upxs*upxs - 1)**(1/2))
    "failed"

  asechIfCan upxs ==
    TRANSFCN =>
      order(upxs,0) < 0 => "failed"
      (rec := recip upxs) case "failed" => "failed"
      log((1 + (1 - upxs*upxs)*(1/2)) * (rec :: UPXS))
    "failed"
```

```
acschIfCan upxs ==
  TRANSFCN =>
    order(upxs,0) < 0 => "failed"
    (rec := recip upxs) case "failed" => "failed"
    log((1 + (1 + upxs*upxs)*(1/2)) * (rec :: UPXS))
  "failed"

applyOrError:(UPXS -> Union(UPXS,"failed"),String,UPXS) -> UPXS
applyOrError(fcn,name,upxs) ==
  ans := fcn upxs
  ans case "failed" =>
    error concat(name," of function with singularity")
  ans :: UPXS

exp upxs   == applyOrError(expIfCan,"exp",upxs)
log upxs   == applyOrError(logIfCan,"log",upxs)
sin upxs   == applyOrError(sinIfCan,"sin",upxs)
cos upxs   == applyOrError(cosIfCan,"cos",upxs)
tan upxs   == applyOrError(tanIfCan,"tan",upxs)
cot upxs   == applyOrError(cotIfCan,"cot",upxs)
sec upxs   == applyOrError(secIfCan,"sec",upxs)
csc upxs   == applyOrError(cscIfCan,"csc",upxs)
asin upxs  == applyOrError(asinIfCan,"asin",upxs)
acos upxs  == applyOrError(acosIfCan,"acos",upxs)
atan upxs  == applyOrError(atanIfCan,"atan",upxs)
acot upxs  == applyOrError(acotIfCan,"acot",upxs)
asec upxs  == applyOrError(asecIfCan,"asec",upxs)
acsc upxs  == applyOrError(acscIfCan,"acsc",upxs)
sinh upxs  == applyOrError(sinhIfCan,"sinh",upxs)
cosh upxs  == applyOrError(coshIfCan,"cosh",upxs)
tanh upxs  == applyOrError(tanhIfCan,"tanh",upxs)
coth upxs  == applyOrError(cothIfCan,"coth",upxs)
sech upxs  == applyOrError(sechIfCan,"sech",upxs)
csch upxs  == applyOrError(cschIfCan,"csch",upxs)
asinh upxs == applyOrError(asinhIfCan,"asinh",upxs)
acosh upxs == applyOrError(acoshIfCan,"acosh",upxs)
atanh upxs == applyOrError(atanhIfCan,"atanh",upxs)
acoth upxs == applyOrError(acothIfCan,"acoth",upxs)
asech upxs == applyOrError(asechIfCan,"asech",upxs)
acsch upxs == applyOrError(acschIfCan,"acsch",upxs)
```

———-

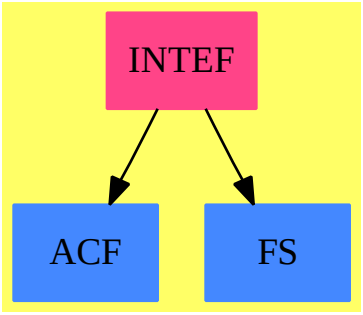
— EFUPXS.dotabb —

```
"EFUPXS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EFUPXS"]
"UPXSCCA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UPXSCCA"]
"EFUPXS" -> "UPXSCCA"
```

———-

6.10 package INTEF ElementaryIntegration

6.10.1 ElementaryIntegration (INTEF)



Exports:
lfextendedint lfextlimint lfinfieldint lfintegrate lflimitedint

— **package INTEF ElementaryIntegration** —

```
)abbrev package INTEF ElementaryIntegration
++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 24 October 1995
++ Keywords: elementary, function, integration.
++ Description:
++ This package provides functions for integration, limited integration,
++ extended integration and the risch differential equation for
++ elementary functions.

ElementaryIntegration(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  SE     ==> Symbol
  K      ==> Kernel F
  P      ==> SparseMultivariatePolynomial(R, K)
  UP     ==> SparseUnivariatePolynomial F
  RF     ==> Fraction UP
  IR     ==> IntegrationResult F
```

```
FF     ==> Record(ratpart:RF, coeff:RF)
LLG    ==> List Record(coeff:F, logand:F)
U2     ==> Union(Record(ratpart:F, coeff:F), "failed")
U3     ==> Union(Record(mainpart:F, limitedlogs:LLG), "failed")
ANS    ==> Record(special:F, integrand:F)
PSOL   ==> Record(ans:F, right:F, sol?:Boolean)
FAIL   ==> error "failed - cannot handle that integrand"
ALGOP  ==> "%alg"
OPDIFF ==> "%diff"::SE

Exports ==> with
  lfextendedint: (F, SE, F) -> U2
     ++ lfextendedint(f, x, g) returns functions \spad{[h, c]} such that
     ++ \spad{dh/dx = f - cg}, if (h, c) exist, "failed" otherwise.
  lflimitedint : (F, SE, List F) -> U3
     ++ lflimitedint(f,x,[g1,...,gn]) returns functions \spad{[h,[[ci, gi]]]}
     ++ such that the gi's are among \spad{[g1,...,gn]}, and
     ++ \spad{d(h+sum(ci log(gi)))/dx = f}, if possible, "failed" otherwise.
  lfinfieldint : (F, SE) -> Union(F, "failed")
     ++ lfinfieldint(f, x) returns a function g such that \spad{dg/dx = f}
     ++ if g exists, "failed" otherwise.
  lfintegrate  : (F, SE) -> IR
     ++ lfintegrate(f, x) = g such that \spad{dg/dx = f}.
  lfextlimint  : (F, SE, K, List K) -> U2
     ++ lfextlimint(f,x,k,[k1,...,kn]) returns functions \spad{[h, c]}
     ++ such that \spad{dh/dx = f - c dk/dx}. Value h is looked for in a field
     ++ containing f and k1,...,kn (the ki's must be logs).

Implementation ==> add
  import IntegrationTools(R, F)
  import ElementaryRischDE(R, F)
  import RationalIntegration(F, UP)
  import AlgebraicIntegration(R, F)
  import AlgebraicManipulations(R, F)
  import ElementaryRischDESystem(R, F)
  import TranscendentalIntegration(F, UP)
  import PureAlgebraicIntegration(R, F, F)
  import IntegrationResultFunctions2(F, F)
  import IntegrationResultFunctions2(RF, F)
  import FunctionSpacePrimitiveElement(R, F)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                        K, R, P, F)

  alglfint    : (F, K, List K, SE) -> IR
  alglfextint : (F, K, List K, SE, F) -> U2
  alglflimint : (F, K, List K, SE, List F) -> U3
  primextint  : (F, SE, K, F) -> U2
  expextint   : (F, SE, K, F) -> U2
  primlimint  : (F, SE, K, List F) -> U3
  explimint   : (F, SE, K, List F) -> U3
```

```
    algprimint  : (F, K, K, SE) -> IR
    algexpint   : (F, K, K, SE) -> IR
    primint     : (F, SE, K) -> IR
    expint      : (F, SE, K) -> IR
    tanint      : (F, SE, K) -> IR
    prim?       : (K, SE)  -> Boolean
    isx?        : (F, SE)  -> Boolean
    addx        : (IR, F) -> IR
    cfind       : (F, LLG) -> F
    lfintegrate0: (F, SE) -> IR
    unknownint  : (F, SE) -> IR
    unkextint   : (F, SE, F) -> U2
    unklimint   : (F, SE, List F) -> U3
    tryChangeVar: (F, K, SE) -> Union(IR, "failed")
    droponex    : (F, F, K, F) -> Union(F, "failed")

    prim?(k, x)      == is?(k, "log"::SE) or has?(operator k, "prim")

    tanint(f, x, k) ==
      eta' := differentiate(eta := first argument k, x)
      r1  :=
       tanintegrate(univariate(f, k),
        (x1:UP):UP +-> differentiate(x1,
         (x2:F):F +-> differentiate(x2, x),
          monomial(eta', 2) + eta'::UP),
           (x3:Integer,x4:F,x5:F):Union(List F,"failed") +->
             rischDEsys(x3, 2 * eta, x4, x5, x,
              (x6:F,x7:List F):U3 +-> lflimitedint(x6, x, x7),
               (x8:F,x9:F):U2 +-> lfextendedint(x8, x, x9)))
      map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

-- tries various tricks since the integrand contains something not elementary
    unknownint(f, x) ==
      ((r := retractIfCan(f)@Union(K, "failed")) case K) and
        is?(k := r::K, OPDIFF) and
         ((ka:=retractIfCan(a:=second(l:=argument k))@Union(K,"failed"))case K)
           and ((z := retractIfCan(zz := third l)@Union(SE, "failed")) case SE)
              and (z::SE = x)
                and ((u := droponex(first l, a, ka, zz)) case F) => u::F::IR
      (da := differentiate(a := denom(f)::F, x)) ^= 0 and
        zero? differentiate(c := numer(f)::F / da, x) => (c * log a)::IR
      mkAnswer(0, empty(), [[f, x::F]])

    droponex(f, a, ka, x) ==
      (r := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
      is?(op := operator(k := r::K), OPDIFF) =>
        (z := third(arg := argument k)) = a => op [first arg, second arg, x]
        (u := droponex(first arg, a, ka, x)) case "failed" => "failed"
        op [u::F, second arg, z]
      eval(f, [ka], [x])
```

```
unklimint(f, x, lu) ==
  for u in lu | u ^= 0 repeat
    zero? differentiate(c := f * u / differentiate(u, x), x) => [0,[[c,u]]]
  "failed"

unkextint(f, x, g) ==
  zero?(g' := differentiate(g, x)) => "failed"
  zero? differentiate(c := f / g', x) => [0, c]
  "failed"

isx?(f, x) ==
  (k := retractIfCan(f)@Union(K, "failed")) case "failed" => false
  (r := symbolIfCan(k::K)) case "failed" => false
  r::SE = x

alglfint(f, k, l, x) ==
  xf := x::F
  symbolIfCan(kx := ksec(k,l,x)) case SE => addx(palgint(f, kx, k), xf)
  is?(kx, "exp"::SE) => addx(algexpint(f, kx, k, x), xf)
  prim?(kx, x)       => addx(algprimint(f, kx, k, x), xf)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y   := rootOf(rec.prim)
    map((x1:F):F +-> eval(x1, retract(y)@K, rec.primelt),
      lfintegrate(eval(f, [kx,k], [(rec.pol1) y, (rec.pol2) y]), x))
  unknownint(f, x)

alglfextint(f, k, l, x, g) ==
  symbolIfCan(kx := ksec(k,l,x)) case SE => palgextint(f, kx, k, g)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y   := rootOf(rec.prim)
    lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
    (u := lfextendedint(eval(f, [kx, k], lrhs), x,
                eval(g, [kx, k], lrhs))) case "failed" => "failed"
    ky := retract(y)@K
    r := u::Record(ratpart:F, coeff:F)
    [eval(r.ratpart,ky,rec.primelt), eval(r.coeff,ky,rec.primelt)]
  is?(kx, "exp"::SE) or is?(kx, "log"::SE) => FAIL
  unkextint(f, x, g)

alglflimint(f, k, l, x, lu) ==
  symbolIfCan(kx := ksec(k,l,x)) case SE => palglimint(f, kx, k, lu)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y   := rootOf(rec.prim)
    lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
    (u := lflimitedint(eval(f, [kx, k], lrhs), x,
      map((x1:F):F+->eval(x1,[kx, k],lrhs), lu))) case "failed" => "failed"
```

```
      ky := retract(y)@K
      r := u::Record(mainpart:F, limitedlogs:LLG)
      [eval(r.mainpart, ky, rec.primelt),
        [[eval(rc.coeff, ky, rec.primelt),
          eval(rc.logand,ky, rec.primelt)] for rc in r.limitedlogs]]
    is?(kx, "exp"::SE) or is?(kx, "log"::SE) => FAIL
    unklimint(f, x, lu)

  if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
    and F has Join(LiouvillianFunctionCategory, RetractableTo SE) then
      import PatternMatchIntegration(R, F)
      lfintegrate(f, x) == intPatternMatch(f, x, lfintegrate0, pmintegrate)

  else lfintegrate(f, x) == lfintegrate0(f, x)

  lfintegrate0(f, x) ==
    zero? f => 0
    xf := x::F
    empty?(l := varselect(kernels f, x)) => (xf * f)::IR
    symbolIfCan(k := kmax l) case SE =>
      map((x1:RF):F +-> multivariate(x1, k), integrate univariate(f, k))
    is?(k, "tan"::SE)  => addx(tanint(f, x, k), xf)
    is?(k, "exp"::SE)  => addx(expint(f, x, k), xf)
    prim?(k, x)        => addx(primint(f, x, k), xf)
    has?(operator k, ALGOP) => alglfint(f, k, l, x)
    unknownint(f, x)

  addx(i, x) ==
    elem? i => i
    mkAnswer(ratpart i, logpart i,
                              [[ne.integrand, x] for ne in notelem i])

  tryChangeVar(f, t, x) ==
      z := new()$Symbol
      g := subst(f / differentiate(t::F, x), [t], [z::F])
      freeOf?(g, x) =>               -- can we do change of variables?
          map((x1:F):F+->eval(x1, kernel z, t::F), lfintegrate(g, z))
      "failed"

  algexpint(f, t, y, x) ==
      (u := tryChangeVar(f, t, x)) case IR => u::IR
      algint(f, t, y,
             (x1:UP):UP +-> differentiate(x1,
              (x2:F):F +-> differentiate(x2, x),
               monomial(differentiate(first argument t, x), 1)))

  algprimint(f, t, y, x) ==
      (u := tryChangeVar(f, t, x)) case IR => u::IR
      algint(f, t, y,
             (x1:UP):UP +-> differentiate(x1,
```

```
              (x2:F):F +-> differentiate(x2, x),
               differentiate(t::F, x)::UP))
```

Bug #100 is an infinite loop that eventually kills Axiom from the input

```
integrate((z^a+1)^b,z)
```

Line 2 of this function used to read:

```
    symbolIfCan(k := kmax(l := union(l, varselect(kernels g, x))))
```

The loop occurs when the call to union causes

```
                a log(z)
               %e
```

to get added to the list every time. This gives the argument to kmax

```
              a log(z)
    arg1= [z,%e        ]
```

and the result being

```
      a log(z)
    %e
```

We keep coming back to process this term, which ends up putting the same term back on the list and we loop. Waldek's solution is to remove the union call.

The original patch fixed the infinite regression mentioned above but caused Axiom to return a closed form of the integral:

$$integrate(asech(x)/x, x)$$

which should not have a closed form. This is referenced in the FriCAS SVN revision 279.

Essentially this new patch uses only logarithms of rational functions when integrating rational functions. It is unclear whether this is the correct fix.

— package INTEF ElementaryIntegration —

```
lfextendedint(f, x, g) ==
  empty?(l := varselect(kernels f, x)) => [x::F * f, 0]
  symbolIfCan(k := kmax(l))
    case SE =>
     g1 :=
       empty?(l1 := varselect(kernels g,x)) => 0::F
```

```
      kmax(l1) = k => g
      0::F
    map((x1:RF):F +-> multivariate(x1, k),
         extendedint(univariate(f, k),
           univariate(g1, k)))
  is?(k, "exp"::SE) => expextint(f, x, k, g)
  prim?(k, x)       => primextint(f, x, k, g)
  has?(operator k, ALGOP) => alglfextint(f, k, l, x, g)
  unkextint(f, x, g)
```

———-

This is part of the fix for bug 100. Line 2 of this function used to read:

```
symbolIfCan(k := kmax(l := union(l, vark(lu, x)))) case SE =>
```

See the above discussion for why this causes an infinite loop.

— **package INTEF ElementaryIntegration** —

```
lflimitedint(f, x, lu) ==
  empty?(l := varselect(kernels f, x)) => [x::F * f, empty()]
  symbolIfCan(k := kmax(l)) case SE =>
   map((x1:RF):F +-> multivariate(x1, k),
        limitedint(univariate(f, k),
          [univariate(u, k) for u in lu]))
  is?(k, "exp"::SE) => explimint(f, x, k, lu)
  prim?(k, x)       => primlimint(f, x, k, lu)
  has?(operator k, ALGOP) => alglflimint(f, k, l, x, lu)
  unklimint(f, x, lu)

lfinfieldint(f, x) ==
  (u := lfextendedint(f, x, 0)) case "failed" => "failed"
  u.ratpart

primextint(f, x, k, g) ==
  lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
  (u1 := primextendedint(univariate(f, k),
   (x1:UP):UP +-> differentiate(x1,
    (x2:F):F +-> differentiate(x2, x), differentiate(k::F, x)::UP),
     (x3:F):U2+->lfextlimint(x3,x,k,lk), univariate(g, k))) case "failed"
           => "failed"
  u1 case FF =>
    [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
  (u2 := lfextendedint(u1.a0, x, g)) case "failed" => "failed"
  [multivariate(u1.answer, k) + u2.ratpart, u2.coeff]

expextint(f, x, k, g) ==
  (u1 := expextendedint(univariate(f, k),
   (x1:UP):UP +-> differentiate(x1,
```

```
      (x2:F):F +->  differentiate(x2, x),
       monomial(differentiate(first argument k, x), 1)),
        (x3:Integer,x4:F):PSOL+->rischDE(x3, first argument k, x4, x,
         (x5:F,x6:List F):U3 +-> lflimitedint(x5, x, x6),
          (x7:F,x8:F):U2+->lfextendedint(x7, x, x8)), univariate(g, k)))
             case "failed" => "failed"
    u1 case FF =>
      [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
    (u2 := lfextendedint(u1.a0, x, g)) case "failed" => "failed"
    [multivariate(u1.answer, k) + u2.ratpart, u2.coeff]

  primint(f, x, k) ==
    lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
    r1 := primintegrate(univariate(f, k),
           (x1:UP):UP +-> differentiate(x1,
            (x2:F):F +-> differentiate(x2, x), differentiate(k::F, x)::UP),
             (x3:F):U2 +-> lfextlimint(x3, x, k, lk))
    map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

  lfextlimint(f, x, k, lk) ==
    not((u1 := lfextendedint(f, x, differentiate(k::F, x)))
      case "failed") => u1
    twr := tower f
    empty?(lg := [kk for kk in lk | not member?(kk, twr)]) => "failed"
    is?(k, "log"::SE) =>
      (u2 := lflimitedint(f, x,
        [first argument u for u in union(lg, [k])])) case "failed"
                                                       => "failed"
      cf := cfind(first argument k, u2.limitedlogs)
      [u2.mainpart - cf * k::F +
              +/[c.coeff * log(c.logand) for c in u2.limitedlogs], cf]
    "failed"

  cfind(f, l) ==
    for u in l repeat
      f = u.logand => return u.coeff
    0

  expint(f, x, k) ==
    eta := first argument k
    r1  :=
     expintegrate(univariate(f, k),
      (x1:UP):UP +-> differentiate(x1,
       (x2:F):F +-> differentiate(x2, x),
        monomial(differentiate(eta, x), 1)),
         (x3:Integer,x4:F):PSOL+->rischDE(x3, eta, x4, x,
          (x5:F,x6:List F):U3 +-> lflimitedint(x5, x, x6),
           (x7:F,x8:F):U2+->lfextendedint(x7, x, x8)))
    map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)
```

```
    primlimint(f, x, k, lu) ==
      lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
      (u1 :=
        primlimitedint(univariate(f, k),
         (x1:UP):UP+->differentiate(x1,
           (x2:F):F+->differentiate(x2, x), differentiate(k::F, x)::UP),
            (x3:F):U2+->lfextlimint(x3,x,k,lk),
             [univariate(u, k) for u in lu]))
               case "failed" => "failed"
      l := [[multivariate(lg.coeff, k),multivariate(lg.logand, k)]
                                    for lg in u1.answer.limitedlogs]$LLG
      (u2 := lflimitedint(u1.a0, x, lu)) case "failed" => "failed"
      [multivariate(u1.answer.mainpart, k) + u2.mainpart,
                                           concat(u2.limitedlogs, l)]

    explimint(f, x, k, lu) ==
      eta := first argument k
      (u1 :=
        explimitedint(univariate(f, k),
         (x1:UP):UP+->differentiate(x1,
          (x2:F):F+->differentiate(x2,x), monomial(differentiate(eta,x), 1)),
           (x3:Integer,x4:F):PSOL+->rischDE(x3, eta, x4, x,
             (x5:F,x6:List F):U3+->lflimitedint(x5, x, x6),
              (x7:F,x8:F):U2+->lfextendedint(x7, x, x8)),
               [univariate(u, k) for u in lu])) case "failed" => "failed"
      l := [[multivariate(lg.coeff, k),multivariate(lg.logand, k)]
                                    for lg in u1.answer.limitedlogs]$LLG
      (u2 := lflimitedint(u1.a0, x, lu)) case "failed" => "failed"
      [multivariate(u1.answer.mainpart, k) + u2.mainpart,
                                           concat(u2.limitedlogs, l)]
```

———-

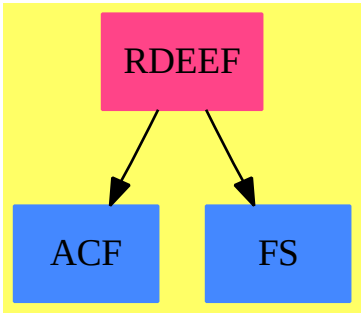
— INTEF.dotabb —

```
"INTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTEF" -> "ACF"
"INTEF" -> "FS"
```

———-

6.11 package RDEEF ElementaryRischDE

6.11.1 ElementaryRischDE (RDEEF)



Exports:
rischDE

— package RDEEF ElementaryRischDE —

```
)abbrev package RDEEF ElementaryRischDE
++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 2 November 1995
++ Keywords: elementary, function, integration.
++ Description:
++ Risch differential equation, elementary case.

ElementaryRischDE(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, AlgebraicallyClosedField,
           FunctionSpace R)

  N   ==> NonNegativeInteger
  Z   ==> Integer
  SE  ==> Symbol
  LF  ==> List F
  K   ==> Kernel F
  LK  ==> List K
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
  GP  ==> LaurentPolynomial(F, UP)
  Data ==> List Record(coeff:Z, argument:P)
  RRF ==> Record(mainpart:F,limitedlogs:List NL)
  NL  ==> Record(coeff:F,logand:F)
```

```
U   ==> Union(RRF, "failed")
UF  ==> Union(F, "failed")
UUP ==> Union(UP, "failed")
UGP ==> Union(GP, "failed")
URF ==> Union(RF, "failed")
UEX ==> Union(Record(ratpart:F,  coeff:F), "failed")
PSOL==> Record(ans:F, right:F, sol?:Boolean)
FAIL==> error("Function not supported by Risch d.e.")
ALGOP ==> "%alg"

Exports ==> with
  rischDE: (Z, F, F, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
       ++ rischDE(n, f, g, x, lim, ext) returns \spad{[y, h, b]} such that
       ++ \spad{dy/dx + n df/dx y = h} and \spad{b := h = g}.
       ++ The equation \spad{dy/dx + n df/dx y = g} has no solution
       ++ if \spad{h \~~= g} (y is a partial solution in that case).
       ++ Notes: lim is a limited integration function, and
       ++ ext is an extended integration function.

Implementation ==> add
  import IntegrationTools(R, F)
  import TranscendentalRischDE(F, UP)
  import TranscendentalIntegration(F, UP)
  import PureAlgebraicIntegration(R, F, F)
  import FunctionSpacePrimitiveElement(R, F)
  import ElementaryFunctionStructurePackage(R, F)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                        K, R, P, F)

  RF2GP:     RF -> GP
  makeData  : (F, SE, K)    -> Data
  normal0   : (Z, F, F, SE) -> UF
  normalise0: (Z, F, F, SE) -> PSOL
  normalise : (Z, F, F, F, SE, K, (F, LF) -> U, (F, F) -> UEX) -> PSOL
  rischDEalg: (Z, F, F, F, K, LK, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
  rischDElog: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
  rischDEexp: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
  polyDElog : (LK, UP, UP,UP,SE,K,UP->UP,(F,LF)->U,(F,F)->UEX) -> UUP
  polyDEexp : (LK, UP, UP,UP,SE,K,UP->UP,(F,LF)->U,(F,F)->UEX) -> UUP
  gpolDEexp : (LK, UP, GP,GP,SE,K,UP->UP,(F,LF)->U,(F,F)->UEX) -> UGP
  boundAt0  : (LK, F, Z,  Z,    SE, K, (F, LF) -> U) -> Z
  boundInf  : (LK, F, Z,  Z, Z, SE, K, (F, LF) -> U) -> Z
  logdegrad : (LK, F, UP, Z, SE, K,(F,LF)->U, (F,F) -> UEX) -> UUP
  expdegrad : (LK, F, UP, Z, SE, K,(F,LF)->U, (F,F) -> UEX) -> UUP
  logdeg    : (UP, F, Z, SE, F, (F, LF) -> U, (F, F) -> UEX) -> UUP
  expdeg    : (UP, F, Z, SE, F, (F, LF) -> U, (F, F) -> UEX) -> UUP
  exppolyint: (UP, (Z, F) -> PSOL) -> UUP
  RRF2F     : RRF -> F
  logdiff   : (List K, List K) -> List K
```

```
    tab:AssociationList(F, Data) := table()

    RF2GP f == (numer(f)::GP exquo denom(f)::GP)::GP

    logdiff(twr, bad) ==
      [u for u in twr | is?(u, "log"::SE) and not member?(u, bad)]

    rischDEalg(n, nfp, f, g, k, l, x, limint, extint) ==
      symbolIfCan(kx := ksec(k, l, x)) case SE =>
        (u := palgRDE(nfp, f, g, kx, k,
          (z1,z2,z3) +-> normal0(n, z1, z2, z3))) case "failed"
            => [0, 0, false]
        [u::F, g, true]
      has?(operator kx, ALGOP) =>
        rec := primitiveElement(kx::F, k::F)
        y   := rootOf(rec.prim)
        lk:LK := [kx, k]
        lv:LF := [(rec.pol1) y, (rec.pol2) y]
        rc := rischDE(n, eval(f, lk, lv), eval(g, lk, lv), x, limint, extint)
        rc.sol? => [eval(rc.ans, retract(y)@K, rec.primelt), rc.right, true]
        [0, 0, false]
      FAIL

-- solve y' + n f'y = g for a rational function y
    rischDE(n, f, g, x, limitedint, extendedint) ==
      zero? g => [0, g, true]
      zero?(nfp := n * differentiate(f, x)) =>
        (u := limitedint(g, empty())) case "failed" => [0, 0, false]
        [u.mainpart, g, true]
      freeOf?(y := g / nfp, x) => [y, g, true]
      vl := varselect(union(kernels nfp, kernels g), x)
      symbolIfCan(k := kmax vl) case SE => normalise0(n, f, g, x)
      is?(k, "log"::SE) or is?(k, "exp"::SE) =>
        normalise(n, nfp, f, g, x, k, limitedint, extendedint)
      has?(operator k, ALGOP) =>
        rischDEalg(n, nfp, f, g, k, vl, x, limitedint, extendedint)
      FAIL

    normal0(n, f, g, x) ==
      rec := normalise0(n, f, g, x)
      rec.sol? => rec.ans
      "failed"

-- solve y' + n f' y = g
-- when f' and g are rational functions over a constant field
    normalise0(n, f, g, x) ==
      k := kernel(x)@K
      if (data1 := search(f, tab)) case "failed" then
        tab.f := data := makeData(f, x, k)
      else data := data1::Data
```

```
      f'  := nfprime := n * differentiate(f, x)
      p:P := 1
      for v in data | (m := n * v.coeff) > 0 repeat
        p  := p * v.argument ** (m::N)
        f' := f' - m * differentiate(v.argument::F, x) / (v.argument::F)
      rec := baseRDE(univariate(f', k), univariate(p::F * g, k))
      y := multivariate(rec.ans, k) / p::F
      rec.nosol => [y, differentiate(y, x) + nfprime * y, false]
      [y, g, true]

-- make f weakly normalized, and solve y' + n f' y = g
    normalise(n, nfp, f, g, x, k, limitedint, extendedint) ==
      if (data1:= search(f, tab)) case "failed" then
        tab.f := data := makeData(f, x, k)
      else data := data1::Data
      p:P := 1
      for v in data | (m := n * v.coeff) > 0 repeat
        p  := p * v.argument ** (m::N)
        f  := f - v.coeff * log(v.argument::F)
        nfp := nfp - m * differentiate(v.argument::F, x) / (v.argument::F)
      newf := univariate(nfp, k)
      newg := univariate(p::F * g, k)
      twr := union(logdiff(tower f, empty()), logdiff(tower g, empty()))
      ans1 :=
        is?(k, "log"::SE) =>
          rischDElog(twr, newf, newg, x, k,
            z1 +-> differentiate(z1,(z2:F):F +-> differentiate(z2, x),
                             differentiate(k::F, x)::UP),
                                        limitedint, extendedint)
        is?(k, "exp"::SE) =>
          rischDEexp(twr, newf, newg, x, k,
            z1 +-> differentiate(z1, (z2:F):F +-> differentiate(z2, x),
                      monomial(differentiate(first argument k, x), 1)),
                                           limitedint, extendedint)
      ans1 case "failed" => [0, 0, false]
      [multivariate(ans1::RF, k) / p::F, g, true]

-- find the n * log(P) appearing in f, where P is in P, n in Z
    makeData(f, x, k) ==
      disasters := empty()$Data
      fnum := numer f
      fden := denom f
      for u in varselect(kernels f, x) | is?(u, "log"::SE) repeat
        logand := first argument u
        if zero?(degree univariate(fden, u)) and
--           one?(degree(num := univariate(fnum, u))) then
           (degree(num := univariate(fnum, u)) = 1) then
            cf := (leadingCoefficient num) / fden
            if (n := retractIfCan(cf)@Union(Z, "failed")) case Z then
              if degree(numer logand, k) > 0 then
```

```
            disasters := concat([n::Z, numer logand], disasters)
          if degree(denom logand, k) > 0 then
            disasters := concat([-(n::Z), denom logand], disasters)
  disasters

rischDElog(twr, f, g, x, theta, driv, limint, extint) ==
  (u := monomRDE(f, g, driv)) case "failed" => "failed"
  (v := polyDElog(twr, u.a, retract(u.b), retract(u.c), x, theta, driv,
                  limint, extint)) case "failed" => "failed"
  v::UP / u.t

rischDEexp(twr, f, g, x, theta, driv, limint, extint) ==
  (u := monomRDE(f, g, driv)) case "failed" => "failed"
  (v := gpolDEexp(twr, u.a, RF2GP(u.b), RF2GP(u.c), x, theta, driv,
                  limint, extint)) case "failed" => "failed"
  convert(v::GP)@RF / u.t::RF

polyDElog(twr, aa, bb, cc, x, t, driv, limint, extint) ==
  zero? cc => 0
  t' := differentiate(t::F, x)
  zero? bb =>
    (u := cc exquo aa) case "failed" => "failed"
    primintfldpoly(u::UP, z1 +-> extint(z1, t'), t')
  n := degree(cc)::Z - (db := degree(bb)::Z)
  if ((da := degree(aa)::Z) = db) and (da > 0) then
    lk0 := tower(f0 :=
                 - (leadingCoefficient bb) / (leadingCoefficient aa))
    lk1 := logdiff(twr, lk0)
    (if0 := limint(f0, [first argument u for u in lk1]))
                  case "failed" => error "Risch's theorem violated"
    (alph := validExponential(lk0, RRF2F(if0::RRF), x)) case F =>
      return
        (ans := polyDElog(twr, alph::F * aa,
          differentiate(alph::F, x) * aa + alph::F * bb,
           cc, x, t, driv, limint, extint)) case "failed" => "failed"
        alph::F * ans::UP
  if (da > db + 1) then n := max(0, degree(cc)::Z - da + 1)
  if (da = db + 1) then
    i := limint(- (leadingCoefficient bb) / (leadingCoefficient aa),
                [first argument t])
    if not(i case "failed") then
      r :=
        null(i.limitedlogs) => 0$F
        i.limitedlogs.first.coeff
      if (nn := retractIfCan(r)@Union(Z, "failed")) case Z then
        n := max(nn::Z, n)
  (v := polyRDE(aa, bb, cc, n, driv)) case ans =>
       v.ans.nosol => "failed"
       v.ans.ans
  w := v.eq
```

```
    zero?(w.b) =>
      degree(w.c) > w.m => "failed"
      (u := primintfldpoly(w.c, z1+->extint(1,t'), t'))
         case "failed" => "failed"
      degree(u::UP) > w.m => "failed"
      w.alpha * u::UP + w.beta
    (u := logdegrad(twr, retract(w.b), w.c, w.m, x, t, limint, extint))
      case "failed" => "failed"
    w.alpha * u::UP + w.beta

  gpolDEexp(twr, a, b, c, x, t, driv, limint, extint) ==
    zero? c => 0
    zero? b =>
      (u := c exquo (a::GP)) case "failed" => "failed"
      expintfldpoly(u::GP,
         (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
    lb := boundAt0(twr, - coefficient(b, 0) / coefficient(a, 0),
                  nb := order b, nc := order c, x, t, limint)
    tm := monomial(1, (m := max(0, max(-nb, lb - nc)))::N)$UP
    (v := polyDEexp(twr,a * tm,lb * differentiate(first argument t, x)
         * a * tm + retract(b * tm::GP)@UP,
             retract(c * monomial(1, m - lb))@UP,
                x, t, driv, limint, extint)) case "failed" => "failed"
    v::UP::GP * monomial(1, lb)

  polyDEexp(twr, aa, bb, cc, x, t, driv, limint, extint) ==
    zero? cc => 0
    zero? bb =>
      (u := cc exquo aa) case "failed" => "failed"
      exppolyint(u::UP,
        (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
    n := boundInf(twr,-leadingCoefficient(bb) / (leadingCoefficient aa),
               degree(aa)::Z, degree(bb)::Z, degree(cc)::Z, x, t, limint)
    (v := polyRDE(aa, bb, cc, n, driv)) case ans =>
         v.ans.nosol => "failed"
         v.ans.ans
    w := v.eq
    zero?(w.b) =>
      degree(w.c) > w.m => "failed"
      (u := exppolyint(w.c,
        (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint)))
                      case "failed" => "failed"
      w.alpha * u::UP + w.beta
    (u := expdegrad(twr, retract(w.b), w.c, w.m, x, t, limint, extint))
      case "failed" => "failed"
    w.alpha * u::UP + w.beta

  exppolyint(p, rischdiffeq) ==
    (u := expintfldpoly(p::GP, rischdiffeq)) case "failed" => "failed"
    retractIfCan(u::GP)@Union(UP, "failed")
```

```
    boundInf(twr, f0, da, db, dc, x, t, limitedint) ==
      da < db => dc - db
      da > db => max(0, dc - da)
      l1 := logdiff(twr, l0 := tower f0)
      (if0 := limitedint(f0, [first argument u for u in l1]))
                      case "failed" => error "Risch's theorem violated"
      (alpha := validExponential(concat(t, l0), RRF2F(if0::RRF), x))
       case F =>
        al := separate(univariate(alpha::F, t))$GP
        zero?(al.fracPart) and monomial?(al.polyPart) =>
                              max(0, max(degree(al.polyPart), dc - db))
        dc - db
      dc - db

    boundAt0(twr, f0, nb, nc, x, t, limitedint) ==
      nb ^= 0 => min(0, nc - min(0, nb))
      l1 := logdiff(twr, l0 := tower f0)
      (if0 := limitedint(f0, [first argument u for u in l1]))
                      case "failed" => error "Risch's theorem violated"
      (alpha := validExponential(concat(t, l0), RRF2F(if0::RRF), x))
       case F =>
        al := separate(univariate(alpha::F, t))$GP
        zero?(al.fracPart) and monomial?(al.polyPart) =>
                                    min(0, min(degree(al.polyPart), nc))
        min(0, nc)
      min(0, nc)

-- case a = 1, deg(B) = 0, B <> 0
-- cancellation at infinity is possible
    logdegrad(twr, b, c, n, x, t, limitedint, extint) ==
      t'  := differentiate(t::F, x)
      lk1 := logdiff(twr, lk0 := tower(f0 := - b))
      (if0 := limitedint(f0, [first argument u for u in lk1]))
                      case "failed" => error "Risch's theorem violated"
      (alpha := validExponential(lk0, RRF2F(if0::RRF), x)) case F =>
        (u1 := primintfldpoly(inv(alpha::F) * c, z1+->extint(z1, t'), t'))
                                              case "failed" => "failed"
        degree(u1::UP)::Z > n => "failed"
        alpha::F * u1::UP
      logdeg(c, - if0.mainpart -
               +/[v.coeff * log(v.logand) for v in if0.limitedlogs],
                                         n, x, t', limitedint, extint)

-- case a = 1, degree(b) = 0, and (exp integrate b) is not in F
-- this implies no cancellation at infinity
    logdeg(c, f, n, x, t', limitedint, extint) ==
      answr:UP := 0
      repeat
        zero? c => return answr
```

```
        (n < 0) or ((m := degree c)::Z > n) => return "failed"
        u := rischDE(1, f, leadingCoefficient c, x, limitedint, extint)
        ~u.sol? => return "failed"
        zero? m => return(answr + u.ans::UP)
        n   := m::Z - 1
        c   := (reductum c) - monomial(m::Z * t' * u.ans, (m - 1)::N)
        answr := answr + monomial(u.ans, m)

-- case a = 1, deg(B) = 0, B <> 0
-- cancellation at infinity is possible
    expdegrad(twr, b, c, n, x, t, limint, extint) ==
      lk1 := logdiff(twr, lk0 := tower(f0 := - b))
      (if0 := limint(f0, [first argument u for u in lk1]))
                      case "failed" => error "Risch's theorem violated"
      intf0 := - if0.mainpart -
                   +/[v.coeff * log(v.logand) for v in if0.limitedlogs]
      (alpha := validExponential(concat(t, lk0), RRF2F(if0::RRF), x))
       case F =>
        al := separate(univariate(alpha::F, t))$GP
        zero?(al.fracPart) and monomial?(al.polyPart) and
         (degree(al.polyPart) >= 0) =>
          (u1 := expintfldpoly(c::GP * recip(al.polyPart)::GP,
            (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint)))
                                              case "failed" => "failed"
          degree(u1::GP) > n => "failed"
          retractIfCan(al.polyPart * u1::GP)@Union(UP, "failed")
        expdeg(c, intf0, n, x, first argument t, limint,extint)
      expdeg(c, intf0, n, x, first argument t, limint, extint)

-- case a = 1, degree(b) = 0, and (exp integrate b) is not a monomial
-- this implies no cancellation at infinity
    expdeg(c, f, n, x, eta, limitedint, extint) ==
      answr:UP := 0
      repeat
        zero? c => return answr
        (n < 0) or ((m := degree c)::Z > n) => return "failed"
        u := rischDE(1, f + m * eta, leadingCoefficient c, x,limitedint,extint)
        ~u.sol? => return "failed"
        zero? m => return(answr + u.ans::UP)
        n   := m::Z - 1
        c   := reductum c
        answr := answr + monomial(u.ans, m)

    RRF2F rrf ==
      rrf.mainpart + +/[v.coeff*log(v.logand) for v in rrf.limitedlogs]
```

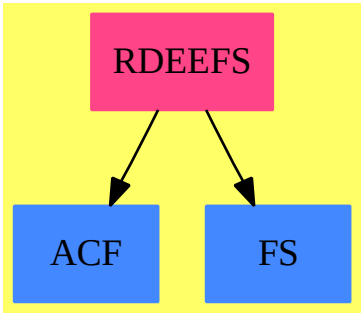
— **RDEEF.dotabb** —

```
"RDEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDEEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"RDEEF" -> "ACF"
"RDEEF" -> "FS"
```

———-

6.12 package RDEEFS ElementaryRischDESystem

6.12.1 ElementaryRischDESystem (RDEEFS)



Exports:
rischDEsys

— **package RDEEFS ElementaryRischDESystem** —

```
)abbrev package RDEEFS ElementaryRischDESystem
++ Author: Manuel Bronstein
++ Date Created: 12 August 1992
++ Date Last Updated: 17 August 1992
++ Keywords: elementary, function, integration.
++ Description:
++ Risch differential equation, elementary case.

ElementaryRischDESystem(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, AlgebraicallyClosedField,
           FunctionSpace R)

  Z   ==> Integer
```

```
  SE  ==> Symbol
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
  NL  ==> Record(coeff:F,logand:F)
  RRF ==> Record(mainpart:F,limitedlogs:List NL)
  U   ==> Union(RRF, "failed")
  ULF ==> Union(List F, "failed")
  UEX ==> Union(Record(ratpart:F, coeff:F), "failed")

  Exports ==> with
    rischDEsys: (Z, F, F, F, SE, (F, List F) -> U, (F, F) -> UEX) -> ULF
      ++ rischDEsys(n, f, g_1, g_2, x,lim,ext) returns \spad{y_1.y_2} such that
      ++ \spad{(dy1/dx,dy2/dx) + ((0, - n df/dx),(n df/dx,0)) (y1,y2) = (g1,g2)}
      ++ if \spad{y_1,y_2} exist, "failed" otherwise.
      ++ lim is a limited integration function,
      ++ ext is an extended integration function.

  Implementation ==> add
    import IntegrationTools(R, F)
    import ElementaryRischDE(R, F)
    import TranscendentalRischDESystem(F, UP)
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                        K, R, P, F)

--  sm1 := sqrt(-1::F)
--  ks1 := retract(sm1)@K

--  gcoeffs    : P -> ULF
--  gets1coeffs: F -> ULF
--  cheat      : (Z, F, F, F, SE, (F, List F) -> U, (F, F) -> UEX) -> ULF
    basecase   : (F, F, F, K) -> ULF

-- solve (y1',y2') + ((0, -nfp), (nfp, 0)) (y1,y2) = (g1, g2), base case
    basecase(nfp, g1, g2, k) ==
      (ans := baseRDEsys(univariate(nfp, k), univariate(g1, k),
                         univariate(g2, k))) case "failed" => "failed"
      l := ans::List(RF)
      [multivariate(first l, k), multivariate(second l, k)]

-- returns [x,y] s.t. f = x + y %i
-- f can be of the form (a + b %i) / (c + d %i)
--  gets1coeffs f ==
--    (lnum := gcoeffs(numer f)) case "failed" => "failed"
--    (lden := gcoeffs(denom f)) case "failed" => "failed"
--    a := first(lnum::List F)
--    b := second(lnum::List F)
--    c := first(lden::List F)
--    zero?(d := second(lden::List F)) => [a/c, b/c]
```

```
--    cd := c * c + d * d
--    [(a * c + b * d) / cd, (b * c - a * d) / cd]

--  gcoeffs p ==
--    degree(q := univariate(p, ks1)) > 1 => "failed"
--    [coefficient(q, 0)::F, coefficient(q, 1)::F]

--  cheat(n, f, g1, g2, x, limint, extint) ==
--    (u := rischDE(n, sm1 * f, g1 + sm1 * g2, x, limint, extint))
--      case "failed" => "failed"
--    (l := gets1coeffs(u::F)) case "failed" =>
--      error "rischDEsys: expect linear result in sqrt(-1)"
--    l::List F

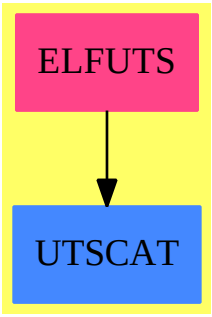
-- solve (y1',y2') + ((0, -n f'), (n f', 0)) (y1,y2) = (g1, g2)
    rischDEsys(n, f, g1, g2, x, limint, extint) ==
      zero? g1 and zero? g2 => [0, 0]
      zero?(nfp := n * differentiate(f, x)) =>
        ((u1 := limint(g1, empty())) case "failed") or
          ((u2 := limint(g1, empty())) case "failed") => "failed"
        [u1.mainpart, u2.mainpart]
      freeOf?(y1 := g2 / nfp, x) and freeOf?(y2 := - g1 / nfp, x) => [y1, y2]
      vl := varselect(union(kernels nfp, union(kernels g1, kernels g2)), x)
      symbolIfCan(k := kmax vl) case SE => basecase(nfp, g1, g2, k)
--    cheat(n, f, g1, g2, x, limint, extint)
      error "rischDEsys: can only handle rational functions for now"
```

— RDEEFS.dotabb —

```
"RDEEFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDEEFS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"RDEEFS" -> "ACF"
"RDEEFS" -> "FS"
```

6.13 package ELFUTS EllipticFunctionsUnivariateTaylorSeries

6.13.1 EllipticFunctionsUnivariateTaylorSeries (ELFUTS)



Exports:
cn dn sn sncndn

— **package ELFUTS EllipticFunctionsUnivariateTaylorSeries** —

```
)abbrev package ELFUTS EllipticFunctionsUnivariateTaylorSeries
++ Author: Bill Burge, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 17 February 1992
++ Keywords: elliptic function, Taylor series
++ Examples:
++ References:
++ Description:
++ The elliptic functions sn, sc and dn are expanded as Taylor series.

EllipticFunctionsUnivariateTaylorSeries(Coef,UTS):
 Exports == Implementation where
  Coef : Field
  UTS  : UnivariateTaylorSeriesCategory Coef

  L   ==> List
  I   ==> Integer
  RN  ==> Fraction Integer
  ST  ==> Stream Coef
  STT ==> StreamTaylorSeriesOperations Coef
  YS  ==> Y$ParadoxicalCombinatorsForStreams(Coef)

  Exports ==> with
    sn     : (UTS,Coef) -> UTS
      ++\spad{sn(x,k)} expands the elliptic function sn as a Taylor
```

```
      ++ series.
    cn     : (UTS,Coef) -> UTS
      ++\spad{cn(x,k)} expands the elliptic function cn as a Taylor
      ++ series.
    dn     : (UTS,Coef) -> UTS
      ++\spad{dn(x,k)} expands the elliptic function dn as a Taylor
      ++ series.
    sncndn: (ST,Coef) -> L ST
       ++\spad{sncndn(s,c)} is used internally.

  Implementation ==> add
    import StreamTaylorSeriesOperations Coef
    UPS==> StreamTaylorSeriesOperations Coef
    integrate ==> lazyIntegrate
    sncndnre:(Coef,L ST,ST,Coef) -> L ST
    sncndnre(k,scd,dx,sign) ==
            [integrate(0,      scd.2*$UPS scd.3*$UPS dx),  _
             integrate(1,  sign*scd.1*$UPS scd.3*$UPS dx),  _
             integrate(1,sign*k**2*$UPS scd.1*$UPS scd.2*$UPS dx)]

    sncndn(z,k) ==
      empty? z => [0 :: ST,1 :: ST,1::ST]
      frst z = 0 => YS(x +-> sncndnre(k,x,deriv z,-1),3)
      error "ELFUTS:sncndn: constant coefficient should be 0"
    sn(x,k)  == series sncndn.(coefficients x,k).1
    cn(x,k)  == series sncndn.(coefficients x,k).2
    dn(x,k)  == series sncndn.(coefficients x,k).3
```

———-

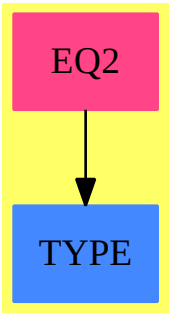
— ELFUTS.dotabb —

```
"ELFUTS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ELFUTS"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"ELFUTS" -> "UTSCAT"
```

———-

6.14 package EQ2 EquationFunctions2

6.14.1 EquationFunctions2 (EQ2)



Exports:
map

— **package EQ2 EquationFunctions2** —

```
)abbrev package EQ2 EquationFunctions2
++ Author:
++ Date Created:
++ Date Last Updated: June 3, 1991
++ Basic Operations:
++ Related Domains: Equation
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ This package provides operations for mapping the sides of equations.

EquationFunctions2(S: Type, R: Type): with
    map: (S ->R ,Equation S) -> Equation R
      ++ map(f,eq) returns an equation where f is applied to the sides of eq
 == add
    map(fn, eqn) == equation(fn lhs eqn, fn rhs eqn)
```

————-

— **EQ2.dotabb** —

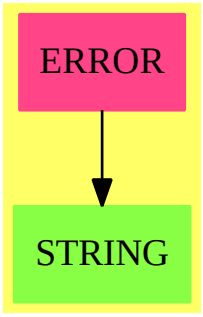
```
"EQ2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EQ2"]
```

```
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"EQ2" -> "TYPE"
```

———-

6.15 package ERROR ErrorFunctions

6.15.1 ErrorFunctions (ERROR)



Exports:
error

— package ERROR ErrorFunctions —

```
)abbrev package ERROR ErrorFunctions
++ Author: Robert S. Sutor
++ Date Created: 29 May 1990
++ Date Last Updated: 29 May 1990
++ Description:
++ ErrorFunctions implements error functions callable from the system
++ interpreter.  Typically, these functions would be called in user
++ functions.  The simple forms of the functions take one argument
++ which is either a string (an error message) or a list of strings
++ which all together make up a message.  The list can contain
++ formatting codes (see below).  The more sophisticated versions takes
++ two arguments where the first argument is the name of the function
++ from which the error was invoked and the second argument is either a
++ string or a list of strings, as above.  When you use the one
++ argument version in an interpreter function, the system will
++ automatically insert the name of the function as the new first
++ argument.  Thus in the user interpreter function\br
++ \tab{5}\spad{f x == if x < 0 then error "negative argument" else x}\br
++ the call to error will actually be of the form\br
++ \tab{5}\spad{error("f","negative argument")}\br
```

```
++ because the interpreter will have created a new first argument.
++
++ Formatting codes:  error messages may contain the following
++ formatting codes (they should either start or end a string or
++ else have blanks around them):\br
++ \spad{%l}\tab{6}start a new line\br
++ \spad{%b}\tab{6}start printing in a bold font (where available)\br
++ \spad{%d}\tab{6}stop  printing in a bold font (where available)\br
++ \spad{%ceon}\tab{3}start centering message lines\br
++ \spad{%ceoff}\tab{2}stop  centering message lines\br
++ \spad{%rjon}\tab{3}start displaying lines "ragged left"\br
++ \spad{%rjoff}\tab{2}stop  displaying lines "ragged left"\br
++ \spad{%i}\tab{6}indent   following lines 3 additional spaces\br
++ \spad{%u}\tab{6}unindent following lines 3 additional spaces\br
++ \spad{%xN}\tab{5}insert N blanks (eg, \spad{%x10} inserts 10 blanks)
++
++ Examples:\br
++ 1.\spad{error "Whoops, you made a %l %ceon %b big %d %ceoff %l mistake!"}\br
++ 2.\spad{error ["Whoops, you made a","%l %ceon %b","big",\br
++ \tab{10}"%d %ceoff %l","mistake!"]}

ErrorFunctions() : Exports == Implementation where
  Exports ==> with
    error: String -> Exit
      ++ error(msg) displays error message msg and terminates.
    error: List String -> Exit
      ++ error(lmsg) displays error message lmsg and terminates.
    error: (String,String) -> Exit
      ++ error(nam,msg) displays error message msg preceded by a
      ++ message containing the name nam of the function in which
      ++ the error is contained.
    error: (String,List String) -> Exit
      ++ error(nam,lmsg) displays error messages lmsg preceded by a
      ++ message containing the name nam of the function in which
      ++ the error is contained.
  Implementation ==> add

    prefix1 : String := "Error signalled from user code: %l "
    prefix2 : String := "Error signalled from user code in function %b "

    doit(s : String) : Exit ==
      throwPatternMsg(s,nil$(List String))$Lisp
      -- there are no objects of type Exit, so we'll fake one,
      -- knowing we will never get to this step anyway.
      "exit" pretend Exit

    error(s : String) : Exit ==
      doit concat [prefix1,s]

    error(l : List String) : Exit ==
```

```
      s : String := prefix1
      for x in l repeat s := concat [s," ",x]
      doit s

    error(fn : String,s : String) : Exit ==
      doit concat [prefix2,fn,": %d %l ",s]

    error(fn : String, l : List String) : Exit ==
      s : String := concat [prefix2,fn,": %d %l"]
      for x in l repeat s := concat [s," ",x]
      doit s
```

———-

— ERROR.dotabb —

```
"ERROR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ERROR"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"ERROR" -> "STRING"
```

———-

6.16 package GBEUCLID EuclideanGroebnerBasisPackage

— EuclideanGroebnerBasisPackage.input —

```
)set break resume
)sys rm -f EuclideanGroebnerBasisPackage.output
)spool EuclideanGroebnerBasisPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 24
a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
--R
--R            2                2
--R   (1)  3y x  + 2y x + y + 9x  + 5x - 3
--R                       Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 1

--S 2 of 24
a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
```

```
--R
--R           3              3     2
--R   (2)  2y x  - y x - y + 6x  - 2x  - 3x + 3
--R                      Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 2

--S 3 of 24
a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
--R
--R          3      2     3     2
--R   (3)  y x  + y x  + 3x  + 2x
--R                      Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 3

--S 4 of 24
an:=[a1,a2,a3]
--R
--R   (4)
--R        2               2              3              3     2
--R   [3y x  + 2y x + y + 9x  + 5x - 3, 2y x  - y x - y + 6x  - 2x  - 3x + 3,
--R       3      2     3     2
--R    y x  + y x  + 3x  + 2x ]
--R                 Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 4

--S 5 of 24
euclideanGroebner(an)
--R
--R                                2           3     2
--R   (5)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--R                 Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 5

--S 6 of 24
euclideanGroebner(an,"redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2              2
--R   - 2y x  - y x - y - 6x  - 3x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
```

```
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   4y + 4x  - 6x - 12
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3      2
--R   - 4x  + 10x  + 10x
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   2y + 2x  - 3x - 6
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3     2
--R   - 2x  + 5x  + 5x
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
```

```
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                  2            3     2
--R   (6)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--R                  Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 6

--S 7 of 24
euclideanGroebner(an,"info")
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R            3              3             2             2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
```

```
--R
--R           2               2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 2,tD= 3]]
--R
--R
--R                                                  3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 3,tD= 4]]
--R
--R
--R   [[ci= y,tci= 4,cj= y,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
--R
--R
--R                                                  3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R          3             3
--R   [[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 2]]
--R
--R
--R           3                               2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R
--R                      3              2
--R   [[ci= y,tci= 4,cj= x ,tcj= 3,c= y x ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                 2            3     2
--R   (7)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--R                 Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 7
```

```
--S 8 of 24
euclideanGroebner(an,"info","redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2               2
--R   - 2y x  - y x - y - 6x  - 3x + 3
--R
--R
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R           3               3              2              2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
--R
--R
--R
--R           2               2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
```

```
--R   4y + 4x  - 6x - 12
--R
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 2,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3      2
--R   - 4x  + 10x  + 10x
--R
--R
--R
--R                                                  3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   2y + 2x  - 3x - 6
--R
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 3,tD= 4]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R   [[ci= y,tci= 4,cj= y,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3     2
```

```
--R   - 2x  + 5x  + 5x
--R
--R
--R
--R                                                  3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R          3             3
--R   [[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R            3                              2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                        3              2
--R   [[ci= y,tci= 4,cj= x ,tcj= 3,c= y x ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
```

```
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                  2            3     2
--R   (8)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--R                 Type: List(DistributedMultivariatePolynomial([y,x],Integer))
--E 8

--S 9 of 24
b1:HDMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
--R
--R            2            2
--R   (9)  3y x  + 2y x + 9x  + y + 5x - 3
--R            Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)
--E 9

--S 10 of 24
b2:HDMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
--R
--R             3     3           2
--R   (10)  2y x  + 6x  - y x - 2x  - y - 3x + 3
--R            Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)
--E 10

--S 11 of 24
b3:HDMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
--R
--R            3      2     3     2
--R   (11)  y x  + y x  + 3x  + 2x
--R            Type: HomogeneousDistributedMultivariatePolynomial([y,x],Integer)
--E 11

--S 12 of 24
bn:=[b1,b2,b3]
--R
--R   (12)
--R        2            2                   3     3           2
--R   [3y x  + 2y x + 9x  + y + 5x - 3, 2y x  + 6x  - y x - 2x  - y - 3x + 3,
--R       3      2     3     2
--R    y x  + y x  + 3x  + 2x ]
--R      Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 12

--S 13 of 24
euclideanGroebner(bn)
--R
--R            2                                 2
--R   (13)  [2y  - 5y - 8x - 3,y x - y + x + 3,2x  + 2y - 3x - 6]
--R      Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 13
```

```
--S 14 of 24
euclideanGroebner(bn,"redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2           2
--R   - 2y x  - y x - 6x  - y - 3x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R     2
--R   4x  + 4y - 6x - 12
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R     2
--R   2x  + 2y - 3x - 6
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
```

```
--R
--R       2
--R   - 2y  + 5y + 8x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R            2                                  2
--R   (14)  [2y  - 5y - 8x - 3,y x - y + x + 3,2x  + 2y - 3x - 6]
--R      Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 14

--S 15 of 24
euclideanGroebner(bn,"info")
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R            3              3             2             2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
```

```
--R
--R
--R           2               2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R           2                                          2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
--R
--R
--R           2                                          2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
--R
--R
--R          2             2
--R   [[ci= x ,tci= 4,cj= x ,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 2,tD= 2]]
--R
--R
--R                         2            2            2
--R   [[ci= y x,tci= 4,cj= x ,tcj= 4,c= y ,tc= 5,rc= y ,trc= 4,tH= 3,tD= 2]]
--R
--R
--R                         2            2
--R   [[ci= y x,tci= 4,cj= y ,tcj= 4,c= y ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R
--R           3                              2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R           2                                   2
--R   (15)  [2y  - 5y - 8x - 3,y x - y + x + 3,2x  + 2y - 3x - 6]
--R      Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 15

--S 16 of 24
euclideanGroebner(bn,"info","redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
```

```
--R
--R         2           2
--R   - 2y x  - y x - 6x  - y - 3x + 3
--R
--R
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R            3               3              2              2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
--R
--R
--R
--R            2               2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R     2
--R   4x  + 4y - 6x - 12
--R
--R
--R
--R            2                                        2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R     2
--R   2x  + 2y - 3x - 6
--R
--R
--R
--R           2                                          2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R         2             2
--R   [[ci= x ,tci= 4,cj= x ,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 2,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       2
--R   - 2y  + 5y + 8x + 3
--R
--R
--R
--R                         2            2            2
--R   [[ci= y x,tci= 4,cj= x ,tcj= 4,c= y ,tc= 5,rc= y ,trc= 4,tH= 3,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                         2            2
--R   [[ci= y x,tci= 4,cj= y ,tcj= 4,c= y ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 1]]
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R            3                                2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R            2                                    2
--R   (16)  [2y  - 5y - 8x - 3,y x - y + x + 3,2x  + 2y - 3x - 6]
--R      Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 16

--S 17 of 24
c1:GDMP([y,x],INT,DIRPROD(2,NNI)):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
--R
--R             2               2
--R   (17)  3y x  + 2y x + y + 9x  + 5x - 3
--IType: GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 17

--S 18 of 24
c2:GDMP([y,x],INT,DIRPROD(2,NNI)):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
--R
--R             3               3     2
--R   (18)  2y x  - y x - y + 6x  - 2x  - 3x + 3
--IType: GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 18

--S 19 of 24
c3:GDMP([y,x],INT,DIRPROD(2,NNI)):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
--R
--R            3      2     3     2
--R   (19)  y x  + y x  + 3x  + 2x
--IType: GeneralDistributedMultivariatePolynomial([y,x],Integer,...
```

```
--E 19

--S 20 of 24
cn:=[c1,c2,c3]
--R
--R   (20)
--R        2               2              3               3     2
--R   [3y x  + 2y x + y + 9x  + 5x - 3, 2y x  - y x - y + 6x  - 2x  - 3x + 3,
--R       3      2     3     2
--R    y x  + y x  + 3x  + 2x ]
--IType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 20

--S 21 of 24
euclideanGroebner(cn)
--R
--R                                  2            3     2
--R   (21)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--IType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 21

--S 22 of 24
euclideanGroebner(cn,"redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2               2
--R   - 2y x  - y x - y - 6x  - 3x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   4y + 4x  - 6x - 12
--R
--R
--R
```

```
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3      2
--R   - 4x  + 10x  + 10x
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   2y + 2x  - 3x - 6
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3     2
--R   - 2x  + 5x  + 5x
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
```

```
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                  2            3     2
--R   (22)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--IType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 22

--S 23 of 24
euclideanGroebner(cn,"info")
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R           3              3             2             2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R
--R           2              2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 2,tD= 3]]
--R
--R
--R                                                 3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
```

```
--R
--R            2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 3,tD= 4]]
--R
--R
--R   [[ci= y,tci= 4,cj= y,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
--R
--R
--R                                                  3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R          3             3
--R   [[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 2]]
--R
--R
--R            3                               2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R
--R                       3              2
--R   [[ci= y,tci= 4,cj= x ,tcj= 3,c= y x ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                    2            3     2
--R   (23)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--IType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 23

--S 24 of 24
euclideanGroebner(cn,"info","redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2              2
--R   - 2y x  - y x - y - 6x  - 3x + 3
--R
--R
--R
```

```
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R           3               3              2              2
--R   [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   y x - y + x + 3
--R
--R
--R
--R           2               2
--R   [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   4y + 4x  - 6x - 12
--R
--R
--R
--R           2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 2,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
```

```
--R       3      2
--R   - 4x  + 10x  + 10x
--R
--R
--R
--R                                                    3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          2
--R   2y + 2x  - 3x - 6
--R
--R
--R
--R            2
--R   [[ci= y x ,tci= 6,cj= y x,tcj= 4,c= y x,tc= 5,rc= y,trc= 4,tH= 3,tD= 4]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R   [[ci= y,tci= 4,cj= y,tcj= 4,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       3     2
--R   - 2x  + 5x  + 5x
--R
--R
--R
--R                                                    3
--R   [[ci= y x,tci= 4,cj= y,tcj= 4,c= y,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
```

```
--R   0
--R
--R
--R
--R          3             3
--R   [[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R           3                              2
--R   [[ci= y x ,tci= 4,cj= y x,tcj= 4,c= y x ,tc= 3,rc= 0,trc= 0,tH= 3,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                       3             2
--R   [[ci= y,tci= 4,cj= x ,tcj= 3,c= y x ,tc= 5,rc= 0,trc= 0,tH= 3,tD= 0]]
--R
--R
--R     There are
--R
--R   3
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R                                  2           3     2
--R   (24)  [y x - y + x + 3,2y + 2x  - 3x - 6,2x  - 5x  - 5x]
--IType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 24

)spool
)lisp (bye)
```

— EuclideanGroebnerBasisPackage.help —

```
=====================================================================
euclideanGroebner examples
=====================================================================

Example to call euclideanGroebner:

  a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
  a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
  a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
  an:=[a1,a2,a3]
  euclideanGroebner(an)

This will return the weak euclidean Groebner basis set.
All reductions are total reductions.

You can get more information by providing a second argument.
To get the reduced critical pairs do:

  euclideanGroebner(an,"redcrit")

You can get other information by calling:

  euclideanGroebner(an,"info")

which returns:
   ci  =>  Leading monomial  for critpair calculation
   tci =>  Number of terms of polynomial i
   cj  =>  Leading monomial  for critpair calculation
   tcj =>  Number of terms of polynomial j
   c   =>  Leading monomial of critpair polynomial
   tc  =>  Number of terms of critpair polynomial
   rc  =>  Leading monomial of redcritpair polynomial
   trc =>  Number of terms of redcritpair polynomial
   tH  =>  Number of polynomials in reduction list H
   tD  =>  Number of critpairs still to do

The three argument form returns all of the information:

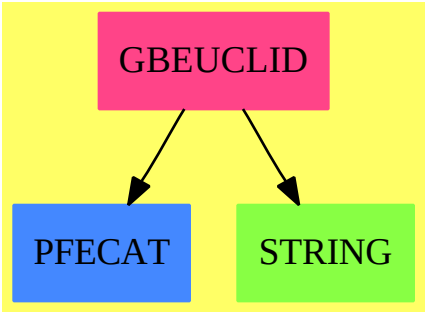
  euclideanGroebner(an,"info","redcrit")


The term ordering is determined by the polynomial type used.
Suggested types include
   DistributedMultivariatePolynomial
   HomogeneousDistributedMultivariatePolynomial
   GeneralDistributedMultivariatePolynomial
```

```
See Also:
o )display operations euclideanGroebner
o )show EuclideanGroebnerBasisPackage
o )show DistributedMultivariatePolynomial
o )show HomogeneousDistributedMultivariatePolynomial
o )show GeneralDistributedMultivariatePolynomial
o )show GroebnerPackage
```

———-

6.16.1 EuclideanGroebnerBasisPackage (GBEUCLID)



Exports:
euclideanGroebner euclideanNormalForm

— package GBEUCLID EuclideanGroebnerBasisPackage —

```
)abbrev package GBEUCLID EuclideanGroebnerBasisPackage
++ Authors: Gebauer, Moeller
++ Date Created: 12-1-86
++ Date Last Updated: 2-28-91
++ Basic Functions:
++ Related Constructors: Ideal, IdealDecompositionPackage, GroebnerPackage
++ Also See:
++ AMS Classifications:
++ Keywords: groebner basis, polynomial ideal, euclidean domain
++ References:
++ Description:
++ \spadtype{EuclideanGroebnerBasisPackage} computes groebner
++ bases for polynomial ideals over euclidean domains.
++ The basic computation provides
++ a distinguished set of generators for these ideals.
++ This basis allows an easy test for membership: the operation
```

```
++ \spadfun{euclideanNormalForm} returns zero on ideal members. The string
++ "info" and "redcrit" can be given as additional args to provide
++ incremental information during the computation. If "info" is given,
++ a computational summary is given for each s-polynomial. If "redcrit"
++ is given, the reduced critical pairs are printed. The term ordering
++ is determined by the polynomial type used. Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

EuclideanGroebnerBasisPackage(Dom, Expon, VarSet, Dpol): T == C where

 Dom: EuclideanDomain
 Expon: OrderedAbelianMonoidSup
 VarSet: OrderedSet
 Dpol: PolynomialCategory(Dom, Expon, VarSet)

 T== with

     euclideanNormalForm: (Dpol, List(Dpol) )  ->  Dpol
       ++ euclideanNormalForm(poly,gb) reduces the polynomial poly modulo the
       ++ precomputed groebner basis gb giving a canonical representative
       ++ of the residue class.
     euclideanGroebner: List(Dpol) -> List(Dpol)
       ++ euclideanGroebner(lp) computes a groebner basis for a polynomial
       ++ ideal over a euclidean domain generated by the list of polys lp.
       ++
       ++X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
       ++X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
       ++X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
       ++X an:=[a1,a2,a3]
       ++X euclideanGroebner(an)

     euclideanGroebner: (List(Dpol), String) -> List(Dpol)
       ++ euclideanGroebner(lp, infoflag) computes a groebner basis
       ++ for a polynomial ideal over a euclidean domain
       ++ generated by the list of polynomials lp.
       ++ During computation, additional information is printed out
       ++ if infoflag is given as
       ++ either "info" (for summary information) or
       ++ "redcrit" (for reduced critical pairs)
       ++
       ++X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
       ++X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
       ++X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
       ++X an:=[a1,a2,a3]
       ++X euclideanGroebner(an,"redcrit")
       ++X euclideanGroebner(an,"info")

     euclideanGroebner: (List(Dpol), String, String ) -> List(Dpol)
```

```
      ++ euclideanGroebner(lp, "info", "redcrit") computes a groebner basis
      ++ for a polynomial ideal generated by the list of polynomials lp.
      ++ If the second argument is "info",
      ++ a summary is given of the critical pairs.
      ++ If the third argument is "redcrit", critical pairs are printed.
      ++
      ++X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
      ++X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
      ++X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
      ++X an:=[a1,a2,a3]
      ++X euclideanGroebner(an,"info","redcrit")

C== add
  Ex ==> OutputForm
  lc ==> leadingCoefficient
  red ==> reductum

  import OutputForm

  ------  Definition list of critPair
  ------  lcmfij is now lcm of headterm of poli and polj
  ------  lcmcij is now lcm of of lc poli and lc polj

  critPair ==>Record(lcmfij: Expon, lcmcij: Dom, poli:Dpol, polj: Dpol )
  Prinp    ==> Record( ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol,
              tc:Integer,rc:Dpol,trc:Integer,tH:Integer,tD:Integer)

  ------  Definition of intermediate functions

  strongGbasis: (List(Dpol), Integer, Integer) -> List(Dpol)
  eminGbasis: List(Dpol) -> List(Dpol)
  ecritT: (critPair ) -> Boolean
  ecritM: (Expon, Dom, Expon, Dom) -> Boolean
  ecritB: (Expon, Dom, Expon, Dom, Expon, Dom) -> Boolean
  ecrithinH: (Dpol, List(Dpol)) -> Boolean
  ecritBonD: (Dpol, List(critPair)) -> List(critPair)
  ecritMTondd1:(List(critPair)) -> List(critPair)
  ecritMondd1:(Expon, Dom, List(critPair)) -> List(critPair)
  crithdelH: (Dpol, List(Dpol)) -> List(Dpol)
  eupdatF: (Dpol, List(Dpol) ) -> List(Dpol)
  updatH: (Dpol, List(Dpol), List(Dpol), List(Dpol) ) -> List(Dpol)
  sortin: (Dpol, List(Dpol) ) -> List(Dpol)
  eRed: (Dpol, List(Dpol), List(Dpol) )  ->  Dpol
  ecredPol: (Dpol, List(Dpol) ) -> Dpol
  esPol: (critPair) -> Dpol
  updatD: (List(critPair), List(critPair)) -> List(critPair)
  lepol: Dpol -> Integer
  prinshINFO : Dpol -> Void
  prindINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
  prinpolINFO: List(Dpol) -> Void
```

```
prinb: Integer -> Void

------    MAIN ALGORITHM GROEBNER ------------------------
euclideanGroebner( Pol: List(Dpol) ) ==
  eminGbasis(strongGbasis(Pol,0,0))

euclideanGroebner( Pol: List(Dpol), xx1: String) ==
  xx1 = "redcrit" =>
    eminGbasis(strongGbasis(Pol,1,0))
  xx1 = "info" =>
    eminGbasis(strongGbasis(Pol,2,1))
  print("   "::Ex)
  print("WARNING: options are - redcrit and/or info - "::Ex)
  print("         you didn't type them correct"::Ex)
  print("         please try again"::Ex)
  print("   "::Ex)
  []

euclideanGroebner( Pol: List(Dpol), xx1: String, xx2: String) ==
  (xx1 = "redcrit" and xx2 = "info") or
   (xx1 = "info" and xx2 = "redcrit")   =>
    eminGbasis(strongGbasis(Pol,1,1))
  xx1 = "redcrit" and xx2 = "redcrit" =>
    eminGbasis(strongGbasis(Pol,1,0))
  xx1 = "info" and xx2 = "info" =>
    eminGbasis(strongGbasis(Pol,2,1))
  print("   "::Ex)
  print("WARNING:  options are - redcrit and/or info - "::Ex)
  print("          you didn't type them correct"::Ex)
  print("          please try again "::Ex)
  print("   "::Ex)
  []

------    calculate basis

strongGbasis(Pol: List(Dpol),xx1: Integer, xx2: Integer ) ==
  dd1, D : List(critPair)

  ---------   create D and Pol

  Pol1:= sort((z1:Dpol,z2:Dpol):Boolean +-> (degree z1 > degree z2) or
                ((degree z1 = degree z2 ) and
                   sizeLess?(leadingCoefficient z2,leadingCoefficient z1)),
              Pol)
  Pol:= [first(Pol1)]
  H:= Pol
  Pol1:= rest(Pol1)
  D:= nil
  while ^null Pol1 repeat
     h:= first(Pol1)
```

```
    Pol1:= rest(Pol1)
    en:= degree(h)
    lch:= lc h
    dd1:=
     [[sup(degree(x), en), lcm(leadingCoefficient x, lch), x, h]$critPair
        for x in Pol]
    D:= updatD(
         ecritMTondd1(
          sort(
           (z1:critPair,z2:critPair):Boolean+->
            (z1.lcmfij < z2.lcmfij) or
             (( z1.lcmfij = z2.lcmfij ) and
               ( sizeLess?(z1.lcmcij,z2.lcmcij)) ), dd1)),
                ecritBonD(h,D))
    Pol:= cons(h, eupdatF(h, Pol))
    ((en = degree(first(H))) and
      (leadingCoefficient(h) = leadingCoefficient(first(H)) ) ) =>
          " go to top of while "
    H:= updatH(h,H,crithdelH(h,H),[h])
    H:= sort((z1,z2) +-> (degree z1 > degree z2) or
              ((degree z1 = degree z2 ) and
               sizeLess?(leadingCoefficient z2,leadingCoefficient z1)), H)
D:= sort((z1,z2) +-> (z1.lcmfij < z2.lcmfij) or
          (( z1.lcmfij = z2.lcmfij ) and
           ( sizeLess?(z1.lcmcij,z2.lcmcij)) ) ,D)
xx:= xx2

--------  loop

while ^null D repeat
    D0:= first D
    ep:=esPol(D0)
    D:= rest(D)
    eh:= ecredPol(eRed(ep,H,H),H)
    if xx1 = 1 then
          prinshINFO(eh)
    eh = 0 =>
         if xx2 = 1 then
             ala:= prindINFO(D0,ep,eh,#H, #D, xx)
             xx:= 2
         " go to top of while "
    eh := unitCanonical eh
    e:= degree(eh)
    leh:= lc eh
    dd1:=
     [[sup(degree(x), e), lcm(leadingCoefficient x, leh), x, eh]$critPair
       for x in Pol]
    D:= updatD(
         ecritMTondd1(
          sort((z1,z2) +-> (z1.lcmfij < z2.lcmfij) or
```

```
                ((  z1.lcmfij = z2.lcmfij ) and
                 ( sizeLess?(z1.lcmcij,z2.lcmcij)) ), dd1)),
                   ecritBonD(eh,D))
      Pol:= cons(eh,eupdatF(eh,Pol))
      ^ecrithinH(eh,H) or
        ((e = degree(first(H))) and
          (leadingCoefficient(eh) = leadingCoefficient(first(H)) ) ) =>
           if xx2 = 1 then
               ala:= prindINFO(D0,ep,eh,#H, #D, xx)
               xx:= 2
           " go to top of while "
      H:= updatH(eh,H,crithdelH(eh,H),[eh])
      H:= sort((z1,z2)+-> (degree z1 > degree z2) or
          ((degree z1 = degree z2 ) and
             sizeLess?(leadingCoefficient z2,leadingCoefficient z1)), H)
      if xx2 = 1 then
        ala:= prindINFO(D0,ep,eh,#H, #D, xx)
        xx:= 2
        " go to top of while "
  if xx2 = 1 then
    prinpolINFO(Pol)
    print("    THE GROEBNER BASIS over EUCLIDEAN DOMAIN"::Ex)
  if xx1 = 1 and xx2 ^= 1 then
    print("    THE GROEBNER BASIS over EUCLIDEAN DOMAIN"::Ex)
  H

          ------------------------------------

          --- erase multiple of e in D2 using crit M

ecritMondd1(e: Expon, c: Dom, D2: List(critPair))==
   null D2 => nil
   x:= first(D2)
   ecritM(e,c, x.lcmfij, lcm(leadingCoefficient(x.poli),
          leadingCoefficient(x.polj)))
      => ecritMondd1(e, c, rest(D2))
   cons(x, ecritMondd1(e, c, rest(D2)))

         -------------------------------

ecredPol(h: Dpol, F: List(Dpol) ) ==
     h0:Dpol:= 0
     null F => h
     while h ^= 0 repeat
        h0:= h0 + monomial(leadingCoefficient(h),degree(h))
        h:= eRed(red(h), F, F)
     h0
          ---------------------------

          --- reduce dd1 using crit T and crit M
```

```
ecritMTondd1(dd1: List(critPair))==
        null dd1 => nil
        f1:= first(dd1)
        s1:= #(dd1)
        cT1:= ecritT(f1)
        s1= 1 and cT1 => nil
        s1= 1 => dd1
        e1:= f1.lcmfij
        r1:= rest(dd1)
        f2:= first(r1)
        e1 = f2.lcmfij and f1.lcmcij = f2.lcmcij =>
           cT1 =>   ecritMTondd1(cons(f1, rest(r1)))
           ecritMTondd1(r1)
        dd1 := ecritMondd1(e1, f1.lcmcij, r1)
        cT1 => ecritMTondd1(dd1)
        cons(f1, ecritMTondd1(dd1))

          ------------------------------

          --- erase elements in D fullfilling crit B

ecritBonD(h:Dpol, D: List(critPair))==
      null D => nil
      x:= first(D)
      x1:= x.poli
      x2:= x.polj
      ecritB(degree(h), leadingCoefficient(h),
             degree(x1),leadingCoefficient(x1),
             degree(x2),leadingCoefficient(x2)) =>
        ecritBonD(h, rest(D))
      cons(x, ecritBonD(h, rest(D)))

          ------------------------------

          --- concat F and h and erase multiples of h in F

eupdatF(h: Dpol, F: List(Dpol)) ==
    null F => nil
    f1:= first(F)
    ecritM(degree h,leadingCoefficient(h), degree f1,leadingCoefficient(f1))
        => eupdatF(h, rest(F))
    cons(f1, eupdatF(h, rest(F)))

          ------------------------------
          --- concat H and h and erase multiples of h in H

updatH(h: Dpol, H: List(Dpol), Hh: List(Dpol), Hhh: List(Dpol)) ==
    null H => append(Hh,Hhh)
    h1:= first(H)
```

```
    hlcm:= sup(degree(h1), degree(h))
    plc:= extendedEuclidean(leadingCoefficient(h), leadingCoefficient(h1))
    hp:= monomial(plc.coef1,subtractIfCan(hlcm, degree(h))::Expon)*h +
         monomial(plc.coef2,subtractIfCan(hlcm, degree(h1))::Expon)*h1
    (ecrithinH(hp, Hh) and ecrithinH(hp, Hhh)) =>
      hpp:= append(rest(H),Hh)
      hp:= ecredPol(eRed(hp,hpp,hpp),hpp)
      updatH(h, rest(H), crithdelH(hp,Hh),cons(hp,crithdelH(hp,Hhh)))
    updatH(h, rest(H), Hh,Hhh)

         ------------------------------------------------
         ---- delete elements in cons(h,H)

crithdelH(h: Dpol, H: List(Dpol))==
     null H => nil
     h1:= first(H)
     dh1:= degree h1
     dh:= degree h
     ecritM(dh, lc h, dh1, lc h1) => crithdelH(h, rest(H))
     dh1 = sup(dh,dh1) =>
      plc:= extendedEuclidean( lc h1, lc h)
      cons(plc.coef1*h1+monomial(plc.coef2,subtractIfCan(dh1,dh)::Expon)*h,
            crithdelH(h,rest(H)))
     cons(h1, crithdelH(h,rest(H)))

eminGbasis(F: List(Dpol)) ==
     null F => nil
     newbas := eminGbasis rest F
     cons(ecredPol( first(F), newbas),newbas)

         ------------------------------------------------
         --- does h belong to H

ecrithinH(h: Dpol, H: List(Dpol))==
     null H  => true
     h1:= first(H)
     ecritM(degree h1, lc h1, degree h, lc h) => false
     ecrithinH(h, rest(H))

        ----------------------------
        --- calculate  euclidean S-polynomial of a critical pair

esPol(p:critPair)==
   Tij := p.lcmfij
   fi := p.poli
   fj := p.polj
   lij:= lcm(leadingCoefficient(fi), leadingCoefficient(fj))
   red(fi)*monomial((lij exquo leadingCoefficient(fi))::Dom,
                    subtractIfCan(Tij, degree fi)::Expon) -
     red(fj)*monomial((lij exquo leadingCoefficient(fj))::Dom,
```

```
                    subtractIfCan(Tij, degree fj)::Expon)

        ----------------------------

        --- euclidean reduction mod F

eRed(s: Dpol, H: List(Dpol), Hh: List(Dpol)) ==
  ( s = 0 or null H ) => s
  f1:= first(H)
  ds:= degree s
  lf1:= leadingCoefficient(f1)
  ls:= leadingCoefficient(s)
  e: Union(Expon, "failed")
  (((e:= subtractIfCan(ds, degree f1))  case "failed" ) or sizeLess?(ls, lf1) ) =>
     eRed(s, rest(H), Hh)
  sdf1:= divide(ls, lf1)
  q1:= sdf1.quotient
  sdf1.remainder = 0 =>
     eRed(red(s) - monomial(q1,e)*reductum(f1), Hh, Hh)
  eRed(s -(monomial(q1, e)*f1), rest(H), Hh)

        ----------------------------

        --- crit T  true, if e1 and e2 are disjoint

ecritT(p: critPair) ==
       pi:= p.poli
       pj:= p.polj
       ci:= lc pi
       cj:= lc pj
       (p.lcmfij = degree pi + degree pj) and  (p.lcmcij = ci*cj)

        ----------------------------

        --- crit M - true, if lcm#2 multiple of lcm#1

ecritM(e1: Expon, c1: Dom, e2: Expon, c2: Dom) ==
  en: Union(Expon, "failed")
  ((en:=subtractIfCan(e2, e1)) case "failed") or
    ((c2 exquo c1) case "failed") => false
  true
        ----------------------------

        --- crit B - true, if eik is a multiple of eh and eik ^equal
        ---          lcm(eh,ei) and eik ^equal lcm(eh,ek)

ecritB(eh:Expon, ch: Dom, ei:Expon, ci: Dom, ek:Expon, ck: Dom) ==
    eik:= sup(ei, ek)
    cik:= lcm(ci, ck)
    ecritM(eh, ch, eik, cik) and
```

```
           ^ecritM(eik, cik, sup(ei, eh), lcm(ci, ch)) and
              ^ecritM(eik, cik, sup(ek, eh), lcm(ck, ch))

         ------------------------------

         --- reduce p1 mod lp

euclideanNormalForm(p1: Dpol, lp: List(Dpol))==
    eRed(p1, lp, lp)

         --------------------------------

         ---  insert element in sorted list

sortin(p1: Dpol, lp: List(Dpol))==
   null lp => [p1]
   f1:= first(lp)
   elf1:= degree(f1)
   ep1:= degree(p1)
   ((elf1 < ep1) or ((elf1 = ep1) and
     sizeLess?(leadingCoefficient(f1),leadingCoefficient(p1)))) =>
      cons(f1,sortin(p1, rest(lp)))
   cons(p1,lp)

updatD(D1: List(critPair), D2: List(critPair)) ==
   null D1 => D2
   null D2 => D1
   dl1:= first(D1)
   dl2:= first(D2)
   (dl1.lcmfij  <  dl2.lcmfij) => cons(dl1, updatD(D1.rest, D2))
   cons(dl2, updatD(D1, D2.rest))

         ----  calculate number of terms of polynomial

lepol(p1:Dpol)==
   n: Integer
   n:= 0
   while p1 ^= 0 repeat
      n:= n + 1
      p1:= red(p1)
   n

         ----  print blanc lines

prinb(n: Integer)==
     for i in 1..n repeat messagePrint("    ")

         ----  print reduced critpair polynom

prinshINFO(h: Dpol)==
```

```
          prinb(2)
          messagePrint(" reduced Critpair - Polynom :")
          prinb(2)
          print(h::Ex)
          prinb(2)

           -------------------------------

           ----  print info string

  prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer,
            i2:Integer, n:Integer) ==
      ll: List Prinp
      a: Dom
      cpi:= cp.poli
      cpj:= cp.polj
      if n = 1 then
       prinb(1)
       messagePrint("you choose option  -info-  ")
       messagePrint("abbrev. for the following information strings are")
       messagePrint("  ci  =>  Leading monomial  for critpair calculation")
       messagePrint("  tci =>  Number of terms of polynomial i")
       messagePrint("  cj  =>  Leading monomial  for critpair calculation")
       messagePrint("  tcj =>  Number of terms of polynomial j")
       messagePrint("  c   =>  Leading monomial of critpair polynomial")
       messagePrint("  tc  =>  Number of terms of critpair polynomial")
       messagePrint("  rc  =>  Leading monomial of redcritpair polynomial")
       messagePrint("  trc =>  Number of terms of redcritpair polynomial")
       messagePrint("  tF  =>  Number of polynomials in reduction list F")
       messagePrint("  tD  =>  Number of critpairs still to do")
       prinb(4)
       n:= 2
      prinb(1)
      a:= 1
      ph = 0  =>
         ps = 0 =>
           ll:= [[monomial(a,degree(cpi)),lepol(cpi),monomial(a,degree(cpj)),
            lepol(cpj),ps,0,ph,0,i1,i2]$Prinp]
           print(ll::Ex)
           prinb(1)
           n
         ll:= [[monomial(a,degree(cpi)),lepol(cpi),
           monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
            lepol(ps), ph,0,i1,i2]$Prinp]
         print(ll::Ex)
         prinb(1)
         n
      ll:= [[monomial(a,degree(cpi)),lepol(cpi),
           monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
            lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2]$Prinp]
```

```
      print(ll::Ex)
      prinb(1)
      n

           ------------------------------

           ----  print the groebner basis polynomials

  prinpolINFO(pl: List(Dpol))==
      n:Integer
      n:= #pl
      prinb(1)
      n = 1 =>
        print("  There is 1  Groebner Basis Polynomial "::Ex)
        prinb(2)
      print("  There are "::Ex)
      prinb(1)
      print(n::Ex)
      prinb(1)
      print("  Groebner Basis Polynomials. "::Ex)
      prinb(2)
```

———-

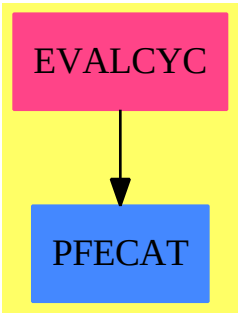
— GBEUCLID.dotabb —

```
"GBEUCLID" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GBEUCLID"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"GBEUCLID" -> "PFECAT"
"GBEUCLID" -> "STRING"
```

———-

6.17 package EVALCYC EvaluateCycleIndicators

6.17.1 EvaluateCycleIndicators (EVALCYC)



Exports:
eval

— package EVALCYC EvaluateCycleIndicators —

```
)abbrev package EVALCYC EvaluateCycleIndicators
++ Author: William H. Burge
++ Date Created: 1986
++ Date Last Updated: Feb 1992
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package is to be used in conjuction with the CycleIndicators package.
++ It provides an evaluation function for SymmetricPolynomials.

EvaluateCycleIndicators(F):T==C where
    F:Algebra Fraction Integer
    I==>Integer
    L==>List
    SPOL==SymmetricPolynomial
    RN==>Fraction Integer
    PR==>Polynomial(RN)
    PTN==>Partition()
    lc ==> leadingCoefficient
    red ==> reductum
    T== with
       eval:((I->F),SPOL RN)->F
```

```
        ++\spad{eval(f,s)} evaluates the cycle index s by applying
        ++ the function f to each integer in a monomial partition,
        ++ forms their product and sums the results over all monomials.
    C== add
       evp:((I->F),PTN)->F
       fn:I->F
       pt:PTN
       spol:SPOL RN
       i:I
       evp(fn, pt)== _*/[fn i for i in pt::(L I)]

       eval(fn,spol)==
        if spol=0
        then 0
        else ((lc spol)* evp(fn,degree spol)) + eval(fn,red spol)
```

———-

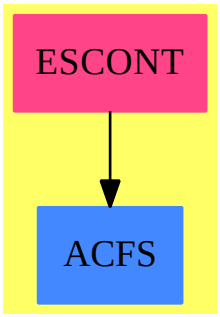
— **EVALCYC.dotabb** —

```
"EVALCYC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EVALCYC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"EVALCYC" -> "PFECAT"
```

———-

6.18 package ESCONT ExpertSystemContinuityPackage

6.18.1 ExpertSystemContinuityPackage (ESCONT)



Exports:

df2st	functionIsFracPolynomial?	gethi	getlo	ldf2lst
polynomialZeros	problemPoints	sdf2lst	singularitiesOf	singularitiesOf
zerosOf				

— package ESCONT ExpertSystemContinuityPackage —

```
)abbrev package ESCONT ExpertSystemContinuityPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
++ Basic Operations: problemPoints, singularitiesOf, zerosOf
++ Related Constructors:
++ Description:
++ ExpertSystemContinuityPackage is a package of functions for the use of
++ domains belonging to the category \axiomType{NumericalIntegration}.

ExpertSystemContinuityPackage(): E == I where
  EF2   ==> ExpressionFunctions2
  FI    ==> Fraction Integer
  EFI   ==> Expression Fraction Integer
  PFI   ==> Polynomial Fraction Integer
  DF    ==> DoubleFloat
  LDF   ==> List DoubleFloat
  EDF   ==> Expression DoubleFloat
  VEDF  ==> Vector Expression DoubleFloat
  SDF   ==> Stream DoubleFloat
  SS    ==> Stream String
  EEDF  ==> Equation Expression DoubleFloat
  LEDF  ==> List Expression DoubleFloat
  KEDF  ==> Kernel Expression DoubleFloat
  LKEDF ==> List Kernel Expression DoubleFloat
  PDF   ==> Polynomial DoubleFloat
  FPDF  ==> Fraction Polynomial DoubleFloat
  OCDF  ==> OrderedCompletion DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA   ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  UP    ==> UnivariatePolynomial
  BO    ==> BasicOperator
  RS    ==> Record(zeros: SDF,ones: SDF,singularities: SDF)

  E ==> with

    getlo : SOCDF -> DF
      ++ getlo(u) gets the \axiomType{DoubleFloat} equivalent of
      ++ the first endpoint of the range \axiom{u}
    gethi : SOCDF -> DF
      ++ gethi(u) gets the \axiomType{DoubleFloat} equivalent of
      ++ the second endpoint of the range \axiom{u}
    functionIsFracPolynomial?: NIA -> Boolean
      ++ functionIsFracPolynomial?(args) tests whether the function
```

```
    ++ can be retracted to \axiomType{Fraction(Polynomial(DoubleFloat))}
  problemPoints:(EDF,Symbol,SOCDF) -> List DF
    ++ problemPoints(f,var,range) returns a list of possible problem points
    ++ by looking at the zeros of the denominator of the function \spad{f}
    ++ if it can be retracted to \axiomType{Polynomial(DoubleFloat)}.
  zerosOf:(EDF,List Symbol,SOCDF) -> SDF
    ++ zerosOf(e,vars,range) returns a list of points
    ++ (\axiomType{Doublefloat}) at which a NAG fortran version of \spad{e}
    ++ will most likely produce an error.
  singularitiesOf: (EDF,List Symbol,SOCDF) -> SDF
    ++ singularitiesOf(e,vars,range) returns a list of points
    ++ (\axiomType{Doublefloat}) at which a NAG fortran
    ++ version of \spad{e} will most likely produce
    ++ an error.  This includes those points which evaluate to 0/0.
  singularitiesOf: (Vector EDF,List Symbol,SOCDF) -> SDF
    ++ singularitiesOf(v,vars,range) returns a list of points
    ++ (\axiomType{Doublefloat}) at which a NAG fortran
    ++ version of \spad{v} will most likely produce
    ++ an error.  This includes those points which evaluate to 0/0.
  polynomialZeros:(PFI,Symbol,SOCDF) -> LDF
    ++ polynomialZeros(fn,var,range) calculates the real zeros of the
    ++ polynomial which are contained in the given interval. It returns
    ++ a list of points (\axiomType{Doublefloat}) for which the univariate
    ++ polynomial \spad{fn} is zero.
  df2st:DF -> String
    ++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
  ldf2lst:LDF -> List String
    ++ ldf2lst(ln) coerces a List of \axiomType{DoubleFloat} to
    ++ \axiomType{List}(\axiomType{String})
  sdf2lst:SDF -> List String
    ++ sdf2lst(ln) coerces a Stream of \axiomType{DoubleFloat} to
    ++ \axiomType{List}(\axiomType{String})

I ==> ExpertSystemToolsPackage add

  import ExpertSystemToolsPackage

  functionIsPolynomial?(args:NIA):Boolean ==
    -- tests whether the function can be retracted to a polynomial
    (retractIfCan(args.fn)@Union(PDF,"failed"))$EDF case PDF

  isPolynomial?(f:EDF):Boolean ==
    -- tests whether the function can be retracted to a polynomial
    (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

  isConstant?(f:EDF):Boolean ==
    -- tests whether the function can be retracted to a constant (DoubleFloat)
    (retractIfCan(f)@Union(DF,"failed"))$EDF case DF

  denominatorIsPolynomial?(args:NIA):Boolean ==
```

```
    -- tests if the denominator can be retracted to polynomial
    a:= copy args
    a.fn:=denominator(args.fn)
    (functionIsPolynomial?(a))@Boolean

  denIsPolynomial?(f:EDF):Boolean ==
    -- tests if the denominator can be retracted to polynomial
    (isPolynomial?(denominator f))@Boolean

  listInRange(l:LDF,range:SOCDF):LDF ==
    -- returns a list with only those elements internal to the range range
    [t for t in l | in?(t,range)]

  loseUntil(l:SDF,a:DF):SDF ==
    empty?(l)$SDF => l
    f := first(l)$SDF
    (abs(f) <= abs(a)) => loseUntil(rest(l)$SDF,a)
    l

  retainUntil(l:SDF,a:DF,b:DF,flag:Boolean):SDF ==
    empty?(l)$SDF => l
    f := first(l)$SDF
    (in?(f)$ExpertSystemContinuityPackage1(a,b)) =>
      concat(f,retainUntil(rest(l),a,b,false))
    flag => empty()$SDF
    retainUntil(rest(l),a,b,true)

  streamInRange(l:SDF,range:SOCDF):SDF ==
    -- returns a stream with only those elements internal to the range range
    a := getlo(range := dfRange(range))
    b := gethi(range)
    explicitlyFinite?(l) =>
      select(in?$ExpertSystemContinuityPackage1(a,b),l)$SDF
    negative?(a*b) => retainUntil(l,a,b,false)
    negative?(a) =>
      l := loseUntil(l,b)
      retainUntil(l,a,b,false)
    l := loseUntil(l,a)
    retainUntil(l,a,b,false)

  getStream(n:Symbol,s:String):SDF ==
    import RS
    entry?(n,bfKeys()$BasicFunctions)$(List(Symbol)) =>
      c := bfEntry(n)$BasicFunctions
      (s = "zeros")@Boolean => c.zeros
      (s = "singularities")@Boolean => c.singularities
      (s = "ones")@Boolean => c.ones
    empty()$SDF

  polynomialZeros(fn:PFI,var:Symbol,range:SOCDF):LDF ==
```

```
      up := unmakeSUP(univariate(fn)$PFI)$UP(var,FI)
      range := dfRange(range)
      r:Record(left:FI,right:FI) := [df2fi(getlo(range)), df2fi(gethi(range))]
      ans:List(Record(left:FI,right:FI)) :=
          realZeros(up,r,1/1000000000000000000)$RealZeroPackageQ(UP(var,FI))
      listInRange(dflist(ans),range)

    functionIsFracPolynomial?(args:NIA):Boolean ==
      -- tests whether the function can be retracted to a fraction
      -- where both numerator and denominator are polynomial
      (retractIfCan(args.fn)@Union(FPDF,"failed"))$EDF case FPDF

    problemPoints(f:EDF,var:Symbol,range:SOCDF):LDF ==
      (denIsPolynomial?(f))@Boolean =>
        c := retract(edf2efi(denominator(f)))@PFI
        polynomialZeros(c,var,range)
      empty()$LDF

    zerosOf(e:EDF,vars:List Symbol,range:SOCDF):SDF ==
      (u := isQuotient(e)) case EDF =>
        singularitiesOf(u,vars,range)
      k := kernels(e)$EDF
      ((nk := # k) = 0)@Boolean => empty()$SDF -- constant found.
      (nk = 1)@Boolean =>                      -- single expression found.
        ker := first(k)$LKEDF
        n := name(operator(ker)$KEDF)$BO
        entry?(n,vars) =>                  -- polynomial found.
          c := retract(edf2efi(e))@PFI
          coerce(polynomialZeros(c,n,range))$SDF
        a := first(argument(ker)$KEDF)$LEDF
        (not (n = log :: Symbol)@Boolean) and ((w := isPlus a) case LEDF) =>
          var:Symbol := first(variables(a))
          c:EDF := w.2
          c1:EDF := w.1
--          entry?(c1,[b::EDF for b in vars]) and (one?(# vars)) =>
          entry?(c1,[b::EDF for b in vars]) and ((# vars) = 1) =>
            c2:DF := edf2df c
            c3 := c2 :: OCDF
            varEdf := var :: EDF
            varEqn := equation(varEdf,c1-c)$EEDF
            range2 := (lo(range)+c3)..(hi(range)+c3)
            s := zerosOf(subst(e,varEqn)$EDF,vars,range2)
            st := map(t1 +-> t1-c2,s)$StreamFunctions2(DF,DF)
            streamInRange(st,range)
          zerosOf(a,vars,range)
        (t := isPlus(e)$EDF) case LEDF =>    -- constant + expression
          # t > 2 => empty()$SDF
          entry?(a,[b::EDF for b in vars]) =>   -- finds entries like sqrt(x)
            st := getStream(n,"ones")
            o := edf2df(second(t)$LEDF)
```

```
--            one?(o) or one?(-o) =>           -- is it like (f(x) -/+ 1)
             (o = 1) or (-o = 1) =>           -- is it like (f(x) -/+ 1)
               st := map(t2 +-> -t2/o,st)$StreamFunctions2(DF,DF)
               streamInRange(st,range)
             empty()$SDF
           empty()$SDF
         entry?(a,[b::EDF for b in vars]) =>     -- finds entries like sqrt(x)
           st := getStream(n,"zeros")
           streamInRange(st,range)
         (n = tan :: Symbol)@Boolean =>
           concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
         (n = sin :: Symbol)@Boolean =>
           concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
         empty()$SDF
       (t := isPlus(e)$EDF) case LEDF => empty()$SDF  -- INCOMPLETE!!!
       (v := isTimes(e)$EDF) case LEDF =>
         concat([zerosOf(u,vars,range) for u in v])
       empty()$SDF

    singularitiesOf(e:EDF,vars:List Symbol,range:SOCDF):SDF ==
       (u := isQuotient(e)) case EDF =>
         zerosOf(u,vars,range)
       (t := isPlus e) case LEDF =>
         concat([singularitiesOf(u,vars,range) for u in t])
       (v := isTimes e) case LEDF =>
         concat([singularitiesOf(u,vars,range) for u in v])
       (k := mainKernel e) case KEDF =>
         n := name(operator k)
         entry?(n,vars) => coerce(problemPoints(e,n,range))$SDF
         a:EDF := (argument k).1
         (not (n = log :: Symbol)@Boolean) and ((w := isPlus a) case LEDF) =>
           var:Symbol := first(variables(a))
           c:EDF := w.2
           c1:EDF := w.1
--           entry?(c1,[b::EDF for b in vars]) and (one?(# vars)) =>
           entry?(c1,[b::EDF for b in vars]) and ((# vars) = 1) =>
             c2:DF := edf2df c
             c3 := c2 :: OCDF
             varEdf := var :: EDF
             varEqn := equation(varEdf,c1-c)$EEDF
             range2 := (lo(range)+c3)..(hi(range)+c3)
             s := singularitiesOf(subst(e,varEqn)$EDF,vars,range2)
             st := map(t3 +-> t3-c2,s)$StreamFunctions2(DF,DF)
             streamInRange(st,range)
           singularitiesOf(a,vars,range)
         entry?(a,[b::EDF for b in vars]) =>
           st := getStream(n,"singularities")
           streamInRange(st,range)
         (n = log :: Symbol)@Boolean =>
           concat([zerosOf(a,vars,range),singularitiesOf(a,vars,range)])
```

```
        singularitiesOf(a,vars,range)
      empty()$SDF

    singularitiesOf(v:VEDF,vars:List Symbol,range:SOCDF):SDF ==
      ls := [singularitiesOf(u,vars,range) for u in entries(v)$VEDF]
      concat(ls)$SDF
```

———-

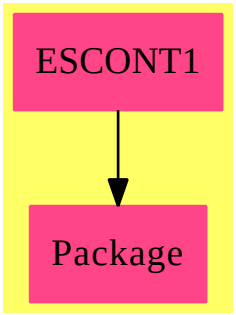
— **ESCONT.dotabb** —

```
"ESCONT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESCONT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ESCONT" -> "ACFS"
```

———-

6.19 package ESCONT1 ExpertSystemContinuityPackage1

6.19.1 ExpertSystemContinuityPackage1 (ESCONT1)



Exports:
in?

— **package ESCONT1 ExpertSystemContinuityPackage1** —

```
)abbrev package ESCONT1 ExpertSystemContinuityPackage1
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
```

```
++ Basic Operations: problemPoints, singularitiesOf, zerosOf
++ Related Constructors:
++ Description:
++ ExpertSystemContinuityPackage1 exports a function to check range inclusion

ExpertSystemContinuityPackage1(A:DF,B:DF): E == I where
  EF2   ==> ExpressionFunctions2
  FI    ==> Fraction Integer
  EFI   ==> Expression Fraction Integer
  PFI   ==> Polynomial Fraction Integer
  DF    ==> DoubleFloat
  LDF   ==> List DoubleFloat
  EDF   ==> Expression DoubleFloat
  VEDF  ==> Vector Expression DoubleFloat
  SDF   ==> Stream DoubleFloat
  SS    ==> Stream String
  EEDF  ==> Equation Expression DoubleFloat
  LEDF  ==> List Expression DoubleFloat
  KEDF  ==> Kernel Expression DoubleFloat
  LKEDF ==> List Kernel Expression DoubleFloat
  PDF   ==> Polynomial DoubleFloat
  FPDF  ==> Fraction Polynomial DoubleFloat
  OCDF  ==> OrderedCompletion DoubleFloat
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  NIA   ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  UP    ==> UnivariatePolynomial
  BO    ==> BasicOperator
  RS    ==> Record(zeros: SDF,ones: SDF,singularities: SDF)

  E ==> with

    in?:DF -> Boolean
      ++ in?(p) tests whether point p is internal to the range [\spad{A..B}]

  I ==> add

    in?(p:DF):Boolean ==
      a:Boolean := (p < B)$DF
      b:Boolean := (A < p)$DF
      (a and b)@Boolean
```

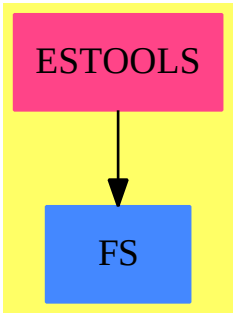
———-

— ESCONT1.dotabb —

```
"ESCONT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESCONT1"]
"Package" [color="#FF4488"]
"ESCONT1" -> "Package"
```

6.20 package ESTOOLS ExpertSystemToolsPackage

6.20.1 ExpertSystemToolsPackage (ESTOOLS)



Exports:

att2Result	concat	convert	dflist	dfRange
df2ef	df2fi	df2mf	df2st	edf2df
edf2ef	edf2efi	edf2fi	ef2edf	expenseOfEvaluation
fi2df	f2df	f2st	gethi	getlo
iflist2Result	in?	isQuotient	ldf2lst	ldf2vmf
mat	measure2Result	measure2Result	numberOfOperations	ocf2ocdf
outputMeasure	pdf2df	pdf2ef	sdf2lst	socf2socdf
vedf2vef				

— package ESTOOLS ExpertSystemToolsPackage —

```
)abbrev package ESTOOLS ExpertSystemToolsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: July 1996
++ Basic Operations:
++ Description:
++ \axiom{ExpertSystemToolsPackage} contains some useful functions for use
++ by the computational agents of numerical solvers.

ExpertSystemToolsPackage():E == I where
  LEDF  ==> List Expression DoubleFloat
  KEDF  ==> Kernel Expression DoubleFloat
  LKEDF  ==> List Kernel Expression DoubleFloat
  VEDF  ==> Vector Expression DoubleFloat
  VEF  ==> Vector Expression Float
```

```
VMF  ==> Vector MachineFloat
EF2  ==> ExpressionFunctions2
EFI  ==> Expression Fraction Integer
MDF  ==> Matrix DoubleFloat
LDF  ==> List DoubleFloat
PDF  ==> Polynomial DoubleFloat
EDF  ==> Expression DoubleFloat
EF  ==> Expression Float
SDF  ==> Stream DoubleFloat
DF  ==> DoubleFloat
F  ==> Float
MF  ==> MachineFloat
INT  ==> Integer
NNI  ==> NonNegativeInteger
LS  ==> List Symbol
ST  ==> String
LST  ==> List String
SS  ==> Stream String
FI  ==> Fraction Integer
R  ==> Ring
OR  ==> OrderedRing
ON  ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
RVE   ==> Record(val:EDF,exponent:INT)
BO  ==> BasicOperator
OCF  ==> OrderedCompletion Float
OCDF  ==> OrderedCompletion DoubleFloat
SOCF  ==> Segment OrderedCompletion Float
SOCDF  ==> Segment OrderedCompletion DoubleFloat
Measure  ==> Record(measure:F, name:String, explanations:List String)
Measure2  ==> Record(measure:F, name:String, explanations:List String, extra:Result)
CTYPE  ==> Union(continuous: "Continuous at the end points",
             lowerSingular: "There is a singularity at the lower end point",
              upperSingular: "There is a singularity at the upper end point",
               bothSingular: "There are singularities at both end points",
                notEvaluated: "End point continuity not yet evaluated")
RTYPE  ==> Union(finite: "The range is finite",
              lowerInfinite: "The bottom of range is infinite",
                upperInfinite: "The top of range is infinite",
                  bothInfinite: "Both top and bottom points are infinite",
                    notEvaluated: "Range not yet evaluated")
STYPE  ==> Union(str:SDF,
                  notEvaluated:"Internal singularities not yet evaluated")
ATT  ==> Record(endPointContinuity:CTYPE,singularitiesStream:STYPE,range:RTYPE)
IFV  ==> Record(stiffness:F,stability:F,expense:F,accuracy:F,intermediateResults:F)

E ==> with

  f2df:F -> DF
    ++ f2df(f) is a function to convert a \axiomType{Float} to a
    ++ \axiomType{DoubleFloat}
```

```
ef2edf:EF -> EDF
  ++ ef2edf(f) is a function to convert an \axiomType{Expression Float}
  ++ to an \axiomType{Expression DoubleFloat}
ocf2ocdf: OCF -> OCDF
  ++ ocf2ocdf(a) is a function to convert an \axiomType{OrderedCompletion
  ++ Float} to an \axiomType{OrderedCompletion DoubleFloat}
socf2socdf: SOCF -> SOCDF
  ++ socf2socdf(a) is a function to convert a \axiomType{Segment OrderedCompletion Float}
  ++ to a \axiomType{Segment OrderedCompletion DoubleFloat}
convert: List SOCF -> List SOCDF
  ++ convert(l) is a function to convert a \axiomType{Segment OrderedCompletion Float}
  ++ to a \axiomType{Segment OrderedCompletion DoubleFloat}
df2fi :DF -> FI
  ++ df2fi(n) is a function to convert a \axiomType{DoubleFloat} to a
  ++ \axiomType{Fraction Integer}
edf2fi :EDF -> FI
  ++ edf2fi(n) maps \axiomType{Expression DoubleFloat} to
  ++ \axiomType{Fraction Integer}
  ++ It is an error if n is not coercible to Fraction Integer
edf2df :EDF -> DF
  ++ edf2df(n) maps \axiomType{Expression DoubleFloat} to
  ++ \axiomType{DoubleFloat}
  ++ It is an error if \spad{n} is not coercible to DoubleFloat
isQuotient:EDF -> Union(EDF,"failed")
  ++ isQuotient(expr) returns the quotient part of the input
  ++ expression or \spad{"failed"} if the expression is not of that form.
expenseOfEvaluation:VEDF -> F
  ++ expenseOfEvaluation(o) gives an approximation of the cost of
  ++ evaluating a list of expressions in terms of the number of basic
  ++ operations.
  ++ < 0.3 inexpensive ; 0.5 neutral ; > 0.7 very expensive
  ++ 400 'operation units' -> 0.75
  ++ 200 'operation units' -> 0.5
  ++ 83 'operation units' -> 0.25
  ++ ** = 4 units , function calls = 10 units.
numberOfOperations:VEDF -> ON
  ++ numberOfOperations(ode) counts additions, multiplications,
  ++ exponentiations and function calls in the input set of expressions.
edf2efi:EDF -> EFI
  ++ edf2efi(e) coerces \axiomType{Expression DoubleFloat} into
  ++ \axiomType{Expression Fraction Integer}
dfRange:SOCDF -> SOCDF
  ++ dfRange(r) converts a range including
  ++ \inputbitmap{\htbmdir{}/plusminus.bitmap} \infty
  ++ to \axiomType{DoubleFloat} equavalents.
dflist:List(Record(left:FI,right:FI)) -> LDF
  ++ dflist(l) returns a list of \axiomType{DoubleFloat} equivalents of list l
df2mf:DF -> MF
  ++ df2mf(n) coerces a \axiomType{DoubleFloat} to \axiomType{MachineFloat}
ldf2vmf:LDF -> VMF
```

```
  ++ ldf2vmf(l) coerces a \axiomType{List DoubleFloat} to
  ++ \axiomType{List MachineFloat}
edf2ef:EDF -> EF
  ++ edf2ef(e) maps \axiomType{Expression DoubleFloat} to
  ++ \axiomType{Expression Float}
vedf2vef:VEDF -> VEF
  ++ vedf2vef(v) maps \axiomType{Vector Expression DoubleFloat} to
  ++ \axiomType{Vector Expression Float}
in?:(DF,SOCDF) -> Boolean
  ++ in?(p,range) tests whether point p is internal to the
  ++ range range
df2st:DF -> ST
  ++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
f2st:F -> ST
  ++ f2st(n) coerces a \axiomType{Float} to \axiomType{String}
ldf2lst:LDF -> LST
  ++ ldf2lst(ln) coerces a \axiomType{List DoubleFloat} to \axiomType{List String}
sdf2lst:SDF -> LST
  ++ sdf2lst(ln) coerces a \axiomType{Stream DoubleFloat} to \axiomType{String}
getlo : SOCDF -> DF
  ++ getlo(u) gets the \axiomType{DoubleFloat} equivalent of
  ++ the first endpoint of the range \spad{u}
gethi : SOCDF -> DF
  ++ gethi(u) gets the \axiomType{DoubleFloat} equivalent of
  ++ the second endpoint of the range \spad{u}
concat:(Result,Result) -> Result
  ++ concat(a,b) adds two aggregates of type \axiomType{Result}.
concat:(List Result) -> Result
  ++ concat(l) concatenates a list of aggregates of type \axiomType{Result}
outputMeasure:F -> ST
  ++ outputMeasure(n) rounds \spad{n} to 3 decimal places and outputs
  ++ it as a string
measure2Result:Measure -> Result
  ++ measure2Result(m) converts a measure record into a \axiomType{Result}
measure2Result:Measure2 -> Result
  ++ measure2Result(m) converts a measure record into a \axiomType{Result}
att2Result:ATT -> Result
  ++ att2Result(m) converts a attributes record into a \axiomType{Result}
iflist2Result:IFV -> Result
  ++ iflist2Result(m) converts a attributes record into a \axiomType{Result}
pdf2ef:PDF -> EF
  ++ pdf2ef(p) coerces a \axiomType{Polynomial DoubleFloat} to
  ++ \axiomType{Expression Float}
pdf2df:PDF -> DF
  ++ pdf2df(p) coerces a \axiomType{Polynomial DoubleFloat} to
  ++ \axiomType{DoubleFloat}. It is an error if \axiom{p} is not
  ++ retractable to DoubleFloat.
df2ef:DF -> EF
  ++ df2ef(a) coerces a \axiomType{DoubleFloat} to \axiomType{Expression Float}
fi2df:FI -> DF
```

```
    ++ fi2df(f) coerces a \axiomType{Fraction Integer} to \axiomType{DoubleFloat}
  mat:(LDF,NNI) -> MDF
    ++ mat(a,n) constructs a one-dimensional matrix of a.

I ==> add

  mat(a:LDF,n:NNI):MDF ==
    empty?(a)$LDF => zero(1,n)$MDF
    matrix(list([i for i in a for j in 1..n])$(List LDF))$MDF

  f2df(f:F):DF == (convert(f)@DF)$F

  ef2edf(f:EF):EDF == map(f2df,f)$EF2(F,DF)

  fi2df(f:FI):DF == coerce(f)$DF

  ocf2ocdf(a:OCF):OCDF ==
    finite? a => (f2df(retract(a)@F))::OCDF
    a pretend OCDF

  socf2socdf(a:SOCF):SOCDF ==
    segment(ocf2ocdf(lo a),ocf2ocdf(hi a))

  convert(l:List SOCF):List SOCDF == [socf2socdf a for a in l]

  pdf2df(p:PDF):DF == retract(p)@DF

  df2ef(a:DF):EF ==
    b := convert(a)@Float
    coerce(b)$EF

  pdf2ef(p:PDF):EF == df2ef(pdf2df(p))

  edf2fi(m:EDF):FI == retract(retract(m)@DF)@FI

  edf2df(m:EDF):DF == retract(m)@DF

  df2fi(r:DF):FI == (retract(r)@FI)$DF

  dfRange(r:SOCDF):SOCDF ==
    if infinite?(lo(r))$OCDF then r := -(max()$DF :: OCDF)..hi(r)$SOCDF
    if infinite?(hi(r))$OCDF then r := lo(r)$SOCDF..(max()$DF :: OCDF)
    r

  dflist(l:List(Record(left:FI,right:FI))):LDF == [u.left :: DF for u in l]

  edf2efi(f:EDF):EFI == map(df2fi,f)$EF2(DF,FI)

  df2st(n:DF):String == (convert((convert(n)@Float)$DF)@ST)$Float
```

```
    f2st(n:F):String == (convert(n)@ST)$Float

    ldf2lst(ln:LDF):LST == [df2st f for f in ln]

    sdf2lst(ln:SDF):LST ==
      explicitlyFinite? ln =>
        m := map(df2st,ln)$StreamFunctions2(DF,ST)
        if index?(20,m)$SS then
          split!(m,20)
          m := concat(m,".......")
        m := complete(m)$SS
        entries(m)$SS
      empty()$LST

    df2mf(n:DF):MF == (df2fi(n))::MF

    ldf2vmf(l:LDF):VMF ==
      m := [df2mf(n) for n in l]
      vector(m)$VMF

    edf2ef(e:EDF):EF == map(convert$DF,e)$EF2(DF,Float)

    vedf2vef(vedf:VEDF):VEF == vector([edf2ef e for e in members(vedf)])

    getlo(u:SOCDF):DF == retract(lo(u))@DF

    gethi(u:SOCDF):DF == retract(hi(u))@DF

    in?(p:DF,range:SOCDF):Boolean ==
      top := gethi(range)
      bottom := getlo(range)
      a:Boolean := (p < top)$DF
      b:Boolean := (p > bottom)$DF
      (a and b)@Boolean

    isQuotient(expr:EDF):Union(EDF,"failed") ==
      (k := mainKernel expr) case KEDF =>
        (expr = inv(f := k :: KEDF :: EDF)$EDF)$EDF => f
--        one?(numerator expr) => denominator expr
        (numerator expr) = 1 => denominator expr
        "failed"
      "failed"

    numberOfOperations1(fn:EDF,numbersSoFar:ON):ON ==
      (u := isQuotient(fn)) case EDF =>
        numbersSoFar := numberOfOperations1(u,numbersSoFar)
      (p := isPlus(fn)) case LEDF =>
        p := coerce(p)@LEDF
        np := #p
        numbersSoFar.additions := (numbersSoFar.additions)+np-1
```

```
    for i in 1..np repeat
      numbersSoFar := numberOfOperations1(p.i,numbersSoFar)
    numbersSoFar
  (t:=isTimes(fn)) case LEDF =>
    t := coerce(t)@LEDF
    nt := #t
    numbersSoFar.multiplications := (numbersSoFar.multiplications)+nt-1
    for i in 1..nt repeat
      numbersSoFar := numberOfOperations1(t.i,numbersSoFar)
    numbersSoFar
  if (e:=isPower(fn)) case RVE then
    e := coerce(e)@RVE
    e.exponent>1 =>
      numbersSoFar.exponentiations := inc(numbersSoFar.exponentiations)
      numbersSoFar := numberOfOperations1(e.val,numbersSoFar)
  lk := kernels(fn)
  #lk = 1 =>        -- #lk = 0 => constant found (no further action)
    k := first(lk)$LKEDF
    n := name(operator(k)$KEDF)$BO
    entry?(n,variables(fn)$EDF)$LS => numbersSoFar  -- solo variable found
    a := first(argument(k)$KEDF)$LEDF
    numbersSoFar.functionCalls := inc(numbersSoFar.functionCalls)$INT
    numbersSoFar := numberOfOperations1(a,numbersSoFar)
  numbersSoFar

numberOfOperations(ode:VEDF):ON ==
  n:ON := [0,0,0,0]
  for i in 1..#ode repeat
    n:ON := numberOfOperations1(ode.i,n)
  n

expenseOfEvaluation(o:VEDF):F ==
  ln:ON := numberOfOperations(o)
  a := ln.additions
  m := ln.multiplications
  e := ln.exponentiations
  f := 10*ln.functionCalls
  n := (a + m + 4*e + 10*e)
  (1.0-exp((-n::F/288.0))$F)

concat(a:Result,b:Result):Result ==
  membersOfa := (members(a)@List(Record(key:Symbol,entry:Any)))
  membersOfb := (members(b)@List(Record(key:Symbol,entry:Any)))
  allMembers:=
    concat(membersOfa,membersOfb)$List(Record(key:Symbol,entry:Any))
  construct(allMembers)

concat(l:List Result):Result ==
  import List Result
  empty? l => empty()$Result
```

```
    f := first l
    if empty?(r := rest l) then
      f
    else
      concat(f,concat r)

  outputMeasure(m:F):ST ==
    fl:Float := round(m*(f:= 1000.0))/f
    convert(fl)@ST

  measure2Result(m:Measure):Result ==
    mm := coerce(m.measure)$AnyFunctions1(Float)
    mmr:Record(key:Symbol,entry:Any) := [bestMeasure@Symbol,mm]
    mn := coerce(m.name)$AnyFunctions1(ST)
    mnr:Record(key:Symbol,entry:Any) := [nameOfRoutine@Symbol,mn]
    me := coerce(m.explanations)$AnyFunctions1(List String)
    mer:Record(key:Symbol,entry:Any) := [allMeasures@Symbol,me]
    mr := construct([mmr,mnr,mer])$Result
    met := coerce(mr)$AnyFunctions1(Result)
    meth:Record(key:Symbol,entry:Any):=[method@Symbol,met]
    construct([meth])$Result

  measure2Result(m:Measure2):Result ==
    mm := coerce(m.measure)$AnyFunctions1(Float)
    mmr:Record(key:Symbol,entry:Any) := [bestMeasure@Symbol,mm]
    mn := coerce(m.name)$AnyFunctions1(ST)
    mnr:Record(key:Symbol,entry:Any) := [nameOfRoutine@Symbol,mn]
    me := coerce(m.explanations)$AnyFunctions1(List String)
    mer:Record(key:Symbol,entry:Any) := [allMeasures@Symbol,me]
    mx := coerce(m.extra)$AnyFunctions1(Result)
    mxr:Record(key:Symbol,entry:Any) := [other@Symbol,mx]
    mr := construct([mmr,mnr,mer,mxr])$Result
    met := coerce(mr)$AnyFunctions1(Result)
    meth:Record(key:Symbol,entry:Any):=[method@Symbol,met]
    construct([meth])$Result

  att2Result(att:ATT):Result ==
    aepc := coerce(att.endPointContinuity)$AnyFunctions1(CTYPE)
    ar := coerce(att.range)$AnyFunctions1(RTYPE)
    as := coerce(att.singularitiesStream)$AnyFunctions1(STYPE)
    aa:List Any := [aepc,ar,as]
    aaa := coerce(aa)$AnyFunctions1(List Any)
    aar:Record(key:Symbol,entry:Any) := [attributes@Symbol,aaa]
    construct([aar])$Result

  iflist2Result(ifv:IFV):Result ==
    ifvs:List String :=
      [concat(["stiffness: ",outputMeasure(ifv.stiffness)]),
        concat(["stability: ",outputMeasure(ifv.stability)]),
         concat(["expense: ",outputMeasure(ifv.expense)]),
```

```
            concat(["accuracy: ",outputMeasure(ifv.accuracy)]),
             concat(["intermediateResults: ",outputMeasure(ifv.intermediateResults)])]
      ifa:= coerce(ifvs)$AnyFunctions1(List String)
      ifr:Record(key:Symbol,entry:Any) := [intensityFunctions@Symbol,ifa]
      construct([ifr])$Result
```

———-

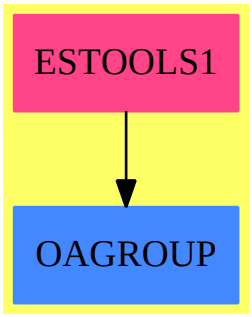
— **ESTOOLS.dotabb** —

```
"ESTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ESTOOLS" -> "FS"
```

———-

6.21 package ESTOOLS1 ExpertSystemToolsPackage1

6.21.1 ExpertSystemToolsPackage1 (ESTOOLS1)



Exports:
neglist

— **package ESTOOLS1 ExpertSystemToolsPackage1** —

```
)abbrev package ESTOOLS1 ExpertSystemToolsPackage1
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: February 1995
++ Basic Operations: neglist
++ Description:
++ \axiom{ExpertSystemToolsPackage1} contains some useful functions for use
```

```
++ by the computational agents of Ordinary Differential Equation solvers.

ExpertSystemToolsPackage1(R1:OR): E == I where
  OR  ==> OrderedRing
  E ==>  with
    neglist:List R1 -> List R1
      ++ neglist(l) returns only the negative elements of the list \spad{l}
  I ==> add
    neglist(l:List R1):List R1 == [u for u in l | negative?(u)$R1]
```

———-

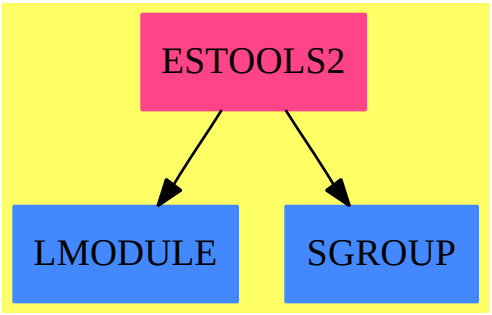
— **ESTOOLS1.dotabb** —

```
"ESTOOLS1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS1"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ESTOOLS1" -> "OAGROUP"
```

———-

6.22 package ESTOOLS2 ExpertSystemToolsPackage2

6.22.1 ExpertSystemToolsPackage2 (ESTOOLS2)



Exports:
map

— **package ESTOOLS2 ExpertSystemToolsPackage2** —

```
)abbrev package ESTOOLS2 ExpertSystemToolsPackage2
++ Author: Brian Dupee
++ Date Created: February 1995
```

```
++ Date Last Updated: July 1996
++ Basic Operations: map
++ Related Constructors: Matrix
++ Description:
++ \axiom{ExpertSystemToolsPackage2} contains some useful functions for use
++ by the computational agents of Ordinary Differential Equation solvers.

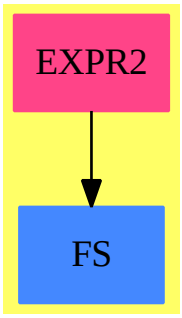
ExpertSystemToolsPackage2(R1:R,R2:R): E == I where
  R  ==> Ring
  E ==>  with
    map:(R1->R2,Matrix R1) -> Matrix R2
      ++ map(f,m) applies a mapping f:R1 -> R2 onto a matrix
      ++ \spad{m} in R1 returning a matrix in R2
  I ==> add
    map(f:R1->R2,m:Matrix R1):Matrix R2 ==
      matrix([[f u for u in v] for v in listOfLists(m)$(Matrix R1)])$(Matrix R2)
```

— **ESTOOLS2.dotabb** —

```
"ESTOOLS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"ESTOOLS2" -> "LMODULE"
"ESTOOLS2" -> "SGROUP"
```

6.23 package EXPR2 ExpressionFunctions2

6.23.1 ExpressionFunctions2 (EXPR2)



Exports:
map

— **package EXPR2 ExpressionFunctions2** —

```
)abbrev package EXPR2 ExpressionFunctions2
++ Lifting of maps to Expressions
++ Author: Manuel Bronstein
++ Date Created: 16 Jan 1989
++ Date Last Updated: 22 Jan 1990
++ Description:
++ Lifting of maps to Expressions.

ExpressionFunctions2(R:OrderedSet, S:OrderedSet):
 Exports == Implementation where
  K   ==> Kernel R
  F2  ==> FunctionSpaceFunctions2(R, Expression R, S, Expression S)
  E2  ==> ExpressionSpaceFunctions2(Expression R, Expression S)

  Exports ==> with
    map: (R -> S, Expression R) -> Expression S
      ++ map(f, e) applies f to all the constants appearing in e.

  Implementation == add
    if S has Ring and R has Ring then
      map(f, r) == map(f, r)$F2
    else
      map(f, r) == map(x1 +-> map(f, x1), retract r)$E2
```

———-

— **EXPR2.dotabb** —

```
"EXPR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPR2"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EXPR2" -> "FS"
```

———-

6.24 package EXPRSOL ExpressionSolve

6.24.1 Bugs

```
seriesSolve(sin f x / cos x, f, x, [1])$EXPRSOL(INT, EXPR INT, UFPS EXPR INT, UFPS SUPEXPR EXPR INT)
```

returns

```
(((0 . 1) 0 . 1) NonNullStream #<compiled-function |STREAM;generate;M$;62!0|> . UNPRINTABLE)
```

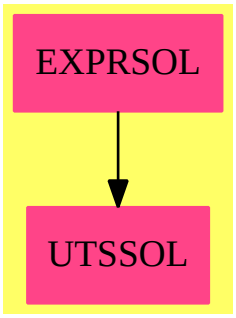
but

```
U ==> UFPS SUPEXPR EXPR INT
seriesSolve(s +-> sin s *((cos monomial(1,1)$U)**-1)$U, f, x, [0])$EXPRSOL(INT, EXPR INT, UFPS E
```

works. This is probably due to missing "/" in UFPS.

```
  I'd really like to be able to specify a function that works for all domains
  in a category. For example, x +-> y(x)^2 + sin x + x should work
  for EXPR INT as well as for UTS INT, both being domains having
  TranscendentalFunctionCategory.
```

6.24.2 ExpressionSolve (EXPRSOL)



Exports:
replaceDiffs seriesSolve

— package EXPRSOL ExpressionSolve —

```
)abbrev package EXPRSOL ExpressionSolve
++ Description:
++ This package has no description

ExpressionSolve(R, F, UTSF, UTSSUPF): Exports == Implementation where
    R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
    F: FunctionSpace R
    UTSF: UnivariateTaylorSeriesCategory F
    SUP  ==> SparseUnivariatePolynomialExpressions
    UTSSUPF: UnivariateTaylorSeriesCategory SUP F
    OP   ==> BasicOperator
```

```
    SY   ==> Symbol
    NNI  ==> NonNegativeInteger
    MKF ==> MakeBinaryCompiledFunction(F, UTSSUPF, UTSSUPF, UTSSUPF)

    Exports == with

        seriesSolve: (F, OP, SY, List F) -> UTSF
        replaceDiffs: (F, OP, Symbol) -> F

    Implementation == add
\getchunk{implementation: EXPRSOL ExpressionSolve}
```

———-

The general method is to transform the given expression into a form which can then be compiled. There is currently no other way in Axiom to transform an expression into a function.

We need to replace the differentiation operator by the corresponding function in the power series category, and make composition explicit. Furthermore, we need to replace the variable by the corresponding variable in the power series. It turns out that the compiler doesn't find the right definition of monomial(1,1). Thus we introduce it as a second argument. In fact, maybe that's even cleaner. Also, we need to tell the compiler that kernels that are independent of the main variable should be coerced to elements of the coefficient ring, since it will complain otherwise.

```
  I cannot find an example for this behaviour right now. However, if I do use
  the coerce, the following fails:

     seriesSolve(h x -1-x*h x *h(q*x), h, x, [1])
```

— package EXPRSOL ExpressionSolve —

```
\getchunk{implementation: EXPRSOL ExpressionSolve}
        opelt := operator("elt"::Symbol)$OP
        opdiff := operator("D"::Symbol)$OP
        opcoerce := operator("coerce"::Symbol)$OP

--        replaceDiffs: (F, OP, Symbol) -> F
        replaceDiffs (expr, op, sy) ==
            lk := kernels expr
            for k in lk repeat
--                if freeOf?(coerce k, sy) then
--                    expr := subst(expr, [k], [opcoerce [coerce k]])

                if is?(k, op) then
                    arg := first argument k
```

```
                    if arg = sy::F
                    then expr := subst(expr, [k], [(name op)::F])
                    else expr := subst(expr, [k], [opelt [(name op)::F,
                                                          replaceDiffs(arg, op,
                                                          sy)]])
--                    => "iterate"

                if is?(k, %diff) then
                    args := argument k
                    differentiand :=
                     replaceDiffs(subst(args.1, args.2 = args.3), op, sy)
                    expr := subst(expr, [k], [opdiff differentiand])
--                    => "iterate"
            expr


        seriesSolve(expr, op, sy, l) ==
            ex := replaceDiffs(expr, op, sy)
            f := compiledFunction(ex, name op, sy)$MKF
            seriesSolve(x+->f(x, monomial(1,1)$UTSSUPF), l)_
              $TaylorSolve(F, UTSF, UTSSUPF)

            ———-
```

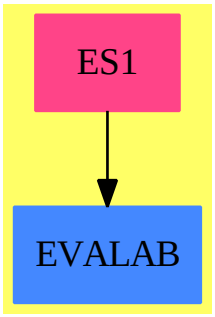
— **EXPRSOL.dotabb** —

```
"EXPRSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRSOL"]
"UTSSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSSOL"]
"EXPRSOL" -> "UTSSOL"
```

```
            ———-
```

6.25 package ES1 ExpressionSpaceFunctions1

6.25.1 ExpressionSpaceFunctions1 (ES1)



Exports:
map

— package ES1 ExpressionSpaceFunctions1 —

```
)abbrev package ES1 ExpressionSpaceFunctions1
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a map from any expression space into any object
++ to be lifted to a kernel over the expression set, using a given
++ property of the operator of the kernel.
-- should not be exposed

ExpressionSpaceFunctions1(F:ExpressionSpace, S:Type): with
    map: (F -> S, String, Kernel F) -> S
      ++ map(f, p, k) uses the property p of the operator
      ++ of k, in order to lift f and apply it to k.

  == add
    --  prop  contains an evaluation function List S -> S
    map(F2S, prop, k) ==
      args := [F2S x for x in argument k]$List(S)
      (p := property(operator k, prop)) case None =>
                                  ((p::None) pretend (List S -> S)) args
      error "Operator does not have required property"
```

———-

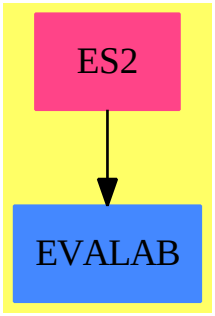
— ES1.dotabb —

```
"ES1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES1"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"ES1" -> "EVALAB"
```

———-

6.26 package ES2 ExpressionSpaceFunctions2

6.26.1 ExpressionSpaceFunctions2 (ES2)



Exports:
map

— **package ES2 ExpressionSpaceFunctions2** —

```
)abbrev package ES2 ExpressionSpaceFunctions2
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a mapping E -> F to be lifted to a kernel over E;
++ This lifting can fail if the operator of the kernel cannot be applied
++ in F; Do not use this package with E = F, since this may
++ drop some properties of the operators.

ExpressionSpaceFunctions2(E:ExpressionSpace, F:ExpressionSpace): with
    map: (E -> F, Kernel E) -> F
      ++ map(f, k) returns \spad{g = op(f(a1),...,f(an))} where
      ++ \spad{k = op(a1,...,an)}.
  == add
    map(f, k) ==
      (operator(operator k)$F) [f x for x in argument k]$List(F)
```

———-

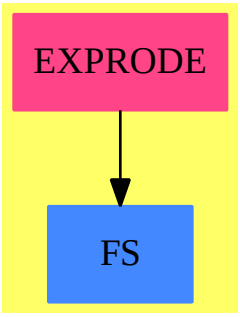
— **ES2.dotabb** —

```
"ES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES2"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"ES2" -> "EVALAB"
```

———-

6.27 package EXPRODE ExpressionSpaceODESolver

6.27.1 ExpressionSpaceODESolver (EXPRODE)



Exports:
seriesSolve

— package EXPRODE ExpressionSpaceODESolver —

```
)abbrev package EXPRODE ExpressionSpaceODESolver
++ Author: Manuel Bronstein
++ Date Created: 5 Mar 1990
++ Date Last Updated: 30 September 1993
++ Keywords: differential equation, ODE, Taylor series
++ Description:
++ Taylor series solutions of explicit ODE's;

ExpressionSpaceODESolver(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
  F: FunctionSpace R

  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  OP  ==> BasicOperator
  SY  ==> Symbol
  UTS ==> UnivariateTaylorSeries(F, x, center)
  MKF ==> MakeUnaryCompiledFunction(F, UTS, UTS)
  MKL ==> MakeUnaryCompiledFunction(F, List UTS, UTS)
  A1  ==> AnyFunctions1(UTS)
  AL1 ==> AnyFunctions1(List UTS)
  EQ  ==> Equation F
  ODE ==> UnivariateTaylorSeriesODESolver(F, UTS)
```

```
Exports ==> with
  seriesSolve: (EQ, OP, EQ, EQ) -> Any
    ++ seriesSolve(eq,y,x=a, y a = b) returns a Taylor series solution
    ++ of eq around x = a with initial condition \spad{y(a) = b}.
    ++ Note that eq must be of the form
    ++ \spad{f(x, y x) y'(x) + g(x, y x) = h(x, y x)}.
  seriesSolve: (EQ, OP, EQ, List F) -> Any
    ++ seriesSolve(eq,y,x=a,[b0,...,b(n-1)]) returns a Taylor series
    ++ solution of eq around \spad{x = a} with initial conditions
    ++ \spad{y(a) = b0}, \spad{y'(a) = b1},
    ++ \spad{y''(a) = b2}, ...,\spad{y(n-1)(a) = b(n-1)}
    ++ eq must be of the form
    ++ \spad{f(x, y x, y'(x),..., y(n-1)(x)) y(n)(x) +
    ++ g(x,y x,y'(x),...,y(n-1)(x)) = h(x,y x, y'(x),..., y(n-1)(x))}.
  seriesSolve: (List EQ, List OP, EQ, List EQ) -> Any
    ++ seriesSolve([eq1,...,eqn],[y1,...,yn],x = a,[y1 a = b1,...,yn a = bn])
    ++ returns a taylor series solution of \spad{[eq1,...,eqn]} around
    ++ \spad{x = a} with initial conditions \spad{yi(a) = bi}.
    ++ Note that eqi must be of the form
    ++ \spad{fi(x, y1 x, y2 x,..., yn x) y1'(x) +
    ++ gi(x, y1 x, y2 x,..., yn x) = h(x, y1 x, y2 x,..., yn x)}.
  seriesSolve: (List EQ, List OP, EQ, List F) -> Any
    ++ seriesSolve([eq1,...,eqn], [y1,...,yn], x=a, [b1,...,bn])
    ++ is equivalent to
    ++ \spad{seriesSolve([eq1,...,eqn], [y1,...,yn], x = a,
    ++ [y1 a = b1,..., yn a = bn])}.
  seriesSolve: (List F, List OP, EQ, List F) -> Any
    ++ seriesSolve([eq1,...,eqn], [y1,...,yn], x=a, [b1,...,bn])
    ++ is equivalent to
    ++ \spad{seriesSolve([eq1=0,...,eqn=0], [y1,...,yn], x=a, [b1,...,bn])}.
  seriesSolve: (List F, List OP, EQ, List EQ) -> Any
    ++ seriesSolve([eq1,...,eqn], [y1,...,yn],
    ++ x = a,[y1 a = b1,..., yn a = bn])
    ++ is equivalent to
    ++ \spad{seriesSolve([eq1=0,...,eqn=0], [y1,...,yn], x = a,
    ++ [y1 a = b1,..., yn a = bn])}.
  seriesSolve: (EQ, OP, EQ, F) -> Any
    ++ seriesSolve(eq,y, x=a, b) is equivalent to
    ++ \spad{seriesSolve(eq, y, x=a, y a = b)}.
  seriesSolve: (F, OP, EQ, F) -> Any
    ++ seriesSolve(eq, y, x = a, b) is equivalent to
    ++ \spad{seriesSolve(eq = 0, y, x = a, y a = b)}.
  seriesSolve: (F, OP, EQ, EQ) -> Any
    ++ seriesSolve(eq, y, x = a, y a = b) is equivalent to
    ++ \spad{seriesSolve(eq=0, y, x=a, y a = b)}.
  seriesSolve: (F, OP, EQ, List F) -> Any
    ++ seriesSolve(eq, y, x = a, [b0,...,bn]) is equivalent to
    ++ \spad{seriesSolve(eq = 0, y, x = a, [b0,...,b(n-1)])}.

Implementation ==> add
```

```
    checkCompat: (OP, EQ, EQ) -> F
    checkOrder1: (F, OP, K, SY, F) -> F
    checkOrderN: (F, OP, K, SY, F, NonNegativeInteger) -> F
    checkSystem: (F, List K, List F) -> F
    div2exquo  : F -> F
    smp2exquo  : P -> F
    k2exquo    : K -> F
    diffRhs    : (F, F) -> F
    diffRhsK   : (K, F) -> F
    findCompat : (F, List EQ) -> F
    findEq     : (K, SY, List F) -> F
    localInteger: F -> F

    opelt := operator("elt"::Symbol)$OP
    --opex  := operator("exquo"::Symbol)$OP
    opex  := operator("fixedPointExquo"::Symbol)$OP
    opint := operator("integer"::Symbol)$OP

    Rint? := R has IntegerNumberSystem

    localInteger n == (Rint? => n; opint n)
    diffRhs(f, g) == diffRhsK(retract(f)@K, g)

    k2exquo k ==
      is?(op := operator k, "%diff"::Symbol) =>
        error "Improper differential equation"
      kernel(op, [div2exquo f for f in argument k]$List(F))

    smp2exquo p ==
      map(k2exquo,x+->x::F,p)_
       $PolynomialCategoryLifting(IndexedExponents K,K, R, P, F)

    div2exquo f ==
--      one?(d := denom f) => f
      ((d := denom f) = 1) => f
      opex(smp2exquo numer f, smp2exquo d)

-- if g is of the form a * k + b, then return -b/a
    diffRhsK(k, g) ==
      h := univariate(g, k)
      (degree(numer h) <= 1) and ground? denom h =>
        - coefficient(numer h, 0) / coefficient(numer h, 1)
      error "Improper differential equation"

    checkCompat(y, eqx, eqy) ==
      lhs(eqy) =$F y(rhs eqx) => rhs eqy
      error "Improper initial value"

    findCompat(yx, l) ==
      for eq in l repeat
```

```
      yx =$F lhs eq => return rhs eq
    error "Improper initial value"

  findEq(k, x, sys) ==
    k := retract(differentiate(k::F, x))@K
    for eq in sys repeat
      member?(k, kernels eq) => return eq
    error "Improper differential equation"

  checkOrder1(diffeq, y, yx, x, sy) ==
    div2exquo subst(diffRhs(differentiate(yx::F,x),diffeq),[yx],[sy])

  checkOrderN(diffeq, y, yx, x, sy, n) ==
    zero? n => error "No initial value(s) given"
    m     := (minIndex(l := [retract(f := yx::F)@K]$List(K)))::F
    lv    := [opelt(sy, localInteger m)]$List(F)
    for i in 2..n repeat
      l  := concat(retract(f := differentiate(f, x))@K, l)
      lv := concat(opelt(sy, localInteger(m := m + 1)), lv)
    div2exquo subst(diffRhs(differentiate(f, x), diffeq), l, lv)

  checkSystem(diffeq, yx, lv) ==
    for k in kernels diffeq repeat
      is?(k, "%diff"::SY) =>
        return div2exquo subst(diffRhsK(k, diffeq), yx, lv)
    0

  seriesSolve(l:List EQ, y:List OP, eqx:EQ, eqy:List EQ) ==
    seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, eqy)

  seriesSolve(l:List EQ, y:List OP, eqx:EQ, y0:List F) ==
    seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, y0)

  seriesSolve(l:List F, ly:List OP, eqx:EQ, eqy:List EQ) ==
    seriesSolve(l, ly, eqx,
                [findCompat(y rhs eqx, eqy) for y in ly]$List(F))

  seriesSolve(diffeq:EQ, y:OP, eqx:EQ, eqy:EQ) ==
    seriesSolve(lhs diffeq - rhs diffeq, y, eqx, eqy)

  seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:F) ==
    seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

  seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:List F) ==
    seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

  seriesSolve(diffeq:F, y:OP, eqx:EQ, eqy:EQ) ==
    seriesSolve(diffeq, y, eqx, checkCompat(y, eqx, eqy))

  seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:F) ==
```

```
      x      := symbolIfCan(retract(lhs eqx)@K)::SY
      sy     := name y
      yx     := retract(y lhs eqx)@K
      f      := checkOrder1(diffeq, y, yx, x, sy::F)
      center := rhs eqx
      coerce(ode1(compiledFunction(f, sy)$MKF, y0)$ODE)$A1

    seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:List F) ==
      x      := symbolIfCan(retract(lhs eqx)@K)::SY
      sy     := new()$SY
      yx     := retract(y lhs eqx)@K
      f      := checkOrderN(diffeq, y, yx, x, sy::F, #y0)
      center := rhs eqx
      coerce(ode(compiledFunction(f, sy)$MKL, y0)$ODE)$A1

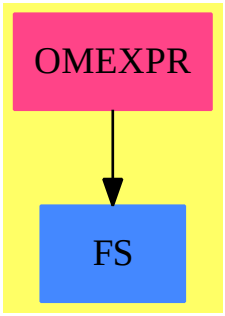
    seriesSolve(sys:List F, ly:List OP, eqx:EQ, l0:List F) ==
      x      := symbolIfCan(kx := retract(lhs eqx)@K)::SY
      fsy    := (sy := new()$SY)::F
      m      := (minIndex(l0) - 1)::F
      yx     := concat(kx, [retract(y lhs eqx)@K for y in ly]$List(K))
      lelt   := [opelt(fsy, localInteger(m := m+1)) for k in yx]$List(F)
      sys    := [findEq(k, x, sys) for k in rest yx]
      l      := [checkSystem(eq, yx, lelt) for eq in sys]$List(F)
      center := rhs eqx
      coerce(mpsode(l0,[compiledFunction(f,sy)$MKL for f in l])$ODE)$AL1
```

— EXPRODE.dotabb —

```
"EXPRODE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRODE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EXPRODE" -> "FS"
```

6.28 package OMEXPR ExpressionToOpenMath

6.28.1 ExpressionToOpenMath (OMEXPR)



Exports:
OMwrite

— package OMEXPR ExpressionToOpenMath —

```
)abbrev package OMEXPR ExpressionToOpenMath
++ Author: Mike Dewar & Vilya Harvey
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{ExpressionToOpenMath} provides support for
++ converting objects of type \spadtype{Expression} into OpenMath.

ExpressionToOpenMath(R: Join(OpenMath, OrderedSet, Ring)): with
  OMwrite  : Expression R -> String
  OMwrite  : (Expression R, Boolean) -> String
  OMwrite  : (OpenMathDevice, Expression R) -> Void
  OMwrite  : (OpenMathDevice, Expression R, Boolean) -> Void
 == add
  import Expression R
  SymInfo ==> Record(cd:String, name:String)
  import SymInfo
  import Record(key: Symbol, entry: SymInfo)
  import AssociationList(Symbol, SymInfo)
  import OMENC
```

```
----------------------------
-- Local translation tables.
----------------------------

nullaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_
  [pi, ["nums1", "pi"]] ]

unaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_
  [exp,  ["transc1", "exp"]],_
  [log,  ["transc1", "ln"]],_
  [sin,  ["transc1", "sin"]],_
  [cos,  ["transc1", "cos"]],_
  [tan,  ["transc1", "tan"]],_
  [cot,  ["transc1", "cot"]],_
  [sec,  ["transc1", "sec"]],_
  [csc,  ["transc1", "csc"]],_
  [asin, ["transc1", "arcsin"]],_
  [acos, ["transc1", "arccos"]],_
  [atan, ["transc1", "arctan"]],_
  [acot, ["transc1", "arccot"]],_
  [asec, ["transc1", "arcsec"]],_
  [acsc, ["transc1", "arccsc"]],_
  [sinh, ["transc1", "sinh"]],_
  [cosh, ["transc1", "cosh"]],_
  [tanh, ["transc1", "tanh"]],_
  [coth, ["transc1", "coth"]],_
  [sech, ["transc1", "sech"]],_
  [csch, ["transc1", "csch"]],_
  [asinh, ["transc1", "arcsinh"]],_
  [acosh, ["transc1", "arccosh"]],_
  [atanh, ["transc1", "arctanh"]],_
  [acoth, ["transc1", "arccoth"]],_
  [asech, ["transc1", "arcsech"]],_
  [acsch, ["transc1", "arccsch"]],_
  [factorial, ["integer1", "factorial"]],_
  [abs, ["arith1", "abs"]] ]

  -- Still need the following unary functions:
  --  digamma
  --  Gamma
  --  airyAi
  --  airyBi
  --  erf
  --  Ei
  --  Si
  --  Ci
  --  li
  --  dilog

  -- Still need the following binary functions:
```

```
  --      Gamma(a, x)
  --      Beta(x,y)
  --      polygamma(k,x)
  --      besselJ(v,x)
  --      besselY(v,x)
  --      besselI(v,x)
  --      besselK(v,x)
  --      permutation(n, m)
  --      summation(x:%, n:Symbol) : as opposed to "definite" sum
  --      product(x:%, n:Symbol)   : ditto

------------------------
-- Forward declarations.
------------------------

outputOMExpr  : (OpenMathDevice, Expression R) -> Void

-------------------------
-- Local helper functions
-------------------------

outputOMArith1(dev: OpenMathDevice, sym: String, args: List Expression R): Void ==
  OMputApp(dev)
  OMputSymbol(dev, "arith1", sym)
  for arg in args repeat
    OMwrite(dev, arg, false)
  OMputEndApp(dev)

outputOMLambda(dev: OpenMathDevice, ex: Expression R, var: Expression R): Void ==
  OMputBind(dev)
  OMputSymbol(dev, "fns1", "lambda")
  OMputBVar(dev)
  OMwrite(dev, var, false)
  OMputEndBVar(dev)
  OMwrite(dev, ex, false)
  OMputEndBind(dev)

outputOMInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
  OMputApp(dev)
  OMputSymbol(dev, "interval1", "interval")
  OMwrite(dev, lo, false)
  OMwrite(dev, hi, false)
  OMputEndApp(dev)

outputOMIntInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
  OMputApp(dev)
  OMputSymbol(dev, "interval1", "integer__interval")
  OMwrite(dev, lo, false)
  OMwrite(dev, hi, false)
  OMputEndApp(dev)
```

```
outputOMBinomial(dev: OpenMathDevice, args: List Expression R): Void ==
  not #args=2 => error "Wrong number of arguments to binomial"
  OMputApp(dev)
  OMputSymbol(dev, "combinat1", "binomial")
  for arg in args repeat
    OMwrite(dev, arg, false)
  OMputEndApp(dev)

outputOMPower(dev: OpenMathDevice, args: List Expression R): Void ==
  not #args=2 => error "Wrong number of arguments to power"
  outputOMArith1(dev, "power", args)

outputOMDefsum(dev: OpenMathDevice, args: List Expression R): Void ==
  #args ^= 5 => error "Unexpected number of arguments to a defsum"
  OMputApp(dev)
  OMputSymbol(dev, "arith1", "sum")
  outputOMIntInterval(dev, args.4, args.5)
  outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
  OMputEndApp(dev)

outputOMDefprod(dev: OpenMathDevice, args: List Expression R): Void ==
  #args ^= 5 => error "Unexpected number of arguments to a defprod"
  OMputApp(dev)
  OMputSymbol(dev, "arith1", "product")
  outputOMIntInterval(dev, args.4, args.5)
  outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
  OMputEndApp(dev)

outputOMDefint(dev: OpenMathDevice, args: List Expression R): Void ==
  #args ^= 5 => error "Unexpected number of arguments to a defint"
  OMputApp(dev)
  OMputSymbol(dev, "calculus1", "defint")
  outputOMInterval(dev, args.4, args.5)
  outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
  OMputEndApp(dev)

outputOMInt(dev: OpenMathDevice, args: List Expression R): Void ==
  #args ^= 3 => error "Unexpected number of arguments to a defint"
  OMputApp(dev)
  OMputSymbol(dev, "calculus1", "int")
  outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
  OMputEndApp(dev)

outputOMFunction(dev: OpenMathDevice, op: Symbol, args: List Expression R): Void ==
  nargs := #args
  zero? nargs =>
    omOp: Union(SymInfo, "failed") := search(op, nullaryFunctionAList)
    omOp case "failed" =>
      error concat ["No OpenMath definition for nullary function ", coerce op]
```

```
      OMputSymbol(dev, omOp.cd, omOp.name)
--    one? nargs =>
    (nargs = 1) =>
      omOp: Union(SymInfo, "failed") := search(op, unaryFunctionAList)
      omOp case "failed" =>
        error concat ["No OpenMath definition for unary function ", coerce op]
      OMputApp(dev)
      OMputSymbol(dev, omOp.cd, omOp.name)
      for arg in args repeat
        OMwrite(dev, arg, false)
      OMputEndApp(dev)
    -- Most of the binary operators cannot be handled trivialy like the
    -- unary ones since they have bound variables of one kind or another.
    -- The special functions should be straightforward, but we don't have
    -- a CD for them yet :-)
    op = %defint  => outputOMDefint(dev, args)
    op = integral => outputOMInt(dev, args)
    op = %defsum  => outputOMDefsum(dev, args)
    op = %defprod => outputOMDefprod(dev, args)
    op = %power   => outputOMPower(dev, args)
    op = binomial => outputOMBinomial(dev, args)
    error concat ["No OpenMath definition for function ", string op]

  outputOMExpr(dev: OpenMathDevice, ex: Expression R): Void ==
    ground? ex => OMwrite(dev, ground ex, false)
    not((v := retractIfCan(ex)@Union(Symbol,"failed")) case "failed") =>
      OMputVariable(dev, v)
    not((w := isPlus ex) case "failed") => outputOMArith1(dev, "plus", w)
    not((w := isTimes ex) case "failed") => outputOMArith1(dev, "times", w)
    --not((y := isMult ex) case "failed") =>
    --  outputOMArith("times", [OMwrite(y.coef)$Integer,
    --          OMwrite(coerce y.var)])
    -- At the time of writing we don't need both isExpt and isPower
    -- here but they may be relevent when we integrate this stuff into
    -- the main Expression code.  Note that if we don't check that
    -- the exponent is non-trivial we get thrown into an infinite recursion.
--    not (((x := isExpt ex) case "failed") or one? x.exponent) =>
    not (((x := isExpt ex) case "failed") or (x.exponent = 1)) =>
      not((s := symbolIfCan(x.var)@Union(Symbol,"failed")) case "failed") =>
        --outputOMPower(dev, [s::Expression(R), (x.exponent)::Expression(R)])
        OMputApp(dev)
        OMputSymbol(dev, "arith1", "power")
        OMputVariable(dev, s)
        OMputInteger(dev, x.exponent)
        OMputEndApp(dev)
      -- TODO: add error handling code here...
--    not (((z := isPower ex) case "failed") or one? z.exponent) =>
    not (((z := isPower ex) case "failed") or (z.exponent = 1)) =>
      outputOMPower(dev, [ z.val, z.exponent::Expression R ])
      --OMputApp(dev)
```

```
    --OMputSymbol(dev, "arith1", "power")
    --outputOMExpr(dev, z.val)
    --OMputInteger(dev, z.exponent)
    --OMputEndApp(dev)
  -- Must only be one top-level Kernel by this point
  k : Kernel Expression R := first kernels ex
  outputOMFunction(dev, name operator k, argument k)


----------
-- Exports
----------

OMwrite(ex: Expression R): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
  OMputObject(dev)
  outputOMExpr(dev, ex)
  OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(ex: Expression R, wholeObj: Boolean): String ==
  s: String := ""
  sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
  dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
  if wholeObj then
    OMputObject(dev)
  outputOMExpr(dev, ex)
  if wholeObj then
    OMputEndObject(dev)
  OMclose(dev)
  s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
  s

OMwrite(dev: OpenMathDevice, ex: Expression R): Void ==
  OMputObject(dev)
  outputOMExpr(dev, ex)
  OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, ex: Expression R, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  outputOMExpr(dev, ex)
  if wholeObj then
    OMputEndObject(dev)
```

———-

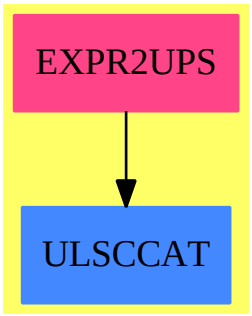
— OMEXPR.dotabb —

```
"OMEXPR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMEXPR"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"OMEXPR" -> "FS"
```

———-

6.29 package EXPR2UPS ExpressionToUnivariatePowerSeries

6.29.1 ExpressionToUnivariatePowerSeries (EXPR2UPS)



Exports:
laurent puiseux series taylor

— package EXPR2UPS ExpressionToUnivariatePowerSeries —

```
)abbrev package EXPR2UPS ExpressionToUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 9 May 1989
++ Date Last Updated: 20 September 1993
++ Basic Operations: taylor, laurent, puiseux, series
++ Related Domains: UnivariateTaylorSeries, UnivariateLaurentSeries,
++   UnivariatePuiseuxSeries, Expression
++ Also See: FunctionSpaceToUnivariatePowerSeries
++ AMS Classifications:
++ Keywords: Taylor series, Laurent series, Puiseux series
++ Examples:
++ References:
```

```
++ Description:
++ This package provides functions to convert functional expressions
++ to power series.

ExpressionToUnivariatePowerSeries(R,FE): Exports == Implementation where
  R  : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
            LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
            FunctionSpace R)

  EQ     ==> Equation
  I      ==> Integer
  NNI    ==> NonNegativeInteger
  RN     ==> Fraction Integer
  SY     ==> Symbol
  UTS    ==> UnivariateTaylorSeries
  ULS    ==> UnivariateLaurentSeries
  UPXS   ==> UnivariatePuiseuxSeries
  GSER   ==> GeneralUnivariatePowerSeries
  EFULS  ==> ElementaryFunctionsUnivariateLaurentSeries
  EFUPXS ==> ElementaryFunctionsUnivariatePuiseuxSeries
  FS2UPS ==> FunctionSpaceToUnivariatePowerSeries
  Prob   ==> Record(func:String,prob:String)
  ANY1   ==> AnyFunctions1

  Exports ==> with
    taylor: SY -> Any
      ++ \spad{taylor(x)} returns x viewed as a Taylor series.
    taylor: FE -> Any
      ++ \spad{taylor(f)} returns a Taylor expansion of the expression f.
      ++ Note that f should have only one variable; the series will be
      ++ expanded in powers of that variable.
    taylor: (FE,NNI) -> Any
      ++ \spad{taylor(f,n)} returns a Taylor expansion of the expression f.
      ++ Note that f should have only one variable; the series will be
      ++ expanded in powers of that variable and terms will be computed
      ++ up to order at least n.
    taylor: (FE,EQ FE) -> Any
      ++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
      ++ in powers of \spad{(x - a)}.
    taylor: (FE,EQ FE,NNI) -> Any
      ++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
      ++ in powers of \spad{(x - a)}; terms will be computed up to order
      ++ at least n.

    laurent: SY -> Any
      ++ \spad{laurent(x)} returns x viewed as a Laurent series.
    laurent: FE -> Any
      ++ \spad{laurent(f)} returns a Laurent expansion of the expression f.
      ++ Note that f should have only one variable; the series will be
```

```
  ++ expanded in powers of that variable.
laurent: (FE,I) -> Any
  ++ \spad{laurent(f,n)} returns a Laurent expansion of the expression f.
  ++ Note that f should have only one variable; the series will be
  ++ expanded in powers of that variable and terms will be computed
  ++ up to order at least n.
laurent: (FE,EQ FE) -> Any
  ++ \spad{laurent(f,x = a)} expands the expression f as a Laurent series
  ++ in powers of \spad{(x - a)}.
laurent: (FE,EQ FE,I) -> Any
  ++ \spad{laurent(f,x = a,n)} expands the expression f as a Laurent
  ++ series in powers of \spad{(x - a)}; terms will be computed up to order
  ++ at least n.
puiseux: SY -> Any
  ++ \spad{puiseux(x)} returns x viewed as a Puiseux series.
puiseux: FE -> Any
  ++ \spad{puiseux(f)} returns a Puiseux expansion of the expression f.
  ++ Note that f should have only one variable; the series will be
  ++ expanded in powers of that variable.
puiseux: (FE,RN) -> Any
  ++ \spad{puiseux(f,n)} returns a Puiseux expansion of the expression f.
  ++ Note that f should have only one variable; the series will be
  ++ expanded in powers of that variable and terms will be computed
  ++ up to order at least n.
puiseux: (FE,EQ FE) -> Any
  ++ \spad{puiseux(f,x = a)} expands the expression f as a Puiseux series
  ++ in powers of \spad{(x - a)}.
puiseux: (FE,EQ FE,RN) -> Any
  ++ \spad{puiseux(f,x = a,n)} expands the expression f as a Puiseux
  ++ series in powers of \spad{(x - a)}; terms will be computed up to order
  ++ at least n.

series: SY -> Any
  ++ \spad{series(x)} returns x viewed as a series.
series: FE -> Any
  ++ \spad{series(f)} returns a series expansion of the expression f.
  ++ Note that f should have only one variable; the series will be
  ++ expanded in powers of that variable.
series: (FE,RN) -> Any
  ++ \spad{series(f,n)} returns a series expansion of the expression f.
  ++ Note that f should have only one variable; the series will be
  ++ expanded in powers of that variable and terms will be computed
  ++ up to order at least n.
series: (FE,EQ FE) -> Any
  ++ \spad{series(f,x = a)} expands the expression f as a series
  ++ in powers of (x - a).
series: (FE,EQ FE,RN) -> Any
  ++ \spad{series(f,x = a,n)} expands the expression f as a series
  ++ in powers of (x - a); terms will be computed up to order
  ++ at least n.
```

```
Implementation ==> add
  performSubstitution: (FE,SY,FE) -> FE
  performSubstitution(fcn,x,a) ==
    zero? a => fcn
    xFE := x :: FE
    eval(fcn,xFE = xFE + a)

  iTaylor: (FE,SY,FE) -> Any
  iTaylor(fcn,x,a) ==
    pack := FS2UPS(R,FE,I,ULS(FE,x,a),_
                   EFULS(FE,UTS(FE,x,a),ULS(FE,x,a)),x)
    ans := exprToUPS(fcn,false,"just do it")$pack
    ans case %problem =>
      ans.%problem.prob = "essential singularity" =>
        error "No Taylor expansion: essential singularity"
      ans.%problem.func = "log" =>
        error "No Taylor expansion: logarithmic singularity"
      ans.%problem.func = "nth root" =>
        error "No Taylor expansion: fractional powers in expansion"
      error "No Taylor expansion"
    uls := ans.%series
    (uts := taylorIfCan uls) case "failed" =>
      error "No Taylor expansion: pole"
    any1 := ANY1(UTS(FE,x,a))
    coerce(uts :: UTS(FE,x,a))$any1

  taylor(x:SY) ==
    uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
    coerce(monomial(1,1)$uts)$any1

  taylor(fcn:FE) ==
    null(vars := variables fcn) =>
      error "taylor: expression has no variables"
    not null rest vars =>
      error "taylor: expression has more than one variable"
    taylor(fcn,(first(vars) :: FE) = 0)

  taylor(fcn:FE,n:NNI) ==
    null(vars := variables fcn) =>
      error "taylor: expression has no variables"
    not null rest vars =>
      error "taylor: expression has more than one variable"
    x := first vars
    uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
    series := retract(taylor(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1

  taylor(fcn:FE,eq:EQ FE) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
```

```
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    iTaylor(performSubstitution(fcn,x,a),x,a)

  taylor(fcn,eq,n) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    any1 := ANY1(UTS(FE,x,a))
    series := retract(iTaylor(performSubstitution(fcn,x,a),x,a))$any1
    coerce(extend(series,n))$any1

  iLaurent: (FE,SY,FE) -> Any
  iLaurent(fcn,x,a) ==
    pack := FS2UPS(R,FE,I,ULS(FE,x,a),_
                   EFULS(FE,UTS(FE,x,a),ULS(FE,x,a)),x)
    ans := exprToUPS(fcn,false,"just do it")$pack
    ans case %problem =>
      ans.%problem.prob = "essential singularity" =>
        error "No Laurent expansion: essential singularity"
      ans.%problem.func = "log" =>
        error "No Laurent expansion: logarithmic singularity"
      ans.%problem.func = "nth root" =>
        error "No Laurent expansion: fractional powers in expansion"
      error "No Laurent expansion"
    any1 := ANY1(ULS(FE,x,a))
    coerce(ans.%series)$any1

  laurent(x:SY) ==
    uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
    coerce(monomial(1,1)$uls)$any1

  laurent(fcn:FE) ==
    null(vars := variables fcn) =>
      error "laurent: expression has no variables"
    not null rest vars =>
      error "laurent: expression has more than one variable"
    laurent(fcn,(first(vars) :: FE) = 0)

  laurent(fcn:FE,n:I) ==
    null(vars := variables fcn) =>
      error "laurent: expression has no variables"
    not null rest vars =>
      error "laurent: expression has more than one variable"
    x := first vars
    uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
    series := retract(laurent(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1

  laurent(fcn:FE,eq:EQ FE) ==
```

```
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    iLaurent(performSubstitution(fcn,x,a),x,a)

  laurent(fcn,eq,n) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    any1 := ANY1(ULS(FE,x,a))
    series := retract(iLaurent(performSubstitution(fcn,x,a),x,a))$any1
    coerce(extend(series,n))$any1

  iPuiseux: (FE,SY,FE) -> Any
  iPuiseux(fcn,x,a) ==
    pack := FS2UPS(R,FE,RN,UPXS(FE,x,a),_
                   EFUPXS(FE,ULS(FE,x,a),UPXS(FE,x,a),_
                   EFULS(FE,UTS(FE,x,a),ULS(FE,x,a))),x)
    ans := exprToUPS(fcn,false,"just do it")$pack
    ans case %problem =>
      ans.%problem.prob = "essential singularity" =>
        error "No Puiseux expansion: essential singularity"
      ans.%problem.func = "log" =>
        error "No Puiseux expansion: logarithmic singularity"
      error "No Puiseux expansion"
    any1 := ANY1(UPXS(FE,x,a))
    coerce(ans.%series)$any1

  puiseux(x:SY) ==
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    coerce(monomial(1,1)$upxs)$any1

  puiseux(fcn:FE) ==
    null(vars := variables fcn) =>
      error "puiseux: expression has no variables"
    not null rest vars =>
      error "puiseux: expression has more than one variable"
    puiseux(fcn,(first(vars) :: FE) = 0)

  puiseux(fcn:FE,n:RN) ==
    null(vars := variables fcn) =>
      error "puiseux: expression has no variables"
    not null rest vars =>
      error "puiseux: expression has more than one variable"
    x := first vars
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    series := retract(puiseux(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1

  puiseux(fcn:FE,eq:EQ FE) ==
```

```
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    iPuiseux(performSubstitution(fcn,x,a),x,a)

  puiseux(fcn,eq,n) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    any1 := ANY1(UPXS(FE,x,a))
    series := retract(iPuiseux(performSubstitution(fcn,x,a),x,a))$any1
    coerce(extend(series,n))$any1

  iSeries: (FE,SY,FE) -> Any
  iSeries(fcn,x,a) ==
    pack := FS2UPS(R,FE,RN,UPXS(FE,x,a), _
                   EFUPXS(FE,ULS(FE,x,a),UPXS(FE,x,a), _
                   EFULS(FE,UTS(FE,x,a),ULS(FE,x,a))),x)
    ans := exprToUPS(fcn,false,"just do it")$pack
    ans case %problem =>
      ansG := exprToGenUPS(fcn,false,"just do it")$pack
      ansG case %problem =>
        ansG.%problem.prob = "essential singularity" =>
          error "No series expansion: essential singularity"
        error "No series expansion"
      anyone := ANY1(GSER(FE,x,a))
      coerce((ansG.%series) :: GSER(FE,x,a))$anyone
    any1 := ANY1(UPXS(FE,x,a))
    coerce(ans.%series)$any1

  series(x:SY) ==
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    coerce(monomial(1,1)$upxs)$any1

  series(fcn:FE) ==
    null(vars := variables fcn) =>
      error "series: expression has no variables"
    not null rest vars =>
      error "series: expression has more than one variable"
    series(fcn,(first(vars) :: FE) = 0)

  series(fcn:FE,n:RN) ==
    null(vars := variables fcn) =>
      error "series: expression has no variables"
    not null rest vars =>
      error "series: expression has more than one variable"
    x := first vars
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    series := retract(series(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1
```

```
    series(fcn:FE,eq:EQ FE) ==
      (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
        error "taylor: left hand side must be a variable"
      x := xx :: SY; a := rhs eq
      iSeries(performSubstitution(fcn,x,a),x,a)

    series(fcn,eq,n) ==
      (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
        error "taylor: left hand side must be a variable"
      x := xx :: SY; a := rhs eq
      any1 := ANY1(UPXS(FE,x,a))
      series := retract(iSeries(performSubstitution(fcn,x,a),x,a))$any1
      coerce(extend(series,n))$any1
```

———-

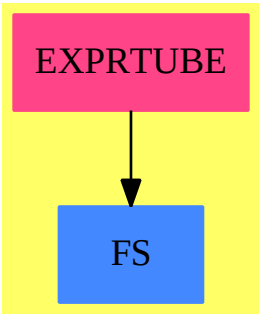
— **EXPR2UPS.dotabb** —

```
"EXPR2UPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPR2UPS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"EXPR2UPS" -> "ULSCCAT"
```

———-

6.30 package EXPRTUBE ExpressionTubePlot

6.30.1 ExpressionTubePlot (EXPRTUBE)



Exports:
constantToUnaryFunction tubePlot

— **package EXPRTUBE ExpressionTubePlot** —

```
)abbrev package EXPRTUBE ExpressionTubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Keywords:
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.

ExpressionTubePlot(): Exports == Implementation where
  B   ==> Boolean
  I   ==> Integer
  FE  ==> Expression Integer
  SY  ==> Symbol
  SF  ==> DoubleFloat
  L   ==> List
  S   ==> String
  SEG ==> Segment
  F2F ==> MakeFloatCompiledFunction(FE)
  Pt  ==> Point SF
  PLOT3 ==> Plot3D
  TUBE  ==> TubePlot Plot3D

  Exports ==> with
    constantToUnaryFunction: SF -> (SF -> SF)
      ++ constantToUnaryFunction(s) is a local function which takes the
      ++ value of s, which may be a function of a constant, and returns
      ++ a function which always returns the value \spadtype{DoubleFloat} s.
    tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF -> SF,I) -> TUBE
      ++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius r(t) with
      ++ n points on each circle about the curve \spad{x = f(t)},
      ++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}.
      ++ The tube is considered to be open.
    tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF -> SF,I,S) -> TUBE
      ++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius \spad{r(t)}
      ++ with n points on each circle about the curve \spad{x = f(t)},
      ++ \spad{y = g(t)},
      ++ \spad{z = h(t)} for t in \spad{[a,b]}. If s = "closed", the tube is
      ++ considered to be closed; if s = "open", the tube is considered
      ++ to be open.
    tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF,I) -> TUBE
      ++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius r with
      ++ n points on each circle about the curve \spad{x = f(t)},
      ++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}.
      ++ The tube is considered to be open.
    tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF,I,S) -> TUBE
      ++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius r with
      ++ n points on each circle about the curve \spad{x = f(t)},
      ++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}.
      ++ If s = "closed", the tube is
      ++ considered to be closed; if s = "open", the tube is considered
```

```
      ++ to be open.

  Implementation ==> add
    import Plot3D
    import F2F
    import TubePlotTools

--% variables

    getVariable: (FE,FE,FE) -> SY
    getVariable(x,y,z) ==
      varList1 := variables x
      varList2 := variables y
      varList3 := variables z
      (not (# varList1 <= 1)) or (not (# varList2 <= 1)) or _
       (not (# varList3 <= 1)) =>
        error "tubePlot: only one variable may be used"
      null varList1 =>
        null varList2 =>
          null varList3 =>
            error "tubePlot: a variable must appear in functions"
          first varList3
        t2 := first varList2
        null varList3 => t2
        not (first varList3 = t2) =>
          error "tubePlot: only one variable may be used"
      t1 := first varList1
      null varList2 =>
        null varList3 => t1
        not (first varList3 = t1) =>
          error "tubePlot: only one variable may be used"
        t1
      t2 := first varList2
      null varList3 =>
        not (t1 = t2) =>
          error "tubePlot: only one variable may be used"
        t1
      not (first varList3 = t1) or not (t2 = t1) =>
        error "tubePlot: only one variable may be used"
      t1

--% tubes: variable radius

    tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
             tRange:SEG SF,radFcn:SF -> SF,n:I,string:S) ==
      -- check value of n
      n < 3 => error "tubePlot: n should be at least 3"
      -- check string
      flag : B :=
        string = "closed" => true
```

```
  string = "open" => false
  error "tubePlot: last argument should be open or closed"
-- check variables
t := getVariable(x,y,z)
-- coordinate functions
xFunc := makeFloatFunction(x,t)
yFunc := makeFloatFunction(y,t)
zFunc := makeFloatFunction(z,t)
-- derivatives of coordinate functions
xp := differentiate(x,t)
yp := differentiate(y,t)
zp := differentiate(z,t)
-- derivative of arc length
sp := sqrt(xp ** 2 + yp ** 2 + zp ** 2)
-- coordinates of unit tangent vector
Tx := xp/sp; Ty := yp/sp; Tz := zp/sp
-- derivatives of coordinates of unit tangent vector
Txp := differentiate(Tx,t)
Typ := differentiate(Ty,t)
Tzp := differentiate(Tz,t)
-- K = curvature = length of curvature vector
K := sqrt(Txp ** 2 + Typ ** 2 + Tzp ** 2)
-- coordinates of principal normal vector
Nx := Txp / K; Ny := Typ / K; Nz := Tzp / K
-- functions SF->SF giving coordinates of principal normal vector
NxFunc := makeFloatFunction(Nx,t);
NyFunc := makeFloatFunction(Ny,t);
NzFunc := makeFloatFunction(Nz,t);
-- coordinates of binormal vector
Bx := Ty * Nz - Tz * Ny
By := Tz * Nx - Tx * Nz
Bz := Tx * Ny - Ty * Nx
-- functions SF -> SF giving coordinates of binormal vector
BxFunc := makeFloatFunction(Bx,t);
ByFunc := makeFloatFunction(By,t);
BzFunc := makeFloatFunction(Bz,t);
-- create Plot3D
parPlot := plot(xFunc,yFunc,zFunc,colorFcn,tRange)
tvals := first tValues parPlot
curvePts := first listBranches parPlot
cosSin := cosSinInfo n
loopList : L L Pt := nil()
while not null tvals repeat
  -- note that tvals and curvePts have the same number of elements
  tval := first tvals; tvals := rest tvals
  ctr := first curvePts; curvePts := rest curvePts
  pNormList : L SF :=
    [NxFunc tval,NyFunc tval,NzFunc tval,colorFcn tval]
  pNorm : Pt := point pNormList
  bNormList : L SF :=
```

```
          [BxFunc tval,ByFunc tval,BzFunc tval,colorFcn tval]
        bNorm : Pt := point bNormList
        lps := loopPoints(ctr,pNorm,bNorm,radFcn tval,cosSin)
        loopList := cons(lps,loopList)
      tube(parPlot,reverse_! loopList,flag)

    tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
             tRange:SEG SF,radFcn:SF -> SF,n:I) ==
      tubePlot(x,y,z,colorFcn,tRange,radFcn,n,"open")

--% tubes: constant radius

    project: (SF,SF) -> SF
    project(x,y) == x

    constantToUnaryFunction x == s +-> project(x,s)

    tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
             tRange:SEG SF,rad:SF,n:I,s:S) ==
      tubePlot(x,y,z,colorFcn,tRange,constantToUnaryFunction rad,n,s)

    tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
             tRange:SEG SF,rad:SF,n:I) ==
      tubePlot(x,y,z,colorFcn,tRange,rad,n,"open")
```

———-

— EXPRTUBE.dotabb —

```
"EXPRTUBE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRTUBE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"EXPRTUBE" -> "FS"
```

———-

6.31 package EXP3D Export3D

— Export3D.input —

```
)set break resume
)sys rm -f Export3D.output
)spool Export3D.output
)set message test on
)set message auto off
```

```
)clear all

--S 1 of 1
)show Export3D
--R Export3D  is a package constructor
--R Abbreviation for Export3D is EXP3D
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXP3D
--R
--R------------------------------ Operations --------------------------------
--R writeObj : (SubSpace(3,DoubleFloat),String) -> Void
--R
--E 1

)spool
)lisp (bye)
```

———-

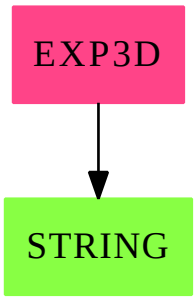
— Export3D.help —

```
========================================================================
Export3D examples
========================================================================

See Also:
o )show Export3D
```

———-

6.31.1 Export3D (EXP3D)



Exports:
writeObj

— package EXP3D Export3D —

```
)abbrev package EXP3D Export3D
++ Author: Martin Baker
++ Date: June, 2010
++ Description:
++ This package provides support for exporting SubSpace and
++ ThreeSpace structures to files.

EF ==> Expression Float
SBF ==> SegmentBinding Float
DF ==> DoubleFloat
I    ==> Integer
PI   ==> PositiveInteger
NNI  ==> NonNegativeInteger
STR ==> String

Export3D(): with

  writeObj:(SubSpace(3,DoubleFloat),String) -> Void
    ++ writes 3D SubSpace to a file in Wavefront (.OBJ) format

 == add
  import List List NNI

  -- return list of indexes
  -- assumes subnodes are leaves containing index
  faceIndex(subSp: SubSpace(3,DoubleFloat)):List NNI ==
    faceIndexList:List NNI := []
    for poly in children(subSp) repeat
      faceIndexList := cons(extractIndex(poly),faceIndexList)
    reverse faceIndexList

  -- called if this component contains a single polygon
  -- write out face information for Wavefront (.OBJ) 3D file format
  -- one face per line, represented by list of vertex indexes
  writePolygon(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
    faceIndexList:List NNI := []
    for curve in curves repeat
      faceIndexList := append(faceIndexList,faceIndex(curve))
    -- write out face information for Wavefront (.OBJ) 3D file format
    -- one face per line, represented by list of vertex indexes
    s:String := "f "
    for i in faceIndexList repeat
      s:=concat(s,string(i))$String
      s:=concat(s," ")$String
    writeLine!(f1,s)

  -- called if this component contains a mesh, the mesh will be rendered
```

```
-- as quad polygons.
-- write out face information for Wavefront (.OBJ) 3D file format
-- one face per line, represented by list of vertex indexes
writeMesh(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
  meshIndexArray:List List NNI := []
  for curve in curves repeat
    -- write out face information for Wavefront (.OBJ) 3D file format
    -- one face per line, represented by list of vertex indexes
    meshIndexArray := cons(faceIndex(curve),meshIndexArray)
  meshIndexArray := reverse meshIndexArray
  rowLength := #meshIndexArray
  colLength := #(meshIndexArray.1)
  for i in 1..(rowLength-1) repeat
    for j in 1..(colLength-1) repeat
      --s1:String := concat["row ",string(i)," col ",string(j)]
      --writeLine!(f1,s1)
      s:String := concat ["f ",string((meshIndexArray.i).j)," ",_
        string((meshIndexArray.(i+1)).j)," ",_
          string((meshIndexArray.(i+1)).(j+1))," ",_
            string((meshIndexArray.i).(j+1))]
      writeLine!(f1,s)

toString(d : DoubleFloat) : String ==
    unparse(convert(d)@InputForm)

-- this writes SubSpace geometry to Wavefront (.OBJ) 3D file format
-- reqires SubSpace to contain 3 or 4 dimensional points over DoubleFloat
-- to export a function plot try:
-- writeObj(subspace(makeObject(x*x-y*y,x=-1..1,y=-1..1)),"myfile.obj")
-- colour dimension is ignored
-- no normals or texture data is generated
writeObj(subSp: SubSpace(3,DoubleFloat), filename:String):Void ==
  f1:TextFile:=open(filename::FileName,"output")
  writeLine!(f1,"# mesh generated by axiom")
  -- write vertex data
  verts := pointData(subSp)
  for v in verts repeat
    #v < 3  => error "Can't write OBJ file from 2D points"
    writeLine!(f1,concat(["v ", toString(v.1), " ",_
               toString(v.2), " ", toString(v.3)])$String)
  for component in children(subSp) repeat
    curves := children(component)
    if #curves < 2 then
      sayTeX$Lisp "Can't write point or curve to OBJ file"
    --writeLine!(f1,"new component")
    if #curves > 1 then
      if numberOfChildren(curves.1) = 1 then writePolygon(f1,curves)
      if numberOfChildren(curves.1) > 1 then writeMesh(f1,curves)
  close! f1
```

———-

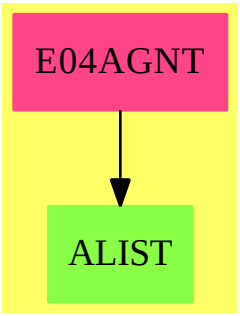
— **EXP3D.dotabb** —

```
"EXP3D" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXP3D"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"EXP3D" -> "STRING"
```

———-

6.32 package E04AGNT e04AgentsPackage

6.32.1 e04AgentsPackage (E04AGNT)



Exports:

changeNameToObjf	expenseOfEvaluation	finiteBound	linear?	linearMatrix
linearPart	nonLinearPart	optAttributes	quadratic?	simpleBounds?
sortConstraints	splitLinear	sumOfSquares	varList	variables

— **package E04AGNT e04AgentsPackage** —

```
)abbrev package E04AGNT e04AgentsPackage
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: June 1996
++ Basic Operations: simple? linear?, quadratic?, nonLinear?
++ Description:
++ \axiomType{e04AgentsPackage} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \axiomFun{measure} of an appropriate numerical optimization routine.

e04AgentsPackage(): E == I where
  MDF  ==> Matrix DoubleFloat
  VEDF  ==> Vector Expression DoubleFloat
```

```
EDF  ==> Expression DoubleFloat
EFI  ==> Expression Fraction Integer
PFI  ==> Polynomial Fraction Integer
FI  ==> Fraction Integer
F  ==> Float
DF  ==> DoubleFloat
OCDF  ==> OrderedCompletion DoubleFloat
LOCDF  ==> List OrderedCompletion DoubleFloat
LEDF  ==> List Expression DoubleFloat
PDF  ==> Polynomial DoubleFloat
LDF  ==> List DoubleFloat
INT  ==> Integer
NNI  ==> NonNegativeInteger
LS  ==> List Symbol
EF2  ==> ExpressionFunctions2
NOA  ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
LSA  ==> Record(lfn:LEDF, init:LDF)

E ==> with
  finiteBound:(LOCDF,DF) -> LDF
    ++ finiteBound(l,b) repaces all instances of an infinite entry in
    ++ \axiom{l} by a finite entry \axiom{b} or \axiom{-b}.
  sortConstraints:NOA -> NOA
    ++ sortConstraints(args) uses a simple bubblesort on the list of
    ++ constraints using the degree of the expression on which to sort.
    ++ Of course, it must match the bounds to the constraints.
  sumOfSquares:EDF -> Union(EDF,"failed")
    ++ sumOfSquares(f) returns either an expression for which the square is
    ++ the original function of "failed".
  splitLinear:EDF -> EDF
    ++ splitLinear(f) splits the linear part from an expression which it
    ++ returns.
  simpleBounds?:LEDF -> Boolean
    ++ simpleBounds?(l) returns true if the list of expressions l are
    ++ simple.
  linear?:LEDF -> Boolean
    ++ linear?(l) returns true if all the bounds l are either linear or
    ++ simple.
  linear?:EDF -> Boolean
    ++ linear?(e) tests if \axiom{e} is a linear function.
  linearMatrix:(LEDF, NNI) -> MDF
    ++ linearMatrix(l,n) returns a matrix of coefficients of the linear
    ++ functions in \axiom{l}.  If l is empty, the matrix has at least one
    ++ row.
  linearPart:LEDF -> LEDF
    ++ linearPart(l) returns the list of linear functions of \axiom{l}.
  nonLinearPart:LEDF -> LEDF
    ++ nonLinearPart(l) returns the list of non-linear functions of l.
  quadratic?:EDF -> Boolean
    ++ quadratic?(e) tests if \axiom{e} is a quadratic function.
```

```
    variables:LSA -> LS
      ++ variables(args) returns the list of variables in \axiom{args.lfn}
    varList:(EDF,NNI) -> LS
      ++ varList(e,n) returns a list of \axiom{n} indexed variables with name
      ++ as in \axiom{e}.
    changeNameToObjf:(Symbol,Result) -> Result
      ++ changeNameToObjf(s,r) changes the name of item \axiom{s} in \axiom{r}
      ++ to objf.
    expenseOfEvaluation:LSA -> F
      ++ expenseOfEvaluation(o) returns the intensity value of the
      ++ cost of evaluating the input set of functions.  This is in terms
      ++ of the number of ``operational units''.  It returns a value
      ++ in the range [0,1].
    optAttributes:Union(noa:NOA,lsa:LSA) -> List String
      ++ optAttributes(o) is a function for supplying a list of attributes
      ++ of an optimization problem.

  I ==> add

    import ExpertSystemToolsPackage, ExpertSystemContinuityPackage

    sumOfSquares2:EFI -> Union(EFI,"failed")
    nonLinear?:EDF -> Boolean
    finiteBound2:(OCDF,DF) -> DF
    functionType:EDF -> String

    finiteBound2(a:OCDF,b:DF):DF ==
      not finite?(a) =>
        positive?(a) => b
        -b
      retract(a)@DF

    finiteBound(l:LOCDF,b:DF):LDF == [finiteBound2(i,b) for i in l]

    sortConstraints(args:NOA):NOA ==
      Args := copy args
      c:LEDF := Args.cf
      l:LOCDF := Args.lb
      u:LOCDF := Args.ub
      m:INT := (# c) - 1
      n:INT := (# l) - m
      for j in m..1 by -1 repeat
        for i in 1..j repeat
          s:EDF := c.i
          t:EDF := c.(i+1)
          if linear?(t) and (nonLinear?(s) or quadratic?(s)) then
            swap!(c,i,i+1)$LEDF
            swap!(l,n+i-1,n+i)$LOCDF
            swap!(u,n+i-1,n+i)$LOCDF
      Args
```

```
changeNameToObjf(s:Symbol,r:Result):Result ==
  a := remove!(s,r)$Result
  a case Any =>
    insert!([objf@Symbol,a],r)$Result
    r
  r

sum(a:EDF,b:EDF):EDF == a+b

variables(args:LSA): LS == variables(reduce(sum,(args.lfn)))

sumOfSquares(f:EDF):Union(EDF,"failed") ==
  e := edf2efi(f)
  s:Union(EFI,"failed") := sumOfSquares2(e)
  s case EFI =>
    map(fi2df,s)$EF2(FI,DF)
  "failed"

sumOfSquares2(f:EFI):Union(EFI,"failed") ==
  p := retractIfCan(f)@Union(PFI,"failed")
  p case PFI =>
    r := squareFreePart(p)$PFI
    (p=r)@Boolean => "failed"
    tp := totalDegree(p)$PFI
    tr := totalDegree(r)$PFI
    t := tp quo tr
    found := false
    q := r
    for i in 2..t by 2 repeat
      s := q**2
      (s=p)@Boolean =>
        found := true
        leave
      q := r**i
    if found then
      q :: EFI
    else
      "failed"
  "failed"

splitLinear(f:EDF):EDF ==
  out := 0$EDF
  (l := isPlus(f)$EDF) case LEDF =>
    for i in l repeat
      if not quadratic? i then
        out := out + i
    out
  out
```

```
    edf2pdf(f:EDF):PDF == (retract(f)@PDF)$EDF

    varList(e:EDF,n:NNI):LS ==
      s := name(first(variables(edf2pdf(e))$PDF)$LS)$Symbol
      [subscript(s,[t::OutputForm]) for t in expand([1..n])$Segment(Integer)]

    functionType(f:EDF):String ==
      n := #(variables(f))$EDF
      p := (retractIfCan(f)@Union(PDF,"failed"))$EDF
      p case PDF =>
        d := totalDegree(p)$PDF
--        one?(n*d) => "simple"
        (n*d) = 1 => "simple"
--        one?(d) => "linear"
        (d = 1) => "linear"
        (d=2)@Boolean => "quadratic"
        "non-linear"
      "non-linear"

    simpleBounds?(l: LEDF):Boolean ==
      a := true
      for e in l repeat
        not (functionType(e) = "simple")@Boolean =>
          a := false
          leave
      a

    simple?(e:EDF):Boolean == (functionType(e) = "simple")@Boolean

    linear?(e:EDF):Boolean == (functionType(e) = "linear")@Boolean

    quadratic?(e:EDF):Boolean == (functionType(e) = "quadratic")@Boolean

    nonLinear?(e:EDF):Boolean == (functionType(e) = "non-linear")@Boolean

    linear?(l: LEDF):Boolean ==
      a := true
      for e in l repeat
        s := functionType(e)
        (s = "quadratic")@Boolean or (s = "non-linear")@Boolean =>
          a := false
          leave
      a

    simplePart(l:LEDF):LEDF == [i for i in l | simple?(i)]

    linearPart(l:LEDF):LEDF == [i for i in l | linear?(i)]

    nonLinearPart(l:LEDF):LEDF ==
      [i for i in l | not linear?(i) and not simple?(i)]
```

```
linearMatrix(l:LEDF, n:NNI):MDF ==
  empty?(l) => mat([],n)
  L := linearPart l
  M := zero(max(1,# L)$NNI,n)$MDF
  vars := varList(first(l)$LEDF,n)
  row:INT := 1
  for a in L repeat
    for j in monomials(edf2pdf(a))$PDF repeat
      col:INT := 1
      for c in vars repeat
        if ((first(variables(j)$PDF)$LS)=c)@Boolean then
          M(row,col):= first(coefficients(j)$PDF)$LDF
        col := col+1
    row := row + 1
  M

expenseOfEvaluation(o:LSA):F ==
  expenseOfEvaluation(vector(copy o.lfn)$VEDF)

optAttributes(o:Union(noa:NOA,lsa:LSA)):List String ==
  o case noa =>
    n := o.noa
    s1:String := "The object function is " functionType(n.fn)
    if empty?(n.lb) then
      s2:String := "There are no bounds on the variables"
    else
      s2:String := "There are simple bounds on the variables"
    c := n.cf
    if empty?(c) then
      s3:String := "There are no constraint functions"
    else
      t := #(c)
      lin := #(linearPart(c))
      nonlin := #(nonLinearPart(c))
      s3:String := "There are " string(lin)$String " linear and "_
                     string(nonlin)$String " non-linear constraints"
    [s1,s2,s3]
  l := o.lsa
  s:String := "non-linear"
  if linear?(l.lfn) then
    s := "linear"
  ["The object functions are " s]
```

———-

— E04AGNT.dotabb —

```
"E04AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=E04AGNT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"E04AGNT" -> "ALIST"
```

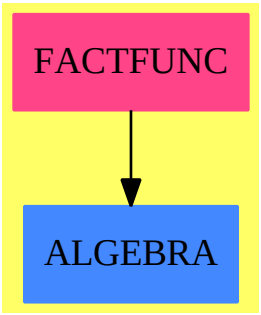
———————-

Chapter 7

Chapter F

7.1 package FACTFUNC FactoredFunctions

7.1.1 FactoredFunctions (FACTFUNC)



Exports:
log nthRoot

— package FACTFUNC FactoredFunctions —

```
)abbrev package FACTFUNC FactoredFunctions
++ Author: Manuel Bronstein
++ Date Created: 2 Feb 1988
++ Date Last Updated: 25 Jun 1990
++ Description:
++ computes various functions on factored arguments.
-- not visible to the user

FactoredFunctions(M:IntegralDomain): Exports == Implementation where
```

```
  N ==> NonNegativeInteger

  Exports ==> with
    nthRoot: (Factored M,N) -> Record(exponent:N,coef:M,radicand:List M)
      ++ nthRoot(f, n) returns \spad{(p, r, [r1,...,rm])} such that
      ++ the nth-root of f is equal to \spad{r * pth-root(r1 * ... * rm)},
      ++ where r1,...,rm are distinct factors of f,
      ++ each of which has an exponent smaller than p in f.
    log : Factored M -> List Record(coef:N, logand:M)
      ++ log(f) returns \spad{[(a1,b1),...,(am,bm)]} such that
      ++ the logarithm of f is equal to \spad{a1*log(b1) + ... + am*log(bm)}.

  Implementation ==> add
    nthRoot(ff, n) ==
      coeff:M       := 1
--      radi:List(M)  := (one? unit ff => empty(); [unit ff])
      radi:List(M)  := (((unit ff) = 1) => empty(); [unit ff])
      lf            := factors ff
      d:N :=
        empty? radi => gcd(concat(n, [t.exponent::N for t in lf]))::N
        1
      n             := n quo d
      for term in lf repeat
        qr    := divide(term.exponent::N quo d, n)
        coeff := coeff * term.factor ** qr.quotient
        not zero?(qr.remainder) =>
          radi := concat_!(radi, term.factor ** qr.remainder)
      [n, coeff, radi]

    log ff ==
      ans := unit ff
      concat([1, unit ff],
             [[term.exponent::N, term.factor] for term in factors ff])
```

— FACTFUNC.dotabb —

```
"FACTFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTFUNC"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FACTFUNC" -> "ALGEBRA"
```

7.2 package FR2 FactoredFunctions2

— **FactoredFunctions2.input** —

```
)set break resume
)sys rm -f FactoredFunctions2.output
)spool FactoredFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 6
double(x) == x + x
--R
--R                                                                    Type: Void
--E 1

--S 2 of 6
f := factor(720)
--R
--R
--R         4 2
--R   (2)  2 3 5
--R                                                  Type: Factored(Integer)
--E 2

--S 3 of 6
map(double,f)
--R
--R   Compiling function double with type Integer -> Integer
--R
--R           4 2
--R   (3)  2 4 6 10
--R                                                  Type: Factored(Integer)
--E 3

--S 4 of 6
makePoly(b) == x + b
--R
--R                                                                    Type: Void
--E 4

--S 5 of 6
g := map(makePoly,f)
--R
--R   Compiling function makePoly with type Integer -> Polynomial(Integer)
--R
--R
--R                      4       2
```

```
--R   (5)  (x + 1)(x + 2) (x + 3) (x + 5)
--R                                          Type: Factored(Polynomial(Integer))
--E 5

--S 6 of 6
nthFlag(g,1)
--R
--R
--R   (6)  "nil"
--R                                                   Type: Union("nil",...)
--E 6
)spool
)lisp (bye)
```

———-

— FactoredFunctions2.help —

```
========================================================================
FactoredFunctions2 examples
========================================================================

The FactoredFunctions2 package implements one operation, map, for
applying an operation to every base in a factored object and to the unit.

  double(x) == x + x
                        Type: Void

  f := factor(720)
     4 2
    2 3 5
                        Type: Factored Integer

Actually, the map operation used in this example comes from Factored
itself, since double takes an integer argument and returns an integer
result.

  map(double,f)
      4 2
    2 4 6 10
                        Type: Factored Integer

If we want to use an operation that returns an object that has a type
different from the operation's argument, the map in Factored cannot be
used and we use the one in FactoredFunctions2.

  makePoly(b) == x + b
                  4       2
    (x + 1)(x + 2) (x + 3) (x + 5)
```

```
                          Type: Factored Polynomial Integer

In fact, the "2" in the name of the package means that we might
be using factored objects of two different types.

  g := map(makePoly,f)

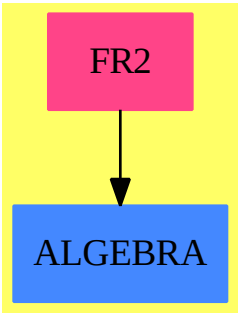
It is important to note that both versions of map destroy any information
known about the bases (the fact that they are prime, for instance).

The flags for each base are set to "nil" in the object returned by map.

  nthFlag(g,1)
    "nil"
                         Type: Union("nil",...)

See Also:
o )help Factored
o )show FactoredFunctions2
```

7.2.1 FactoredFunctions2 (FR2)



Exports:
map

— package FR2 FactoredFunctions2 —

```
)abbrev package FR2 FactoredFunctions2
++ Author: Robert S. Sutor
++ Date Created: 1987
++ Change History:
```

```
++ Basic Operations: map
++ Related Constructors: Factored
++ Also See:
++ AMS Classifications: 11A51, 11Y05
++ Keywords: map, factor
++ References:
++ Description:
++ \spadtype{FactoredFunctions2} contains functions that involve
++ factored objects whose underlying domains may not be the same.
++ For example, \spadfun{map} might be used to coerce an object of
++ type \spadtype{Factored(Integer)} to
++ \spadtype{Factored(Complex(Integer))}.

FactoredFunctions2(R, S): Exports == Implementation where
  R: IntegralDomain
  S: IntegralDomain

  Exports ==> with
    map: (R -> S, Factored R) -> Factored S
      ++ map(fn,u) is used to apply the function \userfun{fn} to every
      ++ factor of \spadvar{u}. The new factored object will have all its
      ++ information flags set to "nil". This function is used, for
      ++ example, to coerce every factor base to another type.

  Implementation ==> add
    map(func, f) ==
      func(unit f) *
             _*/[nilFactor(func(g.factor), g.exponent) for g in factors f]
```

———-

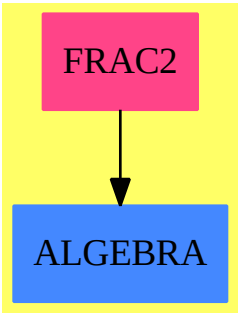
— FR2.dotabb —

```
"FR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FR2"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FR2" -> "ALGEBRA"
```

———-

7.3 package FRUTIL FactoredFunctionUtilities

7.3.1 FactoredFunctionUtilities (FRUTIL)



Exports:
mergeFactors refine

— package FRUTIL FactoredFunctionUtilities —

```
)abbrev package FRUTIL FactoredFunctionUtilities
++ Author:
++ Date Created:
++ Change History:
++ Basic Operations: refine, mergeFactors
++ Related Constructors: Factored
++ Also See:
++ AMS Classifications: 11A51, 11Y05
++ Keywords: factor
++ References:
++ Description:
++ \spadtype{FactoredFunctionUtilities} implements some utility
++ functions for manipulating factored objects.

FactoredFunctionUtilities(R): Exports == Implementation where
  R: IntegralDomain
  FR ==> Factored R

  Exports ==> with
    refine: (FR, R-> FR) -> FR
      ++ refine(u,fn) is used to apply the function \userfun{fn} to
      ++ each factor of \spadvar{u} and then build a new factored
      ++ object from the results.  For example, if \spadvar{u} were
      ++ created by calling \spad{nilFactor(10,2)} then
      ++ \spad{refine(u,factor)} would create a factored object equal
      ++ to that created by \spad{factor(100)} or
      ++ \spad{primeFactor(2,2) * primeFactor(5,2)}.
```

```
    mergeFactors: (FR,FR) -> FR
      ++ mergeFactors(u,v) is used when the factorizations of \spadvar{u}
      ++ and \spadvar{v} are known to be disjoint, e.g. resulting from a
      ++ content/primitive part split. Essentially, it creates a new
      ++ factored object by multiplying the units together and appending
      ++ the lists of factors.

  Implementation ==> add
    fg: FR
    func: R -> FR
    fUnion ==> Union("nil", "sqfr", "irred", "prime")
    FF     ==> Record(flg: fUnion, fctr: R, xpnt: Integer)

    mergeFactors(f,g) ==
      makeFR(unit(f)*unit(g),append(factorList f,factorList g))

    refine(f, func) ==
       u := unit(f)
       l: List FF := empty()
       for item in factorList f repeat
         fitem := func item.fctr
         u := u*unit(fitem) ** (item.xpnt :: NonNegativeInteger)
         if item.xpnt = 1 then
            l := concat(factorList fitem,l)
         else l := concat([[v.flg,v.fctr,v.xpnt*item.xpnt]
                           for v in factorList fitem],l)
       makeFR(u,l)
```

———-

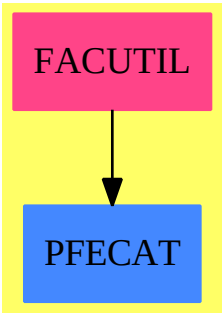
— FRUTIL.dotabb —

```
"FRUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRUTIL"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FRUTIL" -> "ALGEBRA"
```

———-

7.4 package FACUTIL FactoringUtilities

7.4.1 FactoringUtilities (FACUTIL)



Exports:

completeEval	degree	lowerPolynomial	normalDeriv	raisePolynomial
ran	variables			

— package FACUTIL FactoringUtilities —

```
)abbrev package FACUTIL FactoringUtilities
++ Author: Barry Trager
++ Date Created: March 12, 1992
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides utilities used by the factorizers
++ which operate on polynomials represented as univariate polynomials
++ with multivariate coefficients.

FactoringUtilities(E,OV,R,P) : C == T where
   E : OrderedAbelianMonoidSup
   OV : OrderedSet
   R : Ring
   P : PolynomialCategory(R,E,OV)

   SUP ==> SparseUnivariatePolynomial
   NNI ==> NonNegativeInteger
   Z   ==> Integer

   C == with
        completeEval   :      (SUP P,List OV,List R)        -> SUP R
```

```
      ++ completeEval(upoly, lvar, lval) evaluates the polynomial upoly
      ++ with each variable in lvar replaced by the corresponding value
      ++ in lval. Substitutions are done for all variables in upoly
      ++ producing a univariate polynomial over R.
    degree         :       (SUP P,List OV)              -> List NNI
      ++ degree(upoly, lvar) returns a list containing the maximum
      ++ degree for each variable in lvar.
    variables      :           SUP P                    -> List OV
      ++ variables(upoly) returns the list of variables for the coefficients
      ++ of upoly.
    lowerPolynomial:           SUP P                    -> SUP R
      ++ lowerPolynomial(upoly) converts upoly to be a univariate polynomial
      ++ over R. An error if the coefficients contain variables.
    raisePolynomial:           SUP R                    -> SUP P
      ++ raisePolynomial(rpoly) converts rpoly from a univariate polynomial
      ++ over r to be a univariate polynomial with polynomial coefficients.
    normalDeriv     :        (SUP P,Z)                  -> SUP P
      ++ normalDeriv(poly,i) computes the ith derivative of poly divided
      ++ by i!.
    ran        :                Z                      -> R
      ++ ran(k) computes a random integer between -k and k as a member of R.

T == add

  lowerPolynomial(f:SUP P) : SUP R ==
    zero? f => 0$SUP(R)
    monomial(ground leadingCoefficient f, degree f)$SUP(R) +
        lowerPolynomial(reductum f)

  raisePolynomial(u:SUP R) : SUP P ==
    zero? u => 0$SUP(P)
    monomial(leadingCoefficient(u)::P, degree u)$SUP(P) +
        raisePolynomial(reductum u)

  completeEval(f:SUP P,lvar:List OV,lval:List R) : SUP R ==
    zero? f => 0$SUP(R)
    monomial(ground eval(leadingCoefficient f,lvar,lval),degree f)$SUP(R) +
           completeEval(reductum f,lvar,lval)

  degree(f:SUP P,lvar:List OV) : List NNI ==
    coefs := coefficients f
    ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]

  variables(f:SUP P) : List OV ==
    "setUnion"/[variables cf for cf in coefficients f]

  if R has FiniteFieldCategory then
     ran(k:Z):R == random()$R
  else
     ran(k:Z):R == (random(2*k+1)$Z -k)::R
```

```
  -- Compute the normalized m derivative
     normalDeriv(f:SUP P,m:Z) : SUP P==
       (n1:Z:=degree f) < m => 0$SUP(P)
       n1=m => (leadingCoefficient f)::SUP(P)
       k:=binomial(n1,m)
       ris:SUP:=0$SUP(P)
       n:Z:=n1
       while n>= m repeat
         while n1>n repeat
           k:=(k*(n1-m)) quo n1
           n1:=n1-1
         ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
         f:=reductum f
         n:=degree f
       ris
```

———-

— FACUTIL.dotabb —

```
"FACUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACUTIL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FACUTIL" -> "PFECAT"
```

———-

7.5 package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

— FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.input —

```
)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)spool FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
--R
```

```
--R FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber(K: PseudoAlgebraicClosureOfA
--R Abbreviation for FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber is FACTEXT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTEXT
--R
--R------------------------------ Operations ------------------------------
--R factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R factorSqFree : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R
--E 1

)spool
)lisp (bye)
```

———-

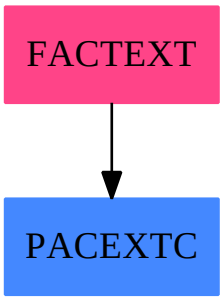
— **FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.help** —

```
=====================================================================
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber examples
=====================================================================

See Also:
o )show FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
```

———-

7.5.1 FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumbe (FACTEXT)



Exports:
factor factorSqFree

— package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber —

```
)abbrev package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber(K):Exports == Implementation where
  K:PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory
  SUP     ==> SparseUnivariatePolynomial
  Q       ==> PseudoAlgebraicClosureOfRationalNumber
  NNI     ==> NonNegativeInteger
  UP      ==> SUP(K)
  UPUP    ==> SUP(SUP(K))
  FACTRNQ ==> FactorisationOverPseudoAlgebraicClosureOfRationalNumber(Q)

  Exports ==> with
    factor: (UP,K) -> Factored UP
    factorSqFree: (UP,K) -> Factored UP

  Implementation ==> add
    up2Rat: UP -> SUP(Q)
    rat2up: SUP(Q) -> UP

    factRat: UP -> Factored UP
    liftPoly: (UP, K) -> UPUP

    liftDefPoly:  UP -> UPUP

    norm: (UP, K) -> UP

    factParPert: ( UP,K,K) -> Factored UP

    trans: (UP, K) -> UP

    swapCoefWithVar: ( UP , NNI) -> UPUP

    frRat2frUP: Factored SUP(Q) -> Factored UP

    factor(pol,a)==
      polSF:= squareFree pol
      reduce("*" , [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI) _
                     for fr in factorList polSF] , 1)

    factorSqFree(pol,a)==
      ratPol:SUP(Q)
      aa:Q
```

```
    ground? a =>
      aa:= retract(a)@Q
      ratPol:= up2Rat pol
      frRat2frUP factor(ratPol,aa)$FACTRNQ::Factored UP
    nPol:= norm(pol,a)
    ta:=previousTower a
    factN := factor( nPol , ta )
    lfactnPol:= factorList factN
    G:UP:=1
    L: Factored UP:= 1
    for fr in lfactnPol repeat
      G:= gcd( [ pol , fr.fctr ] )
      pol:= pol quo$UP G
      if one? fr.xpnt then
        L := L *  flagFactor( G, 1 ,"prime")$Factored(UP)
      else
        L := L *   factParPert( G, a, a )
    L

  factParPert(pol, a, b)==
    polt:=trans(pol,b)
    frpol:= factorList  factor(polt,a)
    sl:= [ fr.fctr for fr in frpol ]
    slt:= [ trans(p , -b) for p in sl ]
    nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
                   for p in slt for fr in frpol ]
    reduce("*" , nfrpol)

  frRat2frUP(fr)==
    frpol:= factorList fr
    sl:= [ fr.fctr for fr in frpol ]
    slt:= [ rat2up p  for p in sl ]
    nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
                  for p in slt for fr in frpol ]
    reduce("*" , nfrpol)

  up2Rat(pol)==
    zero?(pol) => 0
    d:=degree pol
    a:Q:= retract(leadingCoefficient pol)@Q
    monomial(a,d)$SUP(Q) + up2Rat(reductum pol)

  rat2up(pol)==
    zero?(pol) => 0
    d:=degree pol
    a:K:=(leadingCoefficient pol) :: K
    monomial(a,d)$UP + rat2up(reductum pol)

  trans(pol,a)==
    zero? pol => 0
```

```
      lc:=leadingCoefficient pol
      d:=degree pol
      lc*(monomial(1,1)$UP + monomial(-a ,0)$UP)**d + trans(reductum pol ,a)

    liftDefPoly(pol)==
      zero?(pol) => 0
      lc:= leadingCoefficient pol
      d:= degree pol
      monomial( monomial(lc,0)$UP , d )$UPUP + liftDefPoly reductum pol

    norm(pol,a)==
      lpol:=liftPoly(pol,a)
      defPol:=definingPolynomial a
      ldefPol:=liftDefPoly defPol
      resultant(ldefPol,lpol)

    swapCoefWithVar(coef,n)==
      ground? coef =>
        monomial( monomial( retract coef , n)$SUP(K) , 0)$UPUP
      lcoef:=leadingCoefficient(coef)
      d:=degree(coef)
      monomial(monomial(lcoef,n)$SUP(K),d)$UPUP+_
                 swapCoefWithVar(reductum coef,n )

    liftPoly(pol,a)==
      zero? pol => 0
      lcoef:=leadingCoefficient pol
      n:=degree pol
      liftCoef:= lift(lcoef,a)$K
      swapCoefWithVar(liftCoef , n) + liftPoly( reductum pol , a )
```

———-

— **FACTEXT.dotabb** —

```
"FACTEXT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTEXT"]
"PACEXTC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PACEXTC"]
"FACTEXT" -> "PACEXTC"
```

———-

7.6 package FACTRN FactorisationOverPseudoAlgebraic-ClosureOfRationalNumber

— **FactorisationOverPseudoAlgebraicClosureOfRationalNumber.input**

—

```
)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfRationalNumber.output
)spool FactorisationOverPseudoAlgebraicClosureOfRationalNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfRationalNumber
--R
--R FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K: PseudoAlgebraicClosureOfRationalN
--R Abbreviation for FactorisationOverPseudoAlgebraicClosureOfRationalNumber is FACTRN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTRN
--R
--R------------------------------ Operations --------------------------------
--R factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R factorSqFree : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R
--E 1

)spool
)lisp (bye)
```

———-

— **FactorisationOverPseudoAlgebraicClosureOfRationalNumber.help**

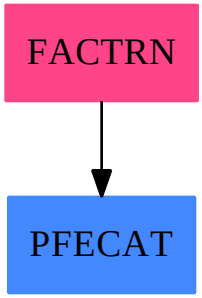
—

```
=====================================================================
FactorisationOverPseudoAlgebraicClosureOfRationalNumber examples
=====================================================================

See Also:
o )show FactorisationOverPseudoAlgebraicClosureOfRationalNumber
```

———-

7.6.1 FactorisationOverPseudoAlgebraicClosureOfRationalNumber (FACTRN)



Exports:
factor factorSqFree

— **package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber** —

```
)abbrev package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K):Exports ==
 Implementation where
  K:PseudoAlgebraicClosureOfRationalNumberCategory
  SUP ==> SparseUnivariatePolynomial
  Q   ==> Fraction Integer
  NNI ==> NonNegativeInteger
  UP ==> SUP(K)
  UPUP ==> SUP(SUP(K))

  Exports ==> with
    factor: (UP,K) -> Factored UP
    factorSqFree: (UP,K) -> Factored UP

  Implementation ==> add
    up2Rat: UP -> SUP(Q)
    rat2up: SUP(Q) -> UP

    factRat: UP -> Factored UP
    liftPoly: (UP, K) -> UPUP

    liftDefPoly:  UP -> UPUP
```

```
norm: (UP, K) -> UP

factParPert: ( UP,K,K) -> Factored UP

trans: (UP, K) -> UP

swapCoefWithVar: ( UP , NNI) -> UPUP

frRat2frUP: Factored SUP(Q) -> Factored UP

factor(pol,a)==
  polSF:= squareFree pol
  reduce("*" , _
   [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI) _
       for fr in factorList polSF] , 1)

factorSqFree(pol,a)==
  ratPol:SUP(Q)
  ground? a =>
    ratPol:= up2Rat pol
    frRat2frUP factor( ratPol )$RationalFactorize( SUP(Q) ) :: Factored UP
  nPol:= norm(pol,a)
  ta:=previousTower a
  factN := factor( nPol , ta )
  lfactnPol:= factorList factN
  G:UP:=1
  L: Factored UP:= 1
  for fr in lfactnPol repeat
    G:= gcd( [ pol , fr.fctr ] )
    pol:= pol quo$UP G
    if one? fr.xpnt then
      L := L *  flagFactor( G, 1 ,"prime")$Factored(UP)
    else
      L := L *   factParPert( G, a, a )
  L

factParPert(pol, a, b)==
  polt:=trans(pol,b)
  frpol:= factorList  factor(polt,a)
  sl:= [ fr.fctr for fr in frpol ]
  slt:= [ trans(p , -b) for p in sl ]
  nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
              for p in slt for fr in frpol ]
  reduce("*" , nfrpol)

frRat2frUP(fr)==
  frpol:= factorList fr
  sl:= [ fr.fctr for fr in frpol ]
  slt:= [ rat2up p  for p in sl ]
```

```
    nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
                for p in slt for fr in frpol ]
    reduce("*" , nfrpol)

  up2Rat(pol)==
    zero?(pol) => 0
    d:=degree pol
    a:Q:= retract(leadingCoefficient pol)@Q
    monomial(a,d)$SUP(Q) + up2Rat(reductum pol)

  rat2up(pol)==
    zero?(pol) => 0
    d:=degree pol
    a:K:=(leadingCoefficient pol) :: K
    monomial(a,d)$UP + rat2up(reductum pol)

  trans(pol,a)==
    zero? pol => 0
    lc:=leadingCoefficient pol
    d:=degree pol

    lc*( monomial(1,1)$UP + monomial(-a,0)$UP )**d + trans(reductum pol , a)

  liftDefPoly(pol)==
    zero?(pol) => 0
    lc:= leadingCoefficient pol
    d:= degree pol
    monomial( monomial(lc,0)$UP , d )$UPUP + liftDefPoly reductum pol

  norm(pol,a)==
    lpol:=liftPoly(pol,a)
    defPol:=definingPolynomial a
    ldefPol:=liftDefPoly defPol
    resultant(ldefPol,lpol)

  swapCoefWithVar(coef,n)==
    ground? coef =>
      monomial( monomial( retract coef , n)$SUP(K) , 0)$UPUP
    lcoef:=leadingCoefficient(coef)
    d:=degree(coef)
    monomial( monomial(lcoef,  n )$SUP(K) , d)$UPUP + _
              swapCoefWithVar( reductum coef, n )

  liftPoly(pol,a)==
    zero? pol => 0
    lcoef:=leadingCoefficient pol
    n:=degree pol
    liftCoef:= lift(lcoef,a)$K
    swapCoefWithVar(liftCoef , n) + liftPoly( reductum pol , a )
```

———-

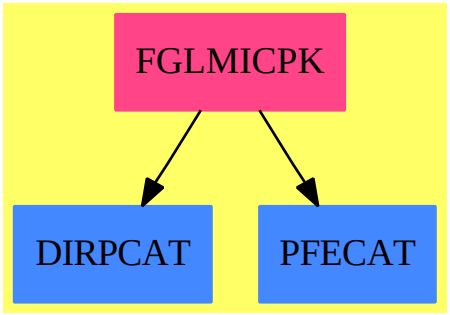
— FACTRN.dotabb —

```
"FACTRN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTRN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FACTRN" -> "PFECAT"
```

———-

7.7 package FGLMICPK FGLMIfCanPackage

7.7.1 FGLMIfCanPackage (FGLMICPK)



Exports:
fglmIfCan groebner zeroDimensional?

— package FGLMICPK FGLMIfCanPackage —

```
)abbrev package FGLMICPK FGLMIfCanPackage
++ Author: Marc Moreno Maza
++ Date Created: 08/02/1999
++ Date Last Updated: 08/02/1999
++ Description:
++ This is just an interface between several packages and domains.
++ The goal is to compute lexicographical Groebner bases
++ of sets of polynomial with type \spadtype{Polynomial R}
++ by the FGLM algorithm if this is possible (i.e.
++ if the input system generates a zero-dimensional ideal).

FGLMIfCanPackage(R,ls): Exports == Implementation where
  R: GcdDomain
  ls: List Symbol
```

```
V ==> OrderedVariableList ls
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
Q1 ==> Polynomial R
Q2 ==> HomogeneousDistributedMultivariatePolynomial(ls,R)
Q3 ==> DistributedMultivariatePolynomial(ls,R)
E2 ==> HomogeneousDirectProduct(#ls,NonNegativeInteger)
E3 ==>  DirectProduct(#ls,NonNegativeInteger)
poltopol ==> PolToPol(ls, R)
lingrobpack ==> LinGroebnerPackage(ls,R)
groebnerpack2 ==> GroebnerPackage(R,E2,V,Q2)
groebnerpack3 ==> GroebnerPackage(R,E3,V,Q3)
Exports ==  with

   zeroDimensional?: List(Q1) -> B
       ++ \axiom{zeroDimensional?(lq1)} returns true iff
       ++ \axiom{lq1} generates a zero-dimensional ideal
       ++ w.r.t. the variables of \axiom{ls}.
   fglmIfCan: List(Q1) -> Union(List(Q1), "failed")
       ++ \axiom{fglmIfCan(lq1)} returns the lexicographical Groebner
       ++ basis of \axiom{lq1} by using the FGLM strategy,
       ++ if \axiom{zeroDimensional?(lq1)} holds.
   groebner: List(Q1) -> List(Q1)
       ++ \axiom{groebner(lq1)} returns the lexicographical Groebner
       ++ basis of \axiom{lq1}. If \axiom{lq1} generates a zero-dimensional
       ++ ideal then the FGLM strategy is used, otherwise
       ++ the Sugar strategy is used.

Implementation == add

   zeroDim?(lq2: List Q2): Boolean ==
     lq2 := groebner(lq2)$groebnerpack2
     empty? lq2 => false
     #lq2 < #ls => false
     lv: List(V) := [(variable(s)$V)::V for s in ls]
     for q2 in lq2 while not empty?(lv) repeat
        m := leadingMonomial(q2)
        x := mainVariable(m)::V
        if ground?(leadingCoefficient(univariate(m,x))) then
             lv := remove(x, lv)
     empty? lv

   zeroDimensional?(lq1: List(Q1)): Boolean ==
     lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
     zeroDim?(lq2)

   fglmIfCan(lq1:List(Q1)): Union(List(Q1),"failed") ==
     lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
     lq2 := groebner(lq2)$groebnerpack2
```

```
      not zeroDim?(lq2) => "failed"::Union(List(Q1),"failed")
      lq3: List(Q3) := totolex(lq2)$lingrobpack
      lq1 := [dmpToP(q3)$poltopol for q3 in lq3]
      lq1::Union(List(Q1),"failed")

    groebner(lq1:List(Q1)): List(Q1) ==
      lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
      lq2 := groebner(lq2)$groebnerpack2
      not zeroDim?(lq2) =>
        lq3: List(Q3) := [pToDmp(q1)$poltopol for q1 in lq1]
        lq3 := groebner(lq3)$groebnerpack3
        [dmpToP(q3)$poltopol for q3 in lq3]
      lq3: List(Q3) := totolex(lq2)$lingrobpack
      [dmpToP(q3)$poltopol for q3 in lq3]
```

———-

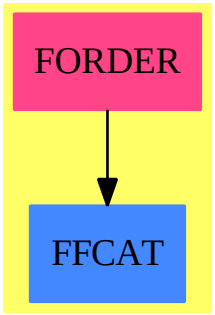
— FGLMICPK.dotabb —

```
"FGLMICPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FGLMICPK"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FGLMICPK" -> "DIRPCAT"
"FGLMICPK" -> "PFECAT"
```

———-

7.8 package FORDER FindOrderFinite

7.8.1 FindOrderFinite (FORDER)



Exports:
order

— **package FORDER FindOrderFinite** —

```
)abbrev package FORDER FindOrderFinite
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Finds the order of a divisor over a finite field

FindOrderFinite(F, UP, UPUP, R): Exports == Implementation where
  F   : Join(Finite, Field)
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R   : FunctionFieldCategory(F, UP, UPUP)

  Exports ==> with
    order: FiniteDivisor(F, UP, UPUP, R) -> NonNegativeInteger
      ++ order(x) \undocumented
  Implementation ==> add
    order d ==
      dd := d := reduce d
      for i in 1.. repeat
        principal? dd => return(i::NonNegativeInteger)
        dd := reduce(d + dd)
```

———-

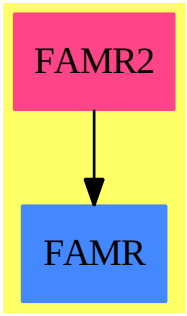
— **FORDER.dotabb** —

```
"FORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORDER"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FORDER" -> "FFCAT"
```

———-

7.9 package FAMR2 FiniteAbelianMonoidRingFunctions2

7.9.1 FiniteAbelianMonoidRingFunctions2 (FAMR2)



Exports:
map

— package FAMR2 FiniteAbelianMonoidRingFunctions2 —

```
)abbrev package FAMR2 FiniteAbelianMonoidRingFunctions2
++ Author: Martin Rubey
++ Description:
++ This package provides a mapping function for
++ \spadtype{FiniteAbelianMonoidRing}
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FiniteAbelianMonoidRingFunctions2(E: OrderedAbelianMonoid,
                                  R1: Ring,
                                  A1: FiniteAbelianMonoidRing(R1, E),
                                  R2: Ring,
                                  A2: FiniteAbelianMonoidRing(R2, E)) _
                                 : Exports == Implementation where
  Exports == with

    map: (R1 -> R2, A1) -> A2
      ++ \spad{map}(f, a) applies the map f to each coefficient in a. It is
      ++ assumed that f maps 0 to 0

  Implementation == add

    map(f: R1 -> R2, a: A1): A2 ==
      if zero? a then 0$A2
      else
        monomial(f leadingCoefficient a, degree a)$A2 + map(f, reductum a)
```

———-

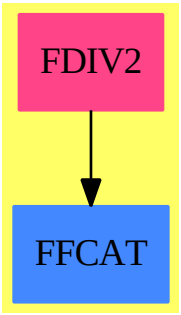
— FAMR2.dotabb —

```
"FAMR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FAMR2"]
"FAMR" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAMR"]
"FAMR2" -> "FAMR"
```

———-

7.10 package FDIV2 FiniteDivisorFunctions2

7.10.1 FiniteDivisorFunctions2 (FDIV2)



Exports:
map

— package FDIV2 FiniteDivisorFunctions2 —

```
)abbrev package FDIV2 FiniteDivisorFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 19 May 1993
++ Description:
++ Lift a map to finite divisors.

FiniteDivisorFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):
 Exports == Implementation where
  R1   : Field
  UP1  : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  F1   : FunctionFieldCategory(R1, UP1, UPUP1)
  R2   : Field
```

```
  UP2  : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2
  F2   : FunctionFieldCategory(R2, UP2, UPUP2)

  Exports ==> with
    map: (R1 -> R2, FiniteDivisor(R1, UP1, UPUP1, F1)) ->
                                       FiniteDivisor(R2, UP2, UPUP2, F2)
      ++ map(f,d) \undocumented{}

  Implementation ==> add
    import UnivariatePolynomialCategoryFunctions2(R1,UP1,R2,UP2)
    import FunctionFieldCategoryFunctions2(R1,UP1,UPUP1,F1,R2,UP2,UPUP2,F2)
    import FractionalIdealFunctions2(UP1, Fraction UP1, UPUP1, F1,
                                     UP2, Fraction UP2, UPUP2, F2)

    map(f, d) ==
      rec := decompose d
      divisor map(f, rec.principalPart) +
        divisor map((s:UP1):UP2 +-> map(f,s), rec.id)
```

———-

— **FDIV2.dotabb** —

```
"FDIV2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FDIV2"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FDIV2" -> "FFCAT"
```

———-

7.11 package FFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect

— **FiniteFieldFactorizationWithSizeParseBySideEffect.input** —

```
)set break resume
)sys rm -f FiniteFieldFactorizationWithSizeParseBySideEffect.output
)spool FiniteFieldFactorizationWithSizeParseBySideEffect.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
```

```
)show FiniteFieldFactorizationWithSizeParseBySideEffect
--R
--R FiniteFieldFactorizationWithSizeParseBySideEffect(K: FiniteFieldCategory,PolK: UnivariatePolynomialCateg
--R Abbreviation for FiniteFieldFactorizationWithSizeParseBySideEffect is FFFACTSE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFACTSE
--R
--R------------------------------ Operations ------------------------------
--R factor : PolK -> Factored(PolK)       irreducible? : PolK -> Boolean
--R factorCantorZassenhaus : (PolK,NonNegativeInteger) -> List(PolK)
--R factorSquareFree : PolK -> List(PolK)
--R factorUsingMusser : PolK -> Factored(PolK)
--R factorUsingYun : PolK -> Factored(PolK)
--R
--E 1

)spool
)lisp (bye)
```

———-

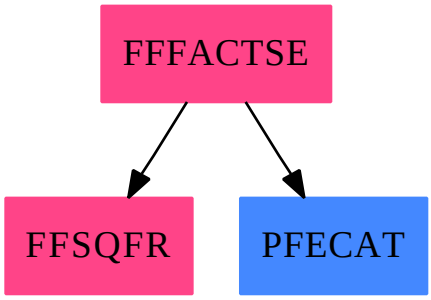
— **FiniteFieldFactorizationWithSizeParseBySideEffect.help** —

```
=====================================================================
FiniteFieldFactorizationWithSizeParseBySideEffect examples
=====================================================================

See Also:
o )show FiniteFieldFactorizationWithSizeParseBySideEffect
```

———-

7.11.1 FiniteFieldFactorizationWithSizeParseBySideEffect (FFFACTSE)



Exports:

factor	factorCantorZassenhaus	factorSquareFree
factorUsingMusser	factorUsingYun	irreducible?

— **package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect** —

```
)abbrev package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect
++ Author: Patrice Naudin, Claude Quitte, Kaj Laursen
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
++ It has been modified (very slitely) so that each time the "factor"
++ function is used, the variable related to the size of the field
++ over which the polynomial is factorized is reset. This is done in
++ order to be used with a "dynamic extension field" which size is not
++ fixed but set before calling the "factor" function and which is
++ parse by side effect to this package via the function "size".  See
++ the local function "initialize" of this package.
FiniteFieldFactorizationWithSizeParseBySideEffect(K : FiniteFieldCategory,
                        PolK : UnivariatePolynomialCategory(K)) : with

     factorSquareFree : PolK -> List(PolK)
     factorCantorZassenhaus : (PolK, NonNegativeInteger) -> List(PolK)
     factor : PolK -> Factored(PolK)
     factorUsingYun : PolK -> Factored(PolK)
     factorUsingMusser : PolK -> Factored(PolK)
     irreducible? : PolK -> Boolean

  == add

     import FiniteFieldSquareFreeDecomposition(K, PolK)
     p : NonNegativeInteger := characteristic()$K
     p' : NonNegativeInteger := p quo 2      -- used for odd p : (p-1)/2
     q : NonNegativeInteger := size()$K
     q' : NonNegativeInteger := q quo 2 -- used for odd q : (q-1)/2
     X : PolK := monomial(1, 1)
     primeKdim : NonNegativeInteger :=
         q_quo_p : NonNegativeInteger := q quo p ;  e : NonNegativeInteger := 1
         while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)
         e

     initialize(): Void() ==
        q : NonNegativeInteger := size()$K
        q' : NonNegativeInteger := q quo 2 -- used for odd q : (q-1)/2
primeKdim : NonNegativeInteger :=
          q_quo_p : NonNegativeInteger := q quo p ;  e:NonNegativeInteger := 1
          while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)
          e
```

```
exp(P : PolK, n : NonNegativeInteger, R : PolK) : PolK ==
   PP : PolK := P rem R ;  Q : PolK := 1
   repeat
      if odd?(n) then Q := Q * PP rem R
      (n := n quo 2) = 0 => leave
      PP := PP * PP rem  R
   return Q

pPowers(P : PolK) : PrimitiveArray(PolK) ==  -- P is monic
   n := degree(P)
   result : PrimitiveArray(PolK) := new(n, 1)
   result(1) := Qi := Q := exp(X, p, P)
   for i in 2 .. n-1 repeat (Qi := Qi*Q rem P ; result(i) := Qi)
   return result

pExp(Q : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
    Q' : PolK := 0
    while Q ^= 0 repeat
        Q':=Q' +primeFrobenius(leadingCoefficient(Q)) * Xpowers(degree(Q))
        Q := reductum(Q)
    return Q'

pTrace(Q : PolK, d : NonNegativeInteger, P : PolK,
       Xpowers : PrimitiveArray(PolK)) : PolK ==
    Q : PolK := Q rem P
    result : PolK := Q
    for i in 1 .. d-1 repeat result := Q + pExp(result, Xpowers)
    return result rem P

random(n : NonNegativeInteger) : PolK ==
   repeat
      if (deg := (random(n)$Integer)::NonNegativeInteger) > 0 then leave
   repeat
      if (x : K := random()$K) ^= 0 then leave
   result : PolK :=
      monomial(x, deg) + +/[monomial(random()$K, i) for i in 0 .. deg-1]
   return result

internalFactorCZ(P : PolK,          -- P monic-squarefree
      d:NonNegativeInteger, Xpowers:PrimitiveArray(PolK)) : List(PolK) ==

    listOfFactors : List(PolK) := [P]
    degree(P) = d => return listOfFactors
    result : List(PolK) := []
    pDim : NonNegativeInteger := d * primeKdim
    Q : PolK := P

    repeat
        G := pTrace(random(degree(Q)), pDim, Q, Xpowers)
```

```
        if p > 2 then G := exp(G, p', Q) - 1
        Q1 := gcd(G, Q) ;  d1 := degree(Q1)
        if d1 > 0 and d1 < degree(Q) then
            listOfFactors := rest(listOfFactors)
            if d1 = d then result := cons(Q1, result)
                      else listOfFactors := cons(Q1, listOfFactors)
            Q1 := Q quo Q1 ;  d1 := degree(Q1)
            if d1 = d then result := cons(Q1, result)
                      else listOfFactors := cons(Q1, listOfFactors)
            if empty?(listOfFactors) then leave
            Q := first(listOfFactors)
    return result

internalFactorSquareFree(P:PolK):List(PolK) ==   -- P is monic-squareFree
    degree(P) = 1 => [P]
    result : List(PolK) := []
    Xpowers : PrimitiveArray(PolK) := pPowers(P)
    S : PolK := Xpowers(1)
    for j in 1..primeKdim-1 repeat S := pExp(S, Xpowers)
    for i in 1 .. repeat  -- S = X**(q**i) mod P
        if degree(R := gcd(S - X, P)) > 0 then
            result := concat(internalFactorCZ(R, i, Xpowers), result)
            if degree (P) = degree (R) then return result
            P := P quo R
            if i >= degree(P) quo 2 then return cons(P, result)
            for j in 0 .. degree(P)-1 repeat Xpowers(j):=Xpowers(j) rem P
            S := S rem P
        else if i >= degree(P) quo 2 then return cons(P, result)
        for j in 1 .. primeKdim repeat S := pExp(S, Xpowers)

internalFactor(P:PolK, sqrfree:PolK -> Factored(PolK)) : Factored(PolK) ==
    result : Factored(PolK)
    if (d := minimumDegree(P)) > 0 then
        P := P quo monomial(1, d)
        result := primeFactor(X, d)
    else
        result := 1
    degree(P) = 0 => P * result
    if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
    degree(P) = 1 => lcP::PolK * primeFactor(P, 1) * result
    sqfP : Factored(PolK) := sqrfree(P)
    for x in factors(sqfP) repeat
        xFactors : List(PolK) := internalFactorSquareFree(x.factor)
        result:=result * */[primeFactor(Q, x.exponent) for Q in xFactors]
    return lcP::PolK * result

factorUsingYun(P : PolK) : Factored(PolK) == internalFactor(P, Yun)

factorUsingMusser(P : PolK) : Factored(PolK) == internalFactor(P, Musser)
```

```
factor(P : PolK) : Factored(PolK) ==
   initialize()
   factorUsingYun(P)

factorSquareFree(P : PolK) : List(PolK) ==
   degree(P) = 0 => []
   discriminant(P) = 0 => error("factorSquareFree : non quadratfrei")
   if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
   return internalFactorSquareFree(P)

factorCantorZassenhaus(P : PolK, d : NonNegativeInteger) : List(PolK) ==
   if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
   degree(P) = 1 => [P]
   return internalFactorCZ(P, d, pPowers(P))

qExp(Q : PolK, XqPowers : PrimitiveArray(PolK)) : PolK ==
   Q' : PolK := 0
   while Q ^= 0 repeat
      Q' := Q' + leadingCoefficient(Q) * XqPowers(degree(Q))
      Q := reductum(Q)
   return Q'

qPowers (Xq:PolK, P:PolK) : PrimitiveArray(PolK) ==  -- Xq = X**q mod P
   n := degree(P)
   result : PrimitiveArray(PolK) := new(n, 1)
   result(1) := Q := Xq
   for i in 2 .. n-1 repeat (Q := Q*Xq rem P ; result(i) := Q)
   return result

discriminantTest?(P : PolK) : Boolean ==
    (delta : K := discriminant(P)) = 0 => true
    StickelbergerTest : Boolean := (delta ** q' = 1) = even?(degree(P))
    return StickelbergerTest

evenCharacteristicIrreducible?(P : PolK) : Boolean ==
    (n := degree(P)) = 0 => false
    n = 1 => true
    degree(gcd(P, D(P))) > 0 => false
    if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
    S : PolK := exp(X, q, P)
    if degree(gcd(S - X, P)) > 0 then
       return false
    if n < 4 then return true
    maxDegreeToTest : NonNegativeInteger := n quo 2
    XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
    for i in 2 .. maxDegreeToTest repeat
       S := qExp(S, XqPowers)
       if degree(gcd(S - X, P)) > 0 then
          return false
    return true
```

```
    oddCharacteristicIrreducible?(P : PolK) : Boolean ==
        (n := degree(P)) = 0 => false
        n = 1 => true
        discriminantTest?(P) => false
        if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
        S : PolK := exp(X, q, P)
        if degree(gcd(S - X, P)) > 0 then
           return false
        if n < 6  then return true
        maxDegreeToTest : NonNegativeInteger := n quo 3
        XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
        for i in 2 .. maxDegreeToTest repeat
           S := qExp(S, XqPowers)
           if degree(gcd(S - X, P)) > 0 then
              return false
        return true
    if p = 2 then
        irreducible?(P : PolK) : Boolean == evenCharacteristicIrreducible?(P)
    else
        irreducible?(P : PolK) : Boolean == oddCharacteristicIrreducible?(P)
```

———-

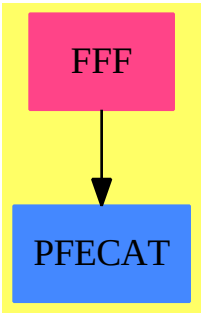
— FFFACTSE.dotabb —

```
"FFFACTSE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFACTSE"]
"FFSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFACTSE" -> "FFSQFR"
"FFFACTSE" -> "PFECAT"
```

———-

7.12 package FFF FiniteFieldFunctions

7.12.1 FiniteFieldFunctions (FFF)



Exports:

createLowComplexityNormalBasis	createLowComplexityTable	createMultiplicationMatrix
createMultiplicationTable	createZechTable	sizeMultiplication

— **package FFF FiniteFieldFunctions** —

```
)abbrev package FFF FiniteFieldFunctions
++ Author: J. Grabmeier, A. Scheerhorn
++ Date Created: 21 March 1991
++ Date Last Updated: 31 March 1991
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: finite field, Zech logarithm, Jacobi logarithm, normal basis
++ References:
++  Lidl, R. & Niederreiter, H., "Finite Fields",
++   Encycl. of Math. 20, Addison-Wesley, 1983
++  J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++   AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldFunctions(GF) is a package with functions
++ concerning finite extension fields of the finite ground field GF,
++ e.g. Zech logarithms.

FiniteFieldFunctions(GF): Exports == Implementation where
  GF  : FiniteFieldCategory  -- the ground field

  PI   ==> PositiveInteger
  NNI  ==> NonNegativeInteger
  I    ==> Integer
  SI   ==> SingleInteger
  SUP  ==> SparseUnivariatePolynomial GF
```

```
V    ==> Vector
M    ==> Matrix
L    ==> List
OUT  ==> OutputForm
SAE  ==> SimpleAlgebraicExtension
ARR  ==> PrimitiveArray(SI)
TERM ==> Record(value:GF,index:SI)
MM   ==> ModMonic(GF,SUP)
PF   ==> PrimeField

Exports ==> with

    createZechTable: SUP -> ARR
      ++ createZechTable(f) generates a Zech logarithm table for the cyclic
      ++ group representation of a extension of the ground field by the
      ++ primitive polynomial f(x), i.e. \spad{Z(i)},
      ++ defined by x**Z(i) = 1+x**i is stored at index i.
      ++ This is needed in particular
      ++ to perform addition of field elements in finite fields represented
      ++ in this way. See \spadtype{FFCGP}, \spadtype{FFCGX}.
    createMultiplicationTable: SUP -> V L TERM
      ++ createMultiplicationTable(f) generates a multiplication
      ++ table for the normal basis of the field extension determined
      ++ by f. This is needed to perform multiplications
      ++ between elements represented as coordinate vectors to this basis.
      ++ See \spadtype{FFNBP}, \spadtype{FFNBX}.
    createMultiplicationMatrix: V L TERM -> M GF
      ++ createMultiplicationMatrix(m) forms the multiplication table
      ++ m into a matrix over the ground field.
      -- only useful for the user to visualise the multiplication table
      -- in a nice form
    sizeMultiplication: V L TERM -> NNI
      ++ sizeMultiplication(m) returns the number of entries
      ++ of the multiplication table m.
      -- the time of the multiplication of field elements depends
      -- on this size
    createLowComplexityTable: PI -> Union(Vector List TERM,"failed")
      ++ createLowComplexityTable(n) tries to find
      ++ a low complexity normal basis of degree n over GF
      ++ and returns its multiplication matrix
      ++ Fails, if it does not find a low complexity basis
    createLowComplexityNormalBasis: PI -> Union(SUP, V L TERM)
      ++ createLowComplexityNormalBasis(n) tries to find a
      ++ a low complexity normal basis of degree n over GF
      ++ and returns its multiplication matrix
      ++ If no low complexity basis is found it calls
      ++ \axiomFunFrom{createNormalPoly}{FiniteFieldPolynomialPackage}(n)
      ++ to produce a normal polynomial of degree n over GF

Implementation ==> add
```

```
    createLowComplexityNormalBasis(n) ==
      (u:=createLowComplexityTable(n)) case "failed" =>
        createNormalPoly(n)$FiniteFieldPolynomialPackage(GF)
      u::(V L TERM)

-- try to find a low complexity normal basis multiplication table
-- of the field of extension degree n
-- the algorithm is from:
-- Wassermann A., Konstruktion von Normalbasen,
-- Bayreuther Mathematische Schriften 31 (1989),1-9.

    createLowComplexityTable(n) ==
      q:=size()$GF
      -- this algorithm works only for prime fields
      p:=characteristic()$GF
      -- search of a suitable parameter k
      k:NNI:=0
      for i in 1..n-1  while (k=0) repeat
        if prime?(i*n+1) and not(p = (i*n+1)) then
          primitive?(q::PF(i*n+1))$PF(i*n+1) =>
              a:NNI:=1
              k:=i
              t1:PF(k*n+1):=(q::PF(k*n+1))**n
          gcd(n,a:=discreteLog(q::PF(n*i+1))$PF(n*i+1))$I = 1 =>
              k:=i
              t1:=primitiveElement()$PF(k*n+1)**n
      k = 0 => "failed"
      -- initialize some start values
      multmat:M PF(p):=zero(n,n)
      p1:=(k*n+1)
      pkn:=q::PF(p1)
      t:=t1 pretend PF(p1)
      if odd?(k) then
          jt:I:=(n quo 2)+1
          vt:I:=positiveRemainder((k-a) quo 2,k)+1
        else
          jt:I:=1
          vt:I:=(k quo 2)+1
      -- compute matrix
      vec:Vector I:=zero(p1 pretend NNI)
      for x in 1..k repeat
        for l in 1..n repeat
          vec.((t**(x-1) * pkn**(l-1)) pretend Integer+1):=_
                                              positiveRemainder(l,p1)
      lvj:M I:=zero(k::NNI,n)
      for v in 1..k repeat
        for j in 1..n repeat
          if (j^=jt) or (v^=vt) then
```

```
        help:PF(p1):=t**(v-1)*pkn**(j-1)+1@PF(p1)
        setelt(lvj,v,j,vec.(help pretend I +1))
    for j in 1..n repeat
      if j^=jt then
        for v in 1..k repeat
          lvjh:=elt(lvj,v,j)
          setelt(multmat,j,lvjh,elt(multmat,j,lvjh)+1)
    for i in 1..n repeat
      setelt(multmat,jt,i,positiveRemainder(-k,p)::PF(p))
    for v in 1..k repeat
      if v^=vt then
        lvjh:=elt(lvj,v,jt)
        setelt(multmat,jt,lvjh,elt(multmat,jt,lvjh)+1)
    -- multmat
    m:=nrows(multmat)$(M PF(p))
    multtable:V L TERM:=new(m,nil()$(L TERM))$(V L TERM)
    for i in 1..m repeat
      l:L TERM:=nil()$(L TERM)
      v:V PF(p):=row(multmat,i)
      for j in (1::I)..(m::I) repeat
        if (v.j ^= 0) then
          -- take -v.j to get trace 1 instead of -1
          term:TERM:=[(convert(-v.j)@I)::GF,(j-2) pretend SI]$TERM
          l:=cons(term,l)$(L TERM)
      qsetelt_!(multtable,i,copy l)$(V L TERM)
    multtable

  sizeMultiplication(m) ==
    s:NNI:=0
    for i in 1..#m repeat
      s := s + #(m.i)
    s

  createMultiplicationTable(f:SUP) ==
    sizeGF:NNI:=size()$GF -- the size of the ground field
    m:PI:=degree(f)$SUP pretend PI
    m=1 =>
      [[[-coefficient(f,0)$SUP,(-1)::SI]$TERM]$(L TERM)]::(V L TERM)
    m1:I:=m-1
    -- initialize basis change matrices
    setPoly(f)$MM
    e:=reduce(monomial(1,1)$SUP)$MM ** sizeGF
    w:=1$MM
    qpow:PrimitiveArray(MM):=new(m,0)
    qpow.0:=1$MM
    for i in 1..m1 repeat
      qpow.i:=(w:=w*e)
    -- qpow.i = x**(i*q)
    qexp:PrimitiveArray(MM):=new(m,0)
    qexp.0:=reduce(monomial(1,1)$SUP)$MM
```

```
  mat:M GF:=zero(m,m)$(M GF)
  qsetelt_!(mat,2,1,1$GF)$(M GF)
  h:=qpow.1
  qexp.1:=h
  setColumn_!(mat,2,Vectorise(h)$MM)$(M GF)
  for i in 2..m1 repeat
    g:=0$MM
    while h ^= 0 repeat
      g:=g + leadingCoefficient(h) * qpow.degree(h)$MM
      h:=reductum(h)$MM
    qexp.i:=g
    setColumn_!(mat,i+1,Vectorise(h:=g)$MM)$(M GF)
  -- loop invariant: qexp.i = x**(q**i)
  mat1:=inverse(mat)$(M GF)
  mat1 = "failed" =>
    error "createMultiplicationTable: polynomial must be normal"
  mat:=mat1 :: (M GF)
  -- initialize multiplication table
  multtable:V L TERM:=new(m,nil()$(L TERM))$(V L TERM)
  for i in 1..m repeat
    l:L TERM:=nil()$(L TERM)
    v:V GF:=mat *$(M GF) Vectorise(qexp.(i-1) *$MM qexp.0)$MM
    for j in (1::SI)..(m::SI) repeat
      if (v.j ^= 0$GF) then
        term:TERM:=[(v.j),j-(2::SI)]$TERM
        l:=cons(term,l)$(L TERM)
    qsetelt_!(multtable,i,copy l)$(V L TERM)
  multtable


createZechTable(f:SUP) ==
  sizeGF:NNI:=size()$GF -- the size of the ground field
  m:=degree(f)$SUP::PI
  qm1:SI:=(sizeGF ** m -1) pretend SI
  zechlog:ARR:=new(((sizeGF ** m + 1) quo 2)::NNI,-1::SI)$ARR
  helparr:ARR:=new(sizeGF ** m::NNI,0$SI)$ARR
  primElement:=reduce(monomial(1,1)$SUP)$SAE(GF,SUP,f)
  a:=primElement
  for i in 1..qm1-1 repeat
    helparr.(lookup(a -$SAE(GF,SUP,f) 1$SAE(GF,SUP,f)_
       )$SAE(GF,SUP,f)):=i::SI
    a:=a * primElement
  characteristic() = 2 =>
    a:=primElement
    for i in 1..(qm1 quo 2) repeat
      zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
      a:=a * primElement
    zechlog
  a:=1$SAE(GF,SUP,f)
  for i in 0..((qm1-2) quo 2) repeat
```

```
        zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
        a:=a * primElement
      zechlog

    createMultiplicationMatrix(m) ==
      n:NNI:=#m
      mat:M GF:=zero(n,n)$(M GF)
      for i in 1..n repeat
        for t in m.i repeat
          qsetelt_!(mat,i,t.index+2,t.value)
      mat
```

———-

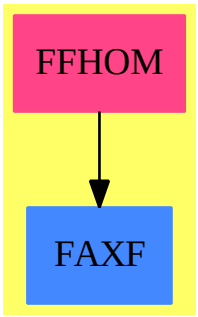
— **FFF.dotabb** —

```
"FFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFF" -> "PFECAT"
```

———-

7.13 package FFHOM FiniteFieldHomomorphisms

7.13.1 FiniteFieldHomomorphisms (FFHOM)



Exports:
coerce

— **package FFHOM FiniteFieldHomomorphisms** —

```
)abbrev package FFHOM FiniteFieldHomomorphisms
```

```
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated:
++ Basic Operations:
++ Related Constructors: FiniteFieldCategory, FiniteAlgebraicExtensionField
++ Also See:
++ AMS Classifications:
++ Keywords: finite field, homomorphism, isomorphism
++ References:
++  R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++  Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++  J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++  AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of
++ elements between the fields F1 and F2, which both must be
++ finite simple algebraic extensions of the finite ground field GF.

FiniteFieldHomomorphisms(F1,GF,F2): Exports == Implementation where
  F1: FiniteAlgebraicExtensionField(GF)
  GF: FiniteFieldCategory
  F2: FiniteAlgebraicExtensionField(GF)
 -- the homorphism can only convert elements w.r.t. the last extension .
  -- Adding a function 'groundField()' which returns the groundfield of GF
  -- as a variable of type FiniteFieldCategory in the new compiler, one
  -- could build up 'convert' recursively to get an homomorphism w.r.t
  -- the whole extension.

  I   ==> Integer
  NNI ==> NonNegativeInteger
  SI  ==> SingleInteger
  PI  ==> PositiveInteger
  SUP ==> SparseUnivariatePolynomial
  M   ==> Matrix GF
  FFP ==> FiniteFieldExtensionByPolynomial
  FFPOL2 ==> FiniteFieldPolynomialPackage2
  FFPOLY ==> FiniteFieldPolynomialPackage
  OUT ==> OutputForm

  Exports ==> with

    coerce: F1  ->  F2
      ++ coerce(x) is the homomorphic image of x from
      ++ F1 in F2. Thus coerce is a
      ++ field homomorphism between the fields extensions
      ++ F1 and F2 both over ground field GF
      ++ (the second argument to the package).
      ++ Error: if the extension degree of F1 doesn't divide
      ++ the extension degree of F2.
      ++ Note that the other coercion function in the
```

```
      ++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.

    coerce: F2  ->  F1
      ++ coerce(x) is the homomorphic image of x from
      ++ F2 in F1, where coerce is a
      ++ field homomorphism between the fields extensions
      ++ F2 and F1 both over ground field GF
      ++ (the second argument to the package).
      ++ Error: if the extension degree of F2 doesn't divide
      ++ the extension degree of F1.
      ++ Note that the other coercion function in the
      ++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.
    -- coerce(coerce(x:F1)@F2)@F1 = x and coerce(coerce(y:F2)@F1)@F2 = y

  Implementation ==> add

-- global variables ================================================

    degree1:NNI:= extensionDegree()$F1
    degree2:NNI:= extensionDegree()$F2
    -- the degrees of the last extension

    -- a necessary condition for the one field being an subfield of
    -- the other one is, that the respective extension degrees are
    -- multiples
    if max(degree1,degree2) rem min(degree1,degree2) ^= 0 then
      error "FFHOM: one extension degree must divide the other one"

    conMat1to2:M:= zero(degree2,degree1)$M
    -- conversion Matix for the conversion direction F1 -> F2
    conMat2to1:M:= zero(degree1,degree2)$M
    -- conversion Matix for the conversion direction F2 -> F1

    repType1:=representationType()$F1
    repType2:=representationType()$F2
    -- the representation types of the fields

    init?:Boolean:=true
    -- gets false after initialization

    defPol1:=definingPolynomial()$F1
    defPol2:=definingPolynomial()$F2
    -- the defining polynomials of the fields


-- functions ===============================================================


    compare: (SUP GF,SUP GF) -> Boolean
    -- compares two polynomials
```

```
convertWRTsameDefPol12: F1  ->  F2
convertWRTsameDefPol21: F2  ->  F1
-- homomorphism if the last extension of F1 and F2 was build up
-- using the same defining polynomials

convertWRTdifferentDefPol12: F1  ->  F2
convertWRTdifferentDefPol21: F2  ->  F1
-- homomorphism if the last extension of F1 and F2 was build up
-- with different defining polynomials

initialize: () -> Void
-- computes the conversion matrices

compare(g:(SUP GF),f:(SUP GF)) ==
  degree(f)$(SUP GF)  >$NNI degree(g)$(SUP GF) => true
  degree(f)$(SUP GF) <$NNI degree(g)$(SUP GF) => false
  equal:Integer:=0
  for i in degree(f)$(SUP GF)..0 by -1 while equal=0 repeat
    not zero?(coefficient(f,i)$(SUP GF))$GF and _
         zero?(coefficient(g,i)$(SUP GF))$GF => equal:=1
    not zero?(coefficient(g,i)$(SUP GF))$GF and _
         zero?(coefficient(f,i)$(SUP GF))$GF => equal:=(-1)
    (f1:=lookup(coefficient(f,i)$(SUP GF))$GF) >$PositiveInteger _
     (g1:=lookup(coefficient(g,i)$(SUP GF))$GF) =>  equal:=1
    f1 <$PositiveInteger g1 => equal:=(-1)
  equal=1 => true
  false

initialize() ==
  -- 1) in the case of equal def. polynomials initialize is called only
  --  if one of the rep. types is "normal" and the other one is "polynomial"
  --  we have to compute the basis change matrix 'mat', which i-th
  --  column are the coordinates of a**(q**i), the i-th component of
  --  the normal basis ('a' the root of the def. polynomial and q the
  --  size of the groundfield)
  defPol1 =$(SUP GF) defPol2 =>
    -- new code using reducedQPowers
    mat:=zero(degree1,degree1)$M
    arr:=reducedQPowers(defPol1)$FFPOLY(GF)
    for i in 1..degree1 repeat
      setColumn_!(mat,i,vectorise(arr.(i-1),degree1)$SUP(GF))$M
      -- old code
      -- here one of the representation types must be "normal"
      --a:=basis()$FFP(GF,defPol1).2  -- the root of the def. polynomial
      --setColumn_!(mat,1,coordinates(a)$FFP(GF,defPol1))$M
      --for i in 2..degree1 repeat
      --  a:= a **$FFP(GF,defPol1) size()$GF
      --  setColumn_!(mat,i,coordinates(a)$FFP(GF,defPol1))$M
      --for the direction "normal" -> "polynomial" we have to multiply the
```

```
    -- coordinate vector of an element of the normal basis field with
    -- the matrix 'mat'. In this case 'mat' is the correct conversion
    -- matrix for the conversion of F1 to F2, its inverse the correct
    -- inversion matrix for the conversion of F2 to F1
  repType1 = "normal" =>  -- repType2 = "polynomial"
    conMat1to2:=copy(mat)
    conMat2to1:=copy(inverse(mat)$M :: M)
    --we finish the function for one case, hence reset initialization flag
    init? := false
    void()$Void
    -- print("'normal' <=> 'polynomial' matrices initialized"::OUT)
  -- in the other case we have to change the matrices
  -- repType2 = "normal" and repType1 = "polynomial"
  conMat2to1:=copy(mat)
  conMat1to2:=copy(inverse(mat)$M :: M)
  -- print("'normal' <=> 'polynomial' matrices initialized"::OUT)
  --we finish the function for one case, hence reset initialization flag
  init? := false
  void()$Void
-- 2) in the case of different def. polynomials we have to order the
--    fields to get the same isomorphism, if the package is called with
--    the fields F1 and F2 swapped.
dPbig:= defPol2
rTbig:= repType2
dPsmall:= defPol1
rTsmall:= repType1
degbig:=degree2
degsmall:=degree1
if compare(defPol2,defPol1) then
  degsmall:=degree2
  degbig:=degree1
  dPbig:= defPol1
  rTbig:= repType1
  dPsmall:= defPol2
  rTsmall:= repType2
-- 3) in every case we need a conversion between the polynomial
--  represented fields. Therefore we compute 'root' as a root of the
--  'smaller' def. polynomial in the 'bigger' field.
--  We compute the matrix 'matsb', which i-th column are the coordinates
--  of the (i-1)-th power of root, i=1..degsmall. Multiplying a
--  coordinate vector of an element of the 'smaller' field by this
--  matrix, we got the coordinates of the corresponding element in the
--  'bigger' field.
-- compute the root of dPsmall in the 'big' field
root:=rootOfIrreduciblePoly(dPsmall)$FFPOL2(FFP(GF,dPbig),GF)
-- set up matrix for polynomial conversion
matsb:=zero(degbig,degsmall)$M
qsetelt_!(matsb,1,1,1$GF)$M
a:=root
for i in 2..degsmall repeat
```

```
  setColumn_!(matsb,i,coordinates(a)$FFP(GF,dPbig))$M
  a := a *$FFP(GF,dPbig) root
--  the conversion from 'big' to 'small': we can't invert matsb
--  directly, because it has degbig rows and degsmall columns and
--  may be no square matrix. Therfore we construct a square matrix
--  mat from degsmall linear independent rows of matsb and invert it.
--  Now we get the conversion matrix 'matbs' for the conversion from
--  'big' to 'small' by putting the columns of mat at the indices
--  of the linear independent rows of matsb to columns of matbs.
ra:I:=1   -- the rank
mat:M:=transpose(row(matsb,1))$M -- has already rank 1
rowind:I:=2
iVec:Vector I:=new(degsmall,1$I)$(Vector I)
while ra < degsmall repeat
  if rank(vertConcat(mat,transpose(row(matsb,rowind))$M)$M)$M > ra then
    mat:=vertConcat(mat,transpose(row(matsb,rowind))$M)$M
    ra:=ra+1
    iVec.ra := rowind
  rowind:=rowind + 1
mat:=inverse(mat)$M :: M
matbs:=zero(degsmall,degbig)$M
for i in 1..degsmall repeat
  setColumn_!(matbs,iVec.i,column(mat,i)$M)$M
-- print(matsb::OUT)
-- print(matbs::OUT)
-- 4) if the 'bigger' field is "normal" we have to compose the
--  polynomial conversion with a conversion from polynomial to normal
--  between the FFP(GF,dPbig) and FFNBP(GF,dPbig) the 'bigger'
--  field. Therefore we compute a conversion matrix 'mat' as in 1)
--  Multiplying with the inverse of 'mat' yields the desired
--  conversion from polynomial to normal. Multiplying this matrix by
--  the above computed 'matsb' we got the matrix for converting form
--  'small polynomial' to 'big normal'.
-- set up matrix 'mat' for polynomial to normal
if rTbig = "normal" then
  arr:=reducedQPowers(dPbig)$FFPOLY(GF)
  mat:=zero(degbig,degbig)$M
  for i in 1..degbig repeat
    setColumn_!(mat,i,vectorise(arr.(i-1),degbig)$SUP(GF))$M
  -- old code
  --a:=basis()$FFP(GF,dPbig).2  -- the root of the def.Polynomial
  --setColumn_!(mat,1,coordinates(a)$FFP(GF,dPbig))$M
  --for i in 2..degbig repeat
  --  a:= a **$FFP(GF,dPbig) size()$GF
  --  setColumn_!(mat,i,coordinates(a)$FFP(GF,dPbig))$M
  -- print(inverse(mat)$M::OUT)
  matsb:= (inverse(mat)$M :: M) * matsb
  -- print("inv *..."::OUT)
  matbs:=matbs * mat
  -- 5) if the 'smaller' field is "normal" we have first to convert
```

```
      --    from 'small normal' to 'small polynomial', that is from
      --    FFNBP(GF,dPsmall) to FFP(GF,dPsmall). Therefore we compute a
      --    conversion matrix 'mat' as in 1). Multiplying with  'mat'
      --    yields the desired conversion from normal to polynomial.
      --    Multiplying the above computed 'matsb' with 'mat' we got the
      --    matrix for converting form 'small normal' to 'big normal'.
    -- set up matrix 'mat' for normal to polynomial
    if rTsmall = "normal" then
      arr:=reducedQPowers(dPsmall)$FFPOLY(GF)
      mat:=zero(degsmall,degsmall)$M
      for i in 1..degsmall repeat
        setColumn_!(mat,i,vectorise(arr.(i-1),degsmall)$SUP(GF))$M
    -- old code
    --b:FFP(GF,dPsmall):=basis()$FFP(GF,dPsmall).2
    --setColumn_!(mat,1,coordinates(b)$FFP(GF,dPsmall))$M
    --for i in 2..degsmall repeat
    --  b:= b **$FFP(GF,dPsmall) size()$GF
    --  setColumn_!(mat,i,coordinates(b)$FFP(GF,dPsmall))$M
      -- print(mat::OUT)
      matsb:= matsb * mat
      matbs:= (inverse(mat) :: M) * matbs
    -- now 'matsb' is the corret conversion matrix for 'small' to 'big'
    -- and 'matbs' the corret one for 'big' to 'small'.
    -- depending on the above ordering the conversion matrices are
    -- initialized
    dPbig =$(SUP GF) defPol2 =>
      conMat1to2 :=matsb
      conMat2to1 :=matbs
      -- print(conMat1to2::OUT)
      -- print(conMat2to1::OUT)
      -- print("conversion matrices initialized"::OUT)
      --we finish the function for one case, hence reset initialization flag
      init? := false
      void()$Void
    conMat1to2 :=matbs
    conMat2to1 :=matsb
    -- print(conMat1to2::OUT)
    -- print(conMat2to1::OUT)
    -- print("conversion matrices initialized"::OUT)
    --we finish the function for one case, hence reset initialization flag
    init? := false
    void()$Void


  coerce(x:F1) ==
    inGroundField?(x)$F1 => retract(x)$F1 :: F2
    -- if x is already in GF then we can use a simple coercion
    defPol1 =$(SUP GF) defPol2 => convertWRTsameDefPol12(x)
    convertWRTdifferentDefPol12(x)
```

```
    convertWRTsameDefPol12(x:F1)  ==
      repType1 = repType2 => x pretend F2
      -- same groundfields, same defining polynomials, same
      -- representation types --> F1 = F2, x is already in F2
      repType1 = "cyclic" =>
        x = 0$F1 => 0$F2
      -- the SI corresponding to the cyclic representation is the exponent of
      -- the primitiveElement, therefore we exponentiate the primitiveElement
      -- of F2 by it.
        primitiveElement()$F2 **$F2 (x pretend SI)
      repType2 = "cyclic" =>
        x = 0$F1 => 0$F2
      -- to get the exponent, we have to take the discrete logarithm of the
      -- element in the given field.
        (discreteLog(x)$F1 pretend SI) pretend F2
      -- here one of the representation types is "normal"
      if init? then initialize()
      -- here a conversion matrix is necessary, (see initialize())
      represents(conMat1to2 *$(Matrix GF) coordinates(x)$F1)$F2

    convertWRTdifferentDefPol12(x:F1) ==
      if init? then initialize()
      -- if we want to convert into a 'smaller' field, we have to test,
      -- whether the element is in the subfield of the 'bigger' field, which
      -- corresponds to the 'smaller' field
      if degree1 > degree2 then
        if positiveRemainder(degree2,degree(x)$F1)^= 0 then
          error "coerce: element doesn't belong to smaller field"
      represents(conMat1to2 *$(Matrix GF) coordinates(x)$F1)$F2

-- the three functions below equal the three functions above up to
-- '1' exchanged by '2' in all domain and variable names


    coerce(x:F2) ==
      inGroundField?(x)$F2 => retract(x)$F2 :: F1
      -- if x is already in GF then we can use a simple coercion
      defPol1 =$(SUP GF) defPol2 => convertWRTsameDefPol21(x)
      convertWRTdifferentDefPol21(x)

    convertWRTsameDefPol21(x:F2)  ==
      repType1 = repType2 => x pretend F1
      -- same groundfields, same defining polynomials,
      -- same representation types --> F1 = F2, that is:
      -- x is already in F1
      repType2 = "cyclic" =>
        x = 0$F2 => 0$F1
        primitiveElement()$F1 **$F1 (x pretend SI)
      repType1 = "cyclic" =>
        x = 0$F2 => 0$F1
```

```
        (discreteLog(x)$F2 pretend SI) pretend F1
      -- here one of the representation types is "normal"
      if init? then initialize()
      represents(conMat2to1 *$(Matrix GF) coordinates(x)$F2)$F1

    convertWRTdifferentDefPol21(x:F2) ==
      if init? then initialize()
      if degree2 > degree1 then
        if positiveRemainder(degree1,degree(x)$F2)^= 0 then
          error "coerce: element doesn't belong to smaller field"
      represents(conMat2to1 *$(Matrix GF) coordinates(x)$F2)$F1
```

———-

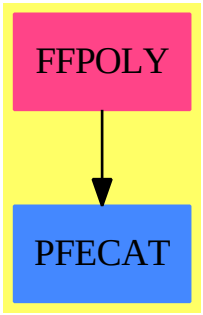
— **FFHOM.dotabb** —

```
"FFHOM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFHOM"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFHOM" -> "FAXF"
```

———-

7.14 package FFPOLY FiniteFieldPolynomialPackage

7.14.1 FiniteFieldPolynomialPackage (FFPOLY)



Exports:

createIrreduciblePoly	createNormalPoly	createNormalPrimitivePoly
createPrimitiveNormalPoly	createPrimitivePoly	leastAffineMultiple
nextIrreduciblePoly	nextNormalPoly	nextNormalPrimitivePoly
nextPrimitiveNormalPoly	nextPrimitivePoly	normal?
numberOfIrreduciblePoly	numberOfNormalPoly	numberOfPrimitivePoly
primitive?	random	reducedQPowers

— package FFPOLY FiniteFieldPolynomialPackage —

```
)abbrev package FFPOLY FiniteFieldPolynomialPackage
++ Author: A. Bouyer, J. Grabmeier, A. Scheerhorn, R. Sutor, B. Trager
++ Date Created: January 1991
++ Date Last Updated: 1 June 1994
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: finite field, polynomial, irreducible polynomial, normal
++   polynomial, primitive polynomial, random polynomials
++ References:
++   [LS] Lenstra, H. W. & Schoof, R. J., "Primitivive Normal Bases
++        for Finite Fields", Math. Comp. 48, 1987, pp. 217-231
++   [LN] Lidl, R. & Niederreiter, H., "Finite Fields",
++        Encycl. of Math. 20, Addison-Wesley, 1983
++  J. Grabmeier, A. Scheerhorn: Finite Fields in Axiom.
++   Axiom Technical Report Series, to appear.
++ Description:
++ This package provides a number of functions for generating, counting
++ and testing irreducible, normal, primitive, random polynomials
++ over finite fields.

FiniteFieldPolynomialPackage GF : Exports == Implementation where

  GF : FiniteFieldCategory

  I    ==> Integer
  L    ==> List
  NNI  ==> NonNegativeInteger
  PI   ==> PositiveInteger
  Rec  ==> Record(expnt:NNI, coeff:GF)
  Repr ==> L Rec
  SUP  ==> SparseUnivariatePolynomial GF

  Exports ==> with
 --    qEulerPhiCyclotomic : PI -> PI
--      ++ qEulerPhiCyclotomic(n)$FFPOLY(GF) yields the q-Euler's function
--      ++ of the n-th cyclotomic polynomial over the field GF of
--      ++ order q (cf. [LN] p.122);
--      ++ error if n is a multiple of the field characteristic.
    primitive? : SUP -> Boolean
      ++ primitive?(f) tests whether the polynomial f over a finite
      ++ field is primitive, i.e. all its roots are primitive.
    normal? : SUP -> Boolean
      ++ normal?(f) tests whether the polynomial f over a finite field is
      ++ normal, i.e. its roots are linearly independent over the field.
    numberOfIrreduciblePoly : PI -> PI
```

```
  ++ numberOfIrreduciblePoly(n)$FFPOLY(GF) yields the number of
  ++ monic irreducible univariate polynomials of degree n
  ++ over the finite field GF.
numberOfPrimitivePoly : PI -> PI
  ++ numberOfPrimitivePoly(n)$FFPOLY(GF) yields the number of
  ++ primitive polynomials of degree n over the finite field GF.
numberOfNormalPoly : PI -> PI
  ++ numberOfNormalPoly(n)$FFPOLY(GF) yields the number of
  ++ normal polynomials of degree n over the finite field GF.
createIrreduciblePoly : PI -> SUP
  ++ createIrreduciblePoly(n)$FFPOLY(GF) generates a monic irreducible
  ++ univariate polynomial of degree n over the finite field GF.
createPrimitivePoly : PI -> SUP
  ++ createPrimitivePoly(n)$FFPOLY(GF) generates a primitive polynomial
  ++ of degree n over the finite field GF.
createNormalPoly : PI -> SUP
  ++ createNormalPoly(n)$FFPOLY(GF) generates a normal polynomial
  ++ of degree n over the finite field GF.
createNormalPrimitivePoly : PI -> SUP
  ++ createNormalPrimitivePoly(n)$FFPOLY(GF) generates a normal and
  ++ primitive polynomial of degree n over the field GF.
  ++ Note that this function is equivalent to createPrimitiveNormalPoly(n)
createPrimitiveNormalPoly : PI -> SUP
  ++ createPrimitiveNormalPoly(n)$FFPOLY(GF) generates a normal and
  ++ primitive polynomial of degree n over the field GF.
  ++ polynomial of degree n over the field GF.
nextIrreduciblePoly : SUP -> Union(SUP, "failed")
  ++ nextIrreduciblePoly(f) yields the next monic irreducible polynomial
  ++ over a finite field GF of the same degree as f in the following
  ++ order, or "failed" if there are no greater ones.
  ++ Error: if f has degree 0.
  ++ Note that the input polynomial f is made monic.
  ++ Also, \spad{f < g} if
  ++ the number of monomials of f is less
  ++ than this number for g.
  ++ If f and g have the same number of monomials,
  ++ the lists of exponents are compared lexicographically.
  ++ If these lists are also equal, the lists of coefficients
  ++ are compared according to the lexicographic ordering induced by
  ++ the ordering of the elements of GF given by lookup.
nextPrimitivePoly : SUP -> Union(SUP, "failed")
  ++ nextPrimitivePoly(f) yields the next primitive polynomial over
  ++ a finite field GF of the same degree as f in the following
  ++ order, or "failed" if there are no greater ones.
  ++ Error: if f has degree 0.
  ++ Note that the input polynomial f is made monic.
  ++ Also, \spad{f < g} if the lookup of the constant term
  ++ of f is less than
  ++ this number for g.
  ++ If these values are equal, then \spad{f < g} if
```

```
    ++ if the number of monomials of f is less than that for g or if
    ++ the lists of exponents of f are lexicographically less than the
    ++ corresponding list for g.
    ++ If these lists are also equal, the lists of coefficients are
    ++ compared according to the lexicographic ordering induced by
    ++ the ordering of the elements of GF given by lookup.
  nextNormalPoly : SUP -> Union(SUP, "failed")
    ++ nextNormalPoly(f) yields the next normal polynomial over
    ++ a finite field GF of the same degree as f in the following
    ++ order, or "failed" if there are no greater ones.
    ++ Error: if f has degree 0.
    ++ Note that the input polynomial f is made monic.
    ++ Also, \spad{f < g} if the lookup of the coefficient
    ++ of the term of degree
    ++ n-1 of f is less than that for g.
    ++ In case these numbers are equal, \spad{f < g} if
    ++ if the number of monomials of f is less that for g or if
    ++ the list of exponents of f are lexicographically less than the
    ++ corresponding list for g.
    ++ If these lists are also equal, the lists of coefficients are
    ++ compared according to the lexicographic ordering induced by
    ++ the ordering of the elements of GF given by lookup.
  nextNormalPrimitivePoly : SUP -> Union(SUP, "failed")
    ++ nextNormalPrimitivePoly(f) yields the next normal primitive polynomial
    ++ over a finite field GF of the same degree as f in the following
    ++ order, or "failed" if there are no greater ones.
    ++ Error: if f has degree 0.
    ++ Note that the input polynomial f is made monic.
    ++ Also, \spad{f < g} if the lookup of the constant
    ++ term of f is less than
    ++ this number for g or if
    ++ lookup of the coefficient of the term of degree n-1
    ++ of f is less than this number for g.
    ++ Otherwise, \spad{f < g}
    ++ if the number of monomials of f is less than
    ++ that for g or if the lists of exponents for f are
    ++ lexicographically less than those for g.
    ++ If these lists are also equal, the lists of coefficients are
    ++ compared according to the lexicographic ordering induced by
    ++ the ordering of the elements of GF given by lookup.
    ++ This operation is equivalent to nextPrimitiveNormalPoly(f).
  nextPrimitiveNormalPoly : SUP -> Union(SUP, "failed")
    ++ nextPrimitiveNormalPoly(f) yields the next primitive normal polynomial
    ++ over a finite field GF of the same degree as f in the following
    ++ order, or "failed" if there are no greater ones.
    ++ Error: if f has degree 0.
    ++ Note that the input polynomial f is made monic.
    ++ Also, \spad{f < g} if the lookup of the
    ++ constant term of f is less than
    ++ this number for g or, in case these numbers are equal, if the
```

```
      ++ lookup of the coefficient of the term of degree n-1
      ++ of f is less than this number for g.
      ++ If these numbers are equals, \spad{f < g}
      ++ if the number of monomials of f is less than
      ++ that for g, or if the lists of exponents for f are lexicographically
      ++ less than those for g.
      ++ If these lists are also equal, the lists of coefficients are
      ++ coefficients according to the lexicographic ordering induced by
      ++ the ordering of the elements of GF given by lookup.
      ++ This operation is equivalent to nextNormalPrimitivePoly(f).
--    random : () -> SUP
--      ++ random()$FFPOLY(GF) generates a random monic polynomial
--      ++ of random degree over the field GF
    random : PI -> SUP
      ++ random(n)$FFPOLY(GF) generates a random monic polynomial
      ++ of degree n over the finite field GF.
    random : (PI, PI) -> SUP
      ++ random(m,n)$FFPOLY(GF) generates a random monic polynomial
      ++ of degree d over the finite field GF, d between m and n.
    leastAffineMultiple: SUP  -> SUP
      ++ leastAffineMultiple(f) computes the least affine polynomial which
      ++ is divisible by the polynomial f over the finite field GF,
      ++ i.e. a polynomial whose exponents are 0 or a power of q, the
      ++ size of GF.
    reducedQPowers: SUP  -> PrimitiveArray SUP
      ++ reducedQPowers(f)
      ++ generates \spad{[x,x**q,x**(q**2),...,x**(q**(n-1))]}
      ++ reduced modulo f where \spad{q = size()$GF} and \spad{n = degree f}.
    --
    -- we intend to implement also the functions
    -- cyclotomicPoly: PI -> SUP, order: SUP -> PI,
    -- and maybe a new version of irreducible?


  Implementation ==> add

    import IntegerNumberTheoryFunctions
    import DistinctDegreeFactorize(GF, SUP)


    MM := ModMonic(GF, SUP)

    sizeGF : PI := size()$GF :: PI

    revListToSUP(l:Repr):SUP ==
        newl:Repr := empty()
        -- cannot use map since copy for Record is an XLAM
        for t in l repeat newl := cons(copy t, newl)
        newl pretend SUP
```

```
listToSUP(l:Repr):SUP ==
    newl:Repr := [copy t for t in l]
    newl pretend SUP

nextSubset : (L NNI, NNI) -> Union(L NNI, "failed")
  -- for a list s of length m with 1 <= s.1 < ... < s.m <= bound,
  -- nextSubset(s, bound) yields the immediate successor of s
  -- (resp. "failed" if s = [1,...,bound])
  -- where s < t if and only if:
  -- (i)  #s < #t; or
  -- (ii) #s = #t and s < t in the lexicographical order;
  -- (we have chosen to fix the signature with NNI instead of PI
  --  to avoid coercions in the main functions)

reducedQPowers(f) ==
  m:PI:=degree(f)$SUP pretend PI
  m1:I:=m-1
  setPoly(f)$MM
  e:=reduce(monomial(1,1)$SUP)$MM ** sizeGF
  w:=1$MM
  qpow:PrimitiveArray SUP:=new(m,0)
  qpow.0:=1$SUP
  for i in 1..m1 repeat  qpow.i:=lift(w:=w*e)$MM
  qexp:PrimitiveArray SUP:=new(m,0)
  m = 1 =>
    qexp.(0$I):= (-coefficient(f,0$NNI)$SUP)::SUP
    qexp
  qexp.0$I:=monomial(1,1)$SUP
  h:=qpow.1
  qexp.1:=h
  for i in 2..m1 repeat
    g:=0$SUP
    while h ^= 0 repeat
      g:=g + leadingCoefficient(h) * qpow.degree(h)
      h:=reductum(h)
    qexp.i:=(h:=g)
  qexp

leastAffineMultiple(f) ==
-- [LS] p.112
  qexp:=reducedQPowers(f)
  n:=degree(f)$SUP
  b:Matrix GF:= transpose matrix [entries vectorise
       (qexp.i,n) for i in 0..n-1]
  col1:Matrix GF:= new(n,1,0)
  col1(1,1)  := 1
  ns : List Vector GF := nullSpace (horizConcat(col1,b) )
  ----------------------------------------------------------------
  -- perhaps one should use that the first vector in ns is already
  -- the right one
```

```
      ----------------------------------------------------------------
      dim:=n+2
      coeffVector : Vector GF
      until empty? ns repeat
        newCoeffVector := ns.1
        i : PI :=(n+1) pretend PI
        while newCoeffVector(i) = 0 repeat
          i := (i - 1) pretend PI
        if i < dim then
          dim := i
          coeffVector := newCoeffVector
        ns := rest ns
      (coeffVector(1)::SUP) +(+/[monomial(coeffVector.k, _
              sizeGF**((k-2)::NNI))$SUP for k in 2..dim])

--    qEulerPhiCyclotomic n ==
--      n = 1 => (sizeGF - 1) pretend PI
--      p : PI := characteristic()$GF :: PI
--      (n rem p) = 0 => error
--        "cyclotomic polynomial not defined for this argument value"
--      q  : PI := sizeGF
--      -- determine the multiplicative order of q modulo n
--      e  : PI := 1
--      qe : PI := q
--      while (qe rem n) ^= 1 repeat
--        e  := e + 1
--        qe := qe * q
--      ((qe - 1) ** ((eulerPhi(n) quo e) pretend PI) ) pretend PI

    numberOfIrreduciblePoly n ==
      -- we compute the number Nq(n) of monic irreducible polynomials
      -- of degree n over the field GF of order q by the formula
      -- Nq(n) = (1/n)* sum(moebiusMu(n/d)*q**d) where the sum extends
      -- over all divisors d of n (cf. [LN] p.93, Th. 3.25)
      n = 1 => sizeGF
      -- the contribution of d = 1 :
      lastd : PI  := 1
      qd    : PI  := sizeGF
      sum   :  I  := moebiusMu(n) * qd
      -- the divisors d > 1 of n :
      divisorsOfn : L PI := rest(divisors n) pretend L PI
      for d in divisorsOfn repeat
        qd := qd * (sizeGF) ** ((d - lastd) pretend PI)
        sum := sum + moebiusMu(n quo d) * qd
        lastd := d
      (sum quo n) :: PI

    numberOfPrimitivePoly n == (eulerPhi((sizeGF ** n) - 1) quo n) :: PI
      -- [each root of a primitive polynomial of degree n over a field
      --  with q elements is a generator of the multiplicative group
```

```
    --  of a field of order q**n (definition), and the number of such
    --  generators is precisely eulerPhi(q**n - 1)]

  numberOfNormalPoly n ==
    -- we compute the number Nq(n) of normal polynomials of degree n
    -- in GF[X], with GF of order q, by the formula
    -- Nq(n) = (1/n) * qPhi(X**n - 1) (cf. [LN] p.124) where,
    -- for any polynomial f in GF[X] of positive degree n,
    -- qPhi(f) = q**n * (1 - q**(-n1)) *...* (1 - q**(-nr)) =
    -- q**n * ((q**(n1)-1) / q**(n1)) *...* ((q**(nr)-1) / q**(n_r)),
    -- the ni being the degrees of the distinct irreducible factors
    -- of f in its canonical factorization over GF
    -- ([LN] p.122, Lemma 3.69).
    -- hence, if n = m * p**r where p is the characteristic of GF
    -- and gcd(m,p) = 1, we get
    -- Nq(n) = (1/n)* q**(n-m) * qPhi(X**m - 1)
    -- now X**m - 1 is the product of the (pairwise relatively prime)
    -- cyclotomic polynomials Qd(X) for which d divides m
    -- ([LN] p.64, Th. 2.45), and each Qd(X) factors into
    -- eulerPhi(d)/e (distinct) monic irreducible polynomials in GF[X]
    -- of the same degree e, where e is the least positive integer k
    -- such that d divides q**k - 1 ([LN] p.65, Th. 2.47)
    n = 1 => (sizeGF - 1) :: NNI :: PI
    m : PI := n
    p : PI := characteristic()$GF :: PI
    q : PI := sizeGF
    while (m rem p) = 0 repeat   -- find m such that
      m := (m quo p) :: PI       -- n = m * p**r and gcd(m,p) = 1
    m = 1 =>
       -- know that n is a power of p
      (((q ** ((n-1)::NNI) )  * (q - 1) ) quo n) :: PI
    prod : I := q - 1
    divisorsOfm : L PI := rest(divisors m) pretend L PI
    for d in divisorsOfm repeat
      -- determine the multiplicative order of q modulo d
      e  : PI := 1
      qe : PI := q
      while (qe rem d) ^= 1 repeat
        e  := e + 1
        qe := qe * q
      prod := prod * _
        ((qe - 1) ** ((eulerPhi(d) quo e) pretend PI) ) pretend PI
    (q**((n-m) pretend PI) * prod quo n) pretend PI

  primitive? f ==
    -- let GF be a field of order q; a monic polynomial f in GF[X]
    -- of degree n is primitive over GF if and only if its constant
    -- term is non-zero, f divides X**(q**n - 1) - 1 and,
    -- for each prime divisor d of q**n - 1,
    -- f does not divide X**((q**n - 1) / d) - 1
```

```
  -- (cf. [LN] p.89, Th. 3.16, and p.87, following Th. 3.11)
  n : NNI := degree f
  n = 0 => false
  leadingCoefficient f ^= 1 => false
  coefficient(f, 0) = 0 => false
  q  : PI := sizeGF
  qn1: PI := (q**n - 1) :: NNI :: PI
  setPoly f
  x := reduce(monomial(1,1)$SUP)$MM -- X rem f represented in MM
  --
  -- may be improved by tabulating the residues x**(i*q)
  -- for i = 0,...,n-1 :
  --
  lift(x ** qn1)$MM ^= 1 => false -- X**(q**n - 1) rem f in GF[X]
  lrec  : L Record(factor:I, exponent:I) := factors(factor qn1)
  lfact : L PI := []              -- collect the prime factors
  for rec in lrec repeat          -- of q**n - 1
    lfact := cons((rec.factor) :: PI, lfact)
  for d in lfact repeat
    if (expt := (qn1 quo d)) >= n then
      lift(x ** expt)$MM = 1 => return false
  true

normal? f ==
  -- let GF be a field with q elements; a monic irreducible
  -- polynomial f in GF[X] of degree n is normal if its roots
  -- x, x**q, ... , x**(q**(n-1)) are linearly independent over GF
  n : NNI := degree f
  n = 0 => false
  leadingCoefficient f ^= 1 => false
  coefficient(f, 0) = 0 => false
  n = 1 => true
  not irreducible? f => false
  g:=reducedQPowers(f)
  l:=[entries vectorise(g.i,n)$SUP for i in 0..(n-1)::NNI]
  rank(matrix(l)$Matrix(GF)) = n => true
  false

nextSubset(s, bound) ==
  m : NNI := #(s)
  m = 0 => [1]
  -- find the first element s(i) of s such that s(i) + 1 < s(i+1) :
  noGap : Boolean := true
  i : NNI := 0
  restOfs : L NNI
  while noGap and not empty?(restOfs := rest s) repeat
  -- after i steps (0 <= i <= m-1) we have s = [s(i), ... , s(m)]
  -- and restOfs = [s(i+1), ... , s(m)]
    secondOfs := first restOfs    -- s(i+1)
    firstOfsPlus1 := first s + 1  -- s(i) + 1
```

```
      secondOfs = firstOfsPlus1 =>
        s := restOfs
        i := i + 1
      setfirst_!(s, firstOfsPlus1)  -- s := [s(i)+1, s(i+1),..., s(m)]
      noGap := false
    if noGap then                   -- here s = [s(m)]
      firstOfs := first s
      firstOfs < bound => setfirst_!(s, firstOfs + 1) -- s := [s(m)+1]
      m < bound =>
          setfirst_!(s, m + 1)      -- s := [m+1]
          i := m
      return "failed"               -- (here m = s(m) = bound)
    for j in i..1 by -1 repeat  -- reconstruct the destroyed
      s := cons(j, s)           -- initial part of s
    s

  nextIrreduciblePoly f ==
    n : NNI := degree f
    n = 0 => error "polynomial must have positive degree"
    -- make f monic
    if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
    -- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
    -- then fRepr := [[n,fn], ... , [i0,f{i0}]]
    fRepr : Repr := f pretend Repr
    fcopy : Repr := []
    -- we can not simply write fcopy := copy fRepr because
    -- the input(!) f would be modified by assigning
    -- a new value to one of its records
    for term in fRepr repeat
      fcopy := cons(copy term, fcopy)
    if term.expnt ^= 0 then
      fcopy := cons([0,0]$Rec, fcopy)
    tailpol : Repr := []
    headpol : Repr := fcopy  -- [[0,f0], ... , [n,fn]] where
                             -- fi is non-zero for i > 0
    fcopy   := reverse fcopy
    weight  : NNI := (#(fcopy) - 1) :: NNI -- #s(f) as explained above
    taillookuplist : L NNI := []
    -- the zeroes in the headlookuplist stand for the fi
    -- whose lookup's were not yet computed :
    headlookuplist : L NNI := new(weight, 0)
    s  : L NNI := [] -- we will compute s(f) only if necessary
    n1 : NNI := (n - 1) :: NNI
    repeat
      -- (run through the possible weights)
      while not empty? headlookuplist repeat
        -- find next polynomial in the above order with fixed weight;
        -- assume at this point we have
        -- headpol = [[i1,f{i1}], [i2,f{i2}], ... , [n,1]]
        -- and tailpol = [[k,fk], ... , [0,f0]] (with k < i1)
```

```
        term := first headpol
        j := first headlookuplist
        if j = 0 then j := lookup(term.coeff)$GF
        j := j + 1 -- lookup(f{i1})$GF + 1
        j rem sizeGF = 0 =>
          -- in this case one has to increase f{i2}
          tailpol := cons(term, tailpol) -- [[i1,f{i1}],...,[0,f0]]
          headpol := rest headpol        -- [[i2,f{i2}],...,[n,1]]
          taillookuplist := cons(j, taillookuplist)
          headlookuplist := rest headlookuplist
        -- otherwise set f{i1} := index(j)$GF
        setelt(first headpol, coeff, index(j :: PI)$GF)
        setfirst_!(headlookuplist, j)
        if empty? taillookuplist then
          pol := revListToSUP(headpol)
          --
          -- may be improved by excluding reciprocal polynomials
          --
          irreducible? pol => return pol
        else
          -- go back to fk
          headpol := cons(first tailpol, headpol) -- [[k,fk],...,[n,1]]
          tailpol := rest tailpol
          headlookuplist := cons(first taillookuplist, headlookuplist)
          taillookuplist := rest taillookuplist
      -- must search for polynomial with greater weight
      if empty? s then -- compute s(f)
        restfcopy := rest fcopy
        for entry in restfcopy repeat s := cons(entry.expnt, s)
      weight = n => return "failed"
      s1 := nextSubset(rest s, n1) :: L NNI
      s := cons(0, s1)
      weight := #s
      taillookuplist := []
      headlookuplist := cons(sizeGF, new((weight-1) :: NNI, 1))
      tailpol := []
      headpol := [] -- [[0,0], [s.2,1], ... , [s.weight,1], [n,1]] :
      s1 := cons(n, reverse s1)
      while not empty? s1 repeat
        headpol := cons([first s1, 1]$Rec, headpol)
        s1 := rest s1
      headpol := cons([0, 0]$Rec, headpol)

  nextPrimitivePoly f ==
    n : NNI := degree f
    n = 0 => error "polynomial must have positive degree"
    -- make f monic
    if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
    -- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
    -- then fRepr := [[n,fn], ... , [i0,f{i0}]]
```

```
fRepr : Repr := f pretend Repr
fcopy : Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
  fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
  term  := [0,0]$Rec
  fcopy := cons(term, fcopy)
fcopy   := reverse fcopy
xn : Rec := first fcopy
c0 : GF  := term.coeff
l  : NNI := lookup(c0)$GF rem sizeGF
n = 1 =>
  -- the polynomial X + c is primitive if and only if -c
  -- is a primitive element of GF
  q1 : NNI  := (sizeGF - 1) :: NNI
  while l < q1 repeat -- find next c such that -c is primitive
    l := l + 1
    c := index(l :: PI)$GF
    primitive?(-c)$GF =>
      return [xn, [0,c]$Rec] pretend SUP
  "failed"
weight : NNI := (#(fcopy) - 1) :: NNI -- #s(f)+1 as explained above
s  : L NNI := [] -- we will compute s(f) only if necessary
n1 : NNI := (n - 1) :: NNI
-- a necessary condition for a monic polynomial f of degree n
-- over GF to be primitive is that (-1)**n * f(0) be a
-- primitive element of GF (cf. [LN] p.90, Th. 3.18)
c  : GF  := c0
while l < sizeGF repeat
  -- (run through the possible values of the constant term)
  noGenerator : Boolean := true
  while noGenerator and l < sizeGF repeat
    -- find least c >= c0 such that (-1)^n c0 is primitive
    primitive?((-1)**n * c)$GF => noGenerator := false
    l := l + 1
    c := index(l :: PI)$GF
  noGenerator => return "failed"
  constterm : Rec := [0, c]$Rec
  if c = c0 and weight > 1 then
    headpol : Repr := rest reverse fcopy -- [[i0,f{i0}],...,[n,1]]
                                         -- fi is non-zero for i>0
    -- the zeroes in the headlookuplist stand for the fi
    -- whose lookup's were not yet computed :
    headlookuplist : L NNI := new(weight, 0)
  else
    -- X**n + c can not be primitive for n > 1 (cf. [LN] p.90,
    -- Th. 3.18); next possible polynomial is X**n + X + c
```

```
  headpol : Repr := [[1,0]$Rec, xn] -- 0*X + X**n
  headlookuplist : L NNI := [sizeGF]
  s := [0,1]
  weight := 2
tailpol : Repr := []
taillookuplist : L NNI := []
notReady : Boolean := true
while notReady repeat
  -- (run through the possible weights)
  while not empty? headlookuplist repeat
    -- find next polynomial in the above order with fixed
    -- constant term and weight; assume at this point we have
    -- headpol = [[i1,f{i1}], [i2,f{i2}], ... , [n,1]] and
    -- tailpol = [[k,fk],...,[k0,fk0]] (k0<...<k<i1<i2<...<n)
    term := first headpol
    j := first headlookuplist
    if j = 0 then j := lookup(term.coeff)$GF
    j := j + 1 -- lookup(f{i1})$GF + 1
    j rem sizeGF = 0 =>
      -- in this case one has to increase f{i2}
      tailpol := cons(term, tailpol) -- [[i1,f{i1}],...,[k0,f{k0}]]
      headpol := rest headpol        -- [[i2,f{i2}],...,[n,1]]
      taillookuplist := cons(j, taillookuplist)
      headlookuplist := rest headlookuplist
    -- otherwise set f{i1} := index(j)$GF
    setelt(first headpol, coeff, index(j :: PI)$GF)
    setfirst_!(headlookuplist, j)
    if empty? taillookuplist then
      pol := revListToSUP cons(constterm, headpol)
      --
      -- may be improved by excluding reciprocal polynomials
      --
      primitive? pol => return pol
    else
      -- go back to fk
      headpol := cons(first tailpol, headpol) -- [[k,fk],...,[n,1]]
      tailpol := rest tailpol
      headlookuplist := cons(first taillookuplist,
                                    headlookuplist)
      taillookuplist := rest taillookuplist
  if weight = n then notReady := false
  else
    -- must search for polynomial with greater weight
    if empty? s then -- compute s(f)
      restfcopy := rest fcopy
      for entry in restfcopy repeat s := cons(entry.expnt, s)
    s1 := nextSubset(rest s, n1) :: L NNI
    s  := cons(0, s1)
    weight := #s
    taillookuplist := []
```

```
        headlookuplist := cons(sizeGF, new((weight-2) :: NNI, 1))
        tailpol := []
        -- headpol = [[s.2,0], [s.3,1], ... , [s.weight,1], [n,1]] :
        headpol := [[first s1, 0]$Rec]
        while not empty? (s1 := rest s1) repeat
          headpol := cons([first s1, 1]$Rec, headpol)
        headpol := reverse cons([n, 1]$Rec, headpol)
      -- next polynomial must have greater constant term
      l := l + 1
      c := index(l :: PI)$GF
    "failed"

  nextNormalPoly f ==
    n : NNI := degree f
    n = 0 => error "polynomial must have positive degree"
    -- make f monic
    if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
    -- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
    -- then fRepr := [[n,fn], ... , [i0,f{i0}]]
    fRepr : Repr := f pretend Repr
    fcopy : Repr := []
    -- we can not simply write fcopy := copy fRepr because
    -- the input(!) f would be modified by assigning
    -- a new value to one of its records
    for term in fRepr repeat
      fcopy := cons(copy term, fcopy)
    if term.expnt ^= 0 then
      term  := [0,0]$Rec
      fcopy := cons(term, fcopy)
    fcopy     := reverse fcopy -- [[n,1], [r,fr], ... , [0,f0]]
    xn : Rec  := first fcopy
    middlepol : Repr := rest fcopy -- [[r,fr], ... , [0,f0]]
    a0 : GF  := (first middlepol).coeff -- fr
    l  : NNI := lookup(a0)$GF rem sizeGF
    n = 1 =>
      -- the polynomial X + a is normal if and only if a is not zero
      l = sizeGF - 1 => "failed"
      [xn, [0, index((l+1) :: PI)$GF]$Rec] pretend SUP
    n1 : NNI := (n  - 1) :: NNI
    n2 : NNI := (n1 - 1) :: NNI
    -- if the polynomial X**n + a * X**(n-1) + ... is normal then
    -- a = -(x + x**q +...+ x**(q**n)) can not be zero (where q = #GF)
    a  : GF  := a0
    -- if a = 0 then set a := 1
    if l = 0 then
      l := 1
      a := 1$GF
    while l < sizeGF repeat
      -- (run through the possible values of a)
      if a = a0 then
```

```
  -- middlepol = [[0,f0], ... , [m,fm]] with m < n-1
  middlepol := reverse rest middlepol
  weight : NNI := #middlepol -- #s(f) as explained above
  -- the zeroes in the middlelookuplist stand for the fi
  -- whose lookup's were not yet computed :
  middlelookuplist : L NNI := new(weight, 0)
  s : L NNI := [] -- we will compute s(f) only if necessary
else
  middlepol := [[0,0]$Rec]
  middlelookuplist : L NNI := [sizeGF]
  s : L NNI := [0]
  weight : NNI := 1
headpol : Repr := [xn, [n1, a]$Rec] -- X**n + a * X**(n-1)
tailpol : Repr := []
taillookuplist : L NNI := []
notReady : Boolean := true
while notReady repeat
  -- (run through the possible weights)
  while not empty? middlelookuplist repeat
    -- find next polynomial in the above order with fixed
    -- a and weight; assume at this point we have
    -- middlepol = [[i1,f{i1}], [i2,f{i2}], ... , [m,fm]] and
    -- tailpol = [[k,fk],...,[0,f0]] ( with k<i1<i2<...<m)
    term := first middlepol
    j := first middlelookuplist
    if j = 0 then j := lookup(term.coeff)$GF
    j := j + 1 -- lookup(f{i1})$GF + 1
    j rem sizeGF = 0 =>
      -- in this case one has to increase f{i2}
      -- tailpol = [[i1,f{i1}],...,[0,f0]]
      tailpol   := cons(term, tailpol)
      middlepol := rest middlepol -- [[i2,f{i2}],...,[m,fm]]
      taillookuplist   := cons(j, taillookuplist)
      middlelookuplist := rest middlelookuplist
    -- otherwise set f{i1} := index(j)$GF
    setelt(first middlepol, coeff, index(j :: PI)$GF)
    setfirst_!(middlelookuplist, j)
    if empty? taillookuplist then
      pol := listToSUP append(headpol, reverse middlepol)
      --
      -- may be improved by excluding reciprocal polynomials
      --
      normal? pol => return pol
    else
      -- go back to fk
      -- middlepol = [[k,fk],...,[m,fm]]
      middlepol := cons(first tailpol, middlepol)
      tailpol := rest tailpol
      middlelookuplist := cons(first taillookuplist,
                                       middlelookuplist)
```

```
            taillookuplist := rest taillookuplist
          if weight = n1 then notReady := false
          else
            -- must search for polynomial with greater weight
            if empty? s then -- compute s(f)
              restfcopy := rest rest fcopy
              for entry in restfcopy repeat s := cons(entry.expnt, s)
            s1 := nextSubset(rest s, n2) :: L NNI
            s  := cons(0, s1)
            weight := #s
            taillookuplist := []
            middlelookuplist := cons(sizeGF, new((weight-1) :: NNI, 1))
            tailpol   := []
            -- middlepol = [[0,0], [s.2,1], ... , [s.weight,1]] :
            middlepol := []
            s1 := reverse s1
            while not empty? s1 repeat
              middlepol := cons([first s1, 1]$Rec, middlepol)
              s1 := rest s1
            middlepol := cons([0,0]$Rec, middlepol)
        -- next polynomial must have greater a
        l := l + 1
        a := index(l :: PI)$GF
      "failed"

    nextNormalPrimitivePoly f ==
      n : NNI := degree f
      n = 0 => error "polynomial must have positive degree"
      -- make f monic
      if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
      -- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
      -- then fRepr := [[n,fn], ... , [i0,f{i0}]]
      fRepr : Repr := f pretend Repr
      fcopy : Repr := []
      -- we can not simply write fcopy := copy fRepr because
      -- the input(!) f would be modified by assigning
      -- a new value to one of its records
      for term in fRepr repeat
        fcopy := cons(copy term, fcopy)
      if term.expnt ^= 0 then
        term  := [0,0]$Rec
        fcopy := cons(term, fcopy)
      fcopy   := reverse fcopy -- [[n,1], [r,fr], ... , [0,f0]]
      xn : Rec := first fcopy
      c0 : GF  := term.coeff
      lc : NNI := lookup(c0)$GF rem sizeGF
      n = 1 =>
        -- the polynomial X + c is primitive if and only if -c
        -- is a primitive element of GF
        q1 : NNI  := (sizeGF - 1) :: NNI
```

```
  while lc < q1 repeat -- find next c such that -c is primitive
    lc := lc + 1
    c  := index(lc :: PI)$GF
    primitive?(-c)$GF =>
      return [xn, [0,c]$Rec] pretend SUP
  "failed"
n1 : NNI := (n  - 1) :: NNI
n2 : NNI := (n1 - 1) :: NNI
middlepol : Repr := rest fcopy -- [[r,fr],...,[i0,f{i0}],[0,f0]]
a0 : GF  := (first middlepol).coeff
la : NNI := lookup(a0)$GF rem sizeGF
-- if the polynomial X**n + a * X**(n-1) +...+ c is primitive and
-- normal over GF then (-1)**n * c is a primitive element of GF
-- (cf. [LN] p.90, Th. 3.18), and a = -(x + x**q +...+ x**(q**n))
-- is not zero (where q = #GF)
c : GF  := c0
a : GF  := a0
-- if a = 0 then set a := 1
if la = 0 then
  la := 1
  a  := 1$GF
while lc < sizeGF repeat
  -- (run through the possible values of the constant term)
  noGenerator : Boolean := true
  while noGenerator and lc < sizeGF repeat
    -- find least c >= c0 such that (-1)**n * c0 is primitive
    primitive?((-1)**n * c)$GF => noGenerator := false
    lc := lc + 1
    c  := index(lc :: PI)$GF
  noGenerator => return "failed"
  constterm : Rec := [0, c]$Rec
  while la < sizeGF repeat
  -- (run through the possible values of a)
    headpol : Repr := [xn, [n1, a]$Rec] -- X**n + a X**(n-1)
    if c = c0 and a = a0 then
      -- middlepol = [[i0,f{i0}], ... , [m,fm]] with m < n-1
      middlepol := rest reverse rest middlepol
      weight : NNI := #middlepol + 1 -- #s(f)+1 as explained above
      -- the zeroes in the middlelookuplist stand for the fi
      -- whose lookup's were not yet computed :
      middlelookuplist : L NNI := new((weight-1) :: NNI, 0)
      s : L NNI := [] -- we will compute s(f) only if necessary
    else
      pol := listToSUP append(headpol, [constterm])
      normal? pol and primitive? pol => return pol
      middlepol := [[1,0]$Rec]
      middlelookuplist : L NNI := [sizeGF]
      s : L NNI := [0,1]
      weight : NNI := 2
    tailpol : Repr := []
```

```
taillookuplist : L NNI := []
notReady : Boolean := true
while notReady repeat
-- (run through the possible weights)
  while not empty? middlelookuplist repeat
    -- find next polynomial in the above order with fixed
    -- c, a and weight; assume at this point we have
    -- middlepol = [[i1,f{i1}], [i2,f{i2}], ... , [m,fm]]
    -- tailpol = [[k,fk],...,[k0,fk0]] (k0<...<k<i1<...<m)
    term := first middlepol
    j := first middlelookuplist
    if j = 0 then j := lookup(term.coeff)$GF
    j := j + 1 -- lookup(f{i1})$GF + 1
    j rem sizeGF = 0 =>
      -- in this case one has to increase f{i2}
      -- tailpol = [[i1,f{i1}],...,[k0,f{k0}]]
      tailpol   := cons(term, tailpol)
      middlepol := rest middlepol -- [[i2,f{i2}],...,[m,fm]]
      taillookuplist   := cons(j, taillookuplist)
      middlelookuplist := rest middlelookuplist
    -- otherwise set f{i1} := index(j)$GF
    setelt(first middlepol, coeff, index(j :: PI)$GF)
    setfirst_!(middlelookuplist, j)
    if empty? taillookuplist then
      pol := listToSUP append(headpol, reverse
                      cons(constterm, middlepol))
      --
      -- may be improved by excluding reciprocal polynomials
      --
      normal? pol and primitive? pol => return pol
    else
      -- go back to fk
      -- middlepol = [[k,fk],...,[m,fm]]
      middlepol := cons(first tailpol, middlepol)
      tailpol := rest tailpol
      middlelookuplist := cons(first taillookuplist,
                                    middlelookuplist)
      taillookuplist := rest taillookuplist
  if weight = n1 then notReady := false
  else
    -- must search for polynomial with greater weight
    if empty? s then -- compute s(f)
      restfcopy := rest rest fcopy
      for entry in restfcopy repeat s := cons(entry.expnt, s)
    s1 := nextSubset(rest s, n2) :: L NNI
    s  := cons(0, s1)
    weight := #s
    taillookuplist := []
    middlelookuplist := cons(sizeGF, new((weight-2)::NNI, 1))
    tailpol   := []
```

```
            -- middlepol = [[s.2,0], [s.3,1], ... , [s.weight,1] :
            middlepol := [[first s1, 0]$Rec]
            while not empty? (s1 := rest s1) repeat
              middlepol := cons([first s1, 1]$Rec, middlepol)
            middlepol := reverse middlepol
          -- next polynomial must have greater a
          la := la + 1
          a  := index(la :: PI)$GF
        -- next polynomial must have greater constant term
        lc := lc + 1
        c  := index(lc :: PI)$GF
        la := 1
        a  := 1$GF
      "failed"

    nextPrimitiveNormalPoly f == nextNormalPrimitivePoly f

    createIrreduciblePoly n ==
      x := monomial(1,1)$SUP
      n = 1 => x
      xn := monomial(1,n)$SUP
      n >= sizeGF => nextIrreduciblePoly(xn + x) :: SUP
      -- (since in this case there is most no irreducible binomial X+a)
      odd? n => nextIrreduciblePoly(xn + 1) :: SUP
      nextIrreduciblePoly(xn) :: SUP

    createPrimitivePoly n ==
    -- (see also the comments in the code of nextPrimitivePoly)
      xn := monomial(1,n)$SUP
      n = 1 => xn + monomial(-primitiveElement()$GF, 0)$SUP
      c0 : GF := (-1)**n * primitiveElement()$GF
      constterm : Rec := [0, c0]$Rec
      -- try first (probably faster) the polynomials
      -- f = X**n + f{n-1}*X**(n-1) +...+ f1*X + c0 for which
      -- fi is 0 or 1 for i=1,...,n-1,
      -- and this in the order used to define nextPrimitivePoly
      s  : L NNI := [0,1]
      weight : NNI := 2
      s1 : L NNI := [1]
      n1 : NNI := (n - 1) :: NNI
      notReady : Boolean := true
      while notReady repeat
        polRepr : Repr := [constterm]
        while not empty? s1 repeat
          polRepr := cons([first s1, 1]$Rec, polRepr)
          s1 := rest s1
        polRepr := cons([n, 1]$Rec, polRepr)
        --
        -- may be improved by excluding reciprocal polynomials
        --
```

```
        primitive? (pol := listToSUP polRepr) => return pol
        if weight = n then notReady := false
        else
          s1 := nextSubset(rest s, n1) :: L NNI
          s  := cons(0, s1)
          weight := #s
      -- if there is no primitive f of the above form
      -- search now from the beginning, allowing arbitrary
      -- coefficients f_i, i = 1,...,n-1
      nextPrimitivePoly(xn + monomial(c0, 0)$SUP) :: SUP

    createNormalPoly n  ==
      n = 1 => monomial(1,1)$SUP + monomial(-1,0)$SUP
      -- get a normal polynomial f = X**n + a * X**(n-1) + ...
      -- with a = -1
      -- [recall that if f is normal over the field GF of order q
      -- then a = -(x + x**q +...+ x**(q**n)) can not be zero;
      -- hence the existence of such an f follows from the
      -- normal basis theorem ([LN] p.60, Th. 2.35) and the
      -- surjectivity of the trace ([LN] p.55, Th. 2.23 (iii))]
      nextNormalPoly(monomial(1,n)$SUP
                     + monomial(-1, (n-1) :: NNI)$SUP) :: SUP

    createNormalPrimitivePoly n ==
      xn := monomial(1,n)$SUP
      n = 1 => xn + monomial(-primitiveElement()$GF, 0)$SUP
      n1  : NNI := (n - 1) :: NNI
      c0  : GF  := (-1)**n * primitiveElement()$GF
      constterm  := monomial(c0, 0)$SUP
      -- try first the polynomials f = X**n + a *  X**(n-1) + ...
      -- with a = -1
      pol := xn + monomial(-1, n1)$SUP + constterm
      normal? pol and primitive? pol => pol
      res := nextNormalPrimitivePoly(pol)
      res case SUP => res
      -- if there is no normal primitive f with a = -1
      -- get now one with arbitrary (non-zero) a
      -- (the existence is proved in [LS])
      pol := xn + monomial(1, n1)$SUP + constterm
      normal? pol and primitive? pol => pol
      nextNormalPrimitivePoly(pol) :: SUP

    createPrimitiveNormalPoly n == createNormalPrimitivePoly n

--    qAdicExpansion m ==
--      ragits : List I := wholeRagits(m :: (RadixExpansion sizeGF))
--      pol  : SUP := 0
--      expt : NNI := #ragits
--      for i in ragits repeat
--        expt := (expt - 1) :: NNI
```

```
--        if i ^= 0 then pol := pol + monomial(index(i::PI)$GF, expt)
--      pol

--    random == qAdicExpansion(random()$I)

--    random n ==
--      pol := monomial(1,n)$SUP
--      n1 : NNI := (n - 1) :: NNI
--      for i in 0..n1 repeat
--        if (c := random()$GF) ^= 0 then
--          pol := pol + monomial(c, i)$SUP
--      pol

    random n ==
      polRepr : Repr := []
      n1 : NNI := (n - 1) :: NNI
      for i in 0..n1 repeat
        if (c := random()$GF) ^= 0 then
          polRepr := cons([i, c]$Rec, polRepr)
      cons([n, 1$GF]$Rec, polRepr) pretend SUP

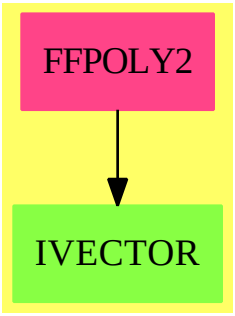
    random(m,n) ==
      if m > n then (m,n) := (n,m)
      d : NNI := (n - m) :: NNI
      if d > 1 then n := ((random()$I rem (d::PI)) + m) :: PI
      random(n)
```

— FFPOLY.dotabb —

```
"FFPOLY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFPOLY"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFPOLY" -> "PFECAT"
```

7.15 package FFPOLY2 FiniteFieldPolynomialPackage2

7.15.1 FiniteFieldPolynomialPackage2 (FFPOLY2)



Exports:
rootOfIrreduciblePoly

— package FFPOLY2 FiniteFieldPolynomialPackage2 —

```
)abbrev package FFPOLY2 FiniteFieldPolynomialPackage2
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated:
++ Basic Operations: rootOfIrreduciblePoly
++ Related Constructors: FiniteFieldCategory
++ Also See:
++ AMS Classifications:
++ Keywords: finite field, zeros of polynomials, Berlekamp's trace algorithm
++ References:
++  R.Lidl, H.Niederreiter: Finite Field, Encycoldia of Mathematics and
++  Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++  AXIOM Technical Report Series, to appear.
++ Description:
++ FiniteFieldPolynomialPackage2(F,GF) exports some functions concerning
++ finite fields, which depend on a finite field GF and an
++ algebraic extension F of GF, e.g. a zero of a polynomial
++ over GF in F.

FiniteFieldPolynomialPackage2(F,GF):Exports == Implementation where
  F:FieldOfPrimeCharacteristic with
      coerce: GF -> F
        ++ coerce(x) \undocumented{}
      lookup: F -> PositiveInteger
        ++ lookup(x) \undocumented{}
      basis: PositiveInteger -> Vector F
        ++ basis(n) \undocumented{}
```

```
      Frobenius: F -> F
       ++ Frobenius(x) \undocumented{}
  -- F should be a algebraic extension of the finite field GF, either an
  -- algebraic closure of GF or a simple algebraic extension field of GF
  GF:FiniteFieldCategory

  I   ==> Integer
  NNI ==> NonNegativeInteger
  PI  ==> PositiveInteger
  SUP ==> SparseUnivariatePolynomial
  MM  ==> ModMonic(GF,SUP GF)
  OUT ==> OutputForm
  M   ==> Matrix
  V   ==> Vector
  L   ==> List
  FFPOLY ==> FiniteFieldPolynomialPackage(GF)
  SUPF2 ==> SparseUnivariatePolynomialFunctions2(GF,F)

  Exports ==> with

    rootOfIrreduciblePoly:SUP GF -> F
      ++ rootOfIrreduciblePoly(f) computes one root of the monic,
      ++ irreducible polynomial f,
      ++ which degree must divide the extension degree
      ++ of F over GF,
      ++ i.e. f splits into linear factors over F.


  Implementation ==> add

-- we use berlekamps trace algorithm
-- it is not checked whether the polynomial is irreducible over GF]]
    rootOfIrreduciblePoly(pf) ==
--    not irreducible(pf)$FFPOLY =>
--      error("polynomial has to be irreducible")
      sizeGF:=size()$GF
      -- if the polynomial is of degree one, we're ready
      deg:=degree(pf)$(SUP GF)::PI
      deg = 0 => error("no roots")
      deg = 1 => -coefficient(pf,0)$(SUP GF)::F
      p : SUP F := map(coerce,pf)$SUPF2
      -- compute qexp, qexp(i) = x **(size()GF ** i) mod p
      -- with this list it's easier to compute the gcd(p(x),trace(x))
      qexp:=reducedQPowers(pf)$FFPOLY
      stillToFactor:=p
      -- take linear independent elements, the basis of F over GF
      basis:Vector F:=basis(deg)$F
      basispointer:I:=1
      -- as p is irreducible over GF, 0 can't be a root of p
      -- therefore we can use the predicate zero?(root) for indicating
```

```
      -- whether a root is found
      root:=0$F
      while zero?(root)$F repeat
        beta:F:=basis.basispointer
        -- gcd(trace(x)+gf,p(x)) has degree 0,that's why we skip beta=1
        if beta = 1$F then
          basispointer:=basispointer + 1
          beta:= basis.basispointer
        basispointer:=basispointer+1
        -- compute the polynomial trace(beta * x) mod p(x) using explist
        trModp:SUP F:= map(coerce,qexp.0)$SUPF2 * beta
        for i in 1..deg-1 repeat
          beta:=Frobenius(beta)
          trModp:=trModp +$(SUP F) beta *$(SUP F) map(coerce,qexp.i)$SUPF2
        -- if it is of degree 0, it doesn't help us finding a root
        if degree(trModp)$(SUP F) > 0 then
          -- for all elements gf of GF do
          for j in 1..sizeGF repeat
            -- compute gcd(trace(beta * x) + gf,stillToFactor)
            h:=gcd(stillToFactor,trModp +$(SUP F) _
             (index(j pretend PI)$GF::F::(SUP F)))$(SUP F)
            -- make the gcd polynomial monic
            if leadingCoefficient(h)$(SUP F) ^= 1$F then
              h:= (inv leadingCoefficient(h)) * h
            degh:=degree(h)$(SUP F)
            degSTF:=degree(stillToFactor)$(SUP F)
            -- if the gcd has degree one we are ready
            degh = 1 => root:=-coefficient(h,0)$(SUP F)
            -- if the quotient of stillToFactor and the gcd has
            -- degree one, we're also ready
            degSTF - degh = 1 =>
              root:= -coefficient(stillToFactor quo h,0)$(SUP F)
            -- otherwise the gcd helps us finding a root, only if its
            -- degree is between 2 and degree(stillToFactor)-2
            if degh > 1 and degh < degSTF then
              2*degh > degSTF => stillToFactor := stillToFactor quo h
              stillToFactor := h
      root
```

———-

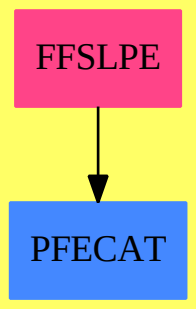
— FFPOLY2.dotabb —

```
"FFPOLY2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFPOLY2"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"FFPOLY2" -> "IVECTOR"
```

———-

7.16 package FFSLPE FiniteFieldSolveLinearPolynomialEquation

7.16.1 FiniteFieldSolveLinearPolynomialEquation (FFSLPE)



Exports:
solveLinearPolynomialEquation

— **package FFSLPE FiniteFieldSolveLinearPolynomialEquation** —

```
)abbrev package FFSLPE FiniteFieldSolveLinearPolynomialEquation
++ Author: Davenport
++ Date Created: 1991
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package solves linear diophantine equations for Bivariate polynomials
++ over finite fields

FiniteFieldSolveLinearPolynomialEquation(F:FiniteFieldCategory,
                                         FP:UnivariatePolynomialCategory F,
                                         FPP:UnivariatePolynomialCategory FP): with
   solveLinearPolynomialEquation: (List FPP, FPP) -> Union(List FPP,"failed")
              ++ solveLinearPolynomialEquation([f1, ..., fn], g)
              ++ (where the fi are relatively prime to each other)
              ++ returns a list of ai such that
              ++ \spad{g/prod fi = sum ai/fi}
              ++ or returns "failed" if no such list of ai's exists.
  == add
     oldlp:List FPP := []
```

```
slpePrime: FP := monomial(1,1)
oldtable:Vector List FPP := []
lp: List FPP
p: FPP
import DistinctDegreeFactorize(F,FP)
solveLinearPolynomialEquation(lp,p) ==
  if (oldlp ^= lp) then
     -- we have to generate a new table
     deg:= +/[degree u for u in lp]
     ans:Union(Vector List FPP,"failed"):="failed"
     slpePrime:=monomial(1,1)+monomial(1,0)   -- x+1: our starting guess
     while (ans case "failed") repeat
       ans:=tablePow(deg,slpePrime,lp)$GenExEuclid(FP,FPP)
       if (ans case "failed") then
          slpePrime:= nextItem(slpePrime)::FP
          while (degree slpePrime > 1) and
                not irreducible? slpePrime repeat
            slpePrime := nextItem(slpePrime)::FP
     oldtable:=(ans:: Vector List FPP)
  answer:=solveid(p,slpePrime,oldtable)
  answer
```

———-

— **FFSLPE.dotabb** —

```
"FFSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFSLPE" -> "PFECAT"
```

———-

7.17 package FFSQFR FiniteFieldSquareFreeDecomposition

— **FiniteFieldSquareFreeDecomposition.input** —

```
)set break resume
)sys rm -f FiniteFieldSquareFreeDecomposition.output
)spool FiniteFieldSquareFreeDecomposition.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 1
)show FiniteFieldSquareFreeDecomposition
--R
--R FiniteFieldSquareFreeDecomposition(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory
--R Abbreviation for FiniteFieldSquareFreeDecomposition is FFSQFR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFSQFR
--R
--R------------------------------ Operations --------------------------------
--R Musser : PolK -> Factored(PolK)       Yun : PolK -> Factored(PolK)
--R
--E 1

)spool
```

———-

— FiniteFieldSquareFreeDecomposition.help —

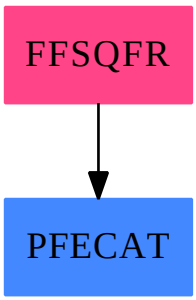
```
=====================================================================
FiniteFieldSquareFreeDecomposition examples
=====================================================================

See also:

o )show FiniteFieldSquareFreeDecomposition
```

———-

7.17.1 FiniteFieldSquareFreeDecomposition (FFSQFR)



Exports:
PolK Yun

— package FFSQFR FiniteFieldSquareFreeDecomposition —

```
)abbrev package FFSQFR FiniteFieldSquareFreeDecomposition
++ Author: Patrice Naudin and Claude Quitte
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
FiniteFieldSquareFreeDecomposition (K : FiniteFieldCategory,
                        PolK : UnivariatePolynomialCategory (K)) : with
    Musser : PolK -> Factored (PolK)
    Yun    : PolK -> Factored (PolK)
  == add

    p : NonNegativeInteger := characteristic()$K
    tableOfSquareFreePolynomials := Table (Integer, PolK)
    oneYunStep2uple := Record (
       simpleDecomposition : tableOfSquareFreePolynomials,
        gcdOfArgumentAndDerivative : PolK
         )

    rawMusser (P : PolK) : Factored(PolK) ==
        Q : PolK := gcd(P, D(P))
        A : PolK := P quo Q
        decomposition : Factored(PolK) := 1
        B : PolK

        for i in 1 .. repeat
            if i rem p ^= 0 then
                B := gcd(A, Q)
                decomposition := sqfrFactor(A quo B, i) * decomposition
                if B = 1 then leave
                A := B
            Q := Q quo A
        if Q ^= 1 then
           decomposition:=decomposition * rawMusser (charthRoot(Q)::PolK) ** p
        return decomposition

    Musser (P : PolK) : Factored(PolK) ==
        degree (P) = 0 => return P::Factored(PolK)
        if (lcP : K := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
        return lcP::PolK * rawMusser (P)

    oneYunStep (P : PolK) : oneYunStep2uple ==
        C : PolK := D (P) ;  A : PolK := gcd(P, C)
        gcd_P_P' : PolK := A ;  B : PolK := P quo A
```

```
        result : tableOfSquareFreePolynomials := empty ()
        i : Integer := 1

        repeat
            C := (C quo A) - D(B)
            if C = 0 then leave
            A := gcd(B, C)
            if A ^= 1 then
                result (i) := A
                B := B quo A
            i := i + 1
        result (i) := B
        return [result, gcd_P_P']

    rawYun (P : PolK) : tableOfSquareFreePolynomials ==
        u : oneYunStep2uple := oneYunStep (P)
        gcd_P_P' : PolK := u.gcdOfArgumentAndDerivative
        U : tableOfSquareFreePolynomials := u.simpleDecomposition

        R : PolK := gcd_P_P'
        for j in indices (U) repeat
            for k in 1 .. j-1 repeat
                R := R quo U(j)
        if R = 1 then return U
        V : tableOfSquareFreePolynomials := rawYun (charthRoot (R)::PolK)

        result : tableOfSquareFreePolynomials := empty ()
        gcd_Uj_Vk : PolK ;
        for k in indices (V) repeat    -- boucle 1

            for j in  indices (U) | not (U(j) = 1)  repeat    -- boucle 2
                gcd_Uj_Vk := gcd (U(j), V(k))
                if not (gcd_Uj_Vk = 1) then
                    result (j+p*k) := gcd_Uj_Vk
                    V (k) := V(k) quo gcd_Uj_Vk
                    U (j) := U(j) quo gcd_Uj_Vk
                if V(k) = 1 then leave

            if not (V(k) = 1) then
                result (p*k):= V (k)

        for j in  indices (U) | not (U(j) = 1)  repeat    -- boucle 3
            result (j) := U (j)

        return result

    Yun(P : PolK) : Factored(PolK) ==
        degree (P) = 0  => P::Factored(PolK)
        if (lcP := leadingCoefficient (P)) ^= 1 then P := inv (lcP)*P
        U : tableOfSquareFreePolynomials := rawYun (P)
```

```
        PFactored : Factored(PolK) := 1
        for i in indices (U) repeat
            PFactored := PFactored * sqfrFactor (U (i), i)
        return (lcP::PolK) * PFactored
```

———-

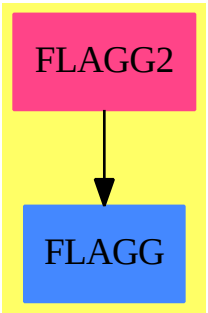
— **FFSQFR.dotabb** —

```
"FFSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFSQFR" -> "PFECAT"
```

———-

7.18 package FLAGG2 FiniteLinearAggregateFunctions2

7.18.1 FiniteLinearAggregateFunctions2 (FLAGG2)



Exports:
map reduce scan

— **package FLAGG2 FiniteLinearAggregateFunctions2** —

```
)abbrev package FLAGG2 FiniteLinearAggregateFunctions2
++ Author: ???
++ Date Created: ???
++ Date Last Updated: ???
++ Description:
++ FiniteLinearAggregateFunctions2 provides functions involving two
++ FiniteLinearAggregates where the underlying domains might be
++ different. An example of this might be creating a list of rational
```

```
++ numbers by mapping a function across a list of integers where the
++ function divides each integer by 1000.

FiniteLinearAggregateFunctions2(S, A, R, B):
 Exports == Implementation where
  S, R: Type
  A   : FiniteLinearAggregate S
  B   : FiniteLinearAggregate R

  Exports ==> with
    map    : (S -> R, A) -> B
     ++ map(f,a) applies function f to each member of aggregate
     ++ \spad{a} resulting in a new aggregate over a
     ++ possibly different underlying domain.
    reduce : ((S, R) -> R, A, R) -> R
     ++ reduce(f,a,r) applies function f to each
     ++ successive element of the
     ++ aggregate \spad{a} and an accumulant initialized to r.
     ++ For example,
     ++ \spad{reduce(_+$Integer,[1,2,3],0)}
     ++ does \spad{3+(2+(1+0))}. Note that third argument r
     ++ may be regarded as the
     ++ identity element for the function f.
    scan   : ((S, R) -> R, A, R) -> B
     ++ scan(f,a,r) successively applies
     ++ \spad{reduce(f,x,r)} to more and more leading sub-aggregates
     ++ x of aggregrate \spad{a}.
     ++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
     ++ \spad{scan(f,a,r)} returns
     ++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
  Implementation ==> add
    if A has ListAggregate(S) then         -- A is a list-oid
      reduce(fn, l, ident) ==
        empty? l => ident
        reduce(fn, rest l, fn(first l, ident))

      if B has ListAggregate(R) or not(B has shallowlyMutable) then
        -- A is a list-oid, and B is either list-oids or not mutable
        map(f, l) == construct [f s for s in entries l]

        scan(fn, l, ident) ==
          empty? l => empty()
          val := fn(first l, ident)
          concat(val, scan(fn, rest l, val))

      else                      -- A is a list-oid, B a mutable array-oid
        map(f, l) ==
          i := minIndex(w := new(#l,NIL$Lisp)$B)
          for a in entries l repeat (qsetelt_!(w, i, f a); i := inc i)
          w
```

```
      scan(fn, l, ident) ==
        i := minIndex(w := new(#l,NIL$Lisp)$B)
        vl := ident
        for a in entries l repeat
          vl := qsetelt_!(w, i, fn(a, vl))
          i := inc i
        w

    else                              -- A is an array-oid
      reduce(fn, v, ident) ==
        val := ident
        for i in minIndex v .. maxIndex v repeat
          val := fn(qelt(v, i), val)
        val

      if B has ListAggregate(R) then   -- A is an array-oid, B a list-oid
        map(f, v) ==
          construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

        scan(fn, v, ident) ==
          w := empty()$B
          for i in minIndex v .. maxIndex v repeat
            ident := fn(qelt(v, i), ident)
            w := concat(ident, w)
          reverse_! w

      else                             -- A and B are array-oid's
        if B has shallowlyMutable then -- B is also mutable
          map(f, v) ==
            w := new(#v,NIL$Lisp)$B
            for i in minIndex w .. maxIndex w repeat
              qsetelt_!(w, i, f qelt(v, i))
            w

          scan(fn, v, ident) ==
            w   := new(#v,NIL$Lisp)$B
            vl := ident
            for i in minIndex v .. maxIndex v repeat
              vl := qsetelt_!(w, i, fn(qelt(v, i), vl))
            w

        else                                  -- B non mutable array-oid
          map(f, v) ==
            construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

          scan(fn, v, ident) ==
            w := empty()$B
            for i in minIndex v .. maxIndex v repeat
              ident := fn(qelt(v, i), ident)
```

```
    w := concat(w, ident)
  w
```

———-

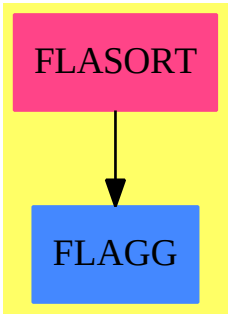
— **FLAGG2.dotabb** —

```
"FLAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLAGG2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG2" -> "FLAGG"
```

———-

7.19 package FLASORT FiniteLinearAggregateSort

7.19.1 FiniteLinearAggregateSort (FLASORT)



Exports:
heapSort quickSort shellSort

— **package FLASORT FiniteLinearAggregateSort** —

```
)abbrev package FLASORT FiniteLinearAggregateSort
++ Author: Michael Monagan Sep/88
++ RelatedOperations: sort
++ Description:
++ This package exports 3 sorting algorithms which work over
++ FiniteLinearAggregates.
++ Sort package (in-place) for shallowlyMutable Finite Linear Aggregates
-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains
```

```
)bo PUSH('FiniteLinearAggregateSort, $mutableDomains)

FiniteLinearAggregateSort(S, V): Exports == Implementation where
  S: Type
  V: FiniteLinearAggregate(S) with shallowlyMutable

  B ==> Boolean
  I ==> Integer

  Exports ==> with
    quickSort: ((S, S) -> B, V) -> V
      ++ quickSort(f, agg) sorts the aggregate agg with the ordering function
      ++ f using the quicksort algorithm.
    heapSort : ((S, S) -> B, V) -> V
      ++ heapSort(f, agg) sorts the aggregate agg with the ordering function
      ++ f using the heapsort algorithm.
    shellSort: ((S, S) -> B, V) -> V
      ++ shellSort(f, agg) sorts the aggregate agg with the ordering function
      ++ f using the shellSort algorithm.

  Implementation ==> add
    siftUp   : ((S, S) -> B, V, I, I) -> Void
    partition: ((S, S) -> B, V, I, I, I) -> I
    QuickSort: ((S, S) -> B, V, I, I) -> V

    quickSort(l, r) == QuickSort(l, r, minIndex r, maxIndex r)

    siftUp(l, r, i, n) ==
      t := qelt(r, i)
      while (j := 2*i+1) < n repeat
        if (k := j+1) < n and l(qelt(r, j), qelt(r, k)) then j := k
        if l(t,qelt(r,j)) then
           qsetelt_!(r, i, qelt(r, j))
           qsetelt_!(r, j, t)
           i := j
        else leave

    heapSort(l, r) ==
      not zero? minIndex r => error "not implemented"
      n := (#r)::I
      for k in shift(n,-1) - 1 .. 0 by -1 repeat siftUp(l, r, k, n)
      for k in n-1 .. 1 by -1 repeat
         swap_!(r, 0, k)
         siftUp(l, r, 0, k)
      r

    partition(l, r, i, j, k) ==
      -- partition r[i..j] such that r.s <= r.k <= r.t
      x := qelt(r, k)
```

```
      t := qelt(r, i)
      qsetelt_!(r, k, qelt(r, j))
      while i < j repeat
         if l(x,t) then
           qsetelt_!(r, j, t)
           j := j-1
           t := qsetelt_!(r, i, qelt(r, j))
         else (i := i+1; t := qelt(r, i))
      qsetelt_!(r, j, x)
      j

    QuickSort(l, r, i, j) ==
      n := j - i
--      if one? n and l(qelt(r, j), qelt(r, i)) then swap_!(r, i, j)
      if (n = 1) and l(qelt(r, j), qelt(r, i)) then swap_!(r, i, j)
      n < 2 => return r
      -- for the moment split at the middle item
      k := partition(l, r, i, j, i + shift(n,-1))
      QuickSort(l, r, i, k - 1)
      QuickSort(l, r, k + 1, j)

    shellSort(l, r) ==
      m := minIndex r
      n := maxIndex r
      -- use Knuths gap sequence: 1,4,13,40,121,...
      g := 1
      while g <= (n-m) repeat g := 3*g+1
      g := g quo 3
      while g > 0 repeat
         for i in m+g..n repeat
            j := i-g
            while j >= m and l(qelt(r, j+g), qelt(r, j)) repeat
               swap_!(r,j,j+g)
               j := j-g
         g := g quo 3
      r
```

———-

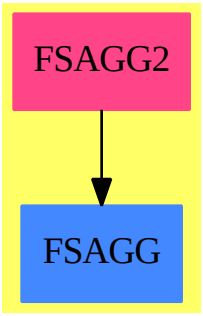
— FLASORT.dotabb —

```
"FLASORT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLASORT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLASORT" -> "FLAGG"
```

———-

7.20 package FSAGG2 FiniteSetAggregateFunctions2

7.20.1 FiniteSetAggregateFunctions2 (FSAGG2)



Exports:
map reduce scan

— package FSAGG2 FiniteSetAggregateFunctions2 —

```
)abbrev package FSAGG2 FiniteSetAggregateFunctions2
++ Author: Robert S. Sutor
++ Date Created: 15 May 1990
++ Date Last Updated: 14 Oct 1993
++ Description:
++ FiniteSetAggregateFunctions2 provides functions involving two
++ finite set aggregates where the underlying domains might be
++ different. An example of this is to create a set of rational
++ numbers by mapping a function across a set of integers, where the
++ function divides each integer by 1000.

FiniteSetAggregateFunctions2(S, A, R, B): Exports == Implementation where
   S, R: SetCategory
   A   : FiniteSetAggregate S
   B   : FiniteSetAggregate R

   Exports ==> with
     map    : (S -> R, A) -> B
      ++ map(f,a) applies function f to each member of
      ++ aggregate \spad{a}, creating a new aggregate with
      ++ a possibly different underlying domain.
     reduce : ((S, R) -> R, A, R) -> R
      ++ reduce(f,a,r) applies function f to each
      ++ successive element of the aggregate \spad{a} and an
      ++ accumulant initialised to r.
      ++ For example,
      ++ \spad{reduce(_+$Integer,[1,2,3],0)}
```

```
      ++ does a \spad{3+(2+(1+0))}.
      ++ Note that third argument r may be regarded
      ++ as an identity element for the function.
    scan   : ((S, R) -> R, A, R) -> B
      ++ scan(f,a,r) successively applies \spad{reduce(f,x,r)}
      ++ to more and more leading sub-aggregates x of
      ++ aggregate \spad{a}.
      ++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
      ++ \spad{scan(f,a,r)} returns
      ++ \spad {[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
  Implementation ==> add
    map(fn, a) ==
      set(map(fn, parts a)$ListFunctions2(S, R))$B
    reduce(fn, a, ident) ==
      reduce(fn, parts a, ident)$ListFunctions2(S, R)
    scan(fn, a, ident) ==
      set(scan(fn, parts a, ident)$ListFunctions2(S, R))$B
```

———-

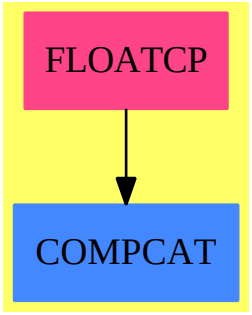
— **FSAGG2.dotabb** —

```
"FSAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSAGG2"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"FSAGG2" -> "FSAGG"
```

———-

7.21 package FLOATCP FloatingComplexPackage

7.21.1 FloatingComplexPackage (FLOATCP)



Exports:
complexRoots complexSolve

— package FLOATCP FloatingComplexPackage —

```
)abbrev package FLOATCP FloatingComplexPackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: SystemSolvePackage, RadicalSolvePackage,
++ FloatingRealPackage
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is a package for the approximation of complex solutions for
++ systems of equations of rational functions with complex rational
++ coefficients. The results are expressed as either complex rational
++ numbers or complex floats depending on the type of the precision
++ parameter which can be either a rational number or a floating point number.

FloatingComplexPackage(Par): Cat == Cap where
    Par : Join(Field, OrderedRing)
    K   ==> GI
    FPK ==> Fraction P K
    C   ==> Complex
    I        ==> Integer
    NNI      ==> NonNegativeInteger
    P        ==> Polynomial
    EQ       ==> Equation
    L        ==> List
    SUP      ==> SparseUnivariatePolynomial
    RN       ==> Fraction Integer
    NF       ==> Float
    CF       ==> Complex Float
    GI       ==> Complex Integer
    GRN      ==> Complex RN
    SE       ==> Symbol
    RFI      ==> Fraction P I
    INFSP ==> InnerNumericFloatSolvePackage


    Cat == with

       complexSolve:    (L FPK,Par) -> L L EQ P C Par
         ++ complexSolve(lp,eps) finds all the complex solutions to
         ++ precision eps of the system lp of rational functions
```

```
      ++ over the complex rationals with respect to all the
      ++ variables appearing in lp.

    complexSolve:    (L EQ FPK,Par) -> L L EQ P C Par
      ++ complexSolve(leq,eps) finds all the complex solutions
      ++ to precision eps of the system leq of equations
      ++ of rational functions over complex rationals
      ++ with respect to all the variables appearing in lp.

    complexSolve:    (FPK,Par) -> L EQ P C Par
      ++ complexSolve(p,eps) find all the complex solutions of the
      ++ rational function p with complex rational coefficients
      ++ with respect to all the variables appearing in p,
      ++ with precision eps.

    complexSolve:    (EQ FPK,Par) ->  L EQ P C Par
      ++ complexSolve(eq,eps) finds all the complex solutions of the
      ++ equation eq of rational functions with rational rational coefficients
      ++ with respect to all the variables appearing in eq,
      ++ with precision eps.

    complexRoots : (FPK,Par) -> L C Par
      ++ complexRoots(rf, eps) finds all the complex solutions of a
      ++ univariate rational function with rational number coefficients.
      ++ The solutions are computed to precision eps.

    complexRoots : (L FPK,L SE,Par) -> L L C Par
      ++ complexRoots(lrf, lv, eps) finds all the complex solutions of a
      ++ list of rational functions with rational number coefficients
      ++ with respect the the variables appearing in lv.
      ++ Each solution is computed to precision eps and returned as
      ++ list corresponding to the order of variables in lv.

  Cap == add

    -- find the complex zeros of an univariate polynomial --
    complexRoots(q:FPK,eps:Par) : L C Par ==
      p:=numer q
      complexZeros(univariate p,eps)$ComplexRootPackage(SUP GI, Par)

    -- find the complex zeros of an univariate polynomial --
    complexRoots(lp:L FPK,lv:L SE,eps:Par) : L L C Par ==
      lnum:=[numer p for p in lp]
      lden:=[dp for p in lp |(dp:=denom p)^=1]
      innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)

    complexSolve(lp:L FPK,eps : Par) : L L EQ  P C Par ==
      lnum:=[numer p for p in lp]
      lden:=[dp for p in lp |(dp:=denom p)^=1]
      lv:="setUnion"/[variables np for np in lnum]
```

```
      if lden^=[] then
       lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
      [[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
        for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

    complexSolve(le:L EQ FPK,eps : Par) : L L EQ  P C Par ==
      lp:=[lhs ep - rhs ep for ep in le]
      lnum:=[numer p for p in lp]
      lden:=[dp for p in lp |(dp:=denom p)^=1]
      lv:="setUnion"/[variables np for np in lnum]
      if lden^=[] then
       lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
      [[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
        for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

    complexSolve(p : FPK,eps : Par) : L EQ  P C Par ==
      (mvar := mainVariable numer p ) case "failed" =>
              error "no variable found"
      x:P C Par:=mvar::SE::(P C Par)
      [equation(x,val::(P C Par)) for val in complexRoots(p,eps)]

    complexSolve(eq : EQ FPK,eps : Par) : L EQ  P C Par ==
      complexSolve(lhs eq - rhs eq,eps)
```

———-

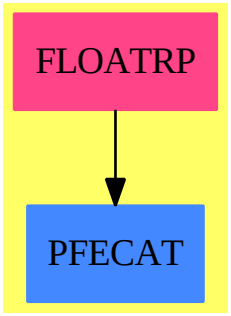
— **FLOATCP.dotabb** —

```
"FLOATCP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLOATCP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"FLOATCP" -> "COMPCAT"
```

———-

7.22 package FLOATRP FloatingRealPackage

7.22.1 FloatingRealPackage (FLOATRP)



Exports:
realRoots solve

— package FLOATRP FloatingRealPackage —

```
)abbrev package FLOATRP FloatingRealPackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: SystemSolvePackage, RadicalSolvePackage,
++ FloatingComplexPackage
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is a package for the approximation of real solutions for
++ systems of polynomial equations over the rational numbers.
++ The results are expressed as either rational numbers or floats
++ depending on the type of the precision parameter which can be
++ either a rational number or a floating point number.

FloatingRealPackage(Par): Cat == Cap where
    I        ==> Integer
    NNI      ==> NonNegativeInteger
    P        ==> Polynomial
    EQ       ==> Equation
    L        ==> List
    SUP      ==> SparseUnivariatePolynomial
    RN       ==> Fraction Integer
    NF       ==> Float
```

```
CF       ==> Complex Float
GI       ==> Complex Integer
GRN      ==> Complex RN
SE       ==> Symbol
RFI      ==> Fraction P I
INFSP ==> InnerNumericFloatSolvePackage

Par : Join(OrderedRing, Field)  -- RN or NewFloat

Cat == with

   solve:    (L RFI,Par) -> L L EQ P Par
     ++ solve(lp,eps) finds all of the real solutions of the
     ++ system lp of rational functions over the rational numbers
     ++ with respect to all the variables appearing in lp,
     ++ with precision eps.

   solve:    (L EQ RFI,Par) -> L L EQ P Par
     ++ solve(leq,eps) finds all of the real solutions of the
     ++ system leq of equationas of rational functions
     ++ with respect to all the variables appearing in lp,
     ++ with precision eps.

   solve:    (RFI,Par) ->  L EQ P Par
     ++ solve(p,eps) finds all of the real solutions of the
     ++ univariate rational function p with rational coefficients
     ++ with respect to the unique variable appearing in p,
     ++ with precision eps.

   solve:    (EQ RFI,Par) ->  L EQ P Par
     ++ solve(eq,eps) finds all of the real solutions of the
     ++ univariate equation eq of rational functions
     ++ with respect to the unique variables appearing in eq,
     ++ with precision eps.

   realRoots:    (L RFI,L SE,Par) -> L L Par
     ++ realRoots(lp,lv,eps) computes the list of the real
     ++ solutions of the list lp of rational functions with rational
     ++ coefficients with respect to the variables in lv,
     ++ with precision eps. Each solution is expressed as a list
     ++ of numbers in order corresponding to the variables in lv.

   realRoots : (RFI,Par) -> L Par
     ++ realRoots(rf, eps) finds the real zeros of a univariate
     ++ rational function with precision given by eps.

Cap == add

   makeEq(nres:L Par,lv:L SE) : L EQ P Par ==
       [equation(x::(P Par),r::(P Par)) for x in lv for r in nres]
```

```
-- find the real zeros of an univariate rational polynomial --
realRoots(p:RFI,eps:Par) : L Par ==
  innerSolve1(numer p,eps)$INFSP(I,Par,Par)

-- real zeros of the system of polynomial lp --
realRoots(lp:L RFI,lv:L SE,eps: Par) : L L Par ==
  lnum:=[numer p for p in lp]
  lden:=[dp for p in lp |(dp:=denom p)^=1]
  innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)

solve(lp:L RFI,eps : Par) : L L EQ  P Par ==
  lnum:=[numer p for p in lp]
  lden:=[dp for p in lp |(dp:=denom p)^=1]
  lv:="setUnion"/[variables np for np in lnum]
  if lden^=[] then
   lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
  [makeEq(numres,lv) for numres
     in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(le:L EQ RFI,eps : Par) : L L EQ  P Par ==
  lp:=[lhs ep - rhs ep for ep in le]
  lnum:=[numer p for p in lp]
  lden:=[dp for p in lp |(dp:=denom p)^=1]
  lv:="setUnion"/[variables np for np in lnum]
  if lden^=[] then
   lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
  [makeEq(numres,lv) for numres
    in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(p : RFI,eps : Par) :  L EQ  P Par ==
  (mvar := mainVariable numer p ) case "failed" =>
       error "no variable found"
  x:P Par:=mvar::SE::(P Par)
  [equation(x,val::(P Par)) for val in realRoots(p,eps)]

solve(eq : EQ RFI,eps : Par) :  L EQ  P Par ==
  solve(lhs eq - rhs eq,eps)
```

———-

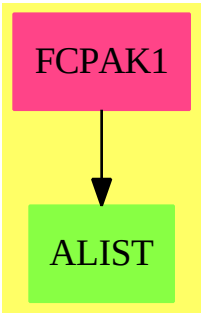
— FLOATRP.dotabb —

```
"FLOATRP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLOATRP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FLOATRP" -> "PFECAT"
```

———-

7.23 package FCPAK1 FortranCodePackage1

7.23.1 FortranCodePackage1 (FCPAK1)



Exports:
identitySquareMatrix zeroMatrix zeroSquareMatrix zeroVector

— package FCPAK1 FortranCodePackage1 —

```
)abbrev package FCPAK1 FortranCodePackage1
++ Author: Grant Keady and Godfrey Nolan
++ Date Created: April 1993
++ Date Last Updated:
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{FortranCodePackage1} provides some utilities for
++ producing useful objects in FortranCode domain.
++ The Package may be used with the FortranCode domain and its
++ \spad{printCode} or possibly via an outputAsFortran.
++ (The package provides items of use in connection with ASPs
++ in the AXIOM-NAG link and, where appropriate, naming accords
++ with that in IRENA.)
++ The easy-to-use functions use Fortran loop variables I1, I2,
++ and it is users' responsibility to check that this is sensible.
++ The advanced functions use SegmentBinding to allow users control
++ over Fortran loop variable names.
-- Later might add functions to build
-- diagonalMatrix from List, i.e. the FC version of the corresponding
-- AXIOM function from MatrixCategory;
-- bandedMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedMatrix Domain
```

```
-- bandedSymmetricMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedSymmetricMatrix Domain

FortranCodePackage1: Exports  == Implementation where

  NNI    ==> NonNegativeInteger
  PI     ==> PositiveInteger
  PIN    ==> Polynomial(Integer)
  SBINT  ==> SegmentBinding(Integer)
  SEGINT ==> Segment(Integer)
  LSBINT ==> List(SegmentBinding(Integer))
  SBPIN  ==> SegmentBinding(Polynomial(Integer))
  SEGPIN ==> Segment(Polynomial(Integer))
  LSBPIN ==> List(SegmentBinding(Polynomial(Integer)))
  FC     ==> FortranCode
  EXPRESSION  ==> Union(Expression Integer,Expression Float,Expression Complex Integer,Expressio

  Exports == with

    zeroVector: (Symbol,PIN) -> FC
      ++ zeroVector(s,p) \undocumented{}

    zeroMatrix: (Symbol,PIN,PIN) -> FC
      ++ zeroMatrix(s,p,q) uses loop variables in the Fortran, I1 and I2

    zeroMatrix: (Symbol,SBPIN,SBPIN) -> FC
      ++ zeroMatrix(s,b,d) in this version gives the user control
      ++ over names of Fortran variables used in loops.

    zeroSquareMatrix: (Symbol,PIN) -> FC
      ++ zeroSquareMatrix(s,p) \undocumented{}

    identitySquareMatrix: (Symbol,PIN) -> FC
      ++ identitySquareMatrix(s,p) \undocumented{}

  Implementation ==> add
    import FC

    zeroVector(fname:Symbol,n:PIN):FC ==
      ue:Expression(Integer) := 0
      i1:Symbol := "I1"::Symbol
      lp1:PIN := 1::PIN
      hp1:PIN := n
      segp1:SEGPIN:= segment(lp1,hp1)$SEGPIN
      segbp1:SBPIN := equation(i1,segp1)$SBPIN
      ip1:PIN := i1::PIN
      indices:List(PIN) := [ip1]
      fa:FC := forLoop(segbp1,assign(fname,indices,ue)$FC)$FC
      fa
```

```
zeroMatrix(fname:Symbol,m:PIN,n:PIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := m
  segp1:SEGPIN:= segment(lp1,hp1)$SEGPIN
  segbp1:SBPIN := equation(i1,segp1)$SBPIN
  i2:Symbol := "I2"::Symbol
  hp2:PIN := n
  segp2:SEGPIN:= segment(lp1,hp2)$SEGPIN
  segbp2:SBPIN := equation(i2,segp2)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

zeroMatrix(fname:Symbol,segbp1:SBPIN,segbp2:SBPIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := variable(segbp1)$SBPIN
  i2:Symbol := variable(segbp2)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

zeroSquareMatrix(fname:Symbol,n:PIN):FC ==
  ue:Expression(Integer) := 0
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := n
  segp1:SEGPIN:= segment(lp1,hp1)$SEGPIN
  segbp1:SBPIN := equation(i1,segp1)$SBPIN
  i2:Symbol := "I2"::Symbol
  segbp2:SBPIN := equation(i2,segp1)$SBPIN
  ip1:PIN := i1::PIN
  ip2:PIN := i2::PIN
  indices:List(PIN) := [ip1,ip2]
  fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
  fa

identitySquareMatrix(fname:Symbol,n:PIN):FC ==
  ue:Expression(Integer) := 0
  u1:Expression(Integer) := 1
  i1:Symbol := "I1"::Symbol
  lp1:PIN := 1::PIN
  hp1:PIN := n
  segp1:SEGPIN:= segment(lp1,hp1)$SEGPIN
  segbp1:SBPIN := equation(i1,segp1)$SBPIN
```

```
i2:Symbol := "I2"::Symbol
segbp2:SBPIN := equation(i2,segp1)$SBPIN
ip1:PIN := i1::PIN
ip2:PIN := i2::PIN
indice1:List(PIN) := [ip1,ip1]
indices:List(PIN) := [ip1,ip2]
fc:FC := forLoop(segbp2,assign(fname,indices,ue)$FC)$FC
f1:FC := assign(fname,indice1,u1)$FC
fl:List(FC) := [fc,f1]
fa:FC := forLoop(segbp1,block(fl)$FC)$FC
fa
```

———-

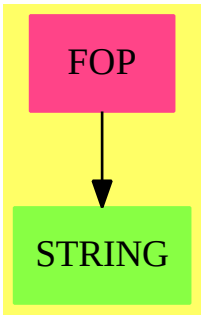
— FCPAK1.dotabb —

```
"FCPAK1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FCPAK1"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"FCPAK1" -> "ALIST"
```

———-

7.24 package FOP FortranOutputStackPackage

7.24.1 FortranOutputStackPackage (FOP)



Exports:

popFortranOutputStack	clearFortranOutputStack	pushFortranOutputStack
pushFortranOutputStack	showFortranOutputStack	topFortranOutputStack

— package FOP FortranOutputStackPackage —

```
)abbrev package FOP FortranOutputStackPackage
```

```
-- Because of a bug in the compiler:
)bo $noSubsumption:=false

++ Author: Mike Dewar
++ Date Created:  October 1992
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Code to manipulate Fortran Output Stack

FortranOutputStackPackage() : specification == implementation where

  specification == with

    clearFortranOutputStack : () -> Stack String
      ++ clearFortranOutputStack() clears the Fortran output stack
    showFortranOutputStack : () -> Stack String
      ++ showFortranOutputStack() returns the Fortran output stack
    popFortranOutputStack : () -> Void
      ++ popFortranOutputStack() pops the Fortran output stack
    pushFortranOutputStack : FileName -> Void
      ++ pushFortranOutputStack(f) pushes f onto the Fortran output stack
    pushFortranOutputStack : String -> Void
      ++ pushFortranOutputStack(f) pushes f onto the Fortran output stack
    topFortranOutputStack : () -> String
      ++ topFortranOutputStack() returns the top element of the Fortran
      ++ output stack

  implementation == add

    import MoreSystemCommands

    -- A stack of filenames for Fortran output.  We are sharing this with
    -- the standard Fortran output code, so want to be a bit careful about
    -- how we interact with what the user does independently.  We get round
    -- potential problems by always examining the top element of the stack
    -- before we push.  If the user has redirected output then we alter our
    -- top value accordingly.
    fortranOutputStack : Stack String := empty()@(Stack String)

    topFortranOutputStack():String == string(_$fortranOutputFile$Lisp)

    pushFortranOutputStack(fn:FileName):Void ==
      if empty? fortranOutputStack then
```

```
      push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
      pop! fortranOutputStack
      push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    push!( fn::String,fortranOutputStack)
    systemCommand concat(["set output fortran quiet ", fn::String])$String
    void()

  pushFortranOutputStack(fn:String):Void ==
    if empty? fortranOutputStack then
      push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
      pop! fortranOutputStack
      push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    push!( fn,fortranOutputStack)
    systemCommand concat(["set output fortran quiet ", fn])$String
    void()

  popFortranOutputStack():Void ==
    if not empty? fortranOutputStack then pop! fortranOutputStack
    if empty? fortranOutputStack then push!("CONSOLE",fortranOutputStack)
    systemCommand concat(["set output fortran quiet append ",_
                          top fortranOutputStack])$String
    void()

  clearFortranOutputStack():Stack String ==
    fortranOutputStack := empty()@(Stack String)

  showFortranOutputStack():Stack String ==
    fortranOutputStack
```

———-

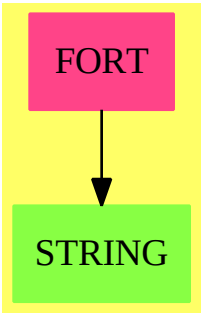
— FOP.dotabb —

```
"FOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FOP"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FOP" -> "STRING"
```

———-

7.25 package FORT FortranPackage

7.25.1 FortranPackage (FORT)



Exports:
outputAsFortran linkToFortran setLegalFortranSourceExtensions

— **package FORT FortranPackage** —

```
)abbrev package FORT FortranPackage
-- Because of a bug in the compiler:
)bo $noSubsumption:=true

++ Author: Mike Dewar
++ Date Created: October 6 1991
++ Date Last Updated: 13 July 1994
++ Basic Operations: linkToFortran
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ provides an interface to the boot code for calling Fortran

FortranPackage(): Exports == Implementation where
 FST ==> FortranScalarType
 SEX ==> SExpression
 L   ==> List
 S   ==> Symbol
 FOP ==> FortranOutputStackPackage
 U   ==> Union(array:L S,scalar:S)

 Exports ==> with
  linkToFortran: (S, L U, L L U, L S) -> SEX
    ++ linkToFortran(s,l,ll,lv) \undocumented{}
```

```
  linkToFortran: (S, L U, L L U, L S, S) -> SEX
    ++ linkToFortran(s,l,ll,lv,t) \undocumented{}
  linkToFortran: (S,L S,TheSymbolTable,L S) -> SEX
    ++ linkToFortran(s,l,t,lv) \undocumented{}
  outputAsFortran: FileName -> Void
    ++ outputAsFortran(fn) \undocumented{}
  setLegalFortranSourceExtensions: List String -> List String
    ++ setLegalFortranSourceExtensions(l) \undocumented{}

 Implementation ==> add

  legalFortranSourceExtensions : List String := ["f"]

  setLegalFortranSourceExtensions(l:List String):List String ==
    legalFortranSourceExtensions := l

  checkExtension(fn : FileName) : String ==
    -- Does it end in a legal extension ?
    stringFn := fn::String
    not member?(extension fn,legalFortranSourceExtensions) =>
      error [stringFn,"is not a legal Fortran Source File."]
    stringFn

  outputAsFortran(fn:FileName):Void ==
--    source : String := checkExtension fn
    source : String := fn::String
    not readable? fn =>
      popFortranOutputStack()$FOP
      error([source,"is not readable"]@List(String))
    target : String := topFortranOutputStack()$FOP
    command : String :=
      concat(["sys rm -f ",target," ; cp ",source," ",target])$String
    systemCommand(command)$MoreSystemCommands
    void()$Void

  linkToFortran(name:S,args:L U, decls:L L U, res:L(S)):SEX ==
    makeFort(name,args,decls,res,NIL$Lisp,NIL$Lisp)$Lisp

  linkToFortran(name:S,args:L U, decls:L L U, res:L(S),returnType:S):SEX ==
    makeFort(name,args,decls,res,returnType,NIL$Lisp)$Lisp

  dimensions(type:FortranType):SEX ==
    convert([convert(convert(u)@InputForm)@SEX _
      for u in dimensionsOf(type)])@SEX

  ftype(name:S,type:FortranType):SEX ==
    [name,scalarTypeOf(type),dimensions(type),external? type]$Lisp

  makeAspList(asp:S,syms:TheSymbolTable):SExpression==
    symtab : SymbolTable := symbolTableOf(asp,syms)
```

```
    [asp,returnTypeOf(asp,syms),argumentListOf(asp,syms), _
     [ftype(u,fortranTypeOf(u,symtab)) for u in parametersOf symtab]]$Lisp

  linkToFortran(name:S,aArgs:L S,syms:TheSymbolTable,res:L S):SEX ==
    arguments : L S := argumentListOf(name,syms)$TheSymbolTable
    dummies : L S := setDifference(arguments,aArgs)
    symbolTable:SymbolTable := symbolTableOf(name,syms)
    symbolList := newTypeLists(symbolTable)
    rt:Union(fst: FST,void: "void") := returnTypeOf(name,syms)$TheSymbolTable

    -- Look for arguments which are subprograms
    asps :=[makeAspList(u,syms) for u in externalList(symbolTable)$SymbolTable]
    rt case fst =>
      makeFort1(name,arguments,aArgs,dummies,symbolList,res,(rt.fst)::S,asps)$Lisp
    makeFort1(name,arguments,aArgs,dummies,symbolList,res,NIL$Lisp,asps)$Lisp
```

———-

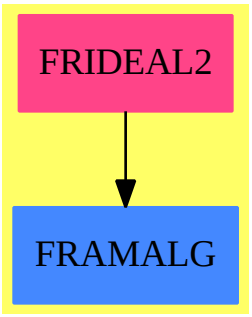
— **FORT.dotabb** —

```
"FORT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FORT" -> "STRING"
```

———-

7.26 package FRIDEAL2 FractionalIdealFunctions2

7.26.1 FractionalIdealFunctions2 (FRIDEAL2)



Exports:
map

— package FRIDEAL2 FractionalIdealFunctions2 —

```
)abbrev package FRIDEAL2 FractionalIdealFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1 Feb 1989
++ Date Last Updated: 27 Feb 1990
++ Keywords: ideal, algebra, module.
++ Description:
++ Lifting of morphisms to fractional ideals.

FractionalIdealFunctions2(R1, F1, U1, A1, R2, F2, U2, A2):
 Exports == Implementation where
  R1, R2: EuclideanDomain
  F1: QuotientFieldCategory R1
  U1: UnivariatePolynomialCategory F1
  A1: Join(FramedAlgebra(F1, U1), RetractableTo F1)
  F2: QuotientFieldCategory R2
  U2: UnivariatePolynomialCategory F2
  A2: Join(FramedAlgebra(F2, U2), RetractableTo F2)

  Exports ==> with
    map: (R1 -> R2, FractionalIdeal(R1, F1, U1, A1)) ->
                                        FractionalIdeal(R2, F2, U2, A2)
      ++ map(f,i) \undocumented{}

  Implementation ==> add
    fmap: (F1 -> F2, A1) -> A2

    fmap(f, a) ==
      v := coordinates a
      represents
        [f qelt(v, i) for i in minIndex v .. maxIndex v]$Vector(F2)

    map(f, i) ==
      b := basis i
      ideal [fmap(s +-> f(numer s) / f(denom s), qelt(b, j))
             for j in minIndex b .. maxIndex b]$Vector(A2)
```

———-

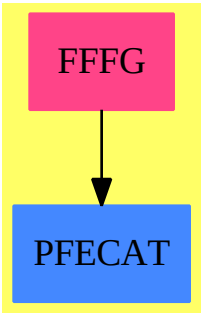
— FRIDEAL2.dotabb —

```
"FRIDEAL2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRIDEAL2"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FRIDEAL2" -> "FRAMALG"
```

———-

7.27 package FFFG FractionFreeFastGaussian

7.27.1 FractionFreeFastGaussian (FFFG)



Exports:

DiffAction	DiffC	ShiftAction	ShiftC
fffg	generalCoefficient	generalInterpolation	interpolate
qShiftAction	qShiftC		

— **package FFFG FractionFreeFastGaussian** —

```
)abbrev package FFFG FractionFreeFastGaussian
++ Author: Martin Rubey
++ Description:
++ This package implements the interpolation algorithm proposed in Beckermann,
++ Bernhard and Labahn, George, Fraction-free computation of matrix rational
++ interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and
++ Applications 22.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussian(D, V): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: AbelianMonoidRing(D, NonNegativeInteger) -- for example, SUP D

  SUP  ==> SparseUnivariatePolynomial

  cFunction ==> (NonNegativeInteger, Vector SUP D) -> D

  CoeffAction ==> (NonNegativeInteger, NonNegativeInteger, V) -> D

  Exports == with

    fffg: (List D, cFunction, List NonNegativeInteger) -> Matrix SUP D
      ++ \spad{fffg} is the general algorithm as proposed by Beckermann and
      ++ Labahn.
      ++
```

```
    ++ The first argument is the list of c_{i,i}.  These are the only values
    ++ of C explicitely needed in \spad{fffg}.
    ++
    ++ The second argument c, computes c_k(M), i.e., c_k(.) is the dual basis
    ++ of the vector space V, but also knows about the special multiplication
    ++ rule as descibed in Equation (2).  Note that the information about f
    ++ is therefore encoded in c.
    ++
    ++ The third argument is the vector of degree bounds n, as introduced in
    ++ Definition 2.1.  In particular, the sum of the entries is the order of
    ++ the Mahler system computed.

  interpolate: (List D, List D, NonNegativeInteger) -> Fraction SUP D
    ++ \spad{interpolate(xlist, ylist, deg} returns the rational function with
    ++ numerator degree at most \spad{deg} and denominator degree at most
    ++ \spad{#xlist-deg-1}  that interpolates the given points using
    ++ fraction free arithmetic. Note that rational interpolation does not
    ++ guarantee that all given points are interpolated correctly:
    ++ unattainable points may make this impossible.
```

```
The following function could be moved to FFFGF, parallel to
generalInterpolation.  However, the reason for moving
generalInterpolation for fractions to a separate package was the need of
a generic signature, hence the extra argument VF to FFFGF.  In the
special case of rational interpolation, this extra argument is not necessary,
since we are always returning a fraction of SUPs, and ignore V.  In
fact, V is not needed for fffg itself, only if we want to specify a
CoeffAction.

Thus, maybe it would be better to move fffg to a separate package?
```

— package FFFG FractionFreeFastGaussian —

```
  interpolate: (List Fraction D, List Fraction D, NonNegativeInteger)
              -> Fraction SUP D
    ++ \spad{interpolate(xlist, ylist, deg} returns the rational function with
    ++ numerator degree \spad{deg} that interpolates the given points using
    ++ fraction free arithmetic.

  generalInterpolation: (List D, CoeffAction,
                         Vector V, List NonNegativeInteger) -> Matrix SUP D
    ++ \spad{generalInterpolation(C, CA, f, eta)} performs Hermite-Pade
    ++ approximation using the given action CA of polynomials on the elements
    ++ of f. The result is guaranteed to be correct up to order
```

```
  ++ |eta|-1. Given that eta is a "normal" point, the degrees on the
  ++ diagonal are given by eta. The degrees of column i are in this case
  ++ eta + e.i - [1,1,...,1], where the degree of zero is -1.
  ++
  ++ The first argument C is the list of coefficients c_{k,k} in the
  ++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
  ++
  ++ The second argument, CA(k, l, f), should return the coefficient of x^k
  ++ in z^l f(x).

generalInterpolation: (List D, CoeffAction,
                       Vector V, NonNegativeInteger, NonNegativeInteger)
                      -> Stream Matrix SUP D
  ++ \spad{generalInterpolation(C, CA, f, sumEta, maxEta)} applies
  ++ \spad{generalInterpolation(C, CA, f, eta)} for all possible \spad{eta}
  ++ with maximal entry \spad{maxEta} and sum of entries at most
  ++ \spad{sumEta}.
  ++
  ++ The first argument C is the list of coefficients c_{k,k} in the
  ++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
  ++
  ++ The second argument, CA(k, l, f), should return the coefficient of x^k
  ++ in z^l f(x).

generalCoefficient: (CoeffAction, Vector V,
                     NonNegativeInteger, Vector SUP D) -> D
  ++ \spad{generalCoefficient(action, f, k, p)} gives the coefficient of
  ++ x^k in p(z)\dot f(x), where the action of z^l on a polynomial in x is
  ++ given by action, i.e., action(k, l, f) should return the coefficient
  ++ of x^k in z^l f(x).

ShiftAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
  ++ \spad{ShiftAction(k, l, g)} gives the coefficient of x^k in z^l g(x),
  ++ where \spad{z*(a+b*x+c*x^2+d*x^3+...) = (b*x+2*c*x^2+3*d*x^3+...)}. In
  ++ terms of sequences, z*u(n)=n*u(n).

ShiftC: NonNegativeInteger -> List D
  ++ \spad{ShiftC} gives the coefficients c_{k,k} in the expansion <x^k> z
  ++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
  ++ shifting. In fact, the result is [0,1,2,...]

DiffAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
  ++ \spad{DiffAction(k, l, g)} gives the coefficient of x^k in z^l g(x),
  ++ where z*(a+b*x+c*x^2+d*x^3+...) = (a*x+b*x^2+c*x^3+...), i.e.,
  ++ multiplication with x.

DiffC: NonNegativeInteger -> List D
  ++ \spad{DiffC} gives the coefficients c_{k,k} in the expansion <x^k> z
  ++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
  ++ shifting. In fact, the result is [0,0,0,...]
```

```
    qShiftAction: (D, NonNegativeInteger, NonNegativeInteger, V) -> D
      ++ \spad{qShiftAction(q, k, l, g)} gives the coefficient of x^k in z^l
      ++ g(x), where z*(a+b*x+c*x^2+d*x^3+...) =
      ++ (a+q*b*x+q^2*c*x^2+q^3*d*x^3+...). In terms of sequences,
      ++ z*u(n)=q^n*u(n).

    qShiftC: (D, NonNegativeInteger) -> List D
      ++ \spad{qShiftC} gives the coefficients c_{k,k} in the expansion <x^k> z
      ++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
      ++ shifting. In fact, the result is [1,q,q^2,...]

  Implementation ==> add

-------------------------------------------------------------------------------
-- Shift Operator
-------------------------------------------------------------------------------

-- ShiftAction(k, l, f) is the CoeffAction appropriate for the shift operator.

    ShiftAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
      k**l*coefficient(f, k)


    ShiftC(total: NonNegativeInteger): List D ==
      [i::D for i in 0..total-1]

-------------------------------------------------------------------------------
-- q-Shift Operator
-------------------------------------------------------------------------------

-- q-ShiftAction(k, l, f) is the CoeffAction appropriate for the q-shift operator.

    qShiftAction(q: D, k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
      q**(k*l)*coefficient(f, k)


    qShiftC(q: D, total: NonNegativeInteger): List D ==
      [q**i for i in 0..total-1]

-------------------------------------------------------------------------------
-- Differentiation Operator
-------------------------------------------------------------------------------

-- DiffAction(k, l, f) is the CoeffAction appropriate for the differentiation
-- operator.

    DiffAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V): D ==
      coefficient(f, (k-l)::NonNegativeInteger)
```

```
    DiffC(total: NonNegativeInteger): List D ==
      [0 for i in 1..total]

--------------------------------------------------------------------------------
-- general - suitable for functions f
--------------------------------------------------------------------------------

-- get the coefficient of z^k in the scalar product of p and f, the action
-- being defined by coeffAction

    generalCoefficient(coeffAction: CoeffAction, f: Vector V,
                       k: NonNegativeInteger, p: Vector SUP D): D ==
      res: D := 0
      for i in 1..#f repeat

-- Defining a and b and summing only over those coefficients that might be
-- nonzero makes a huge difference in speed
        a := f.i
        b := p.i
        for l in minimumDegree b..degree b repeat
            if not zero? coefficient(b, l)
            then res := res + coefficient(b, l) * coeffAction(k, l, a)
      res


    generalInterpolation(C: List D, coeffAction: CoeffAction,
                         f: Vector V,
                         eta: List NonNegativeInteger): Matrix SUP D ==

      c: cFunction := (x,y) +-> generalCoefficient(coeffAction, f,
                                        (x-1)::NonNegativeInteger, y)
      fffg(C, c, eta)



--------------------------------------------------------------------------------
-- general - suitable for functions f - trying all possible degree combinations
--------------------------------------------------------------------------------
```

———-

The following function returns the lexicographically next vector with non-negative components smaller than p with the same sum as v.

— package FFFG FractionFreeFastGaussian —

```
    nextVector!(p: NonNegativeInteger, v: List NonNegativeInteger)
```

```
          : Union("failed", List NonNegativeInteger) ==
  n := #v
  pos := position(x +-> x < p, v)
  zero? pos => return "failed"
  if pos = 1 then
    sum: Integer := v.1
    for i in 2..n repeat
      if v.i < p and sum > 0 then
        v.i := v.i + 1
        sum := sum - 1
        for j in 1..i-1 repeat
          if sum > p then
            v.j := p
            sum := sum - p
          else
            v.j := sum::NonNegativeInteger
            sum := 0
        return v
      else sum := sum + v.i
    return "failed"
  else
    v.pos     := v.pos + 1
    v.(pos-1) := (v.(pos-1) - 1)::NonNegativeInteger

  v
```

———-

The following function returns the stream of all possible degree vectors, beginning with v, where the degree vectors are sorted in reverse lexicographic order. Furthermore, the entries are all less or equal to p and their sum equals the sum of the entries of v. We assume that the entries of v are also all less or equal to p.

— **package FFFG FractionFreeFastGaussian** —

```
vectorStream(p: NonNegativeInteger, v: List NonNegativeInteger)
            : Stream List NonNegativeInteger == delay
  next := nextVector!(p, copy v)
  (next case "failed") => empty()$Stream(List NonNegativeInteger)
  cons(next, vectorStream(p, next))
```

———-

vectorStream2 skips every second entry of vectorStream.

— **package FFFG FractionFreeFastGaussian** —

```
vectorStream2(p: NonNegativeInteger, v: List NonNegativeInteger)
```

```
            : Stream List NonNegativeInteger == delay
      next := nextVector!(p, copy v)
      (next case "failed") => empty()$Stream(List NonNegativeInteger)
      next2 := nextVector!(p, copy next)
      (next2 case "failed") => cons(next, empty())
      cons(next2, vectorStream2(p, next2))
```

———-

This version of generalInterpolation returns a stream of solutions, one for each possible degree vector. Thus, it only needs to apply the previously defined generalInterpolation to each degree vector. These are generated by vectorStream and vectorStream2 respectively.

If f consists of two elements only, we can skip every second degree vector: note that fffg, and thus also generalInterpolation, returns a matrix with `#f` columns, each corresponding to a solution of the interpolation problem. More precisely, the i^{th} column is a solution with degrees $\eta-(1,1,\ldots,1)+e_i$. Thus, in the case of 2×2 matrices, vectorStream would produce solutions corresponding to $(d,0),(d-1,1);(d-1,1),(d-2,2);(d-2,2),(d-3,3)\ldots$, i.e., every second matrix is redundant.

Although some redundancy exists also for higher dimensional f, the scheme becomes much more complicated, thus we did not implement it.

— **package FFFG FractionFreeFastGaussian** —

```
   generalInterpolation(C: List D, coeffAction: CoeffAction,
                        f: Vector V,
                        sumEta: NonNegativeInteger,
                        maxEta: NonNegativeInteger)
                       : Stream Matrix SUP D ==

\getchunk{generate an initial degree vector}

      if #f = 2 then
        map(x +-> generalInterpolation(C, coeffAction, f, x),
            cons(eta, vectorStream2(maxEta, eta)))
           $StreamFunctions2(List NonNegativeInteger,
                             Matrix SUP D)
      else
        map(x +-> generalInterpolation(C, coeffAction, f, x),
            cons(eta, vectorStream(maxEta, eta)))
           $StreamFunctions2(List NonNegativeInteger,
                             Matrix SUP D)
```

———-

We need to generate an initial degree vector, being the minimal element in reverse lexicographic order, i.e., $m,m,\ldots,m,k,0,0,\ldots$, where m is maxEta and k is the remainder of sumEta divided by maxEta. This is done by the following code:

— **generate an initial degree vector** —

```
      sum: Integer := sumEta
      entry: Integer
      eta: List NonNegativeInteger
          := [(if sum < maxEta _
               then (entry := sum; sum := 0) _
               else (entry := maxEta; sum := sum - maxEta); _
               entry::NonNegativeInteger) for i in 1..#f]
```

———-

We want to generate all vectors with sum of entries being at most sumEta. Therefore the following is incorrect.

— **BUG generate an initial degree vector** —

```
-- (sum > 0) => empty()$Stream(Matrix SUP D)
```

———-

— **package FFFG FractionFreeFastGaussian** —

```
-------------------------------------------------------------------------
-- rational interpolation
-------------------------------------------------------------------------

    interpolate(x: List Fraction D, y: List Fraction D, d: NonNegativeInteger)
               : Fraction SUP D ==
      gx := splitDenominator(x)$InnerCommonDenominator(D, Fraction D, _
                                                       List D, _
                                                       List Fraction D)
      gy := splitDenominator(y)$InnerCommonDenominator(D, Fraction D, _
                                                       List D, _
                                                       List Fraction D)
      r := interpolate(gx.num, gy.num, d)
      elt(numer r, monomial(gx.den,1))/(gy.den*elt(denom r, monomial(gx.den,1)))


    interpolate(x: List D, y: List D, d: NonNegativeInteger): Fraction SUP D ==
-- berechne Interpolante mit Graden d und N-d-1
      if (N := #x) ~= #y then
        error "interpolate: number of points and values must match"
      if N <= d then
        error "interpolate: numerator degree must be smaller than number of data points"
      c: cFunction := (s,u) +-> y.s * elt(u.2, x.s) - elt(u.1, x.s)
      eta: List NonNegativeInteger := [d, (N-d)::NonNegativeInteger]
      M := fffg(x, c, eta)
```

```
      if zero?(M.(2,1)) then M.(1,2)/M.(2,2)
                        else M.(1,1)/M.(2,1)
```

———-

Because of Lemma 5.3, M.1.(2,1) and M.1.(2,2) cannot both vanish, since `eta_sigma` is always σ-normal by Theorem 7.2 and therefore also para-normal, see Definition 4.2.

Because of Lemma 5.1 we have that M.1.(*,2) is a solution of the interpolation problem, if M.1.(2,1) vanishes.

— **package FFFG FractionFreeFastGaussian** —

```
------------------------------------------------------------------------------
-- fffg
------------------------------------------------------------------------------
```

———-

recurrence computes the new matrix M, according to the following formulas (cf. Table 2 in Beckermann and Labahn):

$$\begin{gathered}\text{Increase order}\\ \text{for } \ell = 1 \dots m,\ \ell \neq \pi\\ \mathbf{M}_{\sigma+1}^{(.,\ell)} := \left(\mathbf{M}_{\sigma}^{(.,\ell)} r^{(\pi)} - \mathbf{M}_{\sigma}^{(.,\pi)} r^{(\ell)}\right)/d_\sigma\\ \text{Increase order in column } \pi\\ \mathbf{M}_{\sigma+1}^{(.,\pi)} := (z - c_{\sigma,\sigma})\,\mathbf{M}_{\sigma}^{(.,\pi)}\\ \text{Adjust degree constraints:}\\ \mathbf{M}_{\sigma+1}^{(.,\pi)} := \left(\mathbf{M}_{\sigma+1}^{(.,\pi)} r^{(\pi)} - \textstyle\sum_{\ell\neq\pi} \mathbf{M}_{\sigma+1}^{(.,\ell)} p^{(\ell)}\right)/d_\sigma\end{gathered}$$

Since we do not need the matrix $\mathbf{M}_\sigma$ anymore, we drop the index and update the matrix destructively. In the following, we write Ck for $c_{\sigma,\sigma}$.

— **package FFFG FractionFreeFastGaussian** —

```
-- a major part of the time is spent here
    recurrence(M: Matrix SUP D, pi: NonNegativeInteger, m: NonNegativeInteger,
               r: Vector D, d: D, z: SUP D, Ck: D, p: Vector D): Matrix SUP D ==

        rPi: D := qelt(r, pi)
        polyf: SUP D := rPi * (z - Ck::SUP D)

        for i in 1..m repeat
            MiPi: SUP D    := qelt(M, i, pi)
            newMiPi: SUP D := polyf * MiPi
```

```
-- update columns ~= pi and calculate their sum
            for l in 1..m | l ~= pi repeat
                rl: D  := qelt(r, l)
-- I need the coercion to SUP D, since exquo returns an element of
-- Union("failed", SUP D)...
                Mil: SUP D := ((qelt(M, i, l) * rPi - MiPi * rl) exquo d)::SUP D
                qsetelt!(M, i, l, Mil)

                pl: D  := qelt(p, l)
                newMiPi := newMiPi - pl * Mil

-- update column pi
            qsetelt!(M, i, pi, (newMiPi exquo d)::SUP D)

        M


    fffg(C: List D, c: cFunction, eta: List NonNegativeInteger): Matrix SUP D ==
-- eta is the vector of degrees. We compute M with degrees eta+e_i-1, i=1..m
        z: SUP D := monomial(1, 1)
        m: NonNegativeInteger := #eta
        M: Matrix SUP D := scalarMatrix(m, 1)
        d: D := 1
        K: NonNegativeInteger := reduce(_+, eta)
        etak: Vector NonNegativeInteger := zero(m)
        r: Vector D := zero(m)
        p: Vector D := zero(m)
        Lambda: List Integer
        lambdaMax: Integer
        lambda: NonNegativeInteger

        for k in 1..K repeat
-- k = sigma+1

            for l in 1..m repeat r.l := c(k, column(M, l))

            Lambda := [eta.l-etak.l for l in 1..m | r.l ~= 0]

-- if Lambda is empty, then M, d and etak remain unchanged. Otherwise, we look
-- for the next closest para-normal point.

            (empty? Lambda) => "iterate"

            lambdaMax := reduce(max, Lambda)
            lambda := 1
            while eta.lambda-etak.lambda < lambdaMax or r.lambda = 0 repeat
                lambda := lambda + 1

-- Calculate leading coefficients
```

```
            for l in 1..m | l ~= lambda repeat
                if etak.l > 0 then
                    p.l := coefficient(M.(l, lambda),
                                       (etak.l-1)::NonNegativeInteger)
                else
                    p.l := 0

-- increase order and adjust degree constraints

            M := recurrence(M, lambda, m, r, d, z, C.k, p)

            d := r.lambda
            etak.lambda := etak.lambda + 1

        M
```

———-

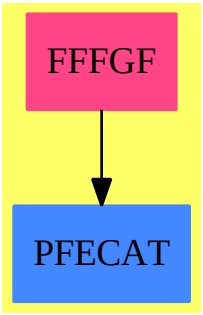
— **FFFG.dotabb** —

```
"FFFG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFG"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFG" -> "PFECAT"
```

———-

7.28 package FFFGF FractionFreeFastGaussianFractions

7.28.1 FractionFreeFastGaussianFractions (FFFGF)



Exports:
generalInterpolation

— package FFFGF FractionFreeFastGaussianFractions —

```
)abbrev package FFFGF FractionFreeFastGaussianFractions
++ Author: Martin Rubey
++ Description:
++ This package lifts the interpolation functions from
++ \spadtype{FractionFreeFastGaussian} to fractions.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussianFractions(D, V, VF): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: FiniteAbelianMonoidRing(D, NonNegativeInteger)
  VF: FiniteAbelianMonoidRing(Fraction D, NonNegativeInteger)

  F     ==> Fraction D
  NNI   ==> NonNegativeInteger
  SUP   ==> SparseUnivariatePolynomial
  FFFG  ==> FractionFreeFastGaussian
  FAMR2 ==> FiniteAbelianMonoidRingFunctions2

  cFunction ==> (NNI, Vector SUP D) -> D

  CoeffAction ==> (NNI, NNI, V) -> D
-- coeffAction(k, l, f) is the coefficient of x^k in z^l f(x)

  Exports == with

    generalInterpolation: (List D, CoeffAction, Vector VF, List NNI)
                          -> Matrix SUP D
      ++ \spad{generalInterpolation(l, CA, f, eta)} performs Hermite-Pade
      ++ approximation using the given action CA of polynomials on the elements
      ++ of f. The result is guaranteed to be correct up to order
      ++ |eta|-1. Given that eta is a "normal" point, the degrees on the
      ++ diagonal are given by eta. The degrees of column i are in this case
      ++ eta + e.i - [1,1,...,1], where the degree of zero is -1.

    generalInterpolation: (List D, CoeffAction, Vector VF, NNI, NNI)
                          -> Stream Matrix SUP D
      ++ \spad{generalInterpolation(l, CA, f, sumEta, maxEta)} applies
      ++ generalInterpolation(l, CA, f, eta) for all possible eta with maximal
      ++ entry maxEta and sum of entries sumEta

  Implementation == add

    multiplyRows!(v: Vector D, M: Matrix SUP D): Matrix SUP D ==
      n := #v
      for i in 1..n repeat
        for j in 1..n repeat
```

```
          M.(i,j) := v.i*M.(i,j)

      M

    generalInterpolation(C: List D, coeffAction: CoeffAction,
                         f: Vector VF, eta: List NNI): Matrix SUP D ==
      n := #f
      g: Vector V   := new(n, 0)
      den: Vector D := new(n, 0)

      for i in 1..n repeat
        c := coefficients(f.i)
        den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
        g.i :=
          map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)

      M := generalInterpolation(C, coeffAction, g, eta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not
-- each column. Possibly I could factor out gcd den, but I'm not sure whether
-- this is efficient.

      multiplyRows!(den, M)

    generalInterpolation(C: List D, coeffAction: CoeffAction,
                         f: Vector VF, sumEta: NNI, maxEta: NNI)
                          : Stream Matrix SUP D ==

      n := #f
      g: Vector V   := new(n, 0)
      den: Vector D := new(n, 0)

      for i in 1..n repeat
        c := coefficients(f.i)
        den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
        g.i :=
          map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)

      c: cFunction :=
       (x,y) +-> generalCoefficient(coeffAction, g, (x-1)::NNI, y)$FFFG(D, V)


      MS: Stream Matrix SUP D
         := generalInterpolation(C, coeffAction, g, sumEta, maxEta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not
-- each column. Possibly I could factor out gcd den, but I'm not sure whether
-- this is efficient.

      map(x +-> multiplyRows!(den, x), MS)$Stream(Matrix SUP D)
```

———-

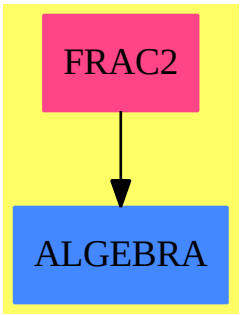
— FFFGF.dotabb —

```
"FFFGF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFGF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFGF" -> "PFECAT"
```

———-

7.29 package FRAC2 FractionFunctions2

7.29.1 FractionFunctions2 (FRAC2)



Exports:
map

— package FRAC2 FractionFunctions2 —

```
)abbrev package FRAC2 FractionFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package extends a map between integral domains to
```

```
++ a map between Fractions over those domains by applying the map to the
++ numerators and denominators.

FractionFunctions2(A, B): Exports == Impl where
  A, B: IntegralDomain

  R ==> Fraction A
  S ==> Fraction B

  Exports ==> with
    map: (A -> B, R) -> S
      ++ map(func,frac) applies the function func to the numerator
      ++ and denominator of the fraction frac.

  Impl ==> add
    map(f, r) == map(f, r)$QuotientFieldCategoryFunctions2(A, B, R, S)
```

———-

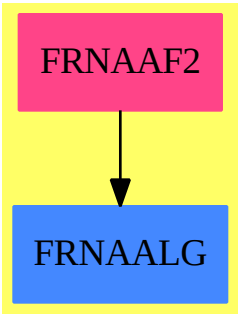
— **FRAC2.dotabb** —

```
"FRAC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRAC2"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FRAC2" -> "ALGEBRA"
```

———-

7.30 package FRNAAF2 FramedNonAssociativeAlgebraFunctions2

7.30.1 FramedNonAssociativeAlgebraFunctions2 (FRNAAF2)



Exports:
map

— **package FRNAAF2 FramedNonAssociativeAlgebraFunctions2** —

```
)abbrev package FRNAAF2 FramedNonAssociativeAlgebraFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 28 February 1992
++ Date Last Updated: 28 February 1992
++ Basic Operations: map
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: non-associative algebra
++ References:
++ Description:
++ FramedNonAssociativeAlgebraFunctions2 implements functions between
++ two framed non associative algebra domains defined over different rings.
++ The function map is used to coerce between algebras over different
++ domains having the same structural constants.

FramedNonAssociativeAlgebraFunctions2(AR,R,AS,S) : Exports ==
  Implementation where
    R  : CommutativeRing
    S  : CommutativeRing
    AR : FramedNonAssociativeAlgebra R
    AS : FramedNonAssociativeAlgebra S
    V ==> Vector
    Exports ==> with
      map:     (R -> S, AR) -> AS
        ++ map(f,u) maps f onto the coordinates of u to get an element
        ++ in \spad{AS} via identification of the basis of \spad{AR}
        ++ as beginning part of the basis of \spad{AS}.
    Implementation ==> add
      map(fn : R -> S, u : AR): AS ==
        rank()$AR > rank()$AS => error("map: ranks of algebras do not fit")
        vr : V R := coordinates u
        vs : V S := map(fn,vr)$VectorFunctions2(R,S)

            ------
```

This line used to read:

```
        rank()$AR = rank()$AR => represents(vs)$AS
```

but the test is clearly always true and cannot be what was intended. Gregory Vanuxem supplied the fix below.

— **package FRNAAF2 FramedNonAssociativeAlgebraFunctions2** —

```
        rank()$AR = rank()$AS => represents(vs)$AS
```

```
ba := basis()$AS
represents(vs,[ba.i for i in 1..rank()$AR])
```

———-

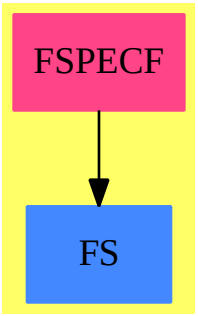
— **FRNAAF2.dotabb** —

```
"FRNAAF2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRNAAF2"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"FRNAAF2" -> "FRNAALG"
```

———-

7.31 package FSPECF FunctionalSpecialFunction

7.31.1 FunctionalSpecialFunction (FSPECF)



Exports:

Beta	Gamma	abs	airyAi	airyBi
belong?	besselI	besselJ	besselK	besselY
digamma	iiAiryAi	iiAiryBi	iiBesselI	iiBesselJ
iiBesselK	iiBesselY	iiBeta	iiGamma	iiabs
iidigamma	iipolygamma	operator	polygamma	

— **package FSPECF FunctionalSpecialFunction** —

```
)abbrev package FSPECF FunctionalSpecialFunction
++ Author: Manuel Bronstein
++ Date Created: 18 Apr 1989
++ Date Last Updated: 4 October 1993
++ Keywords: special, function.
++ Description:
```

```
++ Provides some special functions over an integral domain.

FunctionalSpecialFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R

  OP  ==> BasicOperator
  K   ==> Kernel F
  SE  ==> Symbol
  SPECIALDIFF  ==> "%specialDiff"

  Exports ==> with
    belong? : OP -> Boolean
      ++ belong?(op) is true if op is a special function operator;
    operator: OP -> OP
      ++ operator(op) returns a copy of op with the domain-dependent
      ++ properties appropriate for F;
      ++ error if op is not a special function operator
    abs     : F -> F
      ++ abs(f) returns the absolute value operator applied to f
    Gamma   : F -> F
      ++ Gamma(f) returns the formal Gamma function applied to f
    Gamma   : (F,F) -> F
      ++ Gamma(a,x) returns the incomplete Gamma function applied to a and x
    Beta:      (F,F) -> F
      ++ Beta(x,y) returns the beta function applied to x and y
    digamma:   F->F
      ++ digamma(x) returns the digamma function applied to x
    polygamma: (F,F) ->F
      ++ polygamma(x,y) returns the polygamma function applied to x and y
    besselJ:   (F,F) -> F
      ++ besselJ(x,y) returns the besselj function applied to x and y
    besselY:   (F,F) -> F
      ++ besselY(x,y) returns the bessely function applied to x and y
    besselI:   (F,F) -> F
      ++ besselI(x,y) returns the besseli function applied to x and y
    besselK:   (F,F) -> F
      ++ besselK(x,y) returns the besselk function applied to x and y
    airyAi:    F -> F
      ++ airyAi(x) returns the airyai function applied to x
    airyBi:    F -> F
      ++ airyBi(x) returns the airybi function applied to x
```

In case we want to have more special function operators here, do not forget to add them to the list specop in CommonOperators. Otherwise they will not have the 'special' attribute and will not be recognised here. One effect could be that

```
myNewSpecOp(1::Expression Integer)::Expression DoubleFloat
```

might not re-evaluate the operator.

— package FSPECF FunctionalSpecialFunction —

```
    iiGamma : F -> F
      ++ iiGamma(x) should be local but conditional;
    iiabs     : F -> F
      ++ iiabs(x) should be local but conditional;
    iiBeta     : List F -> F
      ++ iiGamma(x) should be local but conditional;
    iidigamma  : F -> F
      ++ iidigamma(x) should be local but conditional;
    iipolygamma: List F -> F
      ++ iipolygamma(x) should be local but conditional;
    iiBesselJ  : List F -> F
      ++ iiBesselJ(x) should be local but conditional;
    iiBesselY  : List F -> F
      ++ iiBesselY(x) should be local but conditional;
    iiBesselI  : List F -> F
      ++ iiBesselI(x) should be local but conditional;
    iiBesselK  : List F -> F
      ++ iiBesselK(x) should be local but conditional;
    iiAiryAi   : F -> F
      ++ iiAiryAi(x) should be local but conditional;
    iiAiryBi   : F -> F
      ++ iiAiryBi(x) should be local but conditional;

  Implementation ==> add
    iabs      : F -> F
    iGamma    : F -> F
    iBeta     : (F, F) -> F
    idigamma  : F -> F
    iiipolygamma: (F, F) -> F
    iiiBesselJ  : (F, F) -> F
    iiiBesselY  : (F, F) -> F
    iiiBesselI  : (F, F) -> F
    iiiBesselK  : (F, F) -> F
    iAiryAi   : F -> F
    iAiryBi   : F -> F

    opabs       := operator("abs"::Symbol)$CommonOperators
    opGamma     := operator("Gamma"::Symbol)$CommonOperators
    opGamma2    := operator("Gamma2"::Symbol)$CommonOperators
    opBeta      := operator("Beta"::Symbol)$CommonOperators
    opdigamma   := operator("digamma"::Symbol)$CommonOperators
    oppolygamma := operator("polygamma"::Symbol)$CommonOperators
    opBesselJ   := operator("besselJ"::Symbol)$CommonOperators
```

```
    opBesselY   := operator("besselY"::Symbol)$CommonOperators
    opBesselI   := operator("besselI"::Symbol)$CommonOperators
    opBesselK   := operator("besselK"::Symbol)$CommonOperators
    opAiryAi    := operator("airyAi"::Symbol)$CommonOperators
    opAiryBi    := operator("airyBi"::Symbol)$CommonOperators

    abs x         == opabs x
    Gamma(x)      == opGamma(x)
    Gamma(a,x)    == opGamma2(a,x)
    Beta(x,y)     == opBeta(x,y)
    digamma x     == opdigamma(x)
    polygamma(k,x)== oppolygamma(k,x)
    besselJ(a,x)  == opBesselJ(a,x)
    besselY(a,x)  == opBesselY(a,x)
    besselI(a,x)  == opBesselI(a,x)
    besselK(a,x)  == opBesselK(a,x)
    airyAi(x)     == opAiryAi(x)
    airyBi(x)     == opAiryBi(x)

    belong? op == has?(op, "special")

    operator op ==
      is?(op, "abs"::Symbol)      => opabs
      is?(op, "Gamma"::Symbol)    => opGamma
      is?(op, "Gamma2"::Symbol)   => opGamma2
      is?(op, "Beta"::Symbol)     => opBeta
      is?(op, "digamma"::Symbol)  => opdigamma
      is?(op, "polygamma"::Symbol)=> oppolygamma
      is?(op, "besselJ"::Symbol)  => opBesselJ
      is?(op, "besselY"::Symbol)  => opBesselY
      is?(op, "besselI"::Symbol)  => opBesselI
      is?(op, "besselK"::Symbol)  => opBesselK
      is?(op, "airyAi"::Symbol)   => opAiryAi
      is?(op, "airyBi"::Symbol)   => opAiryBi

      error "Not a special operator"

    -- Could put more unconditional special rules for other functions here
    iGamma x ==
--      one? x => x
      (x = 1) => x
      kernel(opGamma, x)

    iabs x ==
      zero? x => 0
      is?(x, opabs) => x
      x < 0 => kernel(opabs, -x)
      kernel(opabs, x)

    iBeta(x, y) == kernel(opBeta, [x, y])
```

```
    idigamma x == kernel(opdigamma, x)
    iiipolygamma(n, x) == kernel(oppolygamma, [n, x])
    iiiBesselJ(x, y) == kernel(opBesselJ, [x, y])
    iiiBesselY(x, y) == kernel(opBesselY, [x, y])
    iiiBesselI(x, y) == kernel(opBesselI, [x, y])
    iiiBesselK(x, y) == kernel(opBesselK, [x, y])
    iAiryAi x == kernel(opAiryAi, x)
    iAiryBi x == kernel(opAiryBi, x)


    -- Could put more conditional special rules for other functions here

    if R has abs : R -> R then
      iiabs x ==
        (r := retractIfCan(x)@Union(Fraction Polynomial R, "failed"))
          case "failed" => iabs x
        f := r::Fraction Polynomial R
        (a := retractIfCan(numer f)@Union(R, "failed")) case "failed" or
          (b := retractIfCan(denom f)@Union(R,"failed")) case "failed" => iabs x
        abs(a::R)::F / abs(b::R)::F

    else iiabs x == iabs x

    if R has SpecialFunctionCategory then
      iiGamma x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iGamma x
        Gamma(r::R)::F

      iiBeta l ==
        (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iBeta(first l, second l)
        Beta(r::R, s::R)::F

      iidigamma x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => idigamma x
        digamma(r::R)::F

      iipolygamma l ==
        (s:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (r:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iiipolygamma(first l, second l)
        polygamma(s::R, r::R)::F

      iiBesselJ l ==
        (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iiiBesselJ(first l, second l)
        besselJ(r::R, s::R)::F
```

```
      iiBesselY l ==
        (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iiiBesselY(first l, second l)
        besselY(r::R, s::R)::F

      iiBesselI l ==
        (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iiiBesselI(first l, second l)
        besselI(r::R, s::R)::F

      iiBesselK l ==
        (r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
        (s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
            => iiiBesselK(first l, second l)
        besselK(r::R, s::R)::F

      iiAiryAi x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryAi x
        airyAi(r::R)::F

      iiAiryBi x ==
        (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iAiryBi x
        airyBi(r::R)::F

    else
      if R has RetractableTo Integer then
        iiGamma x ==
          (r := retractIfCan(x)@Union(Integer, "failed")) case Integer
            and (r::Integer >= 1) => factorial(r::Integer - 1)::F
          iGamma x
      else
        iiGamma x == iGamma x

      iiBeta l == iBeta(first l, second l)
      iidigamma x == idigamma x
      iipolygamma l == iiipolygamma(first l, second l)
      iiBesselJ l == iiiBesselJ(first l, second l)
      iiBesselY l == iiiBesselY(first l, second l)
      iiBesselI l == iiiBesselI(first l, second l)
      iiBesselK l == iiiBesselK(first l, second l)
      iiAiryAi x == iAiryAi x
      iiAiryBi x == iAiryBi x

    -- Default behaviour is to build a kernel
    evaluate(opGamma, iiGamma)$BasicOperatorFunctions1(F)
    evaluate(opabs, iiabs)$BasicOperatorFunctions1(F)
--    evaluate(opGamma2    ,iiGamma2   )$BasicOperatorFunctions1(F)
    evaluate(opBeta      ,iiBeta     )$BasicOperatorFunctions1(F)
```

```
evaluate(opdigamma   ,iidigamma  )$BasicOperatorFunctions1(F)
evaluate(oppolygamma ,iipolygamma)$BasicOperatorFunctions1(F)
evaluate(opBesselJ   ,iiBesselJ  )$BasicOperatorFunctions1(F)
evaluate(opBesselY   ,iiBesselY  )$BasicOperatorFunctions1(F)
evaluate(opBesselI   ,iiBesselI  )$BasicOperatorFunctions1(F)
evaluate(opBesselK   ,iiBesselK  )$BasicOperatorFunctions1(F)
evaluate(opAiryAi    ,iiAiryAi   )$BasicOperatorFunctions1(F)
evaluate(opAiryBi    ,iiAiryBi   )$BasicOperatorFunctions1(F)
```

———-

7.31.2 differentiation of special functions

In the following we define the symbolic derivatives of the special functions we provide. The formulas we use for the Bessel functions can be found in Milton Abramowitz and Irene A. Stegun, eds. (1965). Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. New York: Dover. ISBN 0-486-61272-4, Equations 9.1.27 and 9.6.26. Up to patch–50 the formula for K missed the minus sign. (Issue #355)

We do not attempt to provide formulas for the derivative with respect to the first argument currently. Instead, we leave such derivatives unevaluated.

— **package FSPECF FunctionalSpecialFunction** —

```
import Fraction Integer
ahalf:  F    := recip(2::F)::F
athird: F    := recip(2::F)::F
twothirds: F := 2*recip(3::F)::F
```

———-

We need to get hold of the differentiation operator as modified by FunctionSpace. Otherwise, for example, display will be ugly. We accomplish that by differentiating an operator, which will certainly result in a single kernel only.

— **package FSPECF FunctionalSpecialFunction** —

```
dummyArg: SE := new()$SE
opdiff := operator first kernels D((operator(new()$SE)$BasicOperator)
                                        (dummyArg::F), dummyArg)
```

———-

The differentiation operator opdiff takes three arguments corresponding to

$$F_{,i}(a_1, a_2, \ldots, a_n) :$$

1. $F(a_1, ..., dm, ...a_n)$, where the i^{th} argument is a dummy variable,

2. dm, the dummy variable, and
3. a_i, the point at which the differential is evaluated.

In the following, it seems to be safe to use the same dummy variable throughout. At least, this is done also in FunctionSpace, and did not cause problems.

The operation symbolicGrad returns the first component of the gradient of op l.

— package FSPECF FunctionalSpecialFunction —

```
dm := new()$SE :: F

iBesselJ(l: List F, t: SE): F ==
    n := first l; x := second l
    differentiate(n, t)*kernel(opdiff, [opBesselJ [dm, x], dm, n])
      + differentiate(x, t) * ahalf * (besselJ (n-1,x) - besselJ (n+1,x))

iBesselY(l: List F, t: SE): F ==
    n := first l; x := second l
    differentiate(n, t)*kernel(opdiff, [opBesselY [dm, x], dm, n])
      + differentiate(x, t) * ahalf * (besselY (n-1,x) - besselY (n+1,x))

iBesselI(l: List F, t: SE): F ==
    n := first l; x := second l
    differentiate(n, t)*kernel(opdiff, [opBesselI [dm, x], dm, n])
      + differentiate(x, t)* ahalf * (besselI (n-1,x) + besselI (n+1,x))

iBesselK(l: List F, t: SE): F ==
    n := first l; x := second l
    differentiate(n, t)*kernel(opdiff, [opBesselK [dm, x], dm, n])
      - differentiate(x, t)* ahalf * (besselK (n-1,x) + besselK (n+1,x))
```

———-

For the moment we throw an error if we try to differentiate polygamma with respect to the first argument.

— package FSPECF FunctionalSpecialFunction —

```
ipolygamma(l: List F, x: SE): F ==
    member?(x, variables first l) =>
        error "cannot differentiate polygamma with respect to the first argument"
    n := first l; y := second l
    differentiate(y, x)*polygamma(n+1, y)
iBetaGrad1(l: List F): F ==
    x := first l; y := second l
    Beta(x,y)*(digamma x - digamma(x+y))
```

```
    iBetaGrad2(l: List F): F ==
        x := first l; y := second l
        Beta(x,y)*(digamma y - digamma(x+y))

    if F has ElementaryFunctionCategory then
      iGamma2(l: List F, t: SE): F ==
        a := first l; x := second l
        differentiate(a, t)*kernel(opdiff, [opGamma2 [dm, x], dm, a])
          - differentiate(x, t)* x ** (a - 1) * exp(-x)
      setProperty(opGamma2, SPECIALDIFF, iGamma2@((List F, SE)->F)
                                              pretend None)
```

———-

Finally, we tell Axiom to use these functions for differentiation. Note that up to patch–50, the properties for the Bessel functions were set using derivative(oppolygamma, [lzero, ipolygammaGrad]), where lzero returned zero always. Trying to replace lzero by a function that returns the first component of the gradient failed, it resulted in an infinite loop for integrate(D(besselJ(a,x),a),a).

— **package FSPECF FunctionalSpecialFunction** —

```
    derivative(opabs,       (x:F):F +-> abs(x) * inv(x))
    derivative(opGamma,     (x:F):F +-> digamma x * Gamma x)
    derivative(opBeta,      [iBetaGrad1, iBetaGrad2])
    derivative(opdigamma,   (x:F):F +-> polygamma(1, x))
    setProperty(oppolygamma, SPECIALDIFF, ipolygamma@((List F, SE)->F)
                                                pretend None)
    setProperty(opBesselJ, SPECIALDIFF, iBesselJ@((List F, SE)->F)
                                              pretend None)
    setProperty(opBesselY, SPECIALDIFF, iBesselY@((List F, SE)->F)
                                              pretend None)
    setProperty(opBesselI, SPECIALDIFF, iBesselI@((List F, SE)->F)
                                              pretend None)
    setProperty(opBesselK, SPECIALDIFF, iBesselK@((List F, SE)->F)
                                              pretend None)
```

———-

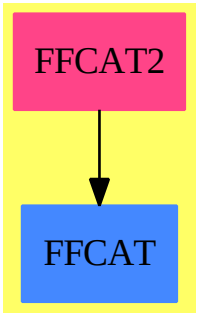
— **FSPECF.dotabb** —

```
"FSPECF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSPECF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSPECF" -> "FS"
```

———-

7.32 package FFCAT2 FunctionFieldCategoryFunctions2

7.32.1 FunctionFieldCategoryFunctions2 (FFCAT2)



Exports:
map

— package FFCAT2 FunctionFieldCategoryFunctions2 —

```
)abbrev package FFCAT2 FunctionFieldCategoryFunctions2
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 26 Jul 1988
++ Description:
++ Lifts a map from rings to function fields over them.

FunctionFieldCategoryFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):
 Exports == Implementation where
  R1   : UniqueFactorizationDomain
  UP1  : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  F1   : FunctionFieldCategory(R1, UP1, UPUP1)
  R2   : UniqueFactorizationDomain
  UP2  : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2
  F2   : FunctionFieldCategory(R2, UP2, UPUP2)

  Exports ==> with
    map: (R1 -> R2, F1) -> F2
      ++ map(f, p) lifts f to F1 and applies it to p.

  Implementation ==> add
    map(f, f1) ==
      reduce(map(f, lift f1)$MultipleMap(R1, UP1, UPUP1, R2, UP2, UPUP2))
```

———-

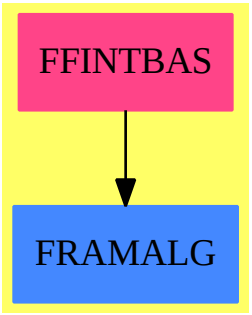
— **FFCAT2.dotabb** —

```
"FFCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFCAT2"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FFCAT2" -> "FFCAT"
```

———-

7.33 package FFINTBAS FunctionFieldIntegralBasis

7.33.1 FunctionFieldIntegralBasis (FFINTBAS)



Exports:
integralBasis localIntegralBasis

— **package FFINTBAS FunctionFieldIntegralBasis** —

```
)abbrev package FFINTBAS FunctionFieldIntegralBasis
++ Author: Victor Miller
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Integral bases for function fields of dimension one
++ In this package R is a Euclidean domain and F is a framed algebra
++ over R.  The package provides functions to compute the integral
++ closure of R in the quotient field of F.  It is assumed that
++ \spad{char(R/P) = char(R)} for any prime P of R.  A typical instance of
++ this is when \spad{R = K[x]} and F is a function field over R.
```

```
FunctionFieldIntegralBasis(R,UP,F): Exports == Implementation where
  R  : EuclideanDomain with
        squareFree: $ -> Factored $
          ++ squareFree(x) returns a square-free factorisation of x
  UP : UnivariatePolynomialCategory R
  F  : FramedAlgebra(R,UP)

  I   ==> Integer
  Mat ==> Matrix R
  NNI ==> NonNegativeInteger

  Exports ==> with
    integralBasis : () -> Record(basis: Mat, basisDen: R, basisInv:Mat)
      ++ \spad{integralBasis()} returns a record
      ++ \spad{[basis,basisDen,basisInv]} containing information regarding
      ++ the integral closure of R in the quotient field of F, where
      ++ F is a framed algebra with R-module basis \spad{w1,w2,...,wn}.
      ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
      ++ the \spad{i}th element of the integral basis is
      ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
      ++ \spad{i}th row of \spad{basis} contains the coordinates of the
      ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
      ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
      ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
      ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
      ++ \spad{wi = sum(bij * vj, j = 1..n)}.
    localIntegralBasis : R -> Record(basis: Mat, basisDen: R, basisInv:Mat)
      ++ \spad{integralBasis(p)} returns a record
      ++ \spad{[basis,basisDen,basisInv]} containing information regarding
      ++ the local integral closure of R at the prime \spad{p} in the quotient
      ++ field of F, where F is a framed algebra with R-module basis
      ++ \spad{w1,w2,...,wn}.
      ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
      ++ the \spad{i}th element of the local integral basis is
      ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
      ++ \spad{i}th row of \spad{basis} contains the coordinates of the
      ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
      ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
      ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
      ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
      ++ \spad{wi = sum(bij * vj, j = 1..n)}.

  Implementation ==> add
    import IntegralBasisTools(R, UP, F)
    import ModularHermitianRowReduction(R)
    import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

    squaredFactors: R -> R
    squaredFactors px ==
```

```
        */[(if ffe.exponent > 1 then ffe.factor else 1$R)
             for ffe in factors squareFree px]

  iIntegralBasis: (Mat,R,R) -> Record(basis: Mat, basisDen: R, basisInv:Mat)
  iIntegralBasis(tfm,disc,sing) ==
    -- tfm = trace matrix of current order
    n := rank()$F; tfm0 := copy tfm; disc0 := disc
    rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
    -- rb    = basis matrix of current order
    -- rbinv = inverse basis matrix of current order
    -- these are wrt the original basis for F
    rbden : R := 1; index : R := 1; oldIndex : R := 1
    -- rbden = denominator for current basis matrix
    -- index = index of original order in current order
    not sizeLess?(1, sing) => [rb, rbden, rbinv]
    repeat
      -- compute the p-radical
      idinv := transpose squareTop rowEchelon(tfm, sing)
      -- [u1,..,un] are the coordinates of an element of the p-radical
      -- iff [u1,..,un] * idinv is in sing * R^n
      id := rowEchelon LowTriBddDenomInv(idinv, sing)
      -- id = basis matrix of the p-radical
      idinv := UpTriBddDenomInv(id, sing)
      -- id * idinv = sing * identity
      -- no need to check for inseparability in this case
      rbinv := idealiser(id * rb, rbinv * idinv, sing * rbden)
      index := diagonalProduct rbinv
      rb := rowEchelon LowTriBddDenomInv(rbinv, rbden * sing)
      g := matrixGcd(rb,sing,n)
      if sizeLess?(1,g) then rb := (rb exquo g) :: Mat
      rbden := rbden * (sing quo g)
      rbinv := UpTriBddDenomInv(rb, rbden)
      disc := disc0 quo (index * index)
      indexChange := index quo oldIndex; oldIndex := index
      sing := gcd(indexChange, squaredFactors disc)
      not sizeLess?(1, sing) => return [rb, rbden, rbinv]
      tfm := ((rb * tfm0 * transpose rb) exquo (rbden * rbden)) :: Mat

  integralBasis() ==
    n := rank()$F; p := characteristic()$F
    (not zero? p) and (n >= p) =>
      error "integralBasis: possible wild ramification"
    tfm := traceMatrix()$F; disc := determinant tfm
    sing := squaredFactors disc    -- singularities of relative Spec
    iIntegralBasis(tfm,disc,sing)

  localIntegralBasis prime ==
    n := rank()$F; p := characteristic()$F
    (not zero? p) and (n >= p) =>
      error "integralBasis: possible wild ramification"
```

```
      tfm := traceMatrix()$F; disc := determinant tfm
      (disc exquo (prime * prime)) case "failed" =>
        [scalarMatrix(n,1),1,scalarMatrix(n,1)]
      iIntegralBasis(tfm,disc,prime)
```

———-

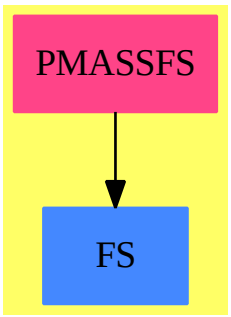
— **FFINTBAS.dotabb** —

```
"FFINTBAS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFINTBAS"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FFINTBAS" -> "FRAMALG"
```

———-

7.34 package PMASSFS FunctionSpaceAssertions

7.34.1 FunctionSpaceAssertions (PMASSFS)



Exports:
assert constant multiple optional

— **package PMASSFS FunctionSpaceAssertions** —

```
)abbrev package PMASSFS FunctionSpaceAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Keywords: pattern, matching.
++ Description:
++ Attaching assertions to symbols for pattern matching;
```

```
FunctionSpaceAssertions(R, F): Exports == Implementation where
  R: OrderedSet
  F: FunctionSpace R

  K  ==> Kernel F
  PMOPT   ==> "%pmoptional"
  PMMULT  ==> "%pmmultiple"
  PMCONST ==> "%pmconstant"

  Exports ==> with
    assert  : (F, String) -> F
      ++ assert(x, s) makes the assertion s about x.
      ++ Error: if x is not a symbol.
    constant: F -> F
      ++ constant(x) tells the pattern matcher that x should
      ++ match only the symbol 'x and no other quantity.
      ++ Error: if x is not a symbol.
    optional: F -> F
      ++ optional(x) tells the pattern matcher that x can match
      ++ an identity (0 in a sum, 1 in a product or exponentiation).
      ++ Error: if x is not a symbol.
    multiple: F -> F
      ++ multiple(x) tells the pattern matcher that x should
      ++ preferably match a multi-term quantity in a sum or product.
      ++ For matching on lists, multiple(x) tells the pattern matcher
      ++ that x should match a list instead of an element of a list.
      ++ Error: if x is not a symbol.

  Implementation ==> add
    ass  : (K, String) -> F
    asst : (K, String) -> F
    mkk  : BasicOperator -> F

    mkk op == kernel(op, empty()$List(F))

    ass(k, s) ==
      has?(op := operator k, s) => k::F
      mkk assert(copy op, s)

    asst(k, s) ==
      has?(op := operator k, s) => k::F
      mkk assert(op, s)

    assert(x, s) ==
      retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
        asst(retract(x)@K, s)
      error "assert must be applied to symbols only"

    constant x ==
      retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
```

```
        ass(retract(x)@K, PMCONST)
      error "constant must be applied to symbols only"

    optional x ==
      retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
        ass(retract(x)@K, PMOPT)
      error "optional must be applied to symbols only"

    multiple x ==
      retractIfCan(x)@Union(Symbol, "failed") case Symbol =>
        ass(retract(x)@K, PMMULT)
      error "multiple must be applied to symbols only"
```

———-

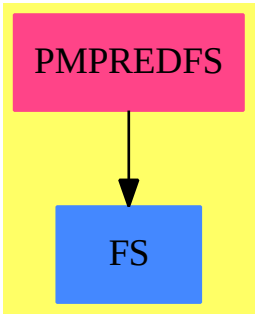
— **PMASSFS.dotabb** —

```
"PMASSFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMASSFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMASSFS" -> "FS"
```

———-

7.35 package PMPREDFS FunctionSpaceAttachPredicates

7.35.1 FunctionSpaceAttachPredicates (PMPREDFS)



Exports:
suchThat

— **package PMPREDFS FunctionSpaceAttachPredicates** —

```
)abbrev package PMPREDFS FunctionSpaceAttachPredicates
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Keywords: pattern, matching.
++ Description:
++ Attaching predicates to symbols for pattern matching.

FunctionSpaceAttachPredicates(R, F, D): Exports == Implementation where
  R: OrderedSet
  F: FunctionSpace R
  D: Type

  K  ==> Kernel F
  PMPRED  ==> "%pmpredicate"

  Exports ==> with
    suchThat: (F, D -> Boolean) -> F
      ++ suchThat(x, foo) attaches the predicate foo to x;
      ++ error if x is not a symbol.
    suchThat: (F, List(D -> Boolean)) -> F
      ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
      ++ f1 and f2 and ... and fn to x.
      ++ Error: if x is not a symbol.

  Implementation ==> add
    import AnyFunctions1(D -> Boolean)

    st   : (K, List Any) -> F
    preds: K -> List Any
    mkk  : BasicOperator -> F

    suchThat(p:F, f:D -> Boolean) == suchThat(p, [f])
    mkk op                        == kernel(op, empty()$List(F))

    preds k ==
      (u := property(operator k, PMPRED)) case "failed" => empty()
      (u::None) pretend List(Any)

    st(k, l) ==
      mkk assert(setProperty(copy operator k, PMPRED,
                  concat(preds k, l) pretend None), string(new()$Symbol))

    suchThat(p:F, l:List(D -> Boolean)) ==
      retractIfCan(p)@Union(Symbol, "failed") case Symbol =>
        st(retract(p)@K, [f::Any for f in l])
      error "suchThat must be applied to symbols only"
```

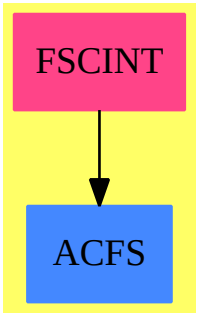
— PMPREDFS.dotabb —

```
"PMPREDFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPREDFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMPREDFS" -> "FS"
```

———-

7.36 package FSCINT FunctionSpaceComplexIntegration

7.36.1 FunctionSpaceComplexIntegration (FSCINT)



Exports:
complexIntegrate internalIntegrate internalIntegrate0

— package FSCINT FunctionSpaceComplexIntegration —

```
)abbrev package FSCINT FunctionSpaceComplexIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 11 June 1993
++ Keywords: function, integration.
++ Description:
++ Top-level complex function integration
++ \spadtype{FunctionSpaceComplexIntegration} provides functions for the
++ indefinite integration of complex-valued functions.

FunctionSpaceComplexIntegration(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory,
           AlgebraicallyClosedFunctionSpace R)
```

```
SE  ==> Symbol
G   ==> Complex R
FG  ==> Expression G
IR  ==> IntegrationResult F

Exports ==> with
  internalIntegrate : (F, SE) -> IR
      ++ internalIntegrate(f, x) returns the integral of \spad{f(x)dx}
      ++ where x is viewed as a complex variable.
  internalIntegrate0: (F, SE) -> IR
      ++ internalIntegrate0 should be a local function, but is conditional.
  complexIntegrate  : (F, SE) -> F
      ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
      ++ where x is viewed as a complex variable.

Implementation ==> add
  import IntegrationTools(R, F)
  import ElementaryIntegration(R, F)
  import ElementaryIntegration(G, FG)
  import AlgebraicManipulations(R, F)
  import AlgebraicManipulations(G, FG)
  import TrigonometricManipulations(R, F)
  import IntegrationResultToFunction(R, F)
  import IntegrationResultFunctions2(FG, F)
  import ElementaryFunctionStructurePackage(R, F)
  import ElementaryFunctionStructurePackage(G, FG)
  import InnerTrigonometricManipulations(R, F, FG)

  K2KG: Kernel F -> Kernel FG

  K2KG k                == retract(tan F2FG first argument k)@Kernel(FG)

  complexIntegrate(f, x) ==
    removeConstantTerm(complexExpand internalIntegrate(f, x), x)

  if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
    and F has Join(LiouvillianFunctionCategory, RetractableTo SE) then
      import PatternMatchIntegration(R, F)
      internalIntegrate0(f, x) ==
        intPatternMatch(f, x, lfintegrate, pmComplexintegrate)

  else internalIntegrate0(f, x) == lfintegrate(f, x)

  internalIntegrate(f, x) ==
    f := distribute(f, x::F)
    any?(x1+->has?(operator x1, "rtrig"),
     [k for k in tower(g := realElementary(f, x))
      | member?(x, variables(k::F))]$List(Kernel F))$List(Kernel F) =>
        h := trigs2explogs(F2FG g, [K2KG k for k in tower f
```

```
                    | is?(k, "tan"::SE) or is?(k, "cot"::SE)], [x])
          real?(g := FG2F h) =>
            internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)
          real?(g := FG2F(h := rootSimp(rischNormalize(h, x).func))) =>
                                                  internalIntegrate0(g, x)
          map(FG2F, lfintegrate(h, x))
      internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)
```

———-

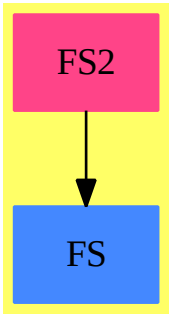
— **FSCINT.dotabb** —

```
"FSCINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSCINT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"FSCINT" -> "ACFS"
```

———-

7.37 package FS2 FunctionSpaceFunctions2

7.37.1 FunctionSpaceFunctions2 (FS2)



Exports:
map

— **package FS2 FunctionSpaceFunctions2** —

```
)abbrev package FS2 FunctionSpaceFunctions2
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 3 May 1994
++ Description:
```

```
++ Lifting of maps to function spaces
++ This package allows a mapping R -> S to be lifted to a mapping
++ from a function space over R to a function space over S;

FunctionSpaceFunctions2(R, A, S, B): Exports == Implementation where
  R, S: Join(Ring, OrderedSet)
  A   : FunctionSpace R
  B   : FunctionSpace S

  K  ==> Kernel A
  P  ==> SparseMultivariatePolynomial(R, K)

  Exports ==> with
    map: (R -> S, A) -> B
      ++ map(f, a) applies f to all the constants in R appearing in \spad{a}.

  Implementation ==> add
    smpmap: (R -> S, P) -> B

    smpmap(fn, p) ==
      map(x+->map(z+->map(fn, z),x)$ExpressionSpaceFunctions2(A,B),
          y+->fn(y)::B,p)_
           $PolynomialCategoryLifting(IndexedExponents K, K, R, P, B)

    if R has IntegralDomain then
      if S has IntegralDomain then
        map(f, x) == smpmap(f, numer x) / smpmap(f, denom x)
      else
        map(f, x) == smpmap(f, numer x) * (recip(smpmap(f, denom x))::B)
    else
      map(f, x) == smpmap(f, numer x)
```

———-

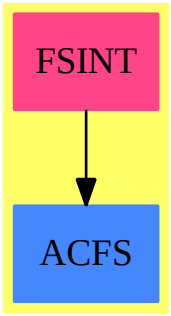
— **FS2.dotabb** —

```
"FS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FS2"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FS2" -> "FS"
```

———-

7.38 package FSINT FunctionSpaceIntegration

7.38.1 FunctionSpaceIntegration (FSINT)



Exports:
integrate

— package FSINT FunctionSpaceIntegration —

```
)abbrev package FSINT FunctionSpaceIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 11 June 1993
++ Keywords: function, integration.
++ Description:
++ Top-level real function integration
++ \spadtype{FunctionSpaceIntegration} provides functions for the
++ indefinite integration of real-valued functions.

FunctionSpaceIntegration(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
           AlgebraicallyClosedFunctionSpace R)

  B   ==> Boolean
  G   ==> Complex R
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  SE  ==> Symbol
  IR  ==> IntegrationResult F
  FG  ==> Expression G
  ALGOP ==> "%alg"
  TANTEMP ==> "%temptan"::SE

  Exports ==> with
```

```
    integrate: (F, SE) -> Union(F, List F)
        ++ integrate(f, x) returns the integral of \spad{f(x)dx}
        ++ where x is viewed as a real variable.

  Implementation ==> add
    import IntegrationTools(R, F)
    import ElementaryIntegration(R, F)
    import ElementaryIntegration(G, FG)
    import AlgebraicManipulations(R, F)
    import TrigonometricManipulations(R, F)
    import IntegrationResultToFunction(R, F)
    import TranscendentalManipulations(R, F)
    import IntegrationResultFunctions2(FG, F)
    import FunctionSpaceComplexIntegration(R, F)
    import ElementaryFunctionStructurePackage(R, F)
    import InnerTrigonometricManipulations(R, F, FG)
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                     K, R, SparseMultivariatePolynomial(R, K), F)

    K2KG      : K -> Kernel FG
    postSubst : (F, List F, List K, B, List K, SE) -> F
    rinteg    : (IR, F, SE, B, B) -> Union(F, List F)
    mkPrimh   : (F, SE, B, B) -> F
    trans?    : F -> B
    goComplex?: (B, List K, List K) -> B
    halfangle : F -> F
    Khalf     : K -> F
    tan2temp  : K -> K

    optemp:BasicOperator := operator(TANTEMP, 1)

    K2KG k     == retract(tan F2FG first argument k)@Kernel(FG)
    tan2temp k == kernel(optemp, argument k, height k)$K

    trans? f ==
      any?(x1+->is?(x1,"log"::SE) or is?(x1,"exp"::SE) or is?(x1,"atan"::SE),
           operators f)$List(BasicOperator)

    mkPrimh(f, x, h, comp) ==
      f := real f
      if comp then f := removeSinSq f
      g := mkPrim(f, x)
      h and trans? g => htrigs g
      g

    rinteg(i, f, x, h, comp) ==
      not elem? i => integral(f, x)$F
      empty? rest(l := [mkPrimh(f, x, h, comp) for f in expand i]) => first l
      l
```

```
-- replace tan(a/2)**2 by (1-cos a)/(1+cos a) if tan(a/2) is in ltan
    halfangle a ==
      a := 2 * a
      (1 - cos a) / (1 + cos a)

    Khalf k ==
      a := 2 * first argument k
      sin(a) / (1 + cos a)

-- ltan = list of tangents in the integrand after real normalization
    postSubst(f, lv, lk, comp, ltan, x) ==
      for v in lv for k in lk repeat
        if ((u := retractIfCan(v)@Union(K, "failed")) case K) then
           if has?(operator(kk := u::K), ALGOP) then
             f := univariate(f, kk, minPoly kk) (kk::F)
           f := eval(f, [u::K], [k::F])
      if not(comp or empty? ltan) then
        ltemp := [tan2temp k for k in ltan]
        f := eval(f, ltan, [k::F for k in ltemp])
        f := eval(f, TANTEMP, 2, halfangle)
        f := eval(f, ltemp, [Khalf k for k in ltemp])
      removeConstantTerm(f, x)

-- can handle a single unnested tangent directly, otherwise go complex for now
-- l is the list of all the kernels containing x
-- ltan is the list of all the tangents in l
    goComplex?(rt, l, ltan) ==
      empty? ltan => rt
      not empty? rest rest l

    integrate(f, x) ==
      not real? f => complexIntegrate(f, x)
      f   := distribute(f, x::F)
      tf  := [k for k in tower f | member?(x,variables(k::F)@List(SE))]$List(K)
      ltf := select(x1+->is?(operator x1, "tan"::SE), tf)
      ht  := any?(x1+->has?(operator x1, "htrig"), tf)
      rec := rischNormalize(realElementary(f, x), x)
      g   := rootSimp(rec.func)
      tg  := [k for k in tower g | member?(x, variables(k::F))]$List(K)
      ltg := select(x1+->is?(operator x1, "tan"::SE), tg)
      rtg := any?(x1+->has?(operator x1, "rtrig"), tg)
      el  := any?(x1+->has?(operator x1, "elem"), tg)
      i:IR
      if (comp := goComplex?(rtg, tg, ltg)) then
        i := map(FG2F, lfintegrate(trigs2explogs(F2FG g,
                       [K2KG k for k in tf | is?(k, "tan"::SE) or
                            is?(k, "cot"::SE)], [x]), x))
      else i := lfintegrate(g, x)
      ltg := setDifference(ltg, ltf)   -- tan's added by normalization
      (u := rinteg(i, f, x, el and ht, comp)) case F =>
```

```
        postSubst(u::F, rec.vals, rec.kers, comp, ltg, x)
      [postSubst(h, rec.vals, rec.kers, comp, ltg, x) for h in u::List(F)]
```

———-

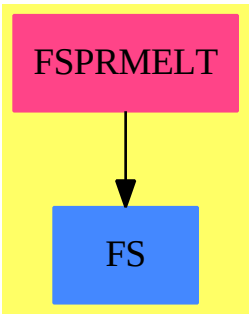
— **FSINT.dotabb** —

```
"FSINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSINT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"FSINT" -> "ACFS"
```

———-

7.39 package FSPRMELT FunctionSpacePrimitiveElement

7.39.1 FunctionSpacePrimitiveElement (FSPRMELT)



Exports:
primitiveElement

— **package FSPRMELT FunctionSpacePrimitiveElement** —

```
)abbrev package FSPRMELT FunctionSpacePrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Keywords: algebraic, extension, primitive.
++ Description:
++ FunctionsSpacePrimitiveElement provides functions to compute
++ primitive elements in functions spaces;
```

```
FunctionSpacePrimitiveElement(R, F): Exports == Implementation where
  R: Join(IntegralDomain, OrderedSet, CharacteristicZero)
  F: FunctionSpace R

  SY  ==> Symbol
  P   ==> Polynomial F
  K   ==> Kernel F
  UP  ==> SparseUnivariatePolynomial F
  REC ==> Record(primelt:F, poly:List UP, prim:UP)

  Exports ==> with
    primitiveElement: List F -> Record(primelt:F, poly:List UP, prim:UP)
      ++ primitiveElement([a1,...,an]) returns \spad{[a, [q1,...,qn], q]}
      ++ such that then \spad{k(a1,...,an) = k(a)},
      ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
      ++ This operation uses the technique of
      ++ \spadglossSee{groebner bases}{Groebner basis}.
    if F has AlgebraicallyClosedField then
      primitiveElement: (F,F)->Record(primelt:F,pol1:UP,pol2:UP,prim:UP)
        ++ primitiveElement(a1, a2) returns \spad{[a, q1, q2, q]}
        ++ such that \spad{k(a1, a2) = k(a)},
        ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
        ++ The minimal polynomial for a2 may involve a1, but the
        ++ minimal polynomial for a1 may not involve a2;
        ++ This operations uses \spadfun{resultant}.

  Implementation ==> add
    import PrimitiveElement(F)
    import AlgebraicManipulations(R, F)
    import PolynomialCategoryLifting(IndexedExponents K,
                            K, R, SparseMultivariatePolynomial(R, K), P)

    F2P: (F, List SY) -> P
    K2P: (K, List SY) -> P

    F2P(f, l) ==
     inv(denom(f)::F)*map((k1:K):P+->K2P(k1,l),(r1:R):P+->r1::F::P, numer f)

    K2P(k, l) ==
      ((v := symbolIfCan k) case SY) and member?(v::SY, l) => v::SY::P
      k::F::P

    primitiveElement l ==
      u    := string(uu := new()$SY)
      vars := [concat(u, string i)::SY for i in 1..#l]
      vv   := [kernel(v)$K :: F for v in vars]
      kers := [retract(a)@K for a in l]
      pols := [F2P(subst(ratDenom((minPoly k) v, kers), kers, vv), vars)
                                              for k in kers for v in vv]
```

```
      rec := primitiveElement(pols, vars, uu)
      [+/[c * a for c in rec.coef for a in l], rec.poly, rec.prim]

    if F has AlgebraicallyClosedField then
      import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                           K, R, SparseMultivariatePolynomial(R, K), F)

      F2UP: (UP, K, UP) -> UP
      getpoly: (UP, F) -> UP

      F2UP(p, k, q) ==
        ans:UP := 0
        while not zero? p repeat
          f   := univariate(leadingCoefficient p, k)
          ans := ans + ((numer f) q)
                       * monomial(inv(retract(denom f)@F), degree p)
          p   := reductum p
        ans

      primitiveElement(a1, a2) ==
        a   := (aa := new()$SY)::F
        b   := (bb := new()$SY)::F
        l   := [aa, bb]$List(SY)
        p1  := minPoly(k1 := retract(a1)@K)
        p2  := map((z1:F):F+->subst(ratDenom(z1, [k1]), [k1], [a]),
                                                 minPoly(retract(a2)@K))
        rec := primitiveElement(F2P(p1 a, l), aa, F2P(p2 b, l), bb)
        w   := rec.coef1 * a1 + rec.coef2 * a2
        g   := rootOf(rec.prim)
        zero?(rec.coef1) =>
          c2g := inv(rec.coef2 :: F) * g
          r := gcd(p1, univariate(p2 c2g, retract(a)@K, p1))
          q := getpoly(r, g)
          [w, q, rec.coef2 * monomial(1, 1)$UP, rec.prim]
        ic1 := inv(rec.coef1 :: F)
        gg  := (ic1 * g)::UP - monomial(rec.coef2 * ic1, 1)$UP
        kg  := retract(g)@K
        r   := gcd(p1 gg, F2UP(p2, retract(a)@K, gg))
        q   := getpoly(r, g)
        [w, monomial(ic1, 1)$UP - rec.coef2 * ic1 * q, q, rec.prim]

      getpoly(r, g) ==
--      one? degree r =>
        (degree r = 1) =>
          k := retract(g)@K
          univariate(-coefficient(r,0)/leadingCoefficient r,k,minPoly k)
        error "GCD not of degree 1"
```

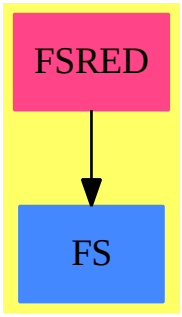
— **FSPRMELT.dotabb** —

```
"FSPRMELT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSPRMELT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSPRMELT" -> "FS"
```

———-

7.40 package FSRED FunctionSpaceReduce

7.40.1 FunctionSpaceReduce (FSRED)



Exports:
bringDown newReduc

— **package FSRED FunctionSpaceReduce** —

```
)abbrev package FSRED FunctionSpaceReduce
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 11 Jul 1990
++ Keywords: function, space, redcution.
++ Description:
++ Reduction from a function space to the rational numbers
++ This package provides function which replaces transcendental kernels
++ in a function space by random integers. The correspondence between
++ the kernels and the integers is fixed between calls to new().

FunctionSpaceReduce(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F: FunctionSpace R
```

```
  Z   ==> Integer
  Q   ==> Fraction Integer
  UP  ==> SparseUnivariatePolynomial Q
  K   ==> Kernel F
  ALGOP  ==> "%alg"

  Exports ==> with
    bringDown: F -> Q
      ++ bringDown(f) \undocumented
    bringDown: (F, K) -> UP
      ++ bringDown(f,k) \undocumented
    newReduc : () -> Void
      ++ newReduc() \undocumented

  Implementation ==> add
    import SparseUnivariatePolynomialFunctions2(F, Q)
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                         K, R, SparseMultivariatePolynomial(R, K), F)

    K2Z : K -> F

    redmap := table()$AssociationList(K, Z)

    newReduc() ==
      for k in keys redmap repeat remove_!(k, redmap)
      void

    bringDown(f, k) ==
      ff := univariate(f, k)
      (bc := extendedEuclidean(map(bringDown, denom ff),
                m := map(bringDown, minPoly k), 1)) case "failed" =>
                     error "denominator is 0"
      (map(bringDown, numer ff) * bc.coef1) rem m

    bringDown f ==
      retract(eval(f, lk := kernels f, [K2Z k for k in lk]))@Q

    K2Z k ==
      has?(operator k, ALGOP) => error "Cannot reduce constant field"
      (u := search(k, redmap)) case "failed" =>
                                      setelt(redmap, k, random()$Z)::F
      u::Z::F
```

———-

— FSRED.dotabb —

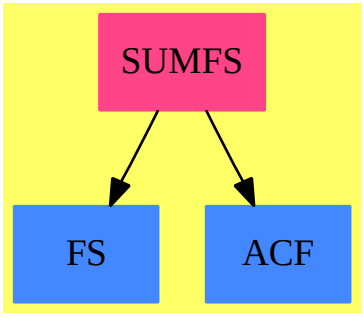
```
"FSRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSRED"]
```

```
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSRED" -> "FS"
```

———-

7.41 package SUMFS FunctionSpaceSum

7.41.1 FunctionSpaceSum (SUMFS)



Exports:
sum

— package SUMFS FunctionSpaceSum —

```
)abbrev package SUMFS FunctionSpaceSum
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description:
++ Computes sums of top-level expressions

FunctionSpaceSum(R, F): Exports == Implementation where
  R: Join(IntegralDomain, OrderedSet,
          RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, CombinatorialOpsCategory,
          AlgebraicallyClosedField, TranscendentalFunctionCategory)

  SE  ==> Symbol
  K   ==> Kernel F

  Exports ==> with
    sum: (F, SE) -> F
      ++ sum(a(n), n) returns A(n) such that A(n+1) - A(n) = a(n);
    sum: (F, SegmentBinding F) -> F
```

```
      ++ sum(f(n), n = a..b) returns f(a) + f(a+1) + ... + f(b);

  Implementation ==> add
    import ElementaryFunctionStructurePackage(R, F)
    import GosperSummationMethod(IndexedExponents K, K, R,
                                 SparseMultivariatePolynomial(R, K), F)

    innersum: (F, K) -> Union(F, "failed")
    notRF?  : (F, K) -> Boolean
    newk    : () -> K

    newk() == kernel(new()$SE)

    sum(x:F, s:SegmentBinding F) ==
      k := kernel(variable s)@K
      (u := innersum(x, k)) case "failed" => summation(x, s)
      eval(u::F, k, 1 + hi segment s) - eval(u::F, k, lo segment s)

    sum(x:F, v:SE) ==
      (u := innersum(x, kernel(v)@K)) case "failed" => summation(x,v)
      u::F

    notRF?(f, k) ==
      for kk in tower f repeat
        member?(k, tower(kk::F)) and (symbolIfCan(kk) case "failed") =>
          return true
      false

    innersum(x, k) ==
      zero? x => 0
      notRF?(f := normalize(x / (x1 := eval(x, k, k::F - 1))), k) =>
        "failed"
      (u := GospersMethod(f, k, newk)) case "failed" => "failed"
      x1 * eval(u::F, k, k::F - 1)
```

———-

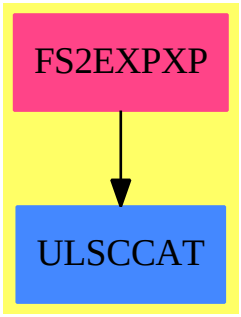
— **SUMFS.dotabb** —

```
"SUMFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUMFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"SUMFS" -> "FS"
"SUMFS" -> "ACF"
```

———-

7.42 package FS2EXPXP FunctionSpaceToExponentialExpansion

7.42.1 FunctionSpaceToExponentialExpansion (FS2EXPXP)



Exports:
localAbs exprToXXP

— package FS2EXPXP FunctionSpaceToExponentialExpansion —

```
)abbrev package FS2EXPXP FunctionSpaceToExponentialExpansion
++ Author: Clifton J. Williamson
++ Date Created: 17 August 1992
++ Date Last Updated: 2 December 1994
++ Basic Operations:
++ Related Domains: ExponentialExpansion, UnivariatePuiseuxSeries(FE,x,cen)
++ Also See: FunctionSpaceToUnivariatePowerSeries
++ AMS Classifications:
++ Keywords: elementary function, power series
++ Examples:
++ References:
++ Description:
++ This package converts expressions in some function space to exponential
++ expansions.

FunctionSpaceToExponentialExpansion(R,FE,x,cen):_
     Exports == Implementation where
  R     : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
               LinearlyExplicitRingOver Integer)
  FE    : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
               FunctionSpace R)
  x     : Symbol
  cen   : FE
  B       ==> Boolean
  BOP     ==> BasicOperator
```

```
Expon    ==> Fraction Integer
I        ==> Integer
NNI      ==> NonNegativeInteger
K        ==> Kernel FE
L        ==> List
RN       ==> Fraction Integer
S        ==> String
SY       ==> Symbol
PCL      ==> PolynomialCategoryLifting(IndexedExponents K,K,R,SMP,FE)
POL      ==> Polynomial R
SMP      ==> SparseMultivariatePolynomial(R,K)
SUP      ==> SparseUnivariatePolynomial Polynomial R
UTS      ==> UnivariateTaylorSeries(FE,x,cen)
ULS      ==> UnivariateLaurentSeries(FE,x,cen)
UPXS     ==> UnivariatePuiseuxSeries(FE,x,cen)
EFULS    ==> ElementaryFunctionsUnivariateLaurentSeries(FE,UTS,ULS)
EFUPXS   ==> ElementaryFunctionsUnivariatePuiseuxSeries(FE,ULS,UPXS,EFULS)
FS2UPS   ==> FunctionSpaceToUnivariatePowerSeries(R,FE,RN,UPXS,EFUPXS,x)
EXPUPXS  ==> ExponentialOfUnivariatePuiseuxSeries(FE,x,cen)
UPXSSING ==> UnivariatePuiseuxSeriesWithExponentialSingularity(R,FE,x,cen)
XXP      ==> ExponentialExpansion(R,FE,x,cen)
Problem  ==> Record(func:String,prob:String)
Result   ==> Union(%series:UPXS,%problem:Problem)
XResult  ==> Union(%expansion:XXP,%problem:Problem)
SIGNEF   ==> ElementaryFunctionSign(R,FE)

Exports ==> with
  exprToXXP : (FE,B) -> XResult
    ++ exprToXXP(fcn,posCheck?) converts the expression \spad{fcn} to
    ++ an exponential expansion.  If \spad{posCheck?} is true,
    ++ log's of negative numbers are not allowed nor are nth roots of
    ++ negative numbers with n even.  If \spad{posCheck?} is false,
    ++ these are allowed.
  localAbs: FE -> FE
    ++ localAbs(fcn) = \spad{abs(fcn)} or \spad{sqrt(fcn**2)} depending
    ++ on whether or not FE has a function \spad{abs}.  This should be
    ++ a local function, but the compiler won't allow it.

Implementation ==> add

  import FS2UPS  -- conversion of functional expressions to Puiseux series
  import EFUPXS  -- partial transcendental funtions on UPXS

  ratIfCan            : FE -> Union(RN,"failed")
  stateSeriesProblem  : (S,S) -> Result
  stateProblem        : (S,S) -> XResult
  newElem             : FE -> FE
  smpElem             : SMP -> FE
  k2Elem              : K -> FE
  iExprToXXP          : (FE,B) -> XResult
```

```
    listToXXP           : (L FE,B,XXP,(XXP,XXP) -> XXP) -> XResult
    isNonTrivPower      : FE -> Union(Record(val:FE,exponent:I),"failed")
    negativePowerOK?    : UPXS -> Boolean
    powerToXXP          : (FE,I,B) -> XResult
    carefulNthRootIfCan : (UPXS,NNI,B) -> Result
    nthRootXXPIfCan     : (XXP,NNI,B) -> XResult
    nthRootToXXP        : (FE,NNI,B) -> XResult
    genPowerToXXP       : (L FE,B) -> XResult
    kernelToXXP         : (K,B) -> XResult
    genExp              : (UPXS,B) -> Result
    exponential         : (UPXS,B) -> XResult
    expToXXP            : (FE,B) -> XResult
    genLog              : (UPXS,B) -> Result
    logToXXP            : (FE,B) -> XResult
    applyIfCan          : (UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
    applyBddIfCan       : (FE,UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
    tranToXXP           : (K,FE,B) -> XResult
    contOnReals?        : S -> B
    bddOnReals?         : S -> B
    opsInvolvingX       : FE -> L BOP
    opInOpList?         : (SY,L BOP) -> B
    exponential?        : FE -> B
    productOfNonZeroes? : FE -> B
    atancotToXXP        : (FE,FE,B,I) -> XResult

    ZEROCOUNT : RN := 1000/1
    -- number of zeroes to be removed when taking logs or nth roots

--% retractions

    ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")

--% 'problems' with conversion

    stateSeriesProblem(function,problem) ==
      -- records the problem which occured in converting an expression
      -- to a power series
      [[function,problem]]

    stateProblem(function,problem) ==
      -- records the problem which occured in converting an expression
      -- to an exponential expansion
      [[function,problem]]

--% normalizations

    newElem f ==
      -- rewrites a functional expression; all trig functions are
      -- expressed in terms of sin and cos; all hyperbolic trig
      -- functions are expressed in terms of exp; all inverse
```

```
      -- hyperbolic trig functions are expressed in terms of exp
      -- and log
      smpElem(numer f) / smpElem(denom f)

    smpElem p == map(k2Elem,(x1:R):FE+->x1::FE,p)$PCL

    k2Elem k ==
    -- rewrites a kernel; all trig functions are
    -- expressed in terms of sin and cos; all hyperbolic trig
    -- functions are expressed in terms of exp
      null(args := [newElem a for a in argument k]) => k :: FE
      iez  := inv(ez  := exp(z := first args))
      sinz := sin z; cosz := cos z
      is?(k,"tan" :: SY)  => sinz / cosz
      is?(k,"cot" :: SY)  => cosz / sinz
      is?(k,"sec" :: SY)  => inv cosz
      is?(k,"csc" :: SY)  => inv sinz
      is?(k,"sinh" :: SY) => (ez - iez) / (2 :: FE)
      is?(k,"cosh" :: SY) => (ez + iez) / (2 :: FE)
      is?(k,"tanh" :: SY) => (ez - iez) / (ez + iez)
      is?(k,"coth" :: SY) => (ez + iez) / (ez - iez)
      is?(k,"sech" :: SY) => 2 * inv(ez + iez)
      is?(k,"csch" :: SY) => 2 * inv(ez - iez)
      is?(k,"acosh" :: SY) => log(sqrt(z**2 - 1) + z)
      is?(k,"atanh" :: SY) => log((z + 1) / (1 - z)) / (2 :: FE)
      is?(k,"acoth" :: SY) => log((z + 1) / (z - 1)) / (2 :: FE)
      is?(k,"asech" :: SY) => log((inv z) + sqrt(inv(z**2) - 1))
      is?(k,"acsch" :: SY) => log((inv z) + sqrt(1 + inv(z**2)))
      (operator k) args

--% general conversion function

    exprToXXP(fcn,posCheck?) == iExprToXXP(newElem fcn,posCheck?)

    iExprToXXP(fcn,posCheck?) ==
      -- converts a functional expression to an exponential expansion
      --!! The following line is commented out so that expressions of
      --!! the form a**b will be normalized to exp(b * log(a)) even if
      --!! 'a' and 'b' do not involve the limiting variable 'x'.
      --!!                          - cjw 1 Dec 94
      --not member?(x,variables fcn) => [monomial(fcn,0)$UPXS :: XXP]
      (poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
        [exprToUPS(fcn,false,"real:two sides").%series :: XXP]
      (sum := isPlus fcn) case L(FE) =>
        listToXXP(sum::L(FE),posCheck?,0,(y1:XXP,y2:XXP):XXP +-> y1+y2)
      (prod := isTimes fcn) case L(FE) =>
        listToXXP(prod :: L(FE),posCheck?,1,(y1:XXP,y2:XXP):XXP +-> y1*y2)
      (expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
        power := expt :: Record(val:FE,exponent:I)
        powerToXXP(power.val,power.exponent,posCheck?)
```

```
      (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
        kernelToXXP(ker :: K,posCheck?)
      error "exprToXXP: neither a sum, product, power, nor kernel"

--% sums and products

    listToXXP(list,posCheck?,ans,op) ==
      -- converts each element of a list of expressions to an exponential
      -- expansion and returns the sum of these expansions, when 'op' is +
      -- and 'ans' is 0, or the product of these expansions, when 'op' is *
      -- and 'ans' is 1
      while not null list repeat
        (term := iExprToXXP(first list,posCheck?)) case %problem =>
          return term
        ans := op(ans,term.%expansion)
        list := rest list
      [ans]

--% nth roots and integral powers

    isNonTrivPower fcn ==
      -- is the function a power with exponent other than 0 or 1?
      (expt := isPower fcn) case "failed" => "failed"
      power := expt :: Record(val:FE,exponent:I)
--      one? power.exponent => "failed"
      (power.exponent = 1) => "failed"
      power

    negativePowerOK? upxs ==
      -- checks the lower order coefficient of a Puiseux series;
      -- the coefficient may be inverted only if
      -- (a) the only function involving x is 'log', or
      -- (b) the lowest order coefficient is a product of exponentials
      --     and functions not involving x
      deg := degree upxs
      if (coef := coefficient(upxs,deg)) = 0 then
        deg := order(upxs,deg + ZEROCOUNT :: Expon)
        (coef := coefficient(upxs,deg)) = 0 =>
          error "inverse of series with many leading zero coefficients"
      xOpList := opsInvolvingX coef
      -- only function involving x is 'log'
      (null xOpList) => true
      (null rest xOpList and is?(first xOpList,"log" :: SY)) => true
      -- lowest order coefficient is a product of exponentials and
      -- functions not involving x
      productOfNonZeroes? coef => true
      false

    powerToXXP(fcn,n,posCheck?) ==
      -- converts an integral power to an exponential expansion
```

```
    (b := iExprToXXP(fcn,posCheck?)) case %problem => b
    xxp := b.%expansion
    n > 0 => [xxp ** n]
    -- a Puiseux series will be reciprocated only if n < 0 and
    -- numerator of 'xxp' has exactly one monomial
    numberOfMonomials(num := numer xxp) > 1 => [xxp ** n]
    negativePowerOK? leadingCoefficient num =>
      (rec := recip num) case "failed" => error "FS2EXPXP: can't happen"
      nn := (-n) :: NNI
      [(((denom xxp) ** nn) * ((rec :: UPXSSING) ** nn)) :: XXP]
    --!! we may want to create a fraction instead of trying to
    --!! reciprocate the numerator
    stateProblem("inv","lowest order coefficient involves x")

  carefulNthRootIfCan(ups,n,posCheck?) ==
    -- similar to 'nthRootIfCan', but it is fussy about the series
    -- it takes as an argument.  If 'n' is EVEN and 'posCheck?'
    -- is truem then the leading coefficient of the series must
    -- be POSITIVE.  In this case, if 'rightOnly?' is false, the
    -- order of the series must be zero.  The idea is that the
    -- series represents a real function of a real variable, and
    -- we want a unique real nth root defined on a neighborhood
    -- of zero.
    n < 1 => error "nthRoot: n must be positive"
    deg := degree ups
    if (coef := coefficient(ups,deg)) = 0 then
      deg := order(ups,deg + ZEROCOUNT :: Expon)
      (coef := coefficient(ups,deg)) = 0 =>
        error "log of series with many leading zero coefficients"
    -- if 'posCheck?' is true, we do not allow nth roots of negative
    -- numbers when n in even
    if even?(n :: I) then
      if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then
        (signum :: I) = -1 =>
          return stateSeriesProblem("nth root","root of negative number")
    (ans := nthRootIfCan(ups,n)) case "failed" =>
      stateSeriesProblem("nth root","no nth root")
    [ans :: UPXS]

  nthRootXXPIfCan(xxp,n,posCheck?) ==
    num := numer xxp; den := denom xxp
    not zero?(reductum num) or not zero?(reductum den) =>
     stateProblem("nth root","several monomials in numerator or denominator")
    nInv : RN := 1/n
    newNum :=
      coef : UPXS :=
        root := carefulNthRootIfCan(leadingCoefficient num,n,posCheck?)
        root case %problem => return [root.%problem]
        root.%series
      deg := (nInv :: FE) * (degree num)
```

```
        monomial(coef,deg)
      newDen :=
        coef : UPXS :=
          root := carefulNthRootIfCan(leadingCoefficient den,n,posCheck?)
          root case %problem => return [root.%problem]
          root.%series
        deg := (nInv :: FE) * (degree den)
        monomial(coef,deg)
      [newNum/newDen]

    nthRootToXXP(arg,n,posCheck?) ==
      -- converts an nth root to a power series
      -- this is not used in the limit package, so the series may
      -- have non-zero order, in which case nth roots may not be unique
      (result := iExprToXXP(arg,posCheck?)) case %problem => [result.%problem]
      ans := nthRootXXPIfCan(result.%expansion,n,posCheck?)
      ans case %problem => [ans.%problem]
      [ans.%expansion]

--% general powers f(x) ** g(x)

    genPowerToXXP(args,posCheck?) ==
      -- converts a power f(x) ** g(x) to an exponential expansion
      (logBase := logToXXP(first args,posCheck?)) case %problem =>
        logBase
      (expon := iExprToXXP(second args,posCheck?)) case %problem =>
        expon
      xxp := (expon.%expansion) * (logBase.%expansion)
      (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
        stateProblem("exp","multiply nested exponential")
      exponential(f,posCheck?)

--% kernels

    kernelToXXP(ker,posCheck?) ==
      -- converts a kernel to a power series
      (sym := symbolIfCan(ker)) case Symbol =>
        (sym :: Symbol) = x => [monomial(1,1)$UPXS :: XXP]
        [monomial(ker :: FE,0)$UPXS :: XXP]
      empty?(args := argument ker) => [monomial(ker :: FE,0)$UPXS :: XXP]
      empty? rest args =>
        arg := first args
        is?(ker,"%paren" :: Symbol) => iExprToXXP(arg,posCheck?)
        is?(ker,"log" :: Symbol) => logToXXP(arg,posCheck?)
        is?(ker,"exp" :: Symbol) => expToXXP(arg,posCheck?)
        tranToXXP(ker,arg,posCheck?)
      is?(ker,"%power" :: Symbol) => genPowerToXXP(args,posCheck?)
      is?(ker,"nthRoot" :: Symbol) =>
        n := retract(second args)@I
        nthRootToXXP(first args,n :: NNI,posCheck?)
```

```
      stateProblem(string name ker,"unknown kernel")

--% exponentials and logarithms

    genExp(ups,posCheck?) ==
      -- If the series has order zero and the constant term a0 of the
      -- series involves x, the function tries to expand exp(a0) as
      -- a power series.
      (deg := order(ups,1)) < 0 =>
        -- this "can't happen"
        error "exp of function with sigularity"
      deg > 0 => [exp(ups)]
      lc := coefficient(ups,0); varOpList := opsInvolvingX lc
      not opInOpList?("log" :: Symbol,varOpList) => [exp(ups)]
      -- try to fix exp(lc) if necessary
      expCoef := normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
      result := exprToGenUPS(expCoef,posCheck?,"real:right side")$FS2UPS
      --!! will deal with problems in limitPlus in EXPEXPAN
      --result case %problem => result
      result case %problem => [exp(ups)]
      [(result.%series) * exp(ups - monomial(lc,0))]

    exponential(f,posCheck?) ==
      singPart := truncate(f,0) - (coefficient(f,0) :: UPXS)
      taylorPart := f - singPart
      expon := exponential(singPart)$EXPUPXS
      (coef := genExp(taylorPart,posCheck?)) case %problem => [coef.%problem]
      [monomial(coef.%series,expon)$UPXSSING :: XXP]

    expToXXP(arg,posCheck?) ==
      (result := iExprToXXP(arg,posCheck?)) case %problem => result
      xxp := result.%expansion
      (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
        stateProblem("exp","multiply nested exponential")
      exponential(f,posCheck?)

    genLog(ups,posCheck?) ==
      deg := degree ups
      if (coef := coefficient(ups,deg)) = 0 then
        deg := order(ups,deg + ZEROCOUNT)
        (coef := coefficient(ups,deg)) = 0 =>
          error "log of series with many leading zero coefficients"
      -- if 'posCheck?' is true, we do not allow logs of negative numbers
      if posCheck? then
        if ((signum := sign(coef)$SIGNEF) case I) then
          (signum :: I) = -1 =>
            return stateSeriesProblem("log","negative leading coefficient")
      lt := monomial(coef,deg)$UPXS
      -- check to see if lowest order coefficient is a negative rational
      negRat? : Boolean :=
```

```
        ((rat := ratIfCan coef) case RN) =>
          (rat :: RN) < 0 => true
          false
        false
      logTerm : FE :=
        mon : FE := (x :: FE) - (cen :: FE)
        pow : FE := mon ** (deg :: FE)
        negRat? => log(coef * pow)
        term1 : FE := (deg :: FE) * log(mon)
        log(coef) + term1
      [monomial(logTerm,0)$UPXS + log(ups/lt)]

    logToXXP(arg,posCheck?) ==
      (result := iExprToXXP(arg,posCheck?)) case %problem => result
      xxp := result.%expansion
      num := numer xxp; den := denom xxp
      not zero?(reductum num) or not zero?(reductum den) =>
        stateProblem("log","several monomials in numerator or denominator")
      numCoefLog : UPXS :=
        (res := genLog(leadingCoefficient num,posCheck?)) case %problem =>
          return [res.%problem]
        res.%series
      denCoefLog : UPXS :=
        (res := genLog(leadingCoefficient den,posCheck?)) case %problem =>
          return [res.%problem]
        res.%series
      numLog := (exponent degree num) + numCoefLog
      denLog := (exponent degree den) + denCoefLog  --?? num?
      [(numLog - denLog) :: XXP]

--% other transcendental functions

    applyIfCan(fcn,arg,fcnName,posCheck?) ==
      -- converts fcn(arg) to an exponential expansion
      (xxpArg := iExprToXXP(arg,posCheck?)) case %problem => xxpArg
      xxp := xxpArg.%expansion
      (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
        stateProblem(fcnName,"multiply nested exponential")
      upxs := f :: UPXS
      (deg := order(upxs,1)) < 0 =>
        stateProblem(fcnName,"essential singularity")
      deg > 0 => [fcn(upxs) :: UPXS :: XXP]
      lc := coefficient(upxs,0); xOpList := opsInvolvingX lc
      null xOpList => [fcn(upxs) :: UPXS :: XXP]
      opInOpList?("log" :: SY,xOpList) =>
        stateProblem(fcnName,"logs in constant coefficient")
      contOnReals? fcnName => [fcn(upxs) :: UPXS :: XXP]
      stateProblem(fcnName,"x in constant coefficient")

    applyBddIfCan(fe,fcn,arg,fcnName,posCheck?) ==
```

```
    -- converts fcn(arg) to a generalized power series, where the
    -- function fcn is bounded for real values
    -- if fcn(arg) has an essential singularity as a complex
    -- function, we return fcn(arg) as a monomial of degree 0
    (xxpArg := iExprToXXP(arg,posCheck?)) case %problem =>
      trouble := xxpArg.%problem
      trouble.prob = "essential singularity" => [monomial(fe,0)$UPXS :: XXP]
      xxpArg
    xxp := xxpArg.%expansion
    (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
      stateProblem("exp","multiply nested exponential")
    (ans := fcn(f :: UPXS)) case "failed" => [monomial(fe,0)$UPXS :: XXP]
    [ans :: UPXS :: XXP]

  CONTFCNS : L S := ["sin","cos","atan","acot","exp","asinh"]
  -- functions which are defined and continuous at all real numbers

  BDDFCNS : L S := ["sin","cos","atan","acot"]
  -- functions which are bounded on the reals

  contOnReals? fcn == member?(fcn,CONTFCNS)
  bddOnReals? fcn  == member?(fcn,BDDFCNS)

  opsInvolvingX fcn ==
    opList := [op for k in tower fcn | unary?(op := operator k) _
               and member?(x,variables first argument k)]
    removeDuplicates opList

  opInOpList?(name,opList) ==
    for op in opList repeat
      is?(op,name) => return true
    false

  exponential? fcn ==
    -- is 'fcn' of the form exp(f)?
    (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
      is?(ker :: K,"exp" :: Symbol)
    false

  productOfNonZeroes? fcn ==
    -- is 'fcn' a product of non-zero terms, where 'non-zero'
    -- means an exponential or a function not involving x
    exponential? fcn => true
    (prod := isTimes fcn) case "failed" => false
    for term in (prod :: L(FE)) repeat
      (not exponential? term) and member?(x,variables term) =>
        return false
    true

  tranToXXP(ker,arg,posCheck?) ==
```

```
    -- converts op(arg) to a power series for certain functions
    -- op in trig or hyperbolic trig categories
    -- N.B. when this function is called, 'k2elem' will have been
    -- applied, so the following functions cannot appear:
    -- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
    -- acosh, atanh, acoth, asech, acsch
    is?(ker,"sin" :: SY) =>
      applyBddIfCan(ker :: FE,sinIfCan,arg,"sin",posCheck?)
    is?(ker,"cos" :: SY) =>
      applyBddIfCan(ker :: FE,cosIfCan,arg,"cos",posCheck?)
    is?(ker,"asin" :: SY) =>
      applyIfCan(asinIfCan,arg,"asin",posCheck?)
    is?(ker,"acos" :: SY) =>
      applyIfCan(acosIfCan,arg,"acos",posCheck?)
    is?(ker,"atan" :: SY) =>
      atancotToXXP(ker :: FE,arg,posCheck?,1)
    is?(ker,"acot" :: SY) =>
      atancotToXXP(ker :: FE,arg,posCheck?,-1)
    is?(ker,"asec" :: SY) =>
      applyIfCan(asecIfCan,arg,"asec",posCheck?)
    is?(ker,"acsc" :: SY) =>
      applyIfCan(acscIfCan,arg,"acsc",posCheck?)
    is?(ker,"asinh" :: SY) =>
      applyIfCan(asinhIfCan,arg,"asinh",posCheck?)
    stateProblem(string name ker,"unknown kernel")

  if FE has abs: FE -> FE then
    localAbs fcn == abs fcn
  else
    localAbs fcn == sqrt(fcn * fcn)

  signOfExpression: FE -> FE
  signOfExpression arg == localAbs(arg)/arg

  atancotToXXP(fe,arg,posCheck?,plusMinus) ==
    -- converts atan(f(x)) to a generalized power series
    atanFlag : String := "real: right side"; posCheck? : Boolean := true
    (result := exprToGenUPS(arg,posCheck?,atanFlag)$FS2UPS) case %problem =>
      trouble := result.%problem
      trouble.prob = "essential singularity" => [monomial(fe,0)$UPXS :: XXP]
      [result.%problem]
    ups := result.%series; coef := coefficient(ups,0)
    -- series involves complex numbers
    (ord := order(ups,0)) = 0 and coef * coef = -1 =>
      y := differentiate(ups)/(1 + ups*ups)
      yCoef := coefficient(y,-1)
      [(monomial(log yCoef,0)+integrate(y - monomial(yCoef,-1)$UPXS)) :: XXP]
    cc : FE :=
      ord < 0 =>
        (rn := ratIfCan(ord :: FE)) case "failed" =>
```

```
          -- this condition usually won't occur because exponents will
          -- be integers or rational numbers
          return stateProblem("atan","branch problem")
        lc := coefficient(ups,ord)
        (signum := sign(lc)$SIGNEF) case "failed" =>
          -- can't determine sign
          posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
          plusMinus = 1 => posNegPi2
          pi()/(2 :: FE) - posNegPi2
        (n := signum :: Integer) = -1 =>
          plusMinus = 1 => -pi()/(2 :: FE)
          pi()
        plusMinus = 1 => pi()/(2 :: FE)
        0
      atan coef
    [((cc :: UPXS) + integrate(differentiate(ups)/(1 + ups*ups))) :: XXP]
```

———-

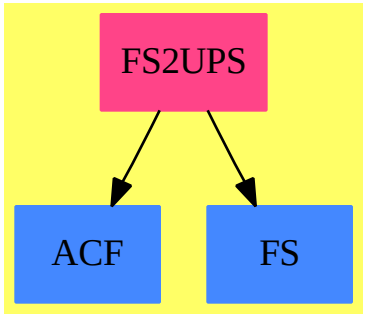
— **FS2EXPXP.dotabb** —

```
"FS2EXPXP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FS2EXPXP"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"FS2EXPXP" -> "ULSCCAT"
```

———-

7.43 package FS2UPS FunctionSpaceToUnivariatePowerSeries

7.43.1 FunctionSpaceToUnivariatePowerSeries (FS2UPS)



Exports:
localAbs exprToGenUPS exprToUPS

— package FS2UPS FunctionSpaceToUnivariatePowerSeries —

```
)abbrev package FS2UPS FunctionSpaceToUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 21 March 1989
++ Date Last Updated: 2 December 1994
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: elementary function, power series
++ Examples:
++ References:
++ Description:
++ This package converts expressions in some function space to power
++ series in a variable x with coefficients in that function space.
++ The function \spadfun{exprToUPS} converts expressions to power series
++ whose coefficients do not contain the variable x. The function
++ \spadfun{exprToGenUPS} converts functional expressions to power series
++ whose coefficients may involve functions of \spad{log(x)}.

FunctionSpaceToUnivariatePowerSeries(R,FE,Expon,UPS,TRAN,x):_
 Exports == Implementation where
  R     : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
               LinearlyExplicitRingOver Integer)
  FE    : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
               FunctionSpace R)
            with
              coerce: Expon -> %
                ++ coerce(e) converts an 'exponent' e to an 'expression'
  Expon : OrderedRing
  UPS   : Join(UnivariatePowerSeriesCategory(FE,Expon),Field,_
               TranscendentalFunctionCategory)
            with
              differentiate: % -> %
                ++ differentiate(x) returns the derivative of x since we
                ++ need to be able to differentiate a power series
              integrate: % -> %
                ++ integrate(x) returns the integral of x since
                ++ we need to be able to integrate a power series
  TRAN  : PartialTranscendentalFunctions UPS
  x     : Symbol
  B       ==> Boolean
  BOP     ==> BasicOperator
  I       ==> Integer
  NNI     ==> NonNegativeInteger
```

```
K       ==> Kernel FE
L       ==> List
RN      ==> Fraction Integer
S       ==> String
SY      ==> Symbol
PCL     ==> PolynomialCategoryLifting(IndexedExponents K,K,R,SMP,FE)
POL     ==> Polynomial R
SMP     ==> SparseMultivariatePolynomial(R,K)
SUP     ==> SparseUnivariatePolynomial Polynomial R
Problem ==> Record(func:String,prob:String)
Result  ==> Union(%series:UPS,%problem:Problem)
SIGNEF  ==> ElementaryFunctionSign(R,FE)

Exports ==> with
  exprToUPS : (FE,B,S) -> Result
    ++ exprToUPS(fcn,posCheck?,atanFlag) converts the expression
    ++ \spad{fcn} to a power series.  If \spad{posCheck?} is true,
    ++ log's of negative numbers are not allowed nor are nth roots of
    ++ negative numbers with n even.  If \spad{posCheck?} is false,
    ++ these are allowed.  \spad{atanFlag} determines how the case
    ++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
    ++ The possible values of \spad{atanFlag} are \spad{"complex"},
    ++ \spad{"real: two sides"}, \spad{"real: left side"},
    ++ \spad{"real: right side"}, and \spad{"just do it"}.
    ++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
    ++ will be computed because, viewed as a function of a complex
    ++ variable, \spad{atan(f(x))} has an essential singularity.
    ++ Otherwise, the sign of the leading coefficient of the series
    ++ expansion of \spad{f(x)} determines the constant coefficient
    ++ in the series expansion of \spad{atan(f(x))}.  If this sign cannot
    ++ be determined, a series expansion is computed only when
    ++ \spad{atanFlag} is \spad{"just do it"}.  When the leading term
    ++ in the series expansion of \spad{f(x)} is of odd degree (or is a
    ++ rational degree with odd numerator), then the constant coefficient
    ++ in the series expansion of \spad{atan(f(x))} for values to the
    ++ left differs from that for values to the right.  If \spad{atanFlag}
    ++ is \spad{"real: two sides"}, no series expansion will be computed.
    ++ If \spad{atanFlag} is \spad{"real: left side"} the constant
    ++ coefficient for values to the left will be used and if \spad{atanFlag}
    ++ \spad{"real: right side"} the constant coefficient for values to the
    ++ right will be used.
    ++ If there is a problem in converting the function to a power series,
    ++ a record containing the name of the function that caused the problem
    ++ and a brief description of the problem is returned.
    ++ When expanding the expression into a series it is assumed that
    ++ the series is centered at 0.  For a series centered at a, the
    ++ user should perform the substitution \spad{x -> x + a} before calling
    ++ this function.

  exprToGenUPS : (FE,B,S) -> Result
```

```
    ++ exprToGenUPS(fcn,posCheck?,atanFlag) converts the expression
    ++ \spad{fcn} to a generalized power series.  If \spad{posCheck?}
    ++ is true, log's of negative numbers are not allowed nor are nth roots
    ++ of negative numbers with n even. If \spad{posCheck?} is false,
    ++ these are allowed.  \spad{atanFlag} determines how the case
    ++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
    ++ The possible values of \spad{atanFlag} are \spad{"complex"},
    ++ \spad{"real: two sides"}, \spad{"real: left side"},
    ++ \spad{"real: right side"}, and \spad{"just do it"}.
    ++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
    ++ will be computed because, viewed as a function of a complex
    ++ variable, \spad{atan(f(x))} has an essential singularity.
    ++ Otherwise, the sign of the leading coefficient of the series
    ++ expansion of \spad{f(x)} determines the constant coefficient
    ++ in the series expansion of \spad{atan(f(x))}.  If this sign cannot
    ++ be determined, a series expansion is computed only when
    ++ \spad{atanFlag} is \spad{"just do it"}.  When the leading term
    ++ in the series expansion of \spad{f(x)} is of odd degree (or is a
    ++ rational degree with odd numerator), then the constant coefficient
    ++ in the series expansion of \spad{atan(f(x))} for values to the
    ++ left differs from that for values to the right.  If \spad{atanFlag}
    ++ is \spad{"real: two sides"}, no series expansion will be computed.
    ++ If \spad{atanFlag} is \spad{"real: left side"} the constant
    ++ coefficient for values to the left will be used and if \spad{atanFlag}
    ++ \spad{"real: right side"} the constant coefficient for values to the
    ++ right will be used.
    ++ If there is a problem in converting the function to a power
    ++ series, we return a record containing the name of the function
    ++ that caused the problem and a brief description of the problem.
    ++ When expanding the expression into a series it is assumed that
    ++ the series is centered at 0.  For a series centered at a, the
    ++ user should perform the substitution \spad{x -> x + a} before calling
    ++ this function.
  localAbs: FE -> FE
    ++ localAbs(fcn) = \spad{abs(fcn)} or \spad{sqrt(fcn**2)} depending
    ++ on whether or not FE has a function \spad{abs}.  This should be
    ++ a local function, but the compiler won't allow it.

Implementation ==> add

  ratIfCan            : FE -> Union(RN,"failed")
  carefulNthRootIfCan : (UPS,NNI,B,B) -> Result
  stateProblem        : (S,S) -> Result
  polyToUPS           : SUP -> UPS
  listToUPS           : (L FE,(FE,B,S) -> Result,B,S,UPS,(UPS,UPS) -> UPS)_
                                        -> Result
  isNonTrivPower      : FE -> Union(Record(val:FE,exponent:I),"failed")
  powerToUPS          : (FE,I,B,S) -> Result
  kernelToUPS         : (K,B,S) -> Result
  nthRootToUPS        : (FE,NNI,B,S) -> Result
```

```
logToUPS            : (FE,B,S) -> Result
atancotToUPS        : (FE,B,S,I) -> Result
applyIfCan          : (UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
tranToUPS           : (K,FE,B,S) -> Result
powToUPS            : (L FE,B,S) -> Result
newElem             : FE -> FE
smpElem             : SMP -> FE
k2Elem              : K -> FE
contOnReals?        : S -> B
bddOnReals?         : S -> B
iExprToGenUPS       : (FE,B,S) -> Result
opsInvolvingX       : FE -> L BOP
opInOpList?         : (SY,L BOP) -> B
exponential?        : FE -> B
productOfNonZeroes? : FE -> B
powerToGenUPS       : (FE,I,B,S) -> Result
kernelToGenUPS      : (K,B,S) -> Result
nthRootToGenUPS     : (FE,NNI,B,S) -> Result
logToGenUPS         : (FE,B,S) -> Result
expToGenUPS         : (FE,B,S) -> Result
expGenUPS           : (UPS,B,S) -> Result
atancotToGenUPS     : (FE,FE,B,S,I) -> Result
genUPSApplyIfCan    : (UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
applyBddIfCan       : (FE,UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
tranToGenUPS        : (K,FE,B,S) -> Result
powToGenUPS         : (L FE,B,S) -> Result

ZEROCOUNT : I := 1000
-- number of zeroes to be removed when taking logs or nth roots

ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")

carefulNthRootIfCan(ups,n,posCheck?,rightOnly?) ==
  -- similar to 'nthRootIfCan', but it is fussy about the series
  -- it takes as an argument.  If 'n' is EVEN and 'posCheck?'
  -- is truem then the leading coefficient of the series must
  -- be POSITIVE.  In this case, if 'rightOnly?' is false, the
  -- order of the series must be zero.  The idea is that the
  -- series represents a real function of a real variable, and
  -- we want a unique real nth root defined on a neighborhood
  -- of zero.
  n < 1 => error "nthRoot: n must be positive"
  deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    deg := order(ups,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(ups,deg)) = 0 =>
      error "log of series with many leading zero coefficients"
  -- if 'posCheck?' is true, we do not allow nth roots of negative
  -- numbers when n in even
  if even?(n :: I) then
```

```
      if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then
        (signum :: I) = -1 =>
          return stateProblem("nth root","negative leading coefficient")
        not rightOnly? and not zero? deg => -- nth root not unique
          return stateProblem("nth root","series of non-zero order")
    (ans := nthRootIfCan(ups,n)) case "failed" =>
      stateProblem("nth root","no nth root")
    [ans :: UPS]

  stateProblem(function,problem) ==
    -- records the problem which occured in converting an expression
    -- to a power series
    [[function,problem]]

  exprToUPS(fcn,posCheck?,atanFlag) ==
    -- converts a functional expression to a power series
    --!! The following line is commented out so that expressions of
    --!! the form a**b will be normalized to exp(b * log(a)) even if
    --!! 'a' and 'b' do not involve the limiting variable 'x'.
    --!!                          - cjw 1 Dec 94
    --not member?(x,variables fcn) => [monomial(fcn,0)]
    (poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
      [polyToUPS univariate(poly :: POL,x)]
    (sum := isPlus fcn) case L(FE) =>
      listToUPS(sum :: L(FE),exprToUPS,posCheck?,atanFlag,0,
         (y1,y2) +-> y1 + y2)
    (prod := isTimes fcn) case L(FE) =>
      listToUPS(prod :: L(FE),exprToUPS,posCheck?,atanFlag,1,
         (y1,y2) +-> y1 * y2)
    (expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
      power := expt :: Record(val:FE,exponent:I)
      powerToUPS(power.val,power.exponent,posCheck?,atanFlag)
    (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
      kernelToUPS(ker :: K,posCheck?,atanFlag)
    error "exprToUPS: neither a sum, product, power, nor kernel"

  polyToUPS poly ==
    -- converts a polynomial to a power series
    zero? poly => 0
    -- we don't start with 'ans := 0' as this may lead to an
    -- enormous number of leading zeroes in the power series
    deg  := degree poly
    coef := leadingCoefficient(poly) :: FE
    ans  := monomial(coef,deg :: Expon)$UPS
    poly := reductum poly
    while not zero? poly repeat
      deg  := degree poly
      coef := leadingCoefficient(poly) :: FE
      ans  := ans + monomial(coef,deg :: Expon)$UPS
      poly := reductum poly
```

```
      ans

    listToUPS(list,feToUPS,posCheck?,atanFlag,ans,op) ==
      -- converts each element of a list of expressions to a power
      -- series and returns the sum of these series, when 'op' is +
      -- and 'ans' is 0, or the product of these series, when 'op' is *
      -- and 'ans' is 1
      while not null list repeat
        (term := feToUPS(first list,posCheck?,atanFlag)) case %problem =>
          return term
        ans := op(ans,term.%series)
        list := rest list
      [ans]

    isNonTrivPower fcn ==
      -- is the function a power with exponent other than 0 or 1?
      (expt := isPower fcn) case "failed" => "failed"
      power := expt :: Record(val:FE,exponent:I)
--      one? power.exponent => "failed"
      (power.exponent = 1) => "failed"
      power

    powerToUPS(fcn,n,posCheck?,atanFlag) ==
      -- converts an integral power to a power series
      (b := exprToUPS(fcn,posCheck?,atanFlag)) case %problem => b
      n > 0 => [(b.%series) ** n]
      -- check lowest order coefficient when n < 0
      ups := b.%series; deg := degree ups
      if (coef := coefficient(ups,deg)) = 0 then
        deg := order(ups,deg + ZEROCOUNT :: Expon)
        (coef := coefficient(ups,deg)) = 0 =>
          error "inverse of series with many leading zero coefficients"
      [ups ** n]

    kernelToUPS(ker,posCheck?,atanFlag) ==
      -- converts a kernel to a power series
      (sym := symbolIfCan(ker)) case Symbol =>
        (sym :: Symbol) = x => [monomial(1,1)]
        [monomial(ker :: FE,0)]
      empty?(args := argument ker) => [monomial(ker :: FE,0)]
      not member?(x, variables(ker :: FE)) => [monomial(ker :: FE,0)]
      empty? rest args =>
        arg := first args
        is?(ker,"abs" :: Symbol) =>
          nthRootToUPS(arg*arg,2,posCheck?,atanFlag)
        is?(ker,"%paren" :: Symbol) => exprToUPS(arg,posCheck?,atanFlag)
        is?(ker,"log" :: Symbol) => logToUPS(arg,posCheck?,atanFlag)
        is?(ker,"exp" :: Symbol) =>
          applyIfCan(expIfCan,arg,"exp",posCheck?,atanFlag)
        tranToUPS(ker,arg,posCheck?,atanFlag)
```

```
    is?(ker,"%power" :: Symbol) => powToUPS(args,posCheck?,atanFlag)
    is?(ker,"nthRoot" :: Symbol) =>
      n := retract(second args)@I
      nthRootToUPS(first args,n :: NNI,posCheck?,atanFlag)
    stateProblem(string name ker,"unknown kernel")

  nthRootToUPS(arg,n,posCheck?,atanFlag) ==
    -- converts an nth root to a power series
    -- this is not used in the limit package, so the series may
    -- have non-zero order, in which case nth roots may not be unique
    (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
    ans := carefulNthRootIfCan(result.%series,n,posCheck?,false)
    ans case %problem => ans
    [ans.%series]

  logToUPS(arg,posCheck?,atanFlag) ==
    -- converts a logarithm log(f(x)) to a power series
    -- f(x) must have order 0 and if 'posCheck?' is true,
    -- then f(x) must have a non-negative leading coefficient
    (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
    ups := result.%series
    not zero? order(ups,1) =>
      stateProblem("log","series of non-zero order")
    coef := coefficient(ups,0)
    -- if 'posCheck?' is true, we do not allow logs of negative numbers
    if posCheck? then
      if ((signum := sign(coef)$SIGNEF) case I) then
        (signum :: I) = -1 =>
          return stateProblem("log","negative leading coefficient")
    [logIfCan(ups) :: UPS]

  if FE has abs: FE -> FE then
    localAbs fcn == abs fcn
  else
    localAbs fcn == sqrt(fcn * fcn)

  signOfExpression: FE -> FE
  signOfExpression arg == localAbs(arg)/arg

  atancotToUPS(arg,posCheck?,atanFlag,plusMinus) ==
    -- converts atan(f(x)) to a power series
    (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
    ups := result.%series; coef := coefficient(ups,0)
    (ord := order(ups,0)) = 0 and coef * coef = -1 =>
      -- series involves complex numbers
      return stateProblem("atan","logarithmic singularity")
    cc : FE :=
      ord < 0 =>
        atanFlag = "complex" =>
          return stateProblem("atan","essential singularity")
```

```
        (rn := ratIfCan(ord :: FE)) case "failed" =>
          -- this condition usually won't occur because exponents will
          -- be integers or rational numbers
          return stateProblem("atan","branch problem")
        if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
          -- expansions to the left and right of zero have different
          -- constant coefficients
          return stateProblem("atan","branch problem")
        lc := coefficient(ups,ord)
        (signum := sign(lc)$SIGNEF) case "failed" =>
          -- can't determine sign
          atanFlag = "just do it" =>
            plusMinus = 1 => pi()/(2 :: FE)
            0
          posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
          plusMinus = 1 => posNegPi2
          pi()/(2 :: FE) - posNegPi2
          --return stateProblem("atan","branch problem")
        left? : B := atanFlag = "real: left side"; n := signum :: Integer
        (left? and n = 1) or (not left? and n = -1) =>
          plusMinus = 1 => -pi()/(2 :: FE)
          pi()
        plusMinus = 1 => pi()/(2 :: FE)
        0
      atan coef
    [(cc :: UPS) + integrate(plusMinus * differentiate(ups)/(1 + ups*ups))]

  applyIfCan(fcn,arg,fcnName,posCheck?,atanFlag) ==
    -- converts fcn(arg) to a power series
    (ups := exprToUPS(arg,posCheck?,atanFlag)) case %problem => ups
    ans := fcn(ups.%series)
    ans case "failed" => stateProblem(fcnName,"essential singularity")
    [ans :: UPS]

  tranToUPS(ker,arg,posCheck?,atanFlag) ==
    -- converts ker to a power series for certain functions
    -- in trig or hyperbolic trig categories
    is?(ker,"sin" :: SY) =>
      applyIfCan(sinIfCan,arg,"sin",posCheck?,atanFlag)
    is?(ker,"cos" :: SY) =>
      applyIfCan(cosIfCan,arg,"cos",posCheck?,atanFlag)
    is?(ker,"tan" :: SY) =>
      applyIfCan(tanIfCan,arg,"tan",posCheck?,atanFlag)
    is?(ker,"cot" :: SY) =>
      applyIfCan(cotIfCan,arg,"cot",posCheck?,atanFlag)
    is?(ker,"sec" :: SY) =>
      applyIfCan(secIfCan,arg,"sec",posCheck?,atanFlag)
    is?(ker,"csc" :: SY) =>
      applyIfCan(cscIfCan,arg,"csc",posCheck?,atanFlag)
    is?(ker,"asin" :: SY) =>
```

```
        applyIfCan(asinIfCan,arg,"asin",posCheck?,atanFlag)
      is?(ker,"acos" :: SY) =>
        applyIfCan(acosIfCan,arg,"acos",posCheck?,atanFlag)
      is?(ker,"atan" :: SY) => atancotToUPS(arg,posCheck?,atanFlag,1)
      is?(ker,"acot" :: SY) => atancotToUPS(arg,posCheck?,atanFlag,-1)
      is?(ker,"asec" :: SY) =>
        applyIfCan(asecIfCan,arg,"asec",posCheck?,atanFlag)
      is?(ker,"acsc" :: SY) =>
        applyIfCan(acscIfCan,arg,"acsc",posCheck?,atanFlag)
      is?(ker,"sinh" :: SY) =>
        applyIfCan(sinhIfCan,arg,"sinh",posCheck?,atanFlag)
      is?(ker,"cosh" :: SY) =>
        applyIfCan(coshIfCan,arg,"cosh",posCheck?,atanFlag)
      is?(ker,"tanh" :: SY) =>
        applyIfCan(tanhIfCan,arg,"tanh",posCheck?,atanFlag)
      is?(ker,"coth" :: SY) =>
        applyIfCan(cothIfCan,arg,"coth",posCheck?,atanFlag)
      is?(ker,"sech" :: SY) =>
        applyIfCan(sechIfCan,arg,"sech",posCheck?,atanFlag)
      is?(ker,"csch" :: SY) =>
        applyIfCan(cschIfCan,arg,"csch",posCheck?,atanFlag)
      is?(ker,"asinh" :: SY) =>
        applyIfCan(asinhIfCan,arg,"asinh",posCheck?,atanFlag)
      is?(ker,"acosh" :: SY) =>
        applyIfCan(acoshIfCan,arg,"acosh",posCheck?,atanFlag)
      is?(ker,"atanh" :: SY) =>
        applyIfCan(atanhIfCan,arg,"atanh",posCheck?,atanFlag)
      is?(ker,"acoth" :: SY) =>
        applyIfCan(acothIfCan,arg,"acoth",posCheck?,atanFlag)
      is?(ker,"asech" :: SY) =>
        applyIfCan(asechIfCan,arg,"asech",posCheck?,atanFlag)
      is?(ker,"acsch" :: SY) =>
        applyIfCan(acschIfCan,arg,"acsch",posCheck?,atanFlag)
      stateProblem(string name ker,"unknown kernel")

    powToUPS(args,posCheck?,atanFlag) ==
      -- converts a power f(x) ** g(x) to a power series
      (logBase := logToUPS(first args,posCheck?,atanFlag)) case %problem =>
        logBase
      (expon := exprToUPS(second args,posCheck?,atanFlag)) case %problem =>
        expon
      ans := expIfCan((expon.%series) * (logBase.%series))
      ans case "failed" => stateProblem("exp","essential singularity")
      [ans :: UPS]

-- Generalized power series: power series in x, where log(x) and
-- bounded functions of x are allowed to appear in the coefficients
-- of the series.  Used for evaluating REAL limits at x = 0.

    newElem f ==
```

```
-- rewrites a functional expression; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
  smpElem(numer f) / smpElem(denom f)

smpElem p == map(k2Elem,(x1:R):FE +-> x1::FE,p)$PCL

k2Elem k ==
-- rewrites a kernel; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
  null(args := [newElem a for a in argument k]) => k::FE
  iez  := inv(ez  := exp(z := first args))
  sinz := sin z; cosz := cos z
  is?(k,"tan" :: Symbol)  => sinz / cosz
  is?(k,"cot" :: Symbol)  => cosz / sinz
  is?(k,"sec" :: Symbol)  => inv cosz
  is?(k,"csc" :: Symbol)  => inv sinz
  is?(k,"sinh" :: Symbol) => (ez - iez) / (2 :: FE)
  is?(k,"cosh" :: Symbol) => (ez + iez) / (2 :: FE)
  is?(k,"tanh" :: Symbol) => (ez - iez) / (ez + iez)
  is?(k,"coth" :: Symbol) => (ez + iez) / (ez - iez)
  is?(k,"sech" :: Symbol) => 2 * inv(ez + iez)
  is?(k,"csch" :: Symbol) => 2 * inv(ez - iez)
  (operator k) args

CONTFCNS : L S := ["sin","cos","atan","acot","exp","asinh"]
-- functions which are defined and continuous at all real numbers

BDDFCNS : L S := ["sin","cos","atan","acot"]
-- functions which are bounded on the reals

contOnReals? fcn == member?(fcn,CONTFCNS)
bddOnReals? fcn  == member?(fcn,BDDFCNS)

exprToGenUPS(fcn,posCheck?,atanFlag) ==
  -- converts a functional expression to a generalized power
  -- series; "generalized" means that log(x) and bounded functions
  -- of x are allowed to appear in the coefficients of the series
  iExprToGenUPS(newElem fcn,posCheck?,atanFlag)

iExprToGenUPS(fcn,posCheck?,atanFlag) ==
  -- converts a functional expression to a generalized power
  -- series without first normalizing the expression
  --!! The following line is commented out so that expressions of
  --!! the form a**b will be normalized to exp(b * log(a)) even if
  --!! 'a' and 'b' do not involve the limiting variable 'x'.
  --!!                          - cjw 1 Dec 94
  --not member?(x,variables fcn) => [monomial(fcn,0)]
  (poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
```

```
      [polyToUPS univariate(poly :: POL,x)]
    (sum := isPlus fcn) case L(FE) =>
      listToUPS(sum :: L(FE),iExprToGenUPS,posCheck?,atanFlag,0,
         (y1,y2) +-> y1 + y2)
    (prod := isTimes fcn) case L(FE) =>
      listToUPS(prod :: L(FE),iExprToGenUPS,posCheck?,atanFlag,1,
         (y1,y2) +-> y1 * y2)
    (expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
      power := expt :: Record(val:FE,exponent:I)
      powerToGenUPS(power.val,power.exponent,posCheck?,atanFlag)
    (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
      kernelToGenUPS(ker :: K,posCheck?,atanFlag)
    error "exprToGenUPS: neither a sum, product, power, nor kernel"

  opsInvolvingX fcn ==
    opList := [op for k in tower fcn | unary?(op := operator k) _
               and member?(x,variables first argument k)]
    removeDuplicates opList

  opInOpList?(name,opList) ==
    for op in opList repeat
      is?(op,name) => return true
    false

  exponential? fcn ==
    -- is 'fcn' of the form exp(f)?
    (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
      is?(ker :: K,"exp" :: Symbol)
    false

  productOfNonZeroes? fcn ==
    -- is 'fcn' a product of non-zero terms, where 'non-zero'
    -- means an exponential or a function not involving x
    exponential? fcn => true
    (prod := isTimes fcn) case "failed" => false
    for term in (prod :: L(FE)) repeat
      (not exponential? term) and member?(x,variables term) =>
        return false
    true

  powerToGenUPS(fcn,n,posCheck?,atanFlag) ==
    -- converts an integral power to a generalized power series
    -- if n < 0 and the lowest order coefficient of the series
    -- involves x, we are careful about inverting this coefficient
    -- the coefficient is inverted only if
    -- (a) the only function involving x is 'log', or
    -- (b) the lowest order coefficient is a product of exponentials
    --     and functions not involving x
    (b := exprToGenUPS(fcn,posCheck?,atanFlag)) case %problem => b
    n > 0 => [(b.%series) ** n]
```

```
    -- check lowest order coefficient when n < 0
    ups := b.%series; deg := degree ups
    if (coef := coefficient(ups,deg)) = 0 then
      deg := order(ups,deg + ZEROCOUNT :: Expon)
      (coef := coefficient(ups,deg)) = 0 =>
        error "inverse of series with many leading zero coefficients"
    xOpList := opsInvolvingX coef
    -- only function involving x is 'log'
    (null xOpList) => [ups ** n]
    (null rest xOpList and is?(first xOpList,"log" :: SY)) =>
      [ups ** n]
    -- lowest order coefficient is a product of exponentials and
    -- functions not involving x
    productOfNonZeroes? coef => [ups ** n]
    stateProblem("inv","lowest order coefficient involves x")

  kernelToGenUPS(ker,posCheck?,atanFlag) ==
    -- converts a kernel to a generalized power series
    (sym := symbolIfCan(ker)) case Symbol =>
      (sym :: Symbol) = x => [monomial(1,1)]
      [monomial(ker :: FE,0)]
    empty?(args := argument ker) => [monomial(ker :: FE,0)]
    empty? rest args =>
      arg := first args
      is?(ker,"abs" :: Symbol) =>
        nthRootToGenUPS(arg*arg,2,posCheck?,atanFlag)
      is?(ker,"%paren" :: Symbol) => iExprToGenUPS(arg,posCheck?,atanFlag)
      is?(ker,"log" :: Symbol) => logToGenUPS(arg,posCheck?,atanFlag)
      is?(ker,"exp" :: Symbol) => expToGenUPS(arg,posCheck?,atanFlag)
      tranToGenUPS(ker,arg,posCheck?,atanFlag)
    is?(ker,"%power" :: Symbol) => powToGenUPS(args,posCheck?,atanFlag)
    is?(ker,"nthRoot" :: Symbol) =>
      n := retract(second args)@I
      nthRootToGenUPS(first args,n :: NNI,posCheck?,atanFlag)
    stateProblem(string name ker,"unknown kernel")

  nthRootToGenUPS(arg,n,posCheck?,atanFlag) ==
    -- convert an nth root to a power series
    -- used for computing right hand limits, so the series may have
    -- non-zero order, but may not have a negative leading coefficient
    -- when n is even
    (result := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
      result
    ans := carefulNthRootIfCan(result.%series,n,posCheck?,true)
    ans case %problem => ans
    [ans.%series]

  logToGenUPS(arg,posCheck?,atanFlag) ==
    -- converts a logarithm log(f(x)) to a generalized power series
    (result := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
```

```
      result
    ups := result.%series; deg := degree ups
    if (coef := coefficient(ups,deg)) = 0 then
      deg := order(ups,deg + ZEROCOUNT :: Expon)
      (coef := coefficient(ups,deg)) = 0 =>
        error "log of series with many leading zero coefficients"
    -- if 'posCheck?' is true, we do not allow logs of negative numbers
    if posCheck? then
      if ((signum := sign(coef)$SIGNEF) case I) then
        (signum :: I) = -1 =>
          return stateProblem("log","negative leading coefficient")
    -- create logarithmic term, avoiding log's of negative rationals
    lt := monomial(coef,deg)$UPS; cen := center lt
    -- check to see if lowest order coefficient is a negative rational
    negRat? : Boolean :=
      ((rat := ratIfCan coef) case RN) =>
        (rat :: RN) < 0 => true
        false
      false
    logTerm : FE :=
      mon : FE := (x :: FE) - (cen :: FE)
      pow : FE := mon ** (deg :: FE)
      negRat? => log(coef * pow)
      term1 : FE := (deg :: FE) * log(mon)
      log(coef) + term1
    [monomial(logTerm,0) + log(ups/lt)]

  expToGenUPS(arg,posCheck?,atanFlag) ==
    -- converts an exponential exp(f(x)) to a generalized
    -- power series
    (ups := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem => ups
    expGenUPS(ups.%series,posCheck?,atanFlag)

  expGenUPS(ups,posCheck?,atanFlag) ==
    -- computes the exponential of a generalized power series.
    -- If the series has order zero and the constant term a0 of the
    -- series involves x, the function tries to expand exp(a0) as
    -- a power series.
    (deg := order(ups,1)) < 0 =>
      stateProblem("exp","essential singularity")
    deg > 0 => [exp ups]
    lc := coefficient(ups,0); xOpList := opsInvolvingX lc
    not opInOpList?("log" :: SY,xOpList) => [exp ups]
    -- try to fix exp(lc) if necessary
    expCoef :=
      normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
    opInOpList?("log" :: SY,opsInvolvingX expCoef) =>
      stateProblem("exp","logs in constant coefficient")
    result := exprToGenUPS(expCoef,posCheck?,atanFlag)
    result case %problem => result
```

```
    [(result.%series) * exp(ups - monomial(lc,0))]

  atancotToGenUPS(fe,arg,posCheck?,atanFlag,plusMinus) ==
    -- converts atan(f(x)) to a generalized power series
    (result := exprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
      trouble := result.%problem
      trouble.prob = "essential singularity" => [monomial(fe,0)$UPS]
      result
    ups := result.%series; coef := coefficient(ups,0)
    -- series involves complex numbers
    (ord := order(ups,0)) = 0 and coef * coef = -1 =>
      y := differentiate(ups)/(1 + ups*ups)
      yCoef := coefficient(y,-1)
      [monomial(log yCoef,0) + integrate(y - monomial(yCoef,-1)$UPS)]
    cc : FE :=
      ord < 0 =>
        atanFlag = "complex" =>
          return stateProblem("atan","essential singularity")
        (rn := ratIfCan(ord :: FE)) case "failed" =>
          -- this condition usually won't occur because exponents will
          -- be integers or rational numbers
          return stateProblem("atan","branch problem")
        if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
          -- expansions to the left and right of zero have different
          -- constant coefficients
          return stateProblem("atan","branch problem")
        lc := coefficient(ups,ord)
        (signum := sign(lc)$SIGNEF) case "failed" =>
          -- can't determine sign
          atanFlag = "just do it" =>
            plusMinus = 1 => pi()/(2 :: FE)
            0
          posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
          plusMinus = 1 => posNegPi2
          pi()/(2 :: FE) - posNegPi2
          --return stateProblem("atan","branch problem")
        left? : B := atanFlag = "real: left side"; n := signum :: Integer
        (left? and n = 1) or (not left? and n = -1) =>
          plusMinus = 1 => -pi()/(2 :: FE)
          pi()
        plusMinus = 1 => pi()/(2 :: FE)
        0
      atan coef
    [(cc :: UPS) + integrate(differentiate(ups)/(1 + ups*ups))]

  genUPSApplyIfCan(fcn,arg,fcnName,posCheck?,atanFlag) ==
    -- converts fcn(arg) to a generalized power series
    (series := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
      series
    ups := series.%series
```

```
    (deg := order(ups,1)) < 0 =>
      stateProblem(fcnName,"essential singularity")
    deg > 0 => [fcn(ups) :: UPS]
    lc := coefficient(ups,0); xOpList := opsInvolvingX lc
    null xOpList => [fcn(ups) :: UPS]
    opInOpList?("log" :: SY,xOpList) =>
      stateProblem(fcnName,"logs in constant coefficient")
    contOnReals? fcnName => [fcn(ups) :: UPS]
    stateProblem(fcnName,"x in constant coefficient")

  applyBddIfCan(fe,fcn,arg,fcnName,posCheck?,atanFlag) ==
    -- converts fcn(arg) to a generalized power series, where the
    -- function fcn is bounded for real values
    -- if fcn(arg) has an essential singularity as a complex
    -- function, we return fcn(arg) as a monomial of degree 0
    (ups := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
      trouble := ups.%problem
      trouble.prob = "essential singularity" => [monomial(fe,0)$UPS]
      ups
    (ans := fcn(ups.%series)) case "failed" => [monomial(fe,0)$UPS]
    [ans :: UPS]

  tranToGenUPS(ker,arg,posCheck?,atanFlag) ==
    -- converts op(arg) to a power series for certain functions
    -- op in trig or hyperbolic trig categories
    -- N.B. when this function is called, 'k2elem' will have been
    -- applied, so the following functions cannot appear:
    -- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
    is?(ker,"sin" :: SY) =>
      applyBddIfCan(ker :: FE,sinIfCan,arg,"sin",posCheck?,atanFlag)
    is?(ker,"cos" :: SY) =>
      applyBddIfCan(ker :: FE,cosIfCan,arg,"cos",posCheck?,atanFlag)
    is?(ker,"asin" :: SY) =>
      genUPSApplyIfCan(asinIfCan,arg,"asin",posCheck?,atanFlag)
    is?(ker,"acos" :: SY) =>
      genUPSApplyIfCan(acosIfCan,arg,"acos",posCheck?,atanFlag)
    is?(ker,"atan" :: SY) =>
      atancotToGenUPS(ker :: FE,arg,posCheck?,atanFlag,1)
    is?(ker,"acot" :: SY) =>
      atancotToGenUPS(ker :: FE,arg,posCheck?,atanFlag,-1)
    is?(ker,"asec" :: SY) =>
      genUPSApplyIfCan(asecIfCan,arg,"asec",posCheck?,atanFlag)
    is?(ker,"acsc" :: SY) =>
      genUPSApplyIfCan(acscIfCan,arg,"acsc",posCheck?,atanFlag)
    is?(ker,"asinh" :: SY) =>
      genUPSApplyIfCan(asinhIfCan,arg,"asinh",posCheck?,atanFlag)
    is?(ker,"acosh" :: SY) =>
      genUPSApplyIfCan(acoshIfCan,arg,"acosh",posCheck?,atanFlag)
    is?(ker,"atanh" :: SY) =>
      genUPSApplyIfCan(atanhIfCan,arg,"atanh",posCheck?,atanFlag)
```

```
      is?(ker,"acoth" :: SY) =>
        genUPSApplyIfCan(acothIfCan,arg,"acoth",posCheck?,atanFlag)
      is?(ker,"asech" :: SY) =>
        genUPSApplyIfCan(asechIfCan,arg,"asech",posCheck?,atanFlag)
      is?(ker,"acsch" :: SY) =>
        genUPSApplyIfCan(acschIfCan,arg,"acsch",posCheck?,atanFlag)
      stateProblem(string name ker,"unknown kernel")

    powToGenUPS(args,posCheck?,atanFlag) ==
      -- converts a power f(x) ** g(x) to a generalized power series
      (logBase := logToGenUPS(first args,posCheck?,atanFlag)) case %problem =>
        logBase
      expon := iExprToGenUPS(second args,posCheck?,atanFlag)
      expon case %problem => expon
      expGenUPS((expon.%series) * (logBase.%series),posCheck?,atanFlag)
```

———-

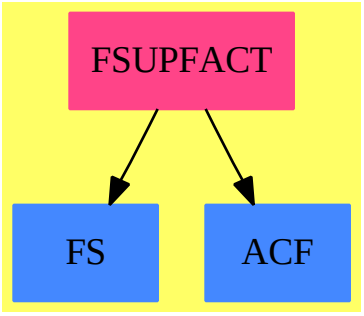
— **FS2UPS.dotabb** —

```
"FS2UPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FS2UPS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FS2UPS" -> "ACF"
"FS2UPS" -> "FS"
```

———-

7.44 package FSUPFACT FunctionSpaceUnivariatePolynomialFactor

7.44.1 FunctionSpaceUnivariatePolynomialFactor (FSUPFACT)



Exports:
anfactor ffactor qfactor UP2ifCan

— **package FSUPFACT FunctionSpaceUnivariatePolynomialFactor** —

```
)abbrev package FSUPFACT FunctionSpaceUnivariatePolynomialFactor
++ Author: Manuel Bronstein
++ Date Created: 12 May 1988
++ Date Last Updated: 22 September 1993
++ Keywords: function, space, polynomial, factoring
++ Description:
++ This package is used internally by IR2F

FunctionSpaceUnivariatePolynomialFactor(R, F, UP):
 Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
  F : FunctionSpace R
  UP: UnivariatePolynomialCategory F

  Q   ==> Fraction Integer
  K   ==> Kernel F
  AN  ==> AlgebraicNumber
  PQ  ==> SparseMultivariatePolynomial(Q, K)
  PR  ==> SparseMultivariatePolynomial(R, K)
  UPQ ==> SparseUnivariatePolynomial Q
  UPA ==> SparseUnivariatePolynomial AN
  FR  ==> Factored UP
  FRQ ==> Factored UPQ
  FRA ==> Factored UPA

  Exports ==> with
    ffactor: UP -> FR
      ++ ffactor(p) tries to factor a univariate polynomial p over F
    qfactor: UP -> Union(FRQ, "failed")
      ++ qfactor(p) tries to factor p over fractions of integers,
      ++ returning "failed" if it cannot
    UP2ifCan: UP  -> Union(overq: UPQ, overan: UPA, failed: Boolean)
      ++ UP2ifCan(x) should be local but conditional.
    if F has RetractableTo AN then
      anfactor: UP -> Union(FRA, "failed")
        ++ anfactor(p) tries to factor p over algebraic numbers,
        ++ returning "failed" if it cannot

  Implementation ==> add
    import AlgFactor(UPA)
    import RationalFactorize(UPQ)

    P2QifCan : PR  -> Union(PQ, "failed")
```

```
    UPQ2UP   : (SparseUnivariatePolynomial PQ, F) -> UP
    PQ2F     : (PQ, F) -> F
    ffactor0 : UP -> FR

    dummy := kernel(new()$Symbol)$K

    if F has RetractableTo AN then
      UPAN2F: UPA -> UP
      UPQ2AN: UPQ -> UPA

      UPAN2F p ==
        map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(AN,UPA,F,UP)

      UPQ2AN p ==
        map(x+->x::AN, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,AN,UPA)

      ffactor p ==
        (pq := anfactor p) case FRA =>
                        map(UPAN2F, pq::FRA)$FactoredFunctions2(UPA, UP)
        ffactor0 p

      anfactor p ==
        (q := UP2ifCan p) case overq =>
                    map(UPQ2AN, factor(q.overq))$FactoredFunctions2(UPQ, UPA)
        q case overan => factor(q.overan)
        "failed"

      UP2ifCan p ==
        ansq := 0$UPQ ; ansa := 0$UPA
        goforq? := true
        while p ^= 0 repeat
          if goforq? then
            rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
            if rq case Q then
              ansq := ansq + monomial(rq::Q, degree p)
              ansa := ansa + monomial(rq::Q::AN, degree p)
            else
              goforq? := false
              ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
              if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
                            else return [true]
          else
            ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
            if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
                          else return [true]
          p := reductum p
        goforq? => [ansq]
        [ansa]

    else
```

```
  UPQ2F: UPQ -> UP

  UPQ2F p ==
    map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

  ffactor p ==
    (pq := qfactor p) case FRQ =>
                      map(UPQ2F, pq::FRQ)$FactoredFunctions2(UPQ, UP)
    ffactor0 p

  UP2ifCan p ==
    ansq := 0$UPQ
    while p ^= 0 repeat
      rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
      if rq case Q then ansq := ansq + monomial(rq::Q, degree p)
                   else return [true]
      p := reductum p
    [ansq]

ffactor0 p ==
  smp := numer(ep := p(dummy::F))
  (q := P2QifCan smp) case "failed" => p::FR
  map(x+->UPQ2UP(univariate(x, dummy), denom(ep)::F), factor(q::PQ
        )$MRationalFactorize(IndexedExponents K, K, Integer,
             PQ))$FactoredFunctions2(PQ, UP)

UPQ2UP(p, d) ==
  map(x+->PQ2F(x, d), p)$UnivariatePolynomialCategoryFunctions2(PQ,
                               SparseUnivariatePolynomial PQ, F, UP)

PQ2F(p, d) ==
  map((x:K):F+->x::F, (y:Q):F+->y::F, p)_
    $PolynomialCategoryLifting(IndexedExponents K, K, Q, PQ, F) / d

qfactor p ==
  (q := UP2ifCan p) case overq => factor(q.overq)
  "failed"

P2QifCan p ==
  and/[retractIfCan(c::F)@Union(Q, "failed") case Q
       for c in coefficients p] =>
        map(x+->x::PQ, y+->retract(y::F)@Q :: PQ, p)_
          $PolynomialCategoryLifting(IndexedExponents K,K,R,PR,PQ)
  "failed"
```

———-

— FSUPFACT.dotabb —

```
"FSUPFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSUPFACT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FSUPFACT" -> "FS"
"FSUPFACT" -> "ACF"
```

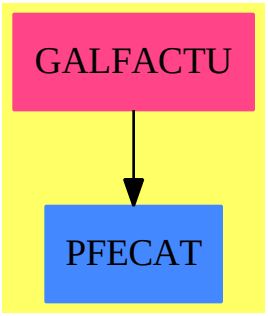
———————-

Chapter 8

Chapter G

8.1 package GALFACTU GaloisGroupFactorizationUtilities

8.1.1 GaloisGroupFactorizationUtilities (GALFACTU)



Exports:

beauzamyBound	bombieriNorm	height	infinityNorm	length
norm	quadraticNorm	rootBound	singleFactorBound	

— package GALFACTU GaloisGroupFactorizationUtilities —

```
)abbrev package GALFACTU GaloisGroupFactorizationUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 19 October 1995
++ Basic Functions:
++ Related Constructors:
++ Also See:
```

```
++ AMS Classifications:
++ Keywords:
++ References:
++ [1] Bernard Beauzamy, Products of polynomials and a priori estimates for
++ coefficients in polynomial decompositions: a sharp result,
++ J. Symbolic Computation (1992) 13, 463-472
++ [2] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri's Norms: I. The Largest Factor,
++ J. Symbolic Computation (1993) 16, 115-130
++ [3] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri's Norms: II. The Smallest Factor,
++ J. Symbolic Computation (1993) 16, 131-145
++ [4] Maurice Mignotte, Some Useful Bounds,
++ Computing, Suppl. 4, 259-263 (1982), Springer-Verlag
++ [5] Donald E. Knuth, The Art of Computer Programming, Vol. 2, (Seminumerical
++ Algorithms) 1st edition, 2nd printing, Addison-Wesley 1971, p. 397-398
++ [6] Bernard Beauzamy, Vilmar Trevisan and Paul S. Wang, Polynomial
++ Factorization: Sharp Bounds, Efficient Algorithms,
++ J. Symbolic Computation (1993) 15, 393-413
++ [7] Augustin-Lux Cauchy, Exercices de Math\'ematiques Quatri\`eme Ann\'ee.
++ De Bure Fr\`eres, Paris 1829 (reprinted Oeuvres, II S\'erie, Tome IX,
++ Gauthier-Villars, Paris, 1891).
++ Description:
++ \spadtype{GaloisGroupFactorizationUtilities} provides functions
++ that will be used by the factorizer.

GaloisGroupFactorizationUtilities(R,UP,F): Exports == Implementation where
  R : Ring
  UP : UnivariatePolynomialCategory R
  F : Join(FloatingPointSystem,RetractableTo(R),Field,
   TranscendentalFunctionCategory,ElementaryFunctionCategory)
  N ==> NonNegativeInteger
  P ==> PositiveInteger
  Z ==> Integer

  Exports ==> with
    beauzamyBound: UP -> Z -- See [1]
      ++ beauzamyBound(p) returns a bound on the larger coefficient of any
      ++ factor of p.
    bombieriNorm: UP -> F -- See [1]
      ++ bombieriNorm(p) returns quadratic Bombieri's norm of p.
    bombieriNorm: (UP,P) -> F -- See [2] and [3]
      ++ bombieriNorm(p,n) returns the nth Bombieri's norm of p.
    rootBound: UP -> Z -- See [4] and [5]
      ++ rootBound(p) returns a bound on the largest norm of the complex roots
      ++ of p.
    singleFactorBound: (UP,N) -> Z -- See [6]
      ++ singleFactorBound(p,r) returns a bound on the infinite norm of
      ++ the factor of p with smallest Bombieri's norm. r is a lower bound
      ++ for the number of factors of p. p shall be of degree higher or equal
```

```
    ++ to 2.
  singleFactorBound: UP -> Z -- See [6]
    ++ singleFactorBound(p,r) returns a bound on the infinite norm of
    ++ the factor of p with smallest Bombieri's norm. p shall be of degree
    ++ higher or equal to 2.
  norm: (UP,P) -> F
    ++ norm(f,p) returns the lp norm of the polynomial f.
  quadraticNorm: UP -> F
    ++ quadraticNorm(f) returns the l2 norm of the polynomial f.
  infinityNorm: UP -> F
    ++ infinityNorm(f) returns the maximal absolute value of the coefficients
    ++ of the polynomial f.
  height: UP -> F
    ++ height(p) returns the maximal absolute value of the coefficients of
    ++ the polynomial p.
  length: UP -> F
    ++ length(p) returns the sum of the absolute values of the coefficients
    ++ of the polynomial p.

Implementation ==> add

  import GaloisGroupUtilities(F)

  height(p:UP):F == infinityNorm(p)

  length(p:UP):F == norm(p,1)

  norm(f:UP,p:P):F ==
    n : F := 0
    for c in coefficients f repeat
      n := n+abs(c::F)**p
    nthRoot(n,p::N)

  quadraticNorm(f:UP):F == norm(f,2)

  infinityNorm(f:UP):F ==
    n : F := 0
    for c in coefficients f repeat
      n := max(n,c::F)
    n

  singleFactorBound(p:UP,r:N):Z == -- See [6]
    n : N := degree p
    r := max(2,r)
    n < r => error "singleFactorBound: Bad arguments."
    nf : F := n :: F
    num : F := nthRoot(bombieriNorm(p),r)
    if F has Gamma: F -> F then
      num := num*nthRoot(Gamma(nf+1$F),2*r)
      den : F := Gamma(nf/((2*r)::F)+1$F)
```

```
    else
      num := num*(2::F)**(5/8+n/2)*exp(1$F/(4*nf))
      den : F := (pi()$F*nf)**(3/8)
    safeFloor( num/den )

  singleFactorBound(p:UP):Z == singleFactorBound(p,2) -- See [6]

  rootBound(p:UP):Z == -- See [4] and [5]
    n := degree p
    zero? n => 0
    lc := abs(leadingCoefficient(p)::F)
    b1 : F := 0 -- Mignotte
    b2 : F := 0 -- Knuth
    b3 : F := 0 -- Zassenhaus in [5]
    b4 : F := 0 -- Cauchy in [7]
    c : F := 0
    cl : F := 0
    for i in 1..n repeat
      c := abs(coefficient(p,(n-i)::N)::F)
      b1 := max(b1,c)
      cl := c/lc
      b2 := max(b2,nthRoot(cl,i))
      b3 := max(b3,nthRoot(cl/pascalTriangle(n,i),i))
      b4 := max(b4,nthRoot(n*cl,i))
    min(1+safeCeiling(b1/lc),min(safeCeiling(2*b2),min(safeCeiling(b3/
     (nthRoot(2::F,n)-1)),safeCeiling(b4))))

  beauzamyBound(f:UP):Z == -- See [1]
    d := degree f
    zero? d => safeFloor bombieriNorm f
    safeFloor( (bombieriNorm(f)*(3::F)**(3/4+d/2))/
     (2*sqrt(pi()$F*(d::F))) )

  bombieriNorm(f:UP,p:P):F == -- See [2] and [3]
    d := degree f
    b := abs(coefficient(f,0)::F)
    if zero? d then return b
     else b := b**p
    b := b+abs(leadingCoefficient(f)::F)**p
    dd := (d-1) quo 2
    for i in 1..dd repeat
      b := b+(abs(coefficient(f,i)::F)**p+abs(coefficient(f,(d-i)::N)::F)**p)
       /pascalTriangle(d,i)
    if even? d then
      dd := dd+1
      b := b+abs(coefficient(f, dd::N)::F)**p/pascalTriangle(d,dd)
    nthRoot(b,p::N)

  bombieriNorm(f:UP):F == bombieriNorm(f,2) -- See [1]
```

———-

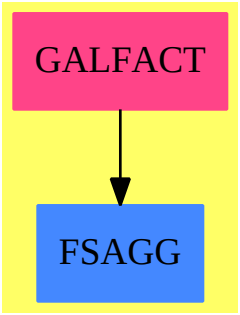
— GALFACTU.dotabb —

```
"GALFACTU" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALFACTU"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GALFACTU" -> "PFECAT"
```

———-

8.2 package GALFACT GaloisGroupFactorizer

8.2.1 GaloisGroupFactorizer (GALFACT)



Exports:

btwFact	degreePartition	eisensteinIrreducible?	factor	f
factorSquareFree	henselFact	makeFR	modularFactor	r
numberOfFactors	stopMusserTrials	tryFunctionalDecomposition	tryFunctionalDecomposition?	u
useEisensteinCriterion?	useSingleFactorBound	useSingleFactorBound?		

— package GALFACT GaloisGroupFactorizer —

```
)abbrev package GALFACT GaloisGroupFactorizer
++ Author: Frederic Lehobey
++ Date Created: 28 June 1994
++ Date Last Updated: 11 July 1997
++ Basic Operations: factor
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: factorization
++ Examples:
++ References:
```

```
++ [1] Bernard Beauzamy, Vilmar Trevisan and Paul S. Wang, Polynomial
++ Factorization: Sharp Bounds, Efficient Algorithms,
++ J. Symbolic Computation (1993) 15, 393-413
++ [2] John Brillhart, Note on Irreducibility Testing,
++ Mathematics of Computation, vol. 35, num. 35, Oct. 1980, 1379-1381
++ [3] David R. Musser, On the Efficiency of a Polynomial Irreducibility Test,
++ Journal of the ACM, Vol. 25, No. 2, April 1978, pp. 271-282
++ Description:
++ \spadtype{GaloisGroupFactorizer} provides functions to factor resolvents.
-- improvements to do :
--   + reformulate the lifting problem in completeFactor -- See [1] (hard)
--   + implement algorithm RC -- See [1] (easy)
--   + use Dedekind's criterion to prove sometimes irreducibility (easy)
--     or even to improve early detection of true factors (hard)
--   + replace Sets by Bits

GaloisGroupFactorizer(UP): Exports == Implementation where
  Z ==> Integer
  UP: UnivariatePolynomialCategory Z
  N ==> NonNegativeInteger
  P ==> PositiveInteger
  CYC ==> CyclotomicPolynomialPackage()
  SUPZ ==> SparseUnivariatePolynomial Z

  ParFact ==> Record(irr: UP, pow: Z)
  FinalFact ==> Record(contp: Z, factors: List ParFact)
  DDRecord ==> Record(factor: UP, degree: Z) -- a Distinct-Degree factor
  DDList ==> List DDRecord
  MFact ==> Record(prime: Z,factors: List UP) -- Modular Factors
  LR ==> Record(left: UP, right: UP) -- Functional decomposition

  Exports ==> with
    makeFR: FinalFact -> Factored UP
      ++ makeFR(flist) turns the final factorization of henselFact into a
      ++ \spadtype{Factored} object.
    degreePartition: DDList -> Multiset N
      ++ degreePartition(ddfactorization) returns the degree partition of
      ++ the polynomial f modulo p where ddfactorization is the distinct
      ++ degree factorization of f computed by ddFact for some prime p.
    musserTrials: () -> P
      ++ musserTrials() returns the number of primes that are tried in
      ++ \spadfun{modularFactor}.
    musserTrials: P -> P
      ++ musserTrials(n) sets to n the number of primes to be tried in
      ++ \spadfun{modularFactor} and returns the previous value.
    stopMusserTrials: () -> P
      ++ stopMusserTrials() returns the bound on the number of factors for
      ++ which \spadfun{modularFactor} stops to look for an other prime. You
      ++ will have to remember that the step of recombining the extraneous
      ++ factors may take up to \spad{2**stopMusserTrials()} trials.
```

```
stopMusserTrials: P -> P
  ++ stopMusserTrials(n) sets to n the bound on the number of factors for
  ++ which \spadfun{modularFactor} stops to look for an other prime. You
  ++ will have to remember that the step of recombining the extraneous
  ++ factors may take up to \spad{2**n} trials. Returns the previous
  ++ value.
numberOfFactors: DDList -> N
  ++ numberOfFactors(ddfactorization) returns the number of factors of
  ++ the polynomial f modulo p where ddfactorization is the distinct
  ++ degree factorization of f computed by ddFact for some prime p.
modularFactor: UP -> MFact
  ++ modularFactor(f) chooses a "good" prime and returns the factorization
  ++ of f modulo this prime in a form that may be used by
  ++ completeHensel. If prime is zero
  ++ it means that f has been proved to be irreducible over the integers
  ++ or that f is a unit (i.e. 1 or -1).
  ++ f shall be primitive (i.e. content(p)=1) and square free (i.e.
  ++ without repeated factors).
useSingleFactorBound?: () -> Boolean
  ++ useSingleFactorBound?() returns \spad{true} if algorithm with single
  ++ factor bound is used for factorization, \spad{false} for algorithm
  ++ with overall bound.
useSingleFactorBound: Boolean -> Boolean
  ++ useSingleFactorBound(b) chooses the algorithm to be used by the
  ++ factorizers: \spad{true} for algorithm with single
  ++ factor bound, \spad{false} for algorithm with overall bound.
  ++ Returns the previous value.
useEisensteinCriterion?: () -> Boolean
  ++ useEisensteinCriterion?() returns \spad{true} if factorizers
  ++ check Eisenstein's criterion before factoring.
useEisensteinCriterion: Boolean -> Boolean
  ++ useEisensteinCriterion(b) chooses whether factorizers check
  ++ Eisenstein's criterion before factoring: \spad{true} for
  ++ using it, \spad{false} else. Returns the previous value.
eisensteinIrreducible?: UP -> Boolean
  ++ eisensteinIrreducible?(p) returns \spad{true} if p can be
  ++ shown to be irreducible by Eisenstein's criterion,
  ++ \spad{false} is inconclusive.
tryFunctionalDecomposition?: () -> Boolean
  ++ tryFunctionalDecomposition?() returns \spad{true} if
  ++ factorizers try functional decomposition of polynomials before
  ++ factoring them.
tryFunctionalDecomposition: Boolean -> Boolean
  ++ tryFunctionalDecomposition(b) chooses whether factorizers have
  ++ to look for functional decomposition of polynomials
  ++ (\spad{true}) or not (\spad{false}). Returns the previous value.
factor: UP -> Factored UP
  ++ factor(p) returns the factorization of p over the integers.
factor: (UP,N) -> Factored UP
  ++ factor(p,r) factorizes the polynomial p using the single factor bound
```

```
  ++ algorithm and knowing that p has at least r factors.
factor: (UP,List N) -> Factored UP
  ++ factor(p,listOfDegrees) factorizes the polynomial p using the single
  ++ factor bound algorithm and knowing that p has for possible
  ++ splitting of its degree listOfDegrees.
factor: (UP,List N,N) -> Factored UP
  ++ factor(p,listOfDegrees,r) factorizes the polynomial p using the single
  ++ factor bound algorithm, knowing that p has for possible
  ++ splitting of its degree listOfDegrees and that p has at least r
  ++ factors.
factor: (UP,N,N) -> Factored UP
  ++ factor(p,d,r) factorizes the polynomial p using the single
  ++ factor bound algorithm, knowing that d divides the degree of all
  ++ factors of p and that p has at least r factors.
factorSquareFree: UP -> Factored UP
  ++ factorSquareFree(p) returns the factorization of p which is supposed
  ++ not having any repeated factor (this is not checked).
factorSquareFree: (UP,N) -> Factored UP
  ++ factorSquareFree(p,r) factorizes the polynomial p using the single
  ++ factor bound algorithm and knowing that p has at least r factors.
  ++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N) -> Factored UP
  ++ factorSquareFree(p,listOfDegrees) factorizes the polynomial p using
  ++ the single factor bound algorithm and knowing that p has for possible
  ++ splitting of its degree listOfDegrees.
  ++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N,N) -> Factored UP
  ++ factorSquareFree(p,listOfDegrees,r) factorizes the polynomial p using
  ++ the single factor bound algorithm, knowing that p has for possible
  ++ splitting of its degree listOfDegrees and that p has at least r
  ++ factors.
  ++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,N,N) -> Factored UP
  ++ factorSquareFree(p,d,r) factorizes the polynomial p using the single
  ++ factor bound algorithm, knowing that d divides the degree of all
  ++ factors of p and that p has at least r factors.
  ++ f is supposed not having any repeated factor (this is not checked).
factorOfDegree: (P,UP) -> Union(UP,"failed")
  ++ factorOfDegree(d,p) returns a factor of p of degree d.
factorOfDegree: (P,UP,N) -> Union(UP,"failed")
  ++ factorOfDegree(d,p,r) returns a factor of p of degree
  ++ d knowing that p has at least r factors.
factorOfDegree: (P,UP,List N) -> Union(UP,"failed")
  ++ factorOfDegree(d,p,listOfDegrees) returns a factor
  ++ of p of degree d knowing that p has for possible splitting of its
  ++ degree listOfDegrees.
factorOfDegree: (P,UP,List N,N) -> Union(UP,"failed")
  ++ factorOfDegree(d,p,listOfDegrees,r) returns a factor
  ++ of p of degree d knowing that p has for possible splitting of its
  ++ degree listOfDegrees, and that p has at least r factors.
```

```
  factorOfDegree: (P,UP,List N,N,Boolean) -> Union(UP,"failed")
    ++ factorOfDegree(d,p,listOfDegrees,r,sqf) returns a
    ++ factor of p of degree d knowing that p has for possible splitting of
    ++ its degree listOfDegrees, and that p has at least r factors.
    ++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
    ++ without repeated factors).
  henselFact: (UP,Boolean) -> FinalFact
    ++ henselFact(p,sqf) returns the factorization of p, the result
    ++ is a Record such that \spad{contp=}content p,
    ++ \spad{factors=}List of irreducible factors of p with exponent.
    ++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
    ++ without repeated factors).
  btwFact: (UP,Boolean,Set N,N) -> FinalFact
    ++ btwFact(p,sqf,pd,r) returns the factorization of p, the result
    ++ is a Record such that \spad{contp=}content p,
    ++ \spad{factors=}List of irreducible factors of p with exponent.
    ++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
    ++ without repeated factors).
    ++ pd is the \spadtype{Set} of possible degrees. r is a lower bound for
    ++ the number of factors of p. Please do not use this function in your
    ++ code because its design may change.

Implementation ==> add

  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FFE ==> Record(flg:fUnion, fctr:UP, xpnt:Z) -- Flag-Factor-Exponent
  DDFact ==> Record(prime:Z, ddfactors:DDList) -- Distinct Degree Factors
  HLR ==> Record(plist:List UP, modulo:Z) -- HenselLift Record

  mussertrials: P := 5
  stopmussertrials: P := 8
  usesinglefactorbound: Boolean := true
  tryfunctionaldecomposition: Boolean := true
  useeisensteincriterion: Boolean := true

  useEisensteinCriterion?():Boolean == useeisensteincriterion

  useEisensteinCriterion(b:Boolean):Boolean ==
    (useeisensteincriterion,b) := (b,useeisensteincriterion)
    b

  tryFunctionalDecomposition?():Boolean == tryfunctionaldecomposition

  tryFunctionalDecomposition(b:Boolean):Boolean ==
    (tryfunctionaldecomposition,b) := (b,tryfunctionaldecomposition)
    b

  useSingleFactorBound?():Boolean == usesinglefactorbound

  useSingleFactorBound(b:Boolean):Boolean ==
```

```
      (usesinglefactorbound,b) := (b,usesinglefactorbound)
      b

    stopMusserTrials():P == stopmussertrials

    stopMusserTrials(n:P):P ==
      (stopmussertrials,n) := (n,stopmussertrials)
      n

    musserTrials():P == mussertrials

    musserTrials(n:P):P ==
      (mussertrials,n) := (n,mussertrials)
      n

    import GaloisGroupFactorizationUtilities(Z,UP,Float)

    import GaloisGroupPolynomialUtilities(Z,UP)

    import IntegerPrimesPackage(Z)
    import IntegerFactorizationPackage(Z)

    import ModularDistinctDegreeFactorizer(UP)

    eisensteinIrreducible?(f:UP):Boolean ==
      rf := reductum f
      c: Z := content rf
      zero? c => false
      unit? c => false
      lc := leadingCoefficient f
      tc := lc
      while not zero? rf repeat
        tc := leadingCoefficient rf
        rf := reductum rf
      for p in factors(factor c)$Factored(Z) repeat
--        if (one? p.exponent) and (not zero? (lc rem p.factor)) and
        if (p.exponent = 1) and (not zero? (lc rem p.factor)) and
         (not zero? (tc rem ((p.factor)**2))) then return true
      false

    numberOfFactors(ddlist:DDList):N ==
      n: N := 0
      d: Z := 0
      for dd in ddlist repeat
        n := n +
          zero? (d := degree(dd.factor)::Z) => 1
          (d quo dd.degree)::N
      n

    -- local function, returns the a Set of shifted elements
```

```
    shiftSet(s:Set N,shift:N):Set N == set [ e+shift for e in parts s ]

    -- local function, returns the "reductum" of an Integer (as chain of bits)
    reductum(n:Z):Z == n-shift(1,length(n)-1)

    -- local function, returns an integer with level lowest bits set to 1
    seed(level:Z):Z == shift(1,level)-1

    -- local function, returns the next number (as a chain of bit) for
    -- factor reconciliation of a given level (which is the number of
    -- extraneaous factors involved) or "End of level" if not any
    nextRecNum(levels:N,level:Z,n:Z):Union("End of level",Z) ==
      if (l := length n)<levels then return(n+shift(1,l-1))
      (n=shift(seed(level),levels-level)) => "End of level"
      b: Z := 1
      while ((l-b) = (lr := length(n := reductum n)))@Boolean repeat b := b+1
      reductum(n)+shift(seed(b+1),lr)

    -- local function, return the set of N, 0..n
    fullSet(n:N):Set N == set [ i for i in 0..n ]

    modularFactor(p:UP):MFact ==
--      not one? abs(content(p)) =>
      not (abs(content(p)) = 1) =>
       error "modularFactor: the polynomial is not primitive."
      zero? (n := degree p) => [0,[p]]

      -- declarations --
      cprime: Z := 2
      trials: List DDFact := empty()
      d: Set N := fullSet(n)
      dirred: Set N := set [0,n]
      s: Set N := empty()
      ddlist: DDList := empty()
      degfact: N := 0
      nf: N := stopmussertrials+1
      i: Z

      -- Musser, see [3] --
      diffp := differentiate p
      for i in 1..mussertrials | nf>stopmussertrials repeat
        -- test 1: cprime divides leading coefficient
        -- test 2: "bad" primes: (in future: use Dedekind's Criterion)
        while (zero? ((leadingCoefficient p) rem cprime)) or
         (not zero? degree gcd(p,diffp,cprime)) repeat
          cprime := nextPrime(cprime)
        ddlist := ddFact(p,cprime)
        -- degree compatibility: See [3] --
        s := set [0]
        for f in ddlist repeat
```

```
      degfact := f.degree::N
      if not zero? degfact then
        for j in 1..(degree(f.factor) quo degfact) repeat
          s := union(s, shiftSet(s,degfact))
    trials := cons([cprime,ddlist]$DDFact,trials)
    d := intersect(d, s)
    d = dirred => return [0,[p]] -- p is irreducible
    cprime := nextPrime(cprime)
    nf := numberOfFactors ddlist

  -- choose the one with the smallest number of factors
  choice := first trials
  nfc := numberOfFactors(choice.ddfactors)
  for t in rest trials repeat
    nf := numberOfFactors(t.ddfactors)
    if nf<nfc or ((nf=nfc) and (t.prime>choice.prime)) then
      nfc := nf
      choice := t
  cprime := choice.prime
  -- HenselLift$GHENSEL expects the degree 0 factor first
  [cprime,separateFactors(choice.ddfactors,cprime)]

degreePartition(ddlist:DDList):Multiset N ==
  dp: Multiset N := empty()
  d: N := 0
  dd: N := 0
  for f in ddlist repeat
    zero? (d := degree(f.factor)) => dp := insert!(0,dp)
    dd := f.degree::N
    dp := insert!(dd,dp,d quo dd)
  dp

import GeneralHenselPackage(Z,UP)
import UnivariatePolynomialDecompositionPackage(Z,UP)
import BrillhartTests(UP) -- See [2]

-- local function, finds the factors of f primitive, square-free, with
-- positive leading coefficient and non zero trailing coefficient,
-- using the overall bound technique. If pdecomp is true then look
-- for a functional decomposition of f.
henselfact(f:UP,pdecomp:Boolean):List UP ==
  if brillhartIrreducible? f or
   (useeisensteincriterion => eisensteinIrreducible? f ; false)
  then return [f]
  cf: Union(LR,"failed")
  if pdecomp and tryfunctionaldecomposition then
    cf := monicDecomposeIfCan f
  else
    cf := "failed"
  cf case "failed" =>
```

```
    m := modularFactor f
    zero? (cprime := m.prime) => m.factors
    b: P := (2*leadingCoefficient(f)*beauzamyBound(f)) :: P
    completeHensel(f,m.factors,cprime,b)
  lrf := cf::LR
  "append"/[ henselfact(g(lrf.right),false) for g in
   henselfact(lrf.left,true) ]

-- local function, returns the complete factorization of its arguments,
-- using the single-factor bound technique
completeFactor(f:UP,lf:List UP,cprime:Z,pk:P,r:N,d:Set N):List UP ==
  lc := leadingCoefficient f
  f0 := coefficient(f,0)
  ltrue: List UP := empty()
  found? := true
  degf: N := 0
  degg: N := 0
  g0: Z := 0
  g: UP := 0
  rg: N := 0
  nb: Z := 0
  lg: List UP := empty()
  b: P := 1
  dg: Set N := empty()
  llg: HLR := [empty(),0]
  levels: N := #lf
  level: Z := 1
  ic: Union(Z,"End of level") := 0
  i: Z := 0
  while level<levels repeat
    -- try all possible factors with degree in d
    ic := seed(level)
    while ((not found?) and (ic case Z)) repeat
      i := ic::Z
      degg := 0
      g0 := 1 -- LC algorithm
      for j in 1..levels repeat
        if bit?(i,j-1) then
          degg := degg+degree lf.j
          g0 := g0*coefficient(lf.j,0) -- LC algorithm
      g0 := symmetricRemainder(lc*g0,pk) -- LC algorithm
      if member?(degg,d) and (((lc*f0) exquo g0) case Z) then
        --                        LC algorithm
        g := lc::UP -- build the possible factor -- LC algorithm
        for j in 1..levels repeat if bit?(i,j-1) then g := g*lf.j
        g := primitivePart reduction(g,pk)
        f1 := f exquo g
        if f1 case UP then -- g is a true factor
          found? := true
          -- remove the factors of g from lf
```

```
                nb := 1
                for j in 1..levels repeat
                  if bit?(i,j-1) then
                    swap!(lf,j,nb)
                    nb := nb+1
                lg := lf
                lf := rest(lf,level::N)
                setrest!(rest(lg,(level-1)::N),empty()$List(UP))
                f := f1::UP
                lc := leadingCoefficient f
                f0 := coefficient(f,0)
                -- is g irreducible?
                dg := select(x+->x <= degg,d)
                if not(dg=set [0,degg]) then -- implies degg >= 2
                  rg := max(2,r+level-levels)::N
                  b := (2*leadingCoefficient(g)*singleFactorBound(g,rg)) :: P
                  if b>pk and (not brillhartIrreducible?(g)) and
                    (useeisensteincriterion => not eisensteinIrreducible?(g) ;
                    true)
                  then
                    -- g may be reducible
                    llg := HenselLift(g,lg,cprime,b)
                    gpk: P := (llg.modulo)::P
                    -- In case exact factorisation has been reached by
                    -- HenselLift before coefficient bound.
                    if gpk<b then
                      lg := llg.plist
                    else
                      lg := completeFactor(g,llg.plist,cprime,gpk,rg,dg)
                  else lg := [ g ] -- g irreducible
                else lg := [ g ] -- g irreducible
                ltrue := append(ltrue,lg)
                r := max(2,(r-#lg))::N
                degf := degree f
                d := select(x+->x <= degf,d)
                if degf<=1 then -- lf exhausted
--                  if one? degf then
                  if (degf = 1) then
                    ltrue := cons(f,ltrue)
                  return ltrue -- 1st exit, all factors found
                else -- can we go on with the same pk?
                  b := (2*lc*singleFactorBound(f,r)) :: P
                  if b>pk then -- unlucky: no we can't
                    llg := HenselLift(f,lf,cprime,b) -- I should reformulate
                     -- the lifting probleme, but hadn't time for that.
                     -- In any case, such case should be quite exceptional.
                    lf := llg.plist
                    pk := (llg.modulo)::P
                    -- In case exact factorisation has been reached by
                    -- HenselLift before coefficient bound.
```

```
              if pk<b then return append(lf,ltrue) -- 2nd exit
              level := 1
        ic := nextRecNum(levels,level,i)
      if found? then
        levels := #lf
        found? := false
      if not (ic case Z) then level := level+1
    cons(f,ltrue) -- 3rd exit, the last factor was irreducible but not "true"

  -- local function, returns the set of elements "divided" by an integer
  divideSet(s:Set N, n:N):Set N ==
    l: List N := [ 0 ]
    for e in parts s repeat
      if (ee := (e exquo n)$N) case N then l := cons(ee::N,l)
    set(l)

  -- Beauzamy-Trevisan-Wang FACTOR, see [1] with some refinements
  -- and some differences. f is assumed to be primitive, square-free
  -- and with positive leading coefficient. If pdecomp is true then
  -- look for a functional decomposition of f.
  btwFactor(f:UP,d:Set N,r:N,pdecomp:Boolean):List UP ==
    df := degree f
    not (max(d) = df) => error "btwFact: Bad arguments"
    reverse?: Boolean := false
    negativelc?: Boolean := false

    (d = set [0,df]) => [ f ]
    if abs(coefficient(f,0))<abs(leadingCoefficient(f)) then
      f := reverse f
      reverse? := true
    brillhartIrreducible? f or
     (useeisensteincriterion => eisensteinIrreducible?(f) ; false) =>
      if reverse? then [ reverse f ] else [ f ]
    if leadingCoefficient(f)<0 then
      f := -f
      negativelc? := true
    cf: Union(LR,"failed")
    if pdecomp and tryfunctionaldecomposition then
      cf := monicDecomposeIfCan f
    else
      cf := "failed"
    if cf case "failed" then
      m := modularFactor f
      zero? (cprime := m.prime) =>
        if reverse? then
          if negativelc? then return [ -reverse f ]
          else return [ reverse f ]
        else if negativelc? then return [ -f ]
             else return [ f ]
      if noLinearFactor? f then d := remove(1,d)
```

```
      lc := leadingCoefficient f
      f0 := coefficient(f,0)
      b: P := (2*lc*singleFactorBound(f,r)) :: P -- LC algorithm
      lm := HenselLift(f,m.factors,cprime,b)
      lf := lm.plist
      pk: P := (lm.modulo)::P
      if ground? first lf then lf := rest lf
      -- in case exact factorisation has been reached by HenselLift
      -- before coefficient bound
      if not pk < b then lf := completeFactor(f,lf,cprime,pk,r,d)
    else
      lrf := cf::LR
      dh := degree lrf.right
      lg := btwFactor(lrf.left,divideSet(d,dh),2,true)
      lf: List UP := empty()
      for i in 1..#lg repeat
        g := lg.i
        dgh := (degree g)*dh
        df := subtractIfCan(df,dgh)::N
        lfg := btwFactor(g(lrf.right),
         select(x+->x <= dgh,d),max(2,r-df)::N,false)
        lf := append(lf,lfg)
        r := max(2,r-#lfg)::N
    if reverse? then lf := [ reverse(fact) for fact in lf ]
    for i in 1..#lf repeat
      if leadingCoefficient(lf.i)<0 then lf.i := -lf.i
      -- because we assume f with positive leading coefficient
    lf

  makeFR(flist:FinalFact):Factored UP ==
    ctp := factor flist.contp
    fflist: List FFE := empty()
    for ff in flist.factors repeat
      fflist := cons(["prime", ff.irr, ff.pow]$FFE, fflist)
    for fc in factorList ctp repeat
      fflist := cons([fc.flg, fc.fctr::UP, fc.xpnt]$FFE, fflist)
    makeFR(unit(ctp)::UP, fflist)

  import IntegerRoots(Z)

  -- local function, factorizes a quadratic polynomial
  quadratic(p:UP):List UP ==
    a := leadingCoefficient p
    b := coefficient(p,1)
    d := b**2-4*a*coefficient(p,0)
    r := perfectSqrt(d)
    r case "failed" => [p]
    b := b+(r::Z)
    a := 2*a
    d := gcd(a,b)
```

```
--      if not one? d then
      if not (d = 1) then
        a := a quo d
        b := b quo d
      f: UP := monomial(a,1)+monomial(b,0)
      cons(f,[(p exquo f)::UP])

    isPowerOf2(n:Z): Boolean ==
       n = 1 => true
       qr: Record(quotient: Z, remainder: Z) := divide(n,2)
       qr.remainder = 1 => false
       isPowerOf2 qr.quotient

    subMinusX(supPol: SUPZ): UP ==
       minusX: SUPZ := monomial(-1,1)$SUPZ
       unmakeSUP(elt(supPol,minusX)$SUPZ)

    henselFact(f:UP, sqf:Boolean):FinalFact ==
      factorlist: List(ParFact) := empty()

      -- make m primitive
      c: Z := content f
      f := (f exquo c)::UP

      -- make the leading coefficient positive
      if leadingCoefficient f < 0 then
        c := -c
        f := -f

      -- is x**d factor of f
      if (d := minimumDegree f) > 0 then
        f := monicDivide(f,monomial(1,d)).quotient
        factorlist := [[monomial(1,1),d]$ParFact]

      d := degree f

      -- is f constant?
      zero? d => [c,factorlist]$FinalFact

      -- is f linear?
--      one? d => [c,cons([f,1]$ParFact,factorlist)]$FinalFact
      (d = 1) => [c,cons([f,1]$ParFact,factorlist)]$FinalFact

      lcPol: UP := leadingCoefficient(f) :: UP

      -- is f cyclotomic (x**n - 1)?
      -lcPol = reductum(f) =>    -- if true, both will = 1
        for fac in map(z+->unmakeSUP(z)$UP,
         cyclotomicDecomposition(d)$CYC)$ListFunctions2(SUPZ,UP) repeat
          factorlist := cons([fac,1]$ParFact,factorlist)
```

```
        [c,factorlist]$FinalFact

      -- is f odd cyclotomic (x**(2*n+1) + 1)?
      odd?(d) and (lcPol = reductum(f)) =>
        for sfac in cyclotomicDecomposition(d)$CYC repeat
           fac := subMinusX sfac
           if leadingCoefficient fac < 0 then fac := -fac
           factorlist := cons([fac,1]$ParFact,factorlist)
        [c,factorlist]$FinalFact

      -- is the poly of the form x**n + 1 with n a power of 2?
      -- if so, then irreducible
      isPowerOf2(d) and (lcPol = reductum(f)) =>
        factorlist := cons([f,1]$ParFact,factorlist)
        [c,factorlist]$FinalFact

      -- other special cases to implement...

      -- f is square-free :
      sqf => [c, append([[pf,1]$ParFact for pf in henselfact(f,true)],
       factorlist)]$FinalFact

      -- f is not square-free :
      sqfflist := factors squareFree f
      for sqfr in sqfflist repeat
        mult := sqfr.exponent
        sqff := sqfr.factor
        d := degree sqff
--        one? d => factorlist := cons([sqff,mult]$ParFact,factorlist)
        (d = 1) => factorlist := cons([sqff,mult]$ParFact,factorlist)
        d=2 =>
          factorlist := append([[pf,mult]$ParFact for pf in quadratic(sqff)],
           factorlist)
        factorlist := append([[pf,mult]$ParFact for pf in
         henselfact(sqff,true)],factorlist)
      [c,factorlist]$FinalFact

    btwFact(f:UP, sqf:Boolean, fd:Set N, r:N):FinalFact ==
      d := degree f
      not(max(fd)=d) => error "btwFact: Bad arguments"
      factorlist: List(ParFact) := empty()

      -- make m primitive
      c: Z := content f
      f := (f exquo c)::UP

      -- make the leading coefficient positive
      if leadingCoefficient f < 0 then
        c := -c
        f := -f
```

```
      -- is x**d factor of f
      if (maxd := minimumDegree f) > 0 then
        f := monicDivide(f,monomial(1,maxd)).quotient
        factorlist := [[monomial(1,1),maxd]$ParFact]
        r := max(2,r-maxd)::N
        d := subtractIfCan(d,maxd)::N
        fd := select(x+->x <= d,fd)

      -- is f constant?
      zero? d => [c,factorlist]$FinalFact

      -- is f linear?
--      one? d => [c,cons([f,1]$ParFact,factorlist)]$FinalFact
      (d = 1) => [c,cons([f,1]$ParFact,factorlist)]$FinalFact

      lcPol: UP := leadingCoefficient(f) :: UP

      -- is f cyclotomic (x**n - 1)?
      -lcPol = reductum(f) =>    -- if true, both will = 1
        for fac in map(z+->unmakeSUP(z)$UP,
         cyclotomicDecomposition(d)$CYC)$ListFunctions2(SUPZ,UP) repeat
          factorlist := cons([fac,1]$ParFact,factorlist)
        [c,factorlist]$FinalFact

      -- is f odd cyclotomic (x**(2*n+1) + 1)?
      odd?(d) and (lcPol = reductum(f)) =>
        for sfac in cyclotomicDecomposition(d)$CYC repeat
           fac := subMinusX sfac
           if leadingCoefficient fac < 0 then fac := -fac
           factorlist := cons([fac,1]$ParFact,factorlist)
        [c,factorlist]$FinalFact

      -- is the poly of the form x**n + 1 with n a power of 2?
      -- if so, then irreducible
      isPowerOf2(d) and (lcPol = reductum(f)) =>
        factorlist := cons([f,1]$ParFact,factorlist)
        [c,factorlist]$FinalFact

      -- other special cases to implement...

      -- f is square-free :
      sqf => [c, append([[pf,1]$ParFact for pf in btwFactor(f,fd,r,true)],
       factorlist)]$FinalFact

      -- f is not square-free :
      sqfflist := factors squareFree(f)
--      if one?(#(sqfflist)) then -- indeed f was a power of a square-free
      if ((#(sqfflist)) = 1) then -- indeed f was a power of a square-free
        r := max(r quo ((first sqfflist).exponent),2)::N
```

```
      else
        r := 2
      for sqfr in sqfflist repeat
        mult := sqfr.exponent
        sqff := sqfr.factor
        d := degree sqff
--        one? d =>
        (d = 1) =>
          factorlist := cons([sqff,mult]$ParFact,factorlist)
          maxd := (max(fd)-mult)::N
          fd := select(x+->x <= maxd,fd)
        d=2 =>
          factorlist := append([[pf,mult]$ParFact for pf in quadratic(sqff)],
           factorlist)
          maxd := (max(fd)-2*mult)::N
          fd := select(x+->x <= maxd,fd)
        factorlist := append([[pf,mult]$ParFact for pf in
         btwFactor(sqff,select(x+->x <= d,fd),r,true)],factorlist)
        maxd := (max(fd)-d*mult)::N
        fd := select(x+->x <= maxd,fd)
      [c,factorlist]$FinalFact

    factor(f:UP):Factored UP ==
      makeFR
        usesinglefactorbound => btwFact(f,false,fullSet(degree f),2)
        henselFact(f,false)

    -- local function, returns true if the sum of the elements of the list
    -- is not the degree.
    errorsum?(d:N,ld:List N):Boolean == not (d = +/ld)

    -- local function, turns list of degrees into a Set
    makeSet(ld:List N):Set N ==
      s := set [0]
      for d in ld repeat s := union(s,shiftSet(s,d))
      s

    factor(f:UP,ld:List N,r:N):Factored UP ==
      errorsum?(degree f,ld) => error "factor: Bad arguments"
      makeFR btwFact(f,false,makeSet(ld),r)

    factor(f:UP,r:N):Factored UP == makeFR btwFact(f,false,fullSet(degree f),r)

    factor(f:UP,ld:List N):Factored UP == factor(f,ld,2)

    factor(f:UP,d:N,r:N):Factored UP ==
      n := (degree f) exquo d
      n case "failed" => error "factor: Bad arguments"
      factor(f,new(n::N,d)$List(N),r)
```

```
    factorSquareFree(f:UP):Factored UP ==
      makeFR
        usesinglefactorbound => btwFact(f,true,fullSet(degree f),2)
        henselFact(f,true)

    factorSquareFree(f:UP,ld:List(N),r:N):Factored UP ==
      errorsum?(degree f,ld) => error "factorSquareFree: Bad arguments"
      makeFR btwFact(f,true,makeSet(ld),r)

    factorSquareFree(f:UP,r:N):Factored UP ==
      makeFR btwFact(f,true,fullSet(degree f),r)

    factorSquareFree(f:UP,ld:List N):Factored UP == factorSquareFree(f,ld,2)

    factorSquareFree(f:UP,d:N,r:N):Factored UP ==
      n := (degree f) exquo d
      n case "failed" => error "factorSquareFree: Bad arguments"
      factorSquareFree(f,new(n::N,d)$List(N),r)

    factorOfDegree(d:P,p:UP,ld:List N,r:N,sqf:Boolean):Union(UP,"failed") ==
      dp := degree p
      errorsum?(dp,ld) => error "factorOfDegree: Bad arguments"
--      (one? (d::N)) and noLinearFactor?(p) => "failed"
      ((d::N) = 1) and noLinearFactor?(p) => "failed"
      lf := btwFact(p,sqf,makeSet(ld),r).factors
      for f in lf repeat
        degree(f.irr)=d => return f.irr
      "failed"

    factorOfDegree(d:P,p:UP,ld:List N,r:N):Union(UP,"failed") ==
      factorOfDegree(d,p,ld,r,false)

    factorOfDegree(d:P,p:UP,r:N):Union(UP,"failed") ==
      factorOfDegree(d,p,new(degree p,1)$List(N),r,false)

    factorOfDegree(d:P,p:UP,ld:List N):Union(UP,"failed") ==
      factorOfDegree(d,p,ld,2,false)

    factorOfDegree(d:P,p:UP):Union(UP,"failed") ==
      factorOfDegree(d,p,new(degree p,1)$List(N),2,false)
```

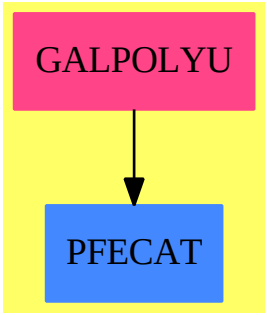
———-

— GALFACT.dotabb —

```
"GALFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALFACT"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"GALFACT" -> "FSAGG"
```

8.3 package GALPOLYU GaloisGroupPolynomialUtilities

8.3.1 GaloisGroupPolynomialUtilities (GALPOLYU)



Exports:

degreePartition	factorOfDegree	factorsOfDegree	monic?	reverse
scaleRoots	shiftRoots	unvectorise		

— package GALPOLYU GaloisGroupPolynomialUtilities —

```
)abbrev package GALPOLYU GaloisGroupPolynomialUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 15 July 1994
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ \spadtype{GaloisGroupPolynomialUtilities} provides useful
++ functions for univariate polynomials which should be added to
++ \spadtype{UnivariatePolynomialCategory} or to \spadtype{Factored}

GaloisGroupPolynomialUtilities(R,UP): Exports == Implementation where
  R : Ring
  UP : UnivariatePolynomialCategory R
```

```
  N ==> NonNegativeInteger
  P ==> PositiveInteger

  Exports ==> with
    monic?: UP -> Boolean
      ++ monic?(p) tests if p is monic (i.e. leading coefficient equal to 1).
    unvectorise: Vector R -> UP
      ++ unvectorise(v) returns the polynomial which has for coefficients the
      ++ entries of v in the increasing order.
    reverse: UP -> UP
      ++ reverse(p) returns the reverse polynomial of p.
    scaleRoots: (UP,R) -> UP
      ++ scaleRoots(p,c) returns the polynomial which has c times the roots
      ++ of p.
    shiftRoots: (UP,R) -> UP
      ++ shiftRoots(p,c) returns the polynomial which has for roots c added
      ++ to the roots of p.
    degreePartition: Factored UP -> Multiset N
      ++ degreePartition(f) returns the degree partition (i.e. the multiset
      ++ of the degrees of the irreducible factors) of
      ++ the polynomial f.
    factorOfDegree: (P, Factored UP) -> UP
      ++ factorOfDegree(d,f) returns a factor of degree d of the factored
      ++ polynomial f. Such a factor shall exist.
    factorsOfDegree: (P, Factored UP) -> List UP
      ++ factorsOfDegree(d,f) returns the factors of degree d of the factored
      ++ polynomial f.

  Implementation ==> add

    import Factored UP

    factorsOfDegree(d:P,r:Factored UP):List UP ==
      lfact : List UP := empty()
      for fr in factors r | degree(fr.factor)=(d::N) repeat
        for i in 1..fr.exponent repeat
          lfact := cons(fr.factor,lfact)
      lfact

    factorOfDegree(d:P,r:Factored UP):UP ==
      factor : UP := 0
      for i in 1..numberOfFactors r repeat
        factor := nthFactor(r,i)
        if degree(factor)=(d::N) then return factor
      error "factorOfDegree: Bad arguments"

    degreePartition(r:Factored UP):Multiset N ==
      multiset([ degree(nthFactor(r,i)) for i in 1..numberOfFactors r ])

--    monic?(p:UP):Boolean == one? leadingCoefficient p
```

```
    monic?(p:UP):Boolean == (leadingCoefficient p) = 1

    unvectorise(v:Vector R):UP ==
      p : UP := 0
      for i in 1..#v repeat p := p + monomial(v(i),(i-1)::N)
      p

    reverse(p:UP):UP ==
      r : UP := 0
      n := degree(p)
      for i in 0..n repeat r := r + monomial(coefficient(p,(n-i)::N),i)
      r

    scaleRoots(p:UP,c:R):UP ==
--      one? c => p
      (c = 1) => p
      n := degree p
      zero? c => monomial(leadingCoefficient p,n)
      r : UP := 0
      mc : R := 1
      for i in n..0 by -1 repeat
        r := r + monomial(mc*coefficient(p,i),i)
        mc := mc*c
      r

    import UnivariatePolynomialCategoryFunctions2(R,UP,UP,
     SparseUnivariatePolynomial UP)

    shiftRoots(p:UP,c:R):UP == elt(map(coerce,p),monomial(1,1)$UP-c::UP)::UP
```

———-

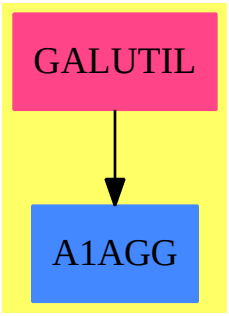
— GALPOLYU.dotabb —

```
"GALPOLYU" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALPOLYU"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GALPOLYU" -> "PFECAT"
```

———-

8.4 package GALUTIL GaloisGroupUtilities

8.4.1 GaloisGroupUtilities (GALUTIL)



Exports:

fillPascalTriangle	pascalTriangle	rangePascalTriangle	safeCeiling	safeFloor
safetyMargin	sizePascalTriangle			

— package GALUTIL GaloisGroupUtilities —

```
)abbrev package GALUTIL GaloisGroupUtilities
++ Author: Frederic Lehobey
++ Date Created: 29 June 1994
++ Date Last Updated: 30 June 1994
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{GaloisGroupUtilities} provides several useful functions.

GaloisGroupUtilities(R): Exports == Implementation where
  N ==> NonNegativeInteger
  Z ==> Integer
  R : Ring

  Exports ==> with
    pascalTriangle: (N,Z) -> R
      ++ pascalTriangle(n,r) returns the binomial coefficient
      ++ \spad{C(n,r)=n!/(r! (n-r)!)}
      ++ and stores it in a table to prevent recomputation.
    rangePascalTriangle: N -> N
      ++ rangePascalTriangle(n) sets the maximal number of lines which
      ++ are stored and returns the previous value.
    rangePascalTriangle: () -> N
```

```
      ++ rangePascalTriangle() returns the maximal number of lines stored.
    sizePascalTriangle: () -> N
      ++ sizePascalTriangle() returns the number of entries currently stored
      ++ in the table.
    fillPascalTriangle: () -> Void
      ++ fillPascalTriangle() fills the stored table.

    if R has FloatingPointSystem then
      safeCeiling: R -> Z
        ++ safeCeiling(x) returns the integer which is greater than any integer
        ++ with the same floating point number representation.
      safeFloor: R -> Z
        ++ safeFloor(x) returns the integer which is lower or equal to the
        ++ largest integer which has the same floating point number
        ++ representation.
      safetyMargin: N -> N
        ++ safetyMargin(n) sets to n the number of low weight digits we do not
        ++ trust in the floating point representation and returns the previous
        ++ value (for use by \spadfun{safeCeiling}).
      safetyMargin: () -> N
        ++ safetyMargin() returns the number of low weight digits we do not
        ++ trust in the floating point representation (used by
        ++ \spadfun{safeCeiling}).

  Implementation ==> add

    if R has FloatingPointSystem then
      safetymargin : N := 6

      safeFloor(x:R):Z ==
        if (shift := order(x)-precision()$R+safetymargin) >= 0 then
          x := x+float(1,shift)
        retract(floor(x))@Z

      safeCeiling(x:R):Z ==
        if (shift := order(x)-precision()$R+safetymargin) >= 0 then
          x := x+float(1,shift)
        retract(ceiling(x))@Z

      safetyMargin(n:N):N ==
        (safetymargin,n) := (n,safetymargin)
        n

      safetyMargin():N == safetymargin

    pascaltriangle : FlexibleArray(R) := empty()
    ncomputed : N := 3
    rangepascaltriangle : N := 216

    pascalTriangle(n:N, r:Z):R ==
```

```
      negative? r => 0
      (d := n-r) < r => pascalTriangle(n,d)
      zero? r => 1$R
--      one? r => n :: R
      (r = 1) => n :: R
      n > rangepascaltriangle =>
       binomial(n,r)$IntegerCombinatoricFunctions(Z) :: R
      n <= ncomputed =>
        m := divide(n-4,2)
        mq := m.quotient
        pascaltriangle((mq+1)*(mq+m.remainder)+r-1)
      -- compute the missing lines
      for i in (ncomputed+1)..n repeat
        for j in 2..(i quo 2) repeat
          pascaltriangle := concat!(pascaltriangle,pascalTriangle((i-1)
           :: N, j-1)+pascalTriangle((i-1) :: N,j))
        ncomputed := i
      pascalTriangle(n,r)

    rangePascalTriangle(n:N):N ==
      if n<ncomputed then
        if n<3 then
          pascaltriangle := delete!(pascaltriangle,1..#pascaltriangle)
          ncomputed := 3
        else
          d := divide(n-3,2)
          dq := d.quotient
          pascaltriangle := delete!(pascaltriangle,((dq+1)*(dq+d.remainder)
           +1)..#pascaltriangle)
          ncomputed := n
      (rangepascaltriangle,n) := (n,rangepascaltriangle)
      n

    rangePascalTriangle():N == rangepascaltriangle

    sizePascalTriangle():N == #pascaltriangle

    fillPascalTriangle():Void == pascalTriangle(rangepascaltriangle,2)
```

———-

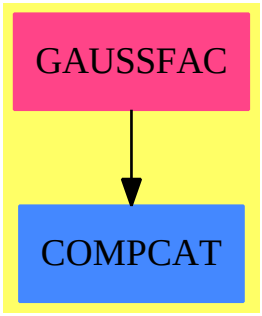
— GALUTIL.dotabb —

```
"GALUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALUTIL"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"GALUTIL" -> "A1AGG"
```

———-

8.5 package GAUSSFAC GaussianFactorizationPackage

8.5.1 GaussianFactorizationPackage (GAUSSFAC)



Exports:
factor prime? sumSquares

— **package GAUSSFAC GaussianFactorizationPackage** —

```
)abbrev package GAUSSFAC GaussianFactorizationPackage
++ Author: Patrizia Gianni
++ Date Created: Summer 1986
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Package for the factorization of complex or gaussian integers.

GaussianFactorizationPackage() : C == T
 where
  NNI  ==  NonNegativeInteger
  Z      ==> Integer
  ZI     ==> Complex Z
  FRZ    ==> Factored ZI
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FFE    ==> Record(flg:fUnion, fctr:ZI, xpnt:Integer)

  C  == with
     factor      :     ZI     ->     FRZ
       ++ factor(zi) produces the complete factorization of the complex
       ++ integer zi.
     sumSquares  :     Z      ->    List Z
```

```
      ++ sumSquares(p) construct \spad{a} and b such that \spad{a**2+b**2}
      ++ is equal to
      ++ the integer prime p, and otherwise returns an error.
      ++ It will succeed if the prime number p is 2 or congruent to 1
      ++ mod 4.
    prime?      :     ZI     ->    Boolean
      ++ prime?(zi) tests if the complex integer zi is prime.

  T  == add
    import IntegerFactorizationPackage Z

    reduction(u:Z,p:Z):Z ==
      p=0 => u
      positiveRemainder(u,p)

    merge(p:Z,q:Z):Union(Z,"failed") ==
      p = q => p
      p = 0 => q
      q = 0 => p
      "failed"

    exactquo(u:Z,v:Z,p:Z):Union(Z,"failed") ==
       p=0 => u exquo v
       v rem p = 0 => "failed"
       positiveRemainder((extendedEuclidean(v,p,u)::Record(coef1:Z,coef2:Z)).coef1,p)

    FMod := ModularRing(Z,Z,reduction,merge,exactquo)

    fact2:ZI:= complex(1,1)

            ----  find the solution of x**2+1 mod q  ----
    findelt(q:Z) : Z ==
      q1:=q-1
      r:=q1
      r1:=r exquo 4
      while ^(r1 case "failed") repeat
        r:=r1::Z
        r1:=r exquo 2
      s : FMod := reduce(1,q)
      qq1:FMod :=reduce(q1,q)
      for i in 2.. while (s=1 or s=qq1) repeat
        s:=reduce(i,q)**(r::NNI)
      t:=s
      while t^=qq1 repeat
        s:=t
        t:=t**2
      s::Z


    ---- write p, congruent to 1 mod 4, as a sum of two squares ----
```

```
sumsq1(p:Z) : List Z ==
  s:= findelt(p)
  u:=p
  while u**2>p repeat
    w:=u rem s
    u:=s
    s:=w
  [u,s]

      ---- factorization of an integer  ----
intfactor(n:Z) : Factored ZI ==
  lfn:= factor n
  r : List FFE :=[]
  unity:ZI:=complex(unit lfn,0)
  for term in (factorList lfn) repeat
    n:=term.fctr
    exp:=term.xpnt
    n=2 =>
      r :=concat(["prime",fact2,2*exp]$FFE,r)
      unity:=unity*complex(0,-1)**(exp rem 4)::NNI

    (n rem 4) = 3 => r:=concat(["prime",complex(n,0),exp]$FFE,r)

    sz:=sumsq1(n)
    z:=complex(sz.1,sz.2)
    r:=concat(["prime",z,exp]$FFE,
            concat(["prime",conjugate(z),exp]$FFE,r))
  makeFR(unity,r)

      ---- factorization of a gaussian number  ----
factor(m:ZI) : FRZ ==
  m=0 => primeFactor(0,1)
  a:= real m

  (b:= imag m)=0 => intfactor(a) :: FRZ

  a=0 =>
    ris:=intfactor(b)
    unity:= unit(ris)*complex(0,1)
    makeFR(unity,factorList ris)

  d:=gcd(a,b)
  result : List FFE :=[]
  unity:ZI:=1$ZI

  if d^=1 then
    a:=(a exquo d)::Z
    b:=(b exquo d)::Z
    r:= intfactor(d)
    result:=factorList r
```

```
    unity:=unit r
    m:=complex(a,b)

  n:Z:=a**2+b**2
  factn:= factorList(factor n)
  part:FFE:=["prime",0$ZI,0]
  for term in factn repeat
    n:=term.fctr
    exp:=term.xpnt
    n=2 =>
      part:= ["prime",fact2,exp]$FFE
      m:=m quo (fact2**exp:NNI)
      result:=concat(part,result)

    (n rem 4) = 3 =>
      g0:=complex(n,0)
      part:= ["prime",g0,exp quo 2]$FFE
      m:=m quo g0
      result:=concat(part,result)

    z:=gcd(m,complex(n,0))
    part:= ["prime",z,exp]$FFE
    z:=z**(exp:NNI)
    m:=m quo z
    result:=concat(part,result)

  if m^=1 then unity:=unity * m
  makeFR(unity,result)

      ----  write p prime like sum of two squares  ----
sumSquares(p:Z) : List Z ==
  p=2 => [1,1]
  p rem 4 ^= 1 => error "no solutions"
  sumsq1(p)


prime?(a:ZI) : Boolean ==
   n : Z := norm a
   n=0 => false            -- zero
   n=1 => false            -- units
   prime?(n)$IntegerPrimesPackage(Z)  => true
   re : Z := real a
   im : Z := imag a
   re^=0 and im^=0 => false
   p : Z := abs(re+im)     -- a is of the form p, -p, %i*p or -%i*p
   p rem 4 ^= 3 => false
   -- return-value true, if p is a rational prime,
   -- and false, otherwise
   prime?(p)$IntegerPrimesPackage(Z)
```

———-

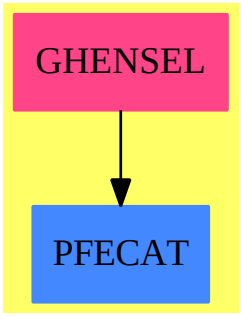
— GAUSSFAC.dotabb —

```
"GAUSSFAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GAUSSFAC"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"GAUSSFAC" -> "COMPCAT"
```

———-

8.6 package GHENSEL GeneralHenselPackage

8.6.1 GeneralHenselPackage (GHENSEL)



Exports:
completeHensel HenselLift reduction

— package GHENSEL GeneralHenselPackage —

```
)abbrev package GHENSEL GeneralHenselPackage
++ Author : P.Gianni
++ Description:
++ General Hensel Lifting
++ Used for Factorization of bivariate polynomials over a finite field.

GeneralHenselPackage(RP,TP):C == T where
   RP :   EuclideanDomain
   TP :   UnivariatePolynomialCategory RP

   PI ==> PositiveInteger

   C == with
      HenselLift: (TP,List(TP),RP,PI) -> Record(plist:List(TP), modulo:RP)
```

```
      ++ HenselLift(pol,lfacts,prime,bound) lifts lfacts,
      ++ that are the factors of pol mod prime,
      ++ to factors of pol mod prime**k > bound. No recombining is done .

     completeHensel: (TP,List(TP),RP,PI) -> List TP
      ++ completeHensel(pol,lfact,prime,bound) lifts lfact,
      ++ the factorization mod prime of pol,
      ++ to the factorization mod prime**k>bound.
      ++ Factors are recombined on the way.

     reduction     :  (TP,RP)  ->  TP
      ++ reduction(u,pol) computes the symmetric reduction of u mod pol

  T == add
    GenExEuclid: (List(FP),List(FP),FP) -> List(FP)
    HenselLift1: (TP,List(TP),List(FP),List(FP),RP,RP,F) -> List(TP)
    mQuo: (TP,RP) -> TP

    reduceCoef(c:RP,p:RP):RP ==
       zero? p => c
       RP is Integer => symmetricRemainder(c,p)
       c rem p

    reduction(u:TP,p:RP):TP ==
       zero? p => u
       RP is Integer => map(x+->symmetricRemainder(x,p),u)
       map(x+->x rem p,u)

    merge(p:RP,q:RP):Union(RP,"failed") ==
        p = q => p
        p = 0 => q
        q = 0 => p
        "failed"

    modInverse(c:RP,p:RP):RP ==
       (extendedEuclidean(c,p,1)::Record(coef1:RP,coef2:RP)).coef1

    exactquo(u:TP,v:TP,p:RP):Union(TP,"failed") ==
       invlcv:=modInverse(leadingCoefficient v,p)
       r:=monicDivide(u,reduction(invlcv*v,p))
       reduction(r.remainder,p) ^=0 => "failed"
       reduction(invlcv*r.quotient,p)

    FP:=EuclideanModularRing(RP,TP,RP,reduction,merge,exactquo)

    mQuo(poly:TP,n:RP) : TP == map(x+->x quo n,poly)

    GenExEuclid(fl:List FP,cl:List FP,rhs:FP) :List FP ==
       [clp*rhs rem flp for clp in cl for flp in fl]
```

```
-- generate the possible factors
genFact(fln:List TP,factlist:List List TP) : List List TP ==
  factlist=[] => [[pol] for pol in fln]
  maxd := +/[degree f for f in fln] quo 2
  auxfl:List List TP := []
  for poly in fln while factlist^=[] repeat
    factlist := [term for term in factlist | ^member?(poly,term)]
    dp := degree poly
    for term in factlist repeat
      (+/[degree f for f in term]) + dp > maxd => "next term"
      auxfl := cons(cons(poly,term),auxfl)
  auxfl

HenselLift1(poly:TP,fln:List TP,fl1:List FP,cl1:List FP,
            prime:RP,Modulus:RP,cinv:RP):List TP ==
   lcp := leadingCoefficient poly
   rhs := reduce(mQuo(poly - lcp * */fln,Modulus),prime)
   zero? rhs => fln
   lcinv:=reduce(cinv::TP,prime)
   vl := GenExEuclid(fl1,cl1,lcinv*rhs)
   [flp + Modulus*(vlp::TP) for flp in fln for vlp in vl]

HenselLift(poly:TP,tl1:List TP,prime:RP,bound:PI) ==
   -- convert tl1
   constp:TP:=0
   if degree first tl1 = 0 then
      constp:=tl1.first
      tl1 := rest tl1
   fl1:=[reduce(ttl,prime) for ttl in tl1]
   cl1 := multiEuclidean(fl1,1)::List FP
   Modulus:=prime
   fln :List TP := [ffl1::TP for ffl1 in fl1]
   lcinv:RP:=retract((inv
             (reduce((leadingCoefficient poly)::TP,prime)))::TP)
   while euclideanSize(Modulus)<bound repeat
      nfln:=HenselLift1(poly,fln,fl1,cl1,prime,Modulus,lcinv)
      fln = nfln and zero?(err:=poly-*/fln) => leave "finished"
      fln := nfln
      Modulus := prime*Modulus
   if constp^=0 then fln:=cons(constp,fln)
   [fln,Modulus]

completeHensel(m:TP,tl1:List TP,prime:RP,bound:PI) ==
 hlift:=HenselLift(m,tl1,prime,bound)
 Modulus:RP:=hlift.modulo
 fln:List TP:=hlift.plist
 nm := degree m
 u:Union(TP,"failed")
 aux,auxl,finallist:List TP
 auxfl,factlist:List List TP
```

```
      factlist := []
      dfn :NonNegativeInteger := nm
      lcm1 := leadingCoefficient m
      mm := lcm1*m
      while dfn>0 and (factlist := genFact(fln,factlist))^=[] repeat
        auxfl := []
        while factlist^=[] repeat
          auxl := factlist.first
          factlist := factlist.rest
          tc := reduceCoef((lcm1 * */[coefficient(poly,0)
                          for poly in auxl]), Modulus)
          coefficient(mm,0) exquo tc case "failed" =>
            auxfl := cons(auxl,auxfl)
          pol := */[poly for poly in auxl]
          poly :=reduction(lcm1*pol,Modulus)
          u := mm exquo poly
          u case "failed"  => auxfl := cons(auxl,auxfl)
          poly1: TP := primitivePart poly
          m := mQuo((u::TP),leadingCoefficient poly1)
          lcm1 := leadingCoefficient(m)
          mm := lcm1*m
          finallist := cons(poly1,finallist)
          dfn := degree m
          aux := []
          for poly in fln repeat
            ^member?(poly,auxl) => aux := cons(poly,aux)
            auxfl := [term for term in auxfl | ^member?(poly,term)]
            factlist := [term for term in factlist |^member?(poly,term)]
          fln := aux
        factlist := auxfl
      if dfn > 0 then finallist := cons(m,finallist)
      finallist
```

———-

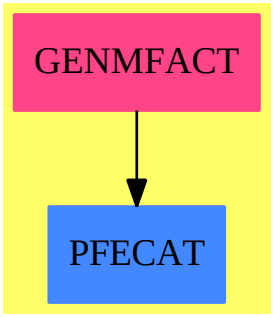
— **GHENSEL.dotabb** —

```
"GHENSEL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GHENSEL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GHENSEL" -> "PFECAT"
```

———-

8.7 package GENMFACT GeneralizedMultivariateFactorize

8.7.1 GeneralizedMultivariateFactorize (GENMFACT)



Exports:
factor

— package GENMFACT GeneralizedMultivariateFactorize —

```
)abbrev package GENMFACT GeneralizedMultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Basic Functions:
++ Related Constructors: MultFiniteFactorize, AlgebraicMultFact,
++   MultivariateFactorize
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \spadtype{Integer} or \spadtype{Fraction Integer}.

GeneralizedMultivariateFactorize(OV,E,S,R,P) : C == T
 where
  R          :   IntegralDomain
                   -- with factor on R[x]
  S          :   IntegralDomain
  OV    :   OrderedSet  with
                convert : % -> Symbol
                  ++ convert(x) converts x to a symbol
                variable: Symbol -> Union(%, "failed")
                  ++ variable(s) makes an element from symbol s or fails.
```

```
  E          :   OrderedAbelianMonoidSup
  P          :   PolynomialCategory(R,E,OV)

  C == with
    factor      :      P  ->  Factored P
      ++ factor(p) factors the multivariate polynomial p over its coefficient
      ++ domain

  T == add
    factor(p:P) : Factored P ==
      R has FiniteFieldCategory => factor(p)$MultFiniteFactorize(OV,E,R,P)
      R is Polynomial(S) and S has EuclideanDomain =>
         factor(p)$MPolyCatPolyFactorizer(E,OV,S,P)
      R is Fraction(S) and S has CharacteristicZero and
        S has EuclideanDomain =>
            factor(p)$MRationalFactorize(E,OV,S,P)
      R is Fraction Polynomial S =>
         factor(p)$MPolyCatRationalFunctionFactorizer(E,OV,S,P)
      R has CharacteristicZero and R has EuclideanDomain =>
               factor(p)$MultivariateFactorize(OV,E,R,P)
      squareFree p
```

———-

— GENMFACT.dotabb —

```
"GENMFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENMFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GENMFACT" -> "PFECAT"
```

———-

8.8 package GPAFF GeneralPackageForAlgebraicFunctionField

— GeneralPackageForAlgebraicFunctionField.input —

```
)set break resume
)sys rm -f GeneralPackageForAlgebraicFunctionField.output
)spool GeneralPackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 1
)show GeneralPackageForAlgebraicFunctionField
--R
--R GeneralPackageForAlgebraicFunctionField(K: Field,symb: List(Symbol),PolyRing: PolynomialCate
--R Abbreviation for GeneralPackageForAlgebraicFunctionField is GPAFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GPAFF
--R
--R------------------------------ Operations --------------------------------
--R adjunctionDivisor : () -> DIVISOR     desingTree : () -> List(DesTree)
--R eval : (PolyRing,Plc) -> K            genus : () -> NonNegativeInteger
--R genusNeg : () -> Integer              placesAbove : ProjPt -> List(Plc)
--R pointDominateBy : Plc -> ProjPt       printInfo : List(Boolean) -> Void
--R reset : () -> Void                    setCurve : PolyRing -> PolyRing
--R theCurve : () -> PolyRing
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R classNumber : () -> Integer if K has FINITE
--R desingTreeWoFullParam : () -> List(DesTree)
--R eval : (PolyRing,PolyRing,Plc) -> K
--R eval : (Fraction(PolyRing),Plc) -> K
--R evalIfCan : (PolyRing,Plc) -> Union(K,"failed")
--R evalIfCan : (PolyRing,PolyRing,Plc) -> Union(K,"failed")
--R evalIfCan : (Fraction(PolyRing),Plc) -> Union(K,"failed")
--R findOrderOfDivisor : (DIVISOR,Integer,Integer) -> Record(ord: Integer,num: PolyRing,den: Pol
--R homogenize : (PolyRing,Integer) -> PolyRing
--R interpolateForms : (DIVISOR,NonNegativeInteger) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R intersectionDivisor : PolyRing -> DIVISOR
--R lBasis : DIVISOR -> Record(num: List(PolyRing),den: PolyRing)
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R parametrize : (PolyRing,Plc) -> PCS
--R placesOfDegree : PositiveInteger -> List(Plc) if K has FINITE
--R rationalPlaces : () -> List(Plc) if K has FINITE
--R rationalPoints : () -> List(ProjPt) if K has FINITE
--R setSingularPoints : List(ProjPt) -> List(ProjPt)
--R singularPoints : () -> List(ProjPt)
--R
--E 1

)spool
)lisp (bye)
```

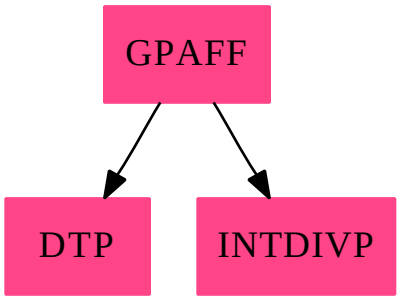
— GeneralPackageForAlgebraicFunctionField.help —

```
=====================================================================
GeneralPackageForAlgebraicFunctionField examples
=====================================================================

See Also:
o )show GeneralPackageForAlgebraicFunctionField
```

———-

8.8.1 GeneralPackageForAlgebraicFunctionField (GPAFF)



Exports:

adjunctionDivisor	classNumber	desingTree
desingTreeWoFullParam	eval	evalIfCan
findOrderOfDivisor	genus	genusNeg
homogenize	interpolateForms	interpolateFormsForFact
intersectionDivisor	lBasis	LPolynomial
numberOfPlacesOfDegree	numberPlacesDegExtDeg	numberRatPlacesExtDeg
parametrize	placesAbove	placesOfDegree
pointDominateBy	printInfo	rationalPlaces
rationalPoints	reset	setCurve
setSingularPoints	singularPoints	theCurve
ZetaFunction		

— package GPAFF GeneralPackageForAlgebraicFunctionField —

```
)abbrev package GPAFF GeneralPackageForAlgebraicFunctionField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
```

```
++ Description:
++ A package that implements the Brill-Noether algorithm. Part of the
++ PAFF package.
GeneralPackageForAlgebraicFunctionField( K,
                                         symb,
                                         PolyRing,
                                         E,
                                         ProjPt,
                                         PCS,
                                         Plc,
                                         DIVISOR,
                                         InfClsPoint,
                                         DesTree,
                                         BLMET
                                        ) :Exports == Implementation where
  K:Field
  symb: List(Symbol)
  OV ==> OrderedVariableList(symb)
  E  : DirectProductCategory(#symb,NonNegativeInteger)
  PolyRing    : PolynomialCategory(K,E,OV)
  ProjPt      : ProjectiveSpaceCategory(K)
  PCS         : LocalPowerSeriesCategory(K)
  Plc         : PlacesCategory(K,PCS)
  DIVISOR     : DivisorCategory(Plc)
  InfClsPoint : InfinitlyClosePointCategory(K,symb,PolyRing,E,ProjPt,_
                                            PCS,Plc,DIVISOR,BLMET)
  DesTree     : DesingTreeCategory(InfClsPoint)
  BLMET       : BlowUpMethodCategory

  FRACPOLY ==> Fraction PolyRing
  OF       ==> OutputForm
  INT      ==> Integer
  NNI      ==> NonNegativeInteger
  PI       ==> PositiveInteger
  UP       ==> UnivariatePolynomial
  UPZ      ==> UP(t, Integer)
  UTSZ     ==> UnivariateTaylorSeriesCZero(Integer,t)
  SUP      ==> SparseUnivariatePolynomial
  PPFC1    ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)


  ParamPackFC   ==> LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_
                                                             E,ProjPt,PCS,Plc)
  ParamPack   ==> ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
  RatSingPack ==> ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
  IntDivPack  ==> IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt,PCS,_
                                        Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
  IntFrmPack  ==> InterpolateFormsPackage(K,symb,PolyRing,E,ProjPt,PCS,_
                                          Plc,DIVISOR)
  DesTrPack   ==> DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,_
```

```
                                        Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
  PackPoly    ==> PackageForPoly(K,PolyRing,E,#symb)

  Exports ==>  with

    reset: () -> Void

    setCurve: PolyRing -> PolyRing

    homogenize: (PolyRing,Integer) -> PolyRing

    printInfo: List Boolean -> Void
      ++ printInfo(lbool) prints some information comming from various
      ++ package and domain used by this package.

    theCurve: () -> PolyRing
      ++ theCurve returns the specified polynomial for the package.

    genus: () -> NNI
      ++ genus returns the genus of the curve defined by the polynomial
      ++ given to the package.

    genusNeg: () -> INT

    desingTree: () -> List DesTree
      ++ desingTree returns the desingularisation trees at all singular
      ++ points of the curve defined by the polynomial given to the package.

    desingTreeWoFullParam  : () -> List DesTree
      ++ desingTreeWoFullParam returns the desingularisation trees at all
      ++ singular points of the curve defined by the polynomial given to
      ++ the package. The local parametrizations are not computed.

    setSingularPoints: List ProjPt -> List ProjPt
      ++ setSingularPoints(lpt) sets the singular points to be used.
      ++ Beware: no attempt is made to check if the points are singular
      ++ or not, nor if all of the singular points are presents. Hence,
      ++ results of some computation maybe false. It is intend to be use
      ++ when one want to compute the singular points are computed by other
      ++ means than to use the function singularPoints.

    singularPoints: () -> List(ProjPt)
      ++ rationalPoints() returns the singular points of the
      ++ curve defined by the polynomial given to the package.
      ++ If the singular points lie in an extension of the specified ground
      ++ field an error message is issued specifying the extension degree
      ++ needed to find all singular points.

    parametrize: (PolyRing,Plc) -> PCS
      ++ parametrize(f,pl) returns a local parametrization of f at the place
```

```
  ++ pl.

lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
  ++ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _
                Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
  ++ interpolateForms(d,n) returns a basis of the interpolate forms of
  ++ degree n of the divisor d.

interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

eval: (PolyRing,Plc) -> K
  ++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
  ++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FRACPOLY,Plc) -> K
  ++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
  ++ evalIfCan(f,pl) evaluate f at the place pl
  ++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
  ++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
  ++ (returns "failed" if it is a pole).

evalIfCan: (FRACPOLY,Plc) -> Union(K,"failed")
  ++ evalIfCan(u,pl) evaluate the function u at the place pl
  ++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
  ++ intersectionDivisor(pol) compute the intersection divisor
  ++ (the Cartier divisor) of the form pol with the curve. If some
  ++ intersection points lie in an extension of the ground field,
  ++ an error message is issued specifying the extension degree
  ++ needed to find all the intersection  points.
  ++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
  ++ adjunctionDivisor computes the adjunction divisor of the plane
  ++ curve given by the polynomial crv.

placesAbove: ProjPt -> List Plc

pointDominateBy : Plc -> ProjPt
```

```
      ++ pointDominateBy(pl) returns the projective point dominated
      ++ by the place pl.

    if K has Finite then --should we say LocallyAlgebraicallyClosedField??

      rationalPlaces: () -> List Plc
        ++ rationalPlaces returns all the rational places of  the
        ++ curve defined by the polynomial given to the package.

      rationalPoints: () -> List(ProjPt)

      LPolynomial: () -> SparseUnivariatePolynomial Integer
        ++ Returns the L-Polynomial of the curve.

      LPolynomial: PI -> SparseUnivariatePolynomial Integer
        ++ LPolynomial(d) returns the L-Polynomial of the curve in
        ++ constant field extension of degree d.

      classNumber: () -> Integer
        ++ Returns the class number of the curve.

      placesOfDegree: PI -> List Plc
        ++ placesOfDegree(d) returns all places of degree d of the
        ++ curve.

      numberOfPlacesOfDegree: PI -> Integer
        ++ returns the number of places of the given degree

      numberRatPlacesExtDeg: PI -> Integer
        ++ numberRatPlacesExtDeg(n) returns the number of rational
        ++ places in the constant field extenstion of degree n

      numberPlacesDegExtDeg: (PI, PI) -> Integer
        ++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
        ++ places of degree d in the constant field extension of
        ++ degree n

      ZetaFunction: () -> UTSZ
        ++ Returns the Zeta function of the curve. Calculated by
        ++ using the L-Polynomial

      ZetaFunction: PI -> UTSZ
        ++ Returns the Zeta function of the curve in constant field
        ++ extension. Calculated by using the L-Polynomial

  Implementation ==>  add
    import PPFC1
    import PPFC2
    import DesTrPack
    import IntFrmPack
```

```
import IntDivPack
import RatSingPack
import ParamPack
import ParamPackFC
import PackPoly

crvLocal:PolyRing:=1$PolyRing

-- flags telling such and such is already computed.

genusCalc?:Boolean:= false()$Boolean
theGenus:INT:=0

desingTreeCalc?:Boolean:=false()$Boolean
theTree:List DesTree := empty()

desingTreeWoFullParamCalc?:Boolean:=false()$Boolean

adjDivCalc?:Boolean:=false()$Boolean
theAdjDiv:DIVISOR:=0

singularPointsCalc?:Boolean:=false()$Boolean
lesPtsSing:List(ProjPt):=empty()

rationalPointsCalc?:Boolean:=false()$Boolean
lesRatPts:List(ProjPt):=empty()

rationalPlacesCalc?:Boolean:=false()$Boolean
lesRatPlcs:List(Plc):=empty()

zf:UTSZ:=1$UTSZ
zfCalc : Boolean := false()$Boolean

DegOfPlacesFound: List Integer := empty()

-- see package IntersectionDivisorPackage
intersectionDivisor(pol)==
  if ^(pol =$PolyRing homogenize(pol,1)) then _
    error _
     "From intersectionDivisor: the input is NOT a homogeneous polynomial"
  intersectionDivisor(pol,theCurve(),desingTree(),singularPoints())

lBasis(divis)==
  d:=degree divis
  d < 0 => [[0$PolyRing],1$PolyRing]
  A:=adjunctionDivisor()
  -- modifie le 08/05/97: avant c'etait formToInterp:=divOfZero(divis) + A
  formToInterp:=  divOfZero(divis + A)
  degDpA:=degree  formToInterp
  degCrb:=totalDegree(theCurve())$PackPoly
```

```
    dd:=divide(degDpA,degCrb pretend Integer)
    dmin:NNI:=
      if ^zero?(dd.remainder) then (dd.quotient+1) pretend NNI
      else dd.quotient pretend NNI
    print("Trying to interpolate with forms of degree:"::OF)
    print(dmin::OF)
    lg0:List PolyRing:=interpolateForms(formToInterp,dmin)
    while zero?(first lg0) repeat
      dmin:=dmin+1
      print("Trying to interpolate with forms of degree:"::OF)
      print(dmin::OF)
      lg0:=interpolateForms(formToInterp,dmin)
    print("Denominator found"::OF)
    g0:PolyRing:=first lg0
    dg0:=intersectionDivisor(g0)
    print("Intersection Divisor of Denominator found"::OF)
    lnumer:List PolyRing:=interpolateForms(dg0-divis,dmin)
    [lnumer,g0]

  genus==
    if ^(genusCalc?) then
      degCrb:=totalDegree(theCurve())$PackPoly
      theGenus:=genusTreeNeg(degCrb,desingTreeWoFullParam())
      genusCalc?:=true()$Boolean
    theGenus < 0 =>
      print(("Too many infinitly near points")::OF)
      print(("The curve may not be absolutely irreducible")::OF)
      error "Have a nice day"
    theGenus pretend NNI

  genusNeg==
    if ^(genusCalc?) then
      degCrb:=totalDegree(theCurve())$PackPoly
      theGenus:=genusTreeNeg(degCrb,desingTreeWoFullParam())
      genusCalc?:=true()$Boolean
    theGenus

  homogenize(pol,n)== homogenize(pol,n)$PackPoly

  fPl(pt:ProjPt,desTr:DesTree):Boolean ==
    nd:=value desTr
    lpt:=pointV nd
    pt = lpt


  placesAbove(pt)==
    -- verifie si le point est simple, si c'est le cas,
    -- on retourne la place correpondante
    -- avec pointToPlace qui cre' la place si necessaire.
    ^member?(pt,singularPoints()) => _
```

```
      [pointToPlace(pt,theCurve())$ParamPackFC]
    -- les quatres lignes suivantes trouvent les feuilles qui
    -- sont au-dessus du point.
    theTree:= desingTree()
    cTree:= find(fPl(pt,#1),theTree)
    cTree case "failed" => error "Big error in placesAbove"
      -- G. Hache, gaetan.hache@inria.fr"
    lvs:=leaves cTree
    -- retourne les places correspondant aux feuilles en "consultant"
    -- les diviseurs exceptionnels.
    concat [supp excpDivV(l) for l in lvs]

  pointDominateBy(pl)== pointDominateBy(pl)$ParamPackFC

  reduceForm(p1:PolyRing,p2:PolyRing):PolyRing==
    normalForm(p1,[p2])$GroebnerPackage(K,E,OV,PolyRing)

  evalIfCan(f:PolyRing,pl:Plc)==
    u:=reduceForm(f, theCurve() )
    zero?(u) => 0
    pf:= parametrize(f,pl)
    ord:INT:=order pf
    ord < 0 => "failed"
    ord > 0 => 0
    coefOfFirstNonZeroTerm pf

  eval(f:PolyRing,pl:Plc)==
    eic:=evalIfCan(f,pl)
    eic case "failed" => _
      error "From eval (function at place): its a pole !!!"
    eic

  setCurve(pol)==
    crvLocal:=pol
    ^(crvLocal =$PolyRing homogenize(crvLocal,1)) =>
      print(("the defining polynomial is not homogeneous")::OF)
      error "Have a nice day"
    reset()
    theCurve()

  reset ==
    setFoundPlacesToEmpty()$Plc
    genusCalc?:Boolean:= false()$Boolean
    theGenus:INT:=0
    desingTreeCalc?:Boolean:=false()$Boolean
    desingTreeWoFullParamCalc?:Boolean:=false()$Boolean
    theTree:List DesTree := empty()
    adjDivCalc?:Boolean:=false()$Boolean
    theAdjDiv:DIVISOR:=0
    singularPointsCalc?:Boolean:=false()$Boolean
```

```
    lesPtsSing:List(ProjPt):=empty()
    rationalPointsCalc?:Boolean:=false()$Boolean
    lesRatPts:List(ProjPt):=empty()
    rationalPlacesCalc?:Boolean:=false()$Boolean
    lesRatPlcs:List(Plc):=empty()
    DegOfPlacesFound: List Integer := empty()
    zf:UTSZ:=1$UTSZ
    zfCalc:Boolean := false$Boolean

  foundPlacesOfDeg?(i:PositiveInteger):Boolean ==
    ld: List Boolean := [zero?(a rem i) for a in DegOfPlacesFound]
    entry?(true$Boolean,ld)

  findOrderOfDivisor(divis,lb,hb) ==
    ^zero?(degree divis) => error("The divisor is NOT of degre zero !!!!")
    A:=adjunctionDivisor()
    formToInterp:=divOfZero ( hb*divis + A )
    degDpA:=degree formToInterp
    degCrb:=totalDegree( theCurve())$PackPoly
    dd:=divide(degDpA,degCrb pretend Integer)
    dmin:NNI:=
      if ^zero?(dd.remainder) then (dd.quotient+1) pretend NNI
      else dd.quotient pretend NNI
    lg0:List PolyRing:=interpolateForms(formToInterp,dmin)
    while zero?(first lg0) repeat
      dmin:=dmin+1
      lg0:=interpolateForms(formToInterp,dmin)
    g0:PolyRing:=first lg0
    dg0:=intersectionDivisor(g0)
    nhb:=hb
    while effective?(dg0 - nhb*divis - A) repeat
      nhb:=nhb+1
    nhb:=nhb-1
    ftry:=lb
    lnumer:List PolyRing:=interpolateForms(dg0-ftry*divis,dmin)
    while zero?(first lnumer) and ftry < nhb repeat
      ftry:=ftry + 1
      lnumer:List PolyRing:=interpolateForms(dg0-ftry*divis,dmin)
    [ftry,first lnumer,g0,nhb]

  theCurve==
    one?(crvLocal) => error "The defining polynomial has not been set yet!"
    crvLocal

  printInfo(lbool)==
    printInfo(lbool.2)$ParamPackFC
    printInfo(lbool.3)$PCS
    void()

  desingTree==
```

```
  theTree:= desingTreeWoFullParam()
  if ^(desingTreeCalc?) then
    for arb in theTree repeat
      fullParamInit(arb)
    desingTreeCalc?:=true()$Boolean
  theTree

desingTreeWoFullParam==
  if ^(desingTreeWoFullParamCalc?) then
    theTree:=[desingTreeAtPoint(pt,theCurve())  for pt in singularPoints()]
    desingTreeWoFullParamCalc?:=true()$Boolean
  theTree

-- compute the adjunction divisor of the curve using adjunctionDivisor
-- from DesingTreePackage
adjunctionDivisor()==
  if ^(adjDivCalc?) then
    theAdjDiv:=_
      reduce("+",[adjunctionDivisor(tr) for tr in desingTree()],0$DIVISOR)
    adjDivCalc?:=true()$Boolean
  theAdjDiv

-- returns the singular points using the function singularPoints
-- from ProjectiveAlgebraicSetPackage
singularPoints==
  if ^(singularPointsCalc?) then
    lesPtsSing:=singularPoints(theCurve())
    singularPointsCalc?:=true()$Boolean
  lesPtsSing

setSingularPoints(lspt)==
  singularPointsCalc?:=true()$Boolean
  lesPtsSing:= lspt

-- returns the rational points using the function rationalPoints
-- from ProjectiveAlgebraicSetPackage

-- compute the local parametrization of f at the place pl
-- (from package ParametrizationPackage)
parametrize(f,pl)==parametrize(f,pl)$ParamPack

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==
  lm:List PolyRing:=listAllMono(n)$PackPoly
  interpolateForms(d,n,theCurve(),lm)

interpolateFormsForFact(d,lm)==
  interpolateFormsForFact(d,lm)$IntFrmPack

evalIfCan(f:PolyRing,g:PolyRing,pl:Plc)==
```

```
    fu:=reduceForm(f,theCurve())
    gu:=reduceForm(g,theCurve())
    zero?(fu) and ^zero?(gu) => 0
    ^zero?(fu) and zero?(gu) => "failed"
    pf:= parametrize(fu,pl)
    pg:= parametrize(gu,pl)
    ordf:INT:=order pf
    ordg:INT:=order pg
    cf:=coefOfFirstNonZeroTerm pf
    cg:=coefOfFirstNonZeroTerm pg
    (ordf - ordg) < 0 => "failed"
    (ordf - ordg) > 0 => 0
    cf * inv cg

  eval(f:PolyRing,g:PolyRing,pl:Plc)==
    eic:=evalIfCan(f,g,pl)
    eic case "failed" => error "From eval (function at place): its a pole"
    eic

  evalIfCan(u:FRACPOLY,pl:Plc)==
    f:PolyRing := numer u
    g:PolyRing := denom u
    evalIfCan(f,g,pl)

  eval(u:FRACPOLY,pl:Plc)==
    f:PolyRing := numer u
    g:PolyRing := denom u
    eval(f,g,pl)

  thedeg:PI := 1

  crap(p:Plc):Boolean ==
    degree(p)$Plc = thedeg

  if K has Finite then
    rationalPlaces ==
      K has PseudoAlgebraicClosureOfFiniteFieldCategory => _
         placesOfDegree(1$PI)
      --non good pour LACF !!!!
      rationalPlacesCalc? => lesRatPlcs
      ltr:List(DesTree):=desingTree()
      ratP:List(ProjPt):=rationalPoints()
      singP:List(ProjPt):=singularPoints()
      simRatP:List(ProjPt):=setDifference(ratP,singP)
      for pt in simRatP repeat
        pointToPlace(pt,theCurve())$ParamPackFC
      rationalPlacesCalc? := true()$Boolean
      lesRatPlcs:=foundPlaces()$Plc
      lesRatPlcs
```

```
      rationalPoints==
        if ^(rationalPointsCalc?) then
          if K has Finite then
            lesRatPts:= rationalPoints(theCurve(),1)$RatSingPack
            rationalPointsCalc?:=true()$Boolean
          else
            error "Can't find rationalPoints when the field is not finite"
        lesRatPts

      ZetaFunction() ==
        if not zfCalc then
  zf:= ZetaFunction(1)
          zfCalc:= true$Boolean
zf

      ZetaFunction(d) ==
  lp:= LPolynomial(d)
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)
  q:INT := size()$K ** d
  lpt:UPZ := unmakeSUP(lp)$UPZ
  lps:UTSZ := coerce(lpt)$UTSZ
  x:= monomial(1,1)$UTSZ
  mul: UTSZ := (1-x)*(1 - q * x)
  invmul:Union(UTSZ,"failed") := recip(mul)$UTSZ
  ivm: UTSZ
  if not (invmul case "failed") then
    ivm := invmul pretend UTSZ
  else
    ivm := 1
          lps * ivm

      calculatedSer: List UTSZ:= [1]
        --in index i is the "almost ZetaFunction" summed to i-1.
--Except calculatedSer.1 which is 1

      numberOfPlacesOfDegreeUsingZeta(degree:PI): Integer  ==
        --is at most called once for each degree. Will calculate the
--entries in calculatdSer.
        ser:UTSZ := 1
x:= monomial(1,1)$UTSZ
pol:UTSZ
polser:Union(UTSZ,"failed")
serdel:UTSZ
i:PI := maxIndex(calculatedSer) pretend PI
while i < degree repeat
  serdel:= 1
  if (n:= numberOfPlacesOfDegree(i)) > 0 then
    pol:= (1-x**i) ** (n pretend PI)
    polser:= recip(pol)$UTSZ -- coerce(pol)$UTSZ)$UTSZ
```

```
    if not (polser case "failed") then
      serdel:= (polser pretend UTSZ)
    else
      error "In numberOfPlacesOfDegreeUsingZeta. This shouldn't happen"
  ser:= serdel * calculatedSer.i
  calculatedSer:= concat(calculatedSer, ser)
  i:= i + 1
if degree = 1 then
  coefficient(ZetaFunction(),degree)
        else
  coefficient(ZetaFunction(),degree) - _
           coefficient(calculatedSer.degree, degree)

      calculatedNP: List Integer := empty()
        --local variable, in index i is number of places of degree i.

      numberOfPlacesOfDegree(i:PI): Integer ==
         if zfCalc then
   if (m := maxIndex(calculatedNP)) < i then
     calculatedNP:= _
               concat(calculatedNP, _
                 [numberOfPlacesOfDegreeUsingZeta(j pretend PI) _
                   for j in ((m+1) pretend PI)..i])
   calculatedNP.i
 else
   # placesOfDegree(i) --maybe we should make an improvement in this

      placesOfDegree(i) ==
        if (not foundPlacesOfDeg?(i)) then
   if characteristic()$K**i > (2**16 - 1) then
     print("If you are using a prime field and"::OF)
            print("GB this will not work."::OF)
   desingTree()
   placesOfDegree(i,theCurve(),singularPoints())
   DegOfPlacesFound:= concat(DegOfPlacesFound, i)
thedeg:= i
select(crap(#1), foundPlaces()$Plc)

      numberRatPlacesExtDeg(extDegree:PI): Integer ==
        numberPlacesDegExtDeg(1,extDegree)

      numberPlacesDegExtDeg(degree:PI, extDegree:PI): Integer ==
        res:Integer:=0
m:PI := degree * extDegree
d: PI
while m > 0 repeat
  d:= gcd(m, extDegree)
  if (m quo d) = degree then
    res:= res + (numberOfPlacesOfDegree(m) * d)
  m:= (m - 1) pretend PI
```

```
res

      calculateS(extDeg:PI): List Integer ==
        g := genus()
sizeK:NNI := size()$K ** extDeg
i:PositiveInteger := g pretend PI
S: List Integer := [0 for j in 1..g]
good:Boolean := true()$Boolean
while good repeat
  S.i := numberRatPlacesExtDeg(i*extDeg) - ((sizeK **$NNI i) + 1)
  j:Integer := i - 1
  if (not (j = 0))  then
    i:= (j pretend PI)
  else good:= false()$Boolean
S

      LPolynomial(): SparseUnivariatePolynomial Integer ==
        LPolynomial(1)

      LPolynomial(extDeg:PI): SparseUnivariatePolynomial Integer ==
        --when translating to AxiomXL rewrite this function!
g := genus()
zero?(g) => 1
coef: List Integer := [1]
if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
  setTower!(1$K)
        sizeK:Integer := size()$K ** extDeg  --need to do a setExtension before
        coef:= concat(coef,[0 for j in 1..(2*g)])
S: List Integer := calculateS(extDeg)
i:PI := 1
tmp:Integer
        while i < g + 1 repeat
          j:PI := 1
  tmp:= 0
  while j < i + 1 repeat
    tmp:= tmp + S.j * coef((i + 1 - j) pretend PI)
    j:= j + 1
  coef.(i+1) := tmp quo i
  i:= i + 1
i:= 1
while i < g + 1 repeat
          ss: Integer := sizeK **$Integer ((g + 1 - i) pretend PI)
          val:Integer := ss * coef.i
          coef.((2*g+2 - i) pretend PI) := val
  i:= i + 1
x:= monomial(1,1)$SUP(INT)
        result: SparseUnivariatePolynomial(Integer):= _
          1$SparseUnivariatePolynomial(Integer)
coef:= rest(coef)
i:= 1
```

```
while i < 2 * g + 1 repeat
  pol: SUP(INT) := (first(coef) :: Integer) * (x ** i)
          result:= result + pol --(first(coef) :: Integer) * (x ** i)
          coef:= rest(coef)
          i:= i + 1
        result

      classNumber():Integer ==
        LPolynomial()(1)
```

———-

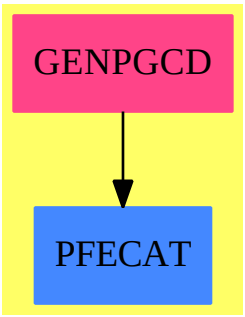
— **GPAFF.dotabb** —

```
"GPAFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GPAFF"]
"DTP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DTP"]
"INTDIVP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTDIVP"]
"GPAFF" -> "INTDIVP"
"GPAFF" -> "DTP"
```

———-

8.9 package GENPGCD GeneralPolynomialGcdPackage

8.9.1 GeneralPolynomialGcdPackage (GENPGCD)



Exports:
gcdPolynomial randomR

— **package GENPGCD GeneralPolynomialGcdPackage** —

```
)abbrev package GENPGCD GeneralPolynomialGcdPackage
```

```
++ Description:
++ This package provides operations for GCD computations on polynomials

GeneralPolynomialGcdPackage(E,OV,R,P):C == T where
    R     :  PolynomialFactorizationExplicit
    P     :  PolynomialCategory(R,E,OV)
    OV    :  OrderedSet
    E     :  OrderedAbelianMonoidSup

    SUPP      ==> SparseUnivariatePolynomial P
--JHD    ContPrim  ==> Record(cont:P,prim:P)

    C == with
           gcdPolynomial     :   (SUPP,SUPP) -> SUPP
             ++ gcdPolynomial(p,q) returns the GCD of p and q
           randomR        : ()               ->R
             ++ randomR() should be local but conditional
--JHD      gcd               :   (P,P)    -> P
--JHD      gcd               :   List P   -> P
--JHD      gcdprim           :   (P,P)    -> P
--JHD      gcdprim           :   List P   -> P

--JHD      gcdcofact         :   List P   -> List P
--JHD      gcdcofactprim     :   List P   -> List P

--JHD      primitate         :   (P,OV)   -> P
--JHD      primitate         :    SUPP    -> SUPP

--JHD      content           :     P      -> P
--JHD      content           :   List P   -> List P
--JHD      contprim          :   List P   -> List ContPrim

--JHD      monomContent      :   (P,OV)   -> P
--JHD      monomContent      :    SUPP    -> SUPP


    T == add

      SUPR     ==> SparseUnivariatePolynomial R
--JHD      SUPLGcd  ==> Record(locgcd:SUPP,goodint:List R)
--JHD      LGcd     ==> Record(locgcd:P,goodint:List R)
--JHD      UTerm    ==> Record(lpol:List SUPR,lint:List R,mpol:P)
--JHD--JHD      pmod:R   :=  (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

--JHD      import MultivariateLifting(E,OV,R,P,pmod)
      import UnivariatePolynomialCategoryFunctions2(R,SUPR,P,SUPP)
      import UnivariatePolynomialCategoryFunctions2(P,SUPP,R,SUPR)
                 --------  Local  Functions  --------

--JHD      abs             :      P       -> P
```

```
      better          :    (P,P)     -> Boolean
--JHD      failtest        :   (P,P,P)    -> Boolean
--JHD      gcdMonom        :  (P,P,OV)    -> P
--JHD      gcdTermList     :    (P,P)     -> P
--JHD      gcdPrim         :  (P,P,OV)    -> P
--JHD      gcdSameMainvar  :  (P,P,OV)    -> P
--JHD      internal        :  (P,P,OV)    -> P
--JHD      good            :  (P,List OV) -> Record(upol:SUPR,inval:List R)
--JHD      gcdPrs          : (P,P,NNI,OV) -> Union(P,"failed")
--JHD
--JHD      chooseVal       :     (P,P,List OV)          -> UTerm
--JHD      localgcd        :     (P,P,List OV)          -> LGcd
--JHD      notCoprime      :  (P,P, List NNI,List OV)   -> P
--JHD      imposelc        : (List SUPR,List OV,List R,List P)  -> List SUPR

--JHD      lift? :(P,P,UTerm,List NNI,List OV)  -> Union("failed",P)
--      lift  :(P,SUPR,SUPR,P,List OV,List NNI,List R) -> P
      lift  :  (SUPR,SUPP,SUPR,List OV,List R) -> Union(SUPP,"failed")
         -- lifts first and third arguments as factors of the second
         -- fourth is number of variables.
--JHD      monomContent      :   (P,OV)   -> P
      monomContentSup   :    SUPP    -> SUPP
--
--JHD  gcdcofact         :   List P   -> List P

      gcdTrivial      :  (SUPP,SUPP)   -> SUPP
      gcdSameVariables:  (SUPP,SUPP,List OV)    -> SUPP
      recursivelyGCDCoefficients: (SUPP,List OV,SUPP,List OV) -> SUPP
      flatten         : (SUPP,List OV) -> SUPP
                        -- evaluates out all variables in the second
                        -- argument, leaving a polynomial of the same
                        -- degree
--    eval            : (SUPP,List OV,List R) -> SUPP
      variables       :  SUPP          -> List OV
                     ---- JHD's exported functions ---
      gcdPolynomial(p1:SUPP,p2:SUPP) ==
        zero? p1 => p2
        zero? p2 => p1
        0=degree p1  => gcdTrivial(p1,p2)
        0=degree p2  => gcdTrivial(p2,p1)
        if degree p1 < degree p2 then (p1,p2):=(p2,p1)
        p1 exquo p2 case SUPP  => (unitNormal p2).canonical
        c1:= monomContentSup(p1)
        c2:= monomContentSup(p2)
        p1:= (p1 exquo c1)::SUPP
        p2:= (p2 exquo c2)::SUPP
        (p1 exquo p2) case SUPP => (unitNormal p2).canonical * gcd(c1,c2)
        vp1:=variables p1
        vp2:=variables p2
        v1:=setDifference(vp1,vp2)
```

```
        v2:=setDifference(vp2,vp1)
        #v1 = 0 and #v2 = 0 => gcdSameVariables(p1,p2,vp1)*gcd(c1,c2)
                 -- all variables are in common
        v:=setDifference(vp1,v1)
        pp1:=flatten(p1,v1)
        pp2:=flatten(p2,v2)
        g:=gcdSameVariables(pp1,pp2,v)
--        one? g => gcd(c1,c2)::SUPP
        (g = 1) => gcd(c1,c2)::SUPP
        (#v1 = 0 or not (p1 exquo g) case "failed") and
                     -- if #vi = 0 then pp1 = p1, so we know g divides
              (#v2 = 0 or not (p2 exquo g) case "failed")
            => g*gcd(c1,c2)  -- divdes them both, so is the gcd
        -- OK, so it's not the gcd: try again
        v:=variables g -- there can be at most these variables in answer
        v1:=setDifference(vp1,v)
        v2:=setDifference(vp2,v)
        if (#v1 = 0) then g:= gcdSameVariables(g,flatten(p2,v2),v)
        else if (#v2=0) then g:=gcdSameVariables(g,flatten(p1,v1),v)
        else g:=gcdSameVariables(g,flatten(p1,v1)-flatten(p2,v2),v)
--        one? g => gcd(c1,c2)::SUPP
        (g = 1) => gcd(c1,c2)::SUPP
        (#v1 = 0 or not (p1 exquo g) case "failed") and
              (#v2 = 0 or not (p2 exquo g) case "failed")
            => g*gcd(c1,c2)::SUPP  -- divdes them both, so is the gcd
        v:=variables g -- there can be at most these variables in answer
        v1:=setDifference(vp1,v)
        if #v1 ^= 0 then
           g:=recursivelyGCDCoefficients(g,v,p1,v1)
--           one? g => return gcd(c1,c2)::SUPP
           (g = 1) => return gcd(c1,c2)::SUPP
           v:=variables g -- there can be at most these variables in answer
        v2:=setDifference(vp2,v)
        recursivelyGCDCoefficients(g,v,p2,v2)*gcd(c1,c2)
      if R has StepThrough then
         randomCount:R := init()
         randomR() ==
            (v:=nextItem(randomCount)) case R =>
                randomCount:=v
                v
            SAY("Taking next stepthrough range in GeneralPolynomialGcdPackage")$Lisp
            randomCount:=init()
            randomCount
      else
            randomR() == (random$Integer() rem 100)::R
                     ---- JHD's local functions ---
      gcdSameVariables(p1:SUPP,p2:SUPP,lv:List OV) ==
            -- two non-trivial primitive (or, at least, we don't care
            -- about content)
            -- polynomials with precisely the same degree
```

```
#lv = 0 => map((x:R):P+->x::P,gcdPolynomial(map(ground,p1),
                                    map(ground,p2)))
degree p2 = 1 =>
  p1 exquo p2 case SUPP => p2
  1
gcdLC:=gcd(leadingCoefficient p1,leadingCoefficient p2)
lr:=[randomR() for vv in lv]
count:NonNegativeInteger:=0
while count<10 repeat
  while zero? eval(gcdLC,lv,lr) and count<10 repeat
    lr:=[randomR() for vv in lv]
    count:=count+1
  count = 10 => error "too many evaluations in GCD code"
  up1:SUPR:=map(y+->ground eval(y,lv,lr),p1)
  up2:SUPR:=map(z+->ground eval(z,lv,lr),p2)
  u:=gcdPolynomial(up1,up2)
  degree u = 0 => return 1
  -- let's pick a second one, just to check
  lrr:=[randomR() for vv in lv]
  while zero? eval(gcdLC,lv,lrr) and count<10 repeat
    lrr:=[randomR() for vv in lv]
    count:=count+1
  count = 10 => error "too many evaluations in GCD code"
  vp1:SUPR:=map(x1+->ground eval(x1,lv,lrr),p1)
  vp2:SUPR:=map(y1+->ground eval(y1,lv,lrr),p2)
  v:=gcdPolynomial(vp1,vp2)
  degree v = 0 => return 1
  if degree v < degree u then
     u:=v
     up1:=vp1
     up2:=vp2
     lr:=lrr
  up1:=(up1 exquo u)::SUPR
  degree gcd(u,up1) = 0 =>
      ans:=lift(u,p1,up1,lv,lr)
      ans case SUPP => return ans
      "next"
  up2:=(up2 exquo u)::SUPR
  degree gcd(u,up2) = 0 =>
      ans:=lift(u,p2,up2,lv,lr)
      ans case SUPP => return ans
      "next"
  -- so neither cofactor is relatively prime
  count:=0
  while count < 10 repeat
     r:=randomR()
     uu:=up1+r*up2
     degree gcd(u,uu)=0 =>
       ans:= lift(u,p1+r::P *p2,uu,lv,lr)
       ans case SUPP => return ans
```

```
            "next"
          error "too many evaluations in GCD code"
        count >= 10 => error "too many evaluations in GCD code"
    lift(gR:SUPR,p:SUPP,cfR:SUPR,lv:List OV,lr:List R) ==
      -- lift the coprime factorisation gR*cfR = (univariate of p)
      -- where the variables lv have been evaluated at lr
      lcp:=leadingCoefficient p
      g:=monomial(lcp,degree gR)+map(x+->x::P,reductum gR)
      cf:=monomial(lcp,degree cfR)+map(y+->y::P,reductum cfR)
      p:=lcp*p       -- impose leaidng coefficient of p on each factor
      while lv ^= [] repeat
         v:=first lv
         r:=first lr
         lv:=rest lv
         lr:=rest lr
         thisp:=map(x1+->eval(x1,lv,lr),p)
         d:="max"/[degree(c,v) for c in coefficients p]
         prime:=v::P - r::P
         pn:=prime
         origFactors:=[g,cf]::List SUPP
         for n in 1..d repeat
            Ecart:=(thisp- g*cf) exquo pn
            Ecart case "failed" =>
               error "failed lifting in hensel in Complex Polynomial GCD"
            zero? Ecart => leave
            step:=solveLinearPolynomialEquation(origFactors,
                      map(x2+->eval(x2,v,r),Ecart::SUPP))
            step case "failed" => return "failed"
            g:=g+pn*first step
            cf:=cf+pn*second step
            pn:=pn*prime
         thisp ^= g*cf => return "failed"
      g
    recursivelyGCDCoefficients(g:SUPP,v:List OV,p:SUPP,pv:List OV) ==
       mv:=first pv   -- take each coefficient w.r.t. mv
       pv:=rest pv    -- and recurse on pv as necessary
       d:="max"/[degree(u,mv) for u in coefficients p]
       for i in 0..d repeat
           p1:=map(x+->coefficient(x,mv,i),p)
           oldg:=g
           if pv = [] then g:=gcdSameVariables(g,p1,v)
           else g:=recursivelyGCDCoefficients(p,v,p1,pv)
--           one? g => return 1
           (g = 1) => return 1
           g^=oldg =>
              oldv:=v
              v:=variables g
              pv:=setUnion(pv,setDifference(v,oldv))
       g
    flatten(p1:SUPP,lv:List OV) ==
```

```
         #lv = 0 => p1
         lr:=[ randomR() for vv in lv]
         dg:=degree p1
         while dg ^= degree (ans:= map(x+->eval(x,lv,lr),p1)) repeat
           lr:=[ randomR() for vv in lv]
         ans
--      eval(p1:SUPP,lv:List OV,lr:List R) == map(eval(#1,lv,lr),p1)
      variables(p1:SUPP) ==
        removeDuplicates ("concat"/[variables u for u in coefficients p1])
      gcdTrivial(p1:SUPP,p2:SUPP) ==
        -- p1 is non-zero, but has degree zero
        -- p2 is non-zero
        cp1:=leadingCoefficient p1
--        one? cp1 => 1
        (cp1 = 1) => 1
        degree p2 = 0 => gcd(cp1,leadingCoefficient p2)::SUPP
        un?:=unit? cp1
        while not zero? p2 and not un? repeat
           cp1:=gcd(leadingCoefficient p2,cp1)
           un?:=unit? cp1
           p2:=reductum p2
        un? => 1
        cp1::SUPP

                     ---- Local  functions ----
--JHD    -- test if something wrong happened in the gcd
--JHD      failtest(f:P,p1:P,p2:P) : Boolean ==
--JHD        (p1 exquo f) case "failed" or (p2 exquo f) case "failed"
--JHD
--JHD    -- Choose the integers
--JHD      chooseVal(p1:P,p2:P,lvar:List OV):UTerm ==
--JHD        x:OV:=lvar.first
--JHD        lvr:=lvar.rest
--JHD        d1:=degree(p1,x)
--JHD        d2:=degree(p2,x)
--JHD        dd:NNI:=0$NNI
--JHD        nvr:NNI:=#lvr
--JHD        lval:List R :=[]
--JHD        range:I:=8
--JHD        for i in 1..  repeat
--JHD          range:=2*range
--JHD          lval:=[(random()$I rem (2*range) - range)::R  for i in 1..nvr]
--JHD          uf1:SUPR:=univariate eval(p1,lvr,lval)
--JHD          degree uf1 ^= d1 => "new point"
--JHD          uf2:SUPR:=univariate eval(p2,lvr,lval)
--JHD          degree uf2 ^= d2 => "new point"
--JHD          u:=gcd(uf1,uf2)
--JHD          du:=degree u
--JHD         --the univariate gcd is 1
--JHD          if du=0 then return [[1$SUPR],lval,0$P]$UTerm
```

```
--JHD
--JHD          ugcd:List SUPR:=[u,(uf1 exquo u)::SUPR,(uf2 exquo u)::SUPR]
--JHD          uterm:=[ugcd,lval,0$P]$UTerm
--JHD          dd=0 => dd:=du
--JHD
--JHD        --the degree is not changed
--JHD          du=dd =>
--JHD
--JHD           --test if one of the polynomials is the gcd
--JHD            dd=d1 =>
--JHD              if ^((f:=p2 exquo p1) case "failed") then
--JHD                return [[u],lval,p1]$UTerm
--JHD              if dd^=d2 then dd:=(dd-1)::NNI
--JHD
--JHD            dd=d2 =>
--JHD              if ^((f:=p1 exquo p2) case "failed") then
--JHD                return [[u],lval,p2]$UTerm
--JHD              dd:=(dd-1)::NNI
--JHD            return uterm
--JHD
--JHD         --the new gcd has degree less
--JHD          du<dd => dd:=du
--JHD
--JHD      good(f:P,lvr:List OV):Record(upol:SUPR,inval:List R) ==
--JHD        nvr:NNI:=#lvr
--JHD        range:I:=1
--JHD        ltry:List List R:=[]
--JHD        while true repeat
--JHD          range:=2*range
--JHD          lval:=[(random()$I rem (2*range) -range)::R  for i in 1..nvr]
--JHD          member?(lval,ltry) => "new point"
--JHD          ltry:=cons(lval,ltry)
--JHD          uf:=univariate eval(f,lvr,lval)
--JHD          if degree gcd(uf,differentiate uf)=0 then return [uf,lval]
--JHD
--JHD      -- impose the right lc
--JHD      imposelc(lipol:List SUPR,
--JHD               lvar:List OV,lval:List R,leadc:List P):List SUPR ==
--JHD        result:List SUPR :=[]
--JHD        lvar:=lvar.rest
--JHD        for pol in lipol for leadpol in leadc repeat
--JHD           p1:= univariate eval(leadpol,lvar,lval) * pol
--JHD           result:= cons((p1 exquo leadingCoefficient pol)::SUPR,result)
--JHD        reverse result
--JHD
--JHD     --Compute the gcd between not coprime polynomials
--JHD      notCoprime(g:P,p2:P,ldeg:List NNI,lvar:List OV) : P ==
--JHD        x:OV:=lvar.first
--JHD        lvar1:List OV:=lvar.rest
--JHD        lg1:=gcdcofact([g,differentiate(g,x)])
```

```
--JHD        g1:=lg1.1
--JHD        lg:LGcd:=localgcd(g1,p2,lvar)
--JHD        (l,lval):=(lg.locgcd,lg.goodint)
--JHD        p2:=(p2 exquo l)::P
--JHD        (gd1,gd2):=(l,l)
--JHD        ul:=univariate(eval(l,lvar1,lval))
--JHD        dl:=degree ul
--JHD        if degree gcd(ul,differentiate ul) ^=0 then
--JHD          newchoice:=good(l,lvar.rest)
--JHD          ul:=newchoice.upol
--JHD          lval:=newchoice.inval
--JHD        ug1:=univariate(eval(g1,lvar1,lval))
--JHD        ulist:=[ug1,univariate eval(p2,lvar1,lval)]
--JHD        lcpol:=[leadingCoefficient univariate(g1,x),
--JHD                leadingCoefficient univariate(p2,x)]
--JHD        while true repeat
--JHD          d:SUPR:=gcd(cons(ul,ulist))
--JHD          if degree d =0 then return gd1
--JHD          lquo:=(ul exquo d)::SUPR
--JHD          if degree lquo ^=0 then
--JHD            lgcd:=gcd(cons(leadingCoefficient univariate(l,x),lcpol))
--JHD            gd2:=lift(l,d,lquo,lgcd,lvar,ldeg,lval)
--JHD            l:=gd2
--JHD            ul:=univariate(eval(l,lvar1,lval))
--JHD            dl:=degree ul
--JHD          gd1:=gd1*gd2
--JHD          ulist:=[(uf exquo d)::SUPR for uf in ulist]
--JHD
--JHD -- we suppose that the poly have the same mainvar, deg p1<deg p2 and the
--JHD -- polys primitive
--JHD      internal(p1:P,p2:P,x:OV) : P ==
--JHD        lvar:List OV:=sort(#1>#2,setUnion(variables p1,variables p2))
--JHD        d1:=degree(p1,x)
--JHD        d2:=degree(p2,x)
--JHD        result: P:=localgcd(p1,p2,lvar).locgcd
--JHD        -- special cases
--JHD        result=1 => 1$P
--JHD        (dr:=degree(result,x))=d1 or dr=d2  => result
--JHD        while failtest(result,p1,p2) repeat
--JHD          SAY$Lisp  "retrying gcd"
--JHD          result:=localgcd(p1,p2,lvar).locgcd
--JHD        result
--JHD
--JHD    --local function for the gcd : it returns the evaluation point too
--JHD      localgcd(p1:P,p2:P,lvar:List(OV)) : LGcd ==
--JHD        x:OV:=lvar.first
--JHD        uterm:=chooseVal(p1,p2,lvar)
--JHD        listpol:= uterm.lpol
--JHD        ud:=listpol.first
--JHD        dd:= degree ud
```

```
--JHD
--JHD        --the univariate gcd is 1
--JHD        dd=0 => [1$P,uterm.lint]$LGcd
--JHD
--JHD        --one of the polynomials is the gcd
--JHD        dd=degree(p1,x) or dd=degree(p2,x) =>
--JHD                                          [uterm.mpol,uterm.lint]$LGcd
--JHD        ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))
--JHD
--JHD       -- if there is a polynomial g s.t. g/gcd and gcd are coprime ...
--JHD        -- I can lift
--JHD        (h:=lift?(p1,p2,uterm,ldeg,lvar)) case "failed" =>
--JHD          [notCoprime(p1,p2,ldeg,lvar),uterm.lint]$LGcd
--JHD        [h::P,uterm.lint]$LGcd
--JHD
--JHD
--JHD  -- content, internal functions return the poly if it is a monomial
--JHD      monomContent(p:P,var:OV):P ==
--JHD        ground? p => 1$P
--JHD        md:= minimumDegree(p,var)
--JHD        ((var::P)**md)*(gcd sort(better,coefficients univariate(p,var)))

      monomContentSup(u:SUPP):SUPP ==
        degree(u) = 0$NonNegativeInteger => 1$SUPP
        md:= minimumDegree u
        gcd(sort(better,coefficients u)) * monomial(1$P,md)$SUPP

--JHD  -- change the polynomials to have positive lc
--JHD      abs(p:P): P == unitNormal(p).canonical

  -- Ordering for gcd purposes
      better(p1:P,p2:P):Boolean ==
        ground? p1 => true
        ground? p2 => false
        degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

   -- PRS algorithm
   -- gcdPrs(p1:P,p2:P,d:NNI,var:OV):Union(P,"failed") ==
   --   u1:= univariate(p1,var)
   --   u2:= univariate(p2,var)
   --   finished:Boolean:= false
   --   until finished repeat
   --     dd:NNI:=(degree u1 - degree u2)::NNI
   --     lc1:SUPP:=leadingCoefficient u2 * reductum u1
   --     lc2:SUPP:=leadingCoefficient u1 * reductum u2
   --     u3:SUPP:= primitate((lc1-lc2)*monomial(1$P,dd))$%
   --     (d3:=degree(u3)) <= d => finished:= true
   --     u1:= u2
   --     u2:= u3
   --     if d3 > degree(u1) then (u1,u2):= (u2,u1)
```

```
   --   g:= (u2 exquo u3)
   --   g case SUPP => abs multivariate(u3,var)
   --   "failed"

  -- Gcd between polynomial p1 and p2 with
  -- mainVariable p1 < x=mainVariable p2
--JHD      gcdTermList(p1:P,p2:P) : P ==
--JHD        termList:=sort(better,
--JHD           cons(p1,coefficients univariate(p2,(mainVariable p2)::OV)))
--JHD        q:P:=termList.first
--JHD        for term in termList.rest until q = 1$P repeat q:= gcd(q,term)
--JHD        q
--JHD
--JHD  -- Gcd between polynomials with the same mainVariable
--JHD      gcdSameMainvar(p1:P,p2:P,mvar:OV): P ==
--JHD        if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
--JHD        (p1 exquo p2) case P => abs p2
--JHD        c1:= monomContent(p1,mvar)$%
--JHD        c1 = p1 => gcdMonom(p1,p2,mvar)
--JHD        c2:= monomContent(p2,mvar)$%
--JHD        c2 = p2 => gcdMonom(p2,p1,mvar)
--JHD        p1:= (p1 exquo c1)::P
--JHD        p2:= (p2 exquo c2)::P
--JHD        if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
--JHD        (p1 exquo p2) case P => abs(p2) * gcd(c1,c2)
--JHD        abs(gcdPrim(p1,p2,mvar)) * gcd(c1,c2)
--JHD
--JHD    --  make the polynomial primitive with respect to var
--JHD      primitate(p:P,var:OV):P == (p exquo monomContent(p,var))::P
--JHD
--JHD      primitate(u:SUPP):SUPP == (u exquo monomContentSup u)::SUPP
--JHD
--JHD    -- gcd between primitive polynomials with the same mainVariable
--JHD      gcdPrim(p1:P,p2:P,mvar:OV):P ==
--JHD        vars:= removeDuplicates append(variables p1,variables p2)
--JHD        #vars=1 => multivariate(gcd(univariate p1,univariate p2),mvar)
--JHD        vars:=delete(vars,position(mvar,vars))
--JHD        --d:= degModGcd(p1,p2,mvar,vars)
--JHD        --d case "failed" => internal(p2,p1,mvar)
--JHD        --deg:= d:NNI
--JHD        --deg = 0$NNI => 1$P
--JHD        --deg = degree(p1,mvar) =>
--JHD        --  (p2 exquo p1) case P => abs(p1)  -- already know that
--JHD                                          -- ^(p1 exquo p2)
--JHD        --  internal(p2,p1,mvar)
--JHD        --cheapPrs?(p1,p2,deg,mvar) =>
--JHD        --  g:= gcdPrs(p1,p2,deg,mvar)
--JHD        --  g case P => g::P
--JHD        --  internal(p2,p1,mvar)
--JHD        internal(p2,p1,mvar)
```

```
--JHD
--JHD   -- gcd between a monomial and a polynomial
--JHD      gcdMonom(m:P,p:P,var:OV):P ==
--JHD        ((var::P) ** min(minimumDegree(m,var),minimumDegree(p,var))) *
--JHD          gcdTermList(leadingCoefficient(univariate(m,var)),p)
--JHD
--JHD    --If there is a pol s.t. pol/gcd and gcd are coprime I can lift
--JHD      lift?(p1:P,p2:P,uterm:UTerm,ldeg:List NNI,
--JHD                      lvar:List OV) : Union("failed",P) ==
--JHD        x:OV:=lvar.first
--JHD        leadpol:Boolean:=false
--JHD        (listpol,lval):=(uterm.lpol,uterm.lint)
--JHD        d:=listpol.first
--JHD        listpol:=listpol.rest
--JHD        nolift:Boolean:=true
--JHD        for uf in listpol repeat
--JHD              --note uf and d not necessarily primitive
--JHD          degree gcd(uf,d) =0 => nolift:=false
--JHD        nolift => "failed"
--JHD        f:P:=([p1,p2]$List(P)).(position(uf,listpol))
--JHD        lgcd:=gcd(leadingCoefficient univariate(p1,x),
--JHD                  leadingCoefficient univariate(p2,x))
--JHD        lift(f,d,uf,lgcd,lvar,ldeg,lval)
--JHD
--JHD   -- interface with the general "lifting" function
--JHD      lift(f:P,d:SUPR,uf:SUPR,lgcd:P,lvar:List OV,
--JHD                         ldeg:List NNI,lval:List R):P ==
--JHD        x:OV:=lvar.first
--JHD        leadpol:Boolean:=false
--JHD        lcf:P
--JHD        lcf:=leadingCoefficient univariate(f,x)
--JHD        df:=degree(f,x)
--JHD        leadlist:List(P):=[]
--JHD
--JHD        if lgcd^=1$P then
--JHD          leadpol:=true
--JHD          f:=lgcd*f
--JHD          ldeg:=[n0+n1 for n0 in ldeg for n1 in degree(lgcd,lvar)]
--JHD          lcd:R:=leadingCoefficient d
--JHD          if ground? lgcd then d:=((retract lgcd) *d exquo lcd)::SUPR
--JHD          else d:=(retract(eval(lgcd,lvar.rest,lval)) * d exquo lcd)::SUPR
--JHD          uf:=lcd*uf
--JHD        leadlist:=[lgcd,lcf]
--JHD        lg:=imposelc([d,uf],lvar,lval,leadlist)
--JHD        plist:=lifting(univariate(f,x),lvar,lg,lval,leadlist,ldeg)::List P
--JHD        (p0:P,p1:P):=(plist.first,plist.2)
--JHD        if univariate eval(p0,rest lvar,lval) ^= lg.first then
--JHD           (p0,p1):=(p1,p0)
--JHD        ^leadpol => p0
--JHD        cprim:=contprim([p0])
```

```
--JHD        cprim.first.prim
--JHD
--JHD  -- Gcd for two multivariate polynomials
--JHD      gcd(p1:P,p2:P) : P ==
--JHD        (p1:= abs(p1)) = (p2:= abs(p2)) => p1
--JHD        ground? p1 =>
--JHD          p1 = 1$P => p1
--JHD          p1 = 0$P => p2
--JHD          ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
--JHD          gcdTermList(p1,p2)
--JHD        ground? p2 =>
--JHD          p2 = 1$P => p2
--JHD          p2 = 0$P => p1
--JHD          gcdTermList(p2,p1)
--JHD        mv1:= mainVariable(p1)::OV
--JHD        mv2:= mainVariable(p2)::OV
--JHD        mv1 = mv2 => gcdSameMainvar(p1,p2,mv1)
--JHD        mv1 < mv2 => gcdTermList(p1,p2)
--JHD        gcdTermList(p2,p1)
--JHD
--JHD  -- Gcd for a list of multivariate polynomials
--JHD      gcd(listp:List P) : P ==
--JHD        lf:=sort(better,listp)
--JHD        f:=lf.first
--JHD        for g in lf.rest repeat
--JHD          f:=gcd(f,g)
--JHD          if f=1$P then return f
--JHD        f
--JHD   -- Gcd and cofactors for a list of polynomials
--JHD      gcdcofact(listp : List P) : List P ==
--JHD         h:=gcd listp
--JHD         cons(h,[(f exquo h) :: P  for f in listp])
--JHD
--JHD   -- Gcd for primitive polynomials
--JHD      gcdprim(p1:P,p2:P):P ==
--JHD        (p1:= abs(p1)) = (p2:= abs(p2)) => p1
--JHD        ground? p1 =>
--JHD          ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
--JHD          p1 = 0$P => p2
--JHD          1$P
--JHD        ground? p2 =>
--JHD          p2 = 0$P => p1
--JHD          1$P
--JHD        mv1:= mainVariable(p1)::OV
--JHD        mv2:= mainVariable(p2)::OV
--JHD        mv1 = mv2 =>
--JHD          md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv1))
--JHD          mp:=1$P
--JHD          if md>1 then
--JHD            mp:=(mv1::P)**md
```

```
--JHD            p1:=(p1 exquo mp)::P
--JHD            p2:=(p2 exquo mp)::P
--JHD          mp*gcdPrim(p1,p2,mv1)
--JHD        1$P
--JHD
--JHD  -- Gcd for a list of primitive multivariate polynomials
--JHD      gcdprim(listp:List P) : P ==
--JHD        lf:=sort(better,listp)
--JHD        f:=lf.first
--JHD        for g in lf.rest repeat
--JHD          f:=gcdprim(f,g)
--JHD          if f=1$P then return f
--JHD        f
--JHD   -- Gcd and cofactors for a list of primitive polynomials
--JHD      gcdcofactprim(listp : List P) : List P ==
--JHD         h:=gcdprim listp
--JHD         cons(h,[(f exquo h) :: P  for f in listp])
--JHD
--JHD   -- content of a polynomial (with respect to its main var)
--JHD      content(f:P):P ==
--JHD        ground? f => f
--JHD        x:OV:=(mainVariable f)::OV
--JHD        gcd sort(better,coefficients univariate(f,x))
--JHD
--JHD   -- contents of a list of polynomials
--JHD      content(listf:List P) : List P == [content f for f in listf]
--JHD
--JHD   -- contents and primitive parts of a list  of polynomials
--JHD      contprim(listf:List P) : List ContPrim ==
--JHD        prelim :List P := content listf
--JHD        [[q,(f exquo q)::P]$ContPrim for q in prelim for f in listf]
--JHD
```

———-

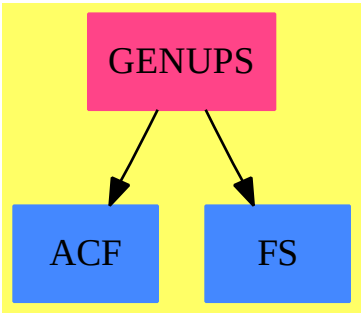
— GENPGCD.dotabb —

```
"GENPGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENPGCD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GENPGCD" -> "PFECAT"
```

———-

8.10 package GENUPS GenerateUnivariatePowerSeries

8.10.1 GenerateUnivariatePowerSeries (GENUPS)



Exports:
laurent puiseux series taylor

— package GENUPS GenerateUnivariatePowerSeries —

```
)abbrev package GENUPS GenerateUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 29 April 1990
++ Date Last Updated: 31 May 1990
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: series, Taylor, Laurent, Puiseux
++ Examples:
++ References:
++ Description:
++ \spadtype{GenerateUnivariatePowerSeries} provides functions that create
++ power series from explicit formulas for their \spad{n}th coefficient.

GenerateUnivariatePowerSeries(R,FE): Exports == Implementation where
  R  : Join(IntegralDomain,OrderedSet,RetractableTo Integer,_
            LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
            FunctionSpace R)
  ANY1 ==> AnyFunctions1
  EQ   ==> Equation
  I    ==> Integer
  NNI  ==> NonNegativeInteger
  RN   ==> Fraction Integer
  SEG  ==> UniversalSegment
  ST   ==> Stream
```

```
SY   ==> Symbol
UTS  ==> UnivariateTaylorSeries
ULS  ==> UnivariateLaurentSeries
UPXS ==> UnivariatePuiseuxSeries

Exports ==> with
  taylor: (I -> FE,EQ FE) -> Any
    ++ \spad{taylor(n +-> a(n),x = a)} returns
    ++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.
  taylor: (FE,SY,EQ FE) -> Any
    ++ \spad{taylor(a(n),n,x = a)} returns \spad{sum(n = 0..,a(n)*(x-a)**n)}.
  taylor: (I -> FE,EQ FE,SEG NNI) -> Any
    ++ \spad{taylor(n +-> a(n),x = a,n0..)} returns
    ++ \spad{sum(n=n0..,a(n)*(x-a)**n)};
    ++ \spad{taylor(n +-> a(n),x = a,n0..n1)} returns
    ++ \spad{sum(n = n0..,a(n)*(x-a)**n)}.
  taylor: (FE,SY,EQ FE,SEG NNI) -> Any
    ++ \spad{taylor(a(n),n,x = a,n0..)} returns
    ++ \spad{sum(n = n0..,a(n)*(x-a)**n)};
    ++ \spad{taylor(a(n),n,x = a,n0..n1)} returns
    ++ \spad{sum(n = n0..,a(n)*(x-a)**n)}.

  laurent: (I -> FE,EQ FE,SEG I) -> Any
    ++ \spad{laurent(n +-> a(n),x = a,n0..)} returns
    ++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
    ++ \spad{laurent(n +-> a(n),x = a,n0..n1)} returns
    ++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.
  laurent: (FE,SY,EQ FE,SEG I) -> Any
    ++ \spad{laurent(a(n),n,x=a,n0..)} returns
    ++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
    ++ \spad{laurent(a(n),n,x=a,n0..n1)} returns
    ++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

  puiseux: (RN -> FE,EQ FE,SEG RN,RN) -> Any
    ++ \spad{puiseux(n +-> a(n),x = a,r0..,r)} returns
    ++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
    ++ \spad{puiseux(n +-> a(n),x = a,r0..r1,r)} returns
    ++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.
  puiseux: (FE,SY,EQ FE,SEG RN,RN) -> Any
    ++ \spad{puiseux(a(n),n,x = a,r0..,r)} returns
    ++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
    ++ \spad{puiseux(a(n),n,x = a,r0..r1,r)} returns
    ++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

  series: (I -> FE,EQ FE) -> Any
    ++ \spad{series(n +-> a(n),x = a)} returns
    ++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.
  series: (FE,SY,EQ FE) -> Any
    ++ \spad{series(a(n),n,x = a)} returns
    ++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.
```

```
  series: (I -> FE,EQ FE,SEG I) -> Any
    ++ \spad{series(n +-> a(n),x = a,n0..)} returns
    ++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
    ++ \spad{series(n +-> a(n),x = a,n0..n1)} returns
    ++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.
  series: (FE,SY,EQ FE,SEG I) -> Any
    ++ \spad{series(a(n),n,x=a,n0..)} returns
    ++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
    ++ \spad{series(a(n),n,x=a,n0..n1)} returns
    ++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.
  series: (RN -> FE,EQ FE,SEG RN,RN) -> Any
    ++ \spad{series(n +-> a(n),x = a,r0..,r)} returns
    ++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
    ++ \spad{series(n +-> a(n),x = a,r0..r1,r)} returns
    ++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.
  series: (FE,SY,EQ FE,SEG RN,RN) -> Any
    ++ \spad{series(a(n),n,x = a,r0..,r)} returns
    ++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
    ++ \spad{series(a(n),n,x = a,r0..r1,r)} returns
    ++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

Implementation ==> add

  genStream: (I -> FE,I) -> ST FE
  genStream(f,n) == delay concat(f(n),genStream(f,n + 1))

  genFiniteStream: (I -> FE,I,I) -> ST FE
  genFiniteStream(f,n,m) == delay
    n > m => empty()
    concat(f(n),genFiniteStream(f,n + 1,m))

  taylor(f,eq) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    coerce(series(genStream(f,0))$UTS(FE,x,a))$ANY1(UTS(FE,x,a))

  taylor(an:FE,n:SY,eq:EQ FE) ==
    taylor((i:I):FE +-> eval(an,(n::FE) = (i::FE)),eq)

  taylor(f:I -> FE,eq:EQ FE,seg:SEG NNI) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    hasHi seg =>
      n0 := lo seg; n1 := hi seg
      if n1 < n0 then (n0,n1) := (n1,n0)
      uts := series(genFiniteStream(f,n0,n1))$UTS(FE,x,a)
      uts := uts * monomial(1,n0)$UTS(FE,x,a)
      coerce(uts)$ANY1(UTS(FE,x,a))
```

```
    n0 := lo seg
    uts := series(genStream(f,n0))$UTS(FE,x,a)
    uts := uts * monomial(1,n0)$UTS(FE,x,a)
    coerce(uts)$ANY1(UTS(FE,x,a))

  taylor(an,n,eq,seg) ==
    taylor((i:I):FE +-> eval(an,(n::FE) = (i::FE)),eq,seg)

  laurent(f,eq,seg) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    hasHi seg =>
      n0 := lo seg; n1 := hi seg
      if n1 < n0 then (n0,n1) := (n1,n0)
      uts := series(genFiniteStream(f,n0,n1))$UTS(FE,x,a)
      coerce(laurent(n0,uts)$ULS(FE,x,a))$ANY1(ULS(FE,x,a))
    n0 := lo seg
    uts := series(genStream(f,n0))$UTS(FE,x,a)
    coerce(laurent(n0,uts)$ULS(FE,x,a))$ANY1(ULS(FE,x,a))

  laurent(an,n,eq,seg) ==
    laurent((i:I):FE +-> eval(an,(n::FE) = (i::FE)),eq,seg)

  modifyFcn:(RN -> FE,I,I,I,I) -> FE
  modifyFcn(f,n0,nn,q,m) == (zero?((m - n0) rem nn) => f(m/q); 0)

  puiseux(f,eq,seg,r) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "puiseux: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    not positive? r => error "puiseux: last argument must be positive"
    hasHi seg =>
      r0 := lo seg; r1 := hi seg
      if r1 < r0 then (r0,r1) := (r1,r0)
      p0 := numer r0; q0 := denom r0
      p1 := numer r1; q1 := denom r1
      p2 := numer r; q2 := denom r
      q := lcm(lcm(q0,q1),q2)
      n0 := p0 * (q quo q0); n1 := p1 * (q quo q1)
      nn := p2 * (q quo q2)
      ulsUnion :=
        laurent((i:I):FE+->modifyFcn(f,n0,nn,q,i),eq,segment(n0,n1))
      uls := retract(ulsUnion)$ANY1(ULS(FE,x,a))
      coerce(puiseux(1/q,uls)$UPXS(FE,x,a))$ANY1(UPXS(FE,x,a))
    p0 := numer(r0 := lo seg); q0 := denom r0
    p2 := numer r; q2 := denom r
    q := lcm(q0,q2)
    n0 := p0 * (q quo q0); nn := p2 * (q quo q2)
    ulsUnion :=
```

```
        laurent((i:I):FE+->modifyFcn(f,n0,nn,q,i),eq,segment n0)
      uls := retract(ulsUnion)$ANY1(ULS(FE,x,a))
      coerce(puiseux(1/q,uls)$UPXS(FE,x,a))$ANY1(UPXS(FE,x,a))

    puiseux(an,n,eq,r0,m) ==
      puiseux((r:RN):FE+->eval(an,(n::FE) = (r::FE)),eq,r0,m)

    series(f:I -> FE,eq:EQ FE) == puiseux(r+->f(numer r),eq,segment 0,1)
    series(an:FE,n:SY,eq:EQ FE) == puiseux(an,n,eq,segment 0,1)
    series(f:I -> FE,eq:EQ FE,seg:SEG I) ==
      ratSeg : SEG RN := map(x+->x::RN,seg)$UniversalSegmentFunctions2(I,RN)
      puiseux((r:RN):FE+->f(numer r),eq,ratSeg,1)
    series(an:FE,n:SY,eq:EQ FE,seg:SEG I) ==
      ratSeg : SEG RN := map(i+->i::RN,seg)$UniversalSegmentFunctions2(I,RN)
      puiseux(an,n,eq,ratSeg,1)
    series(f:RN -> FE,eq:EQ FE,seg:SEG RN,r:RN) == puiseux(f,eq,seg,r)
    series(an:FE,n:SY,eq:EQ FE,seg:SEG RN,r:RN) == puiseux(an,n,eq,seg,r)
```

———-

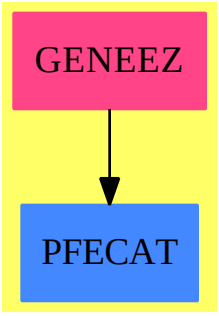
— **GENUPS.dotabb** —

```
"GENUPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENUPS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"GENUPS" -> "ACF"
"GENUPS" -> "FS"
```

———-

8.11 package GENEEZ GenExEuclid

8.11.1 GenExEuclid (GENEEZ)



Exports:
compBound reduction solveid tablePow testModulus

— package GENEEZ GenExEuclid —

```
)abbrev package GENEEZ GenExEuclid
++ Author : P.Gianni.
++ Date Created: January 1990
++ Description:
++ The equation \spad{Af+Bg=h} and its generalization to n polynomials
++ is solved for solutions over the R, euclidean domain.
++ A table containing the solutions of \spad{Af+Bg=x**k} is used.
++ The operations are performed modulus a prime
++ which are in principle big enough,
++ but the solutions are tested and, in case of failure, a hensel
++ lifting process is used to get to the right solutions.
++ It will be used in the factorization of multivariate polynomials
++ over finite field, with \spad{R=F[x]}.

GenExEuclid(R,BP) : C == T
 where
  R   :   EuclideanDomain
  PI  ==> PositiveInteger
  NNI ==> NonNegativeInteger
  BP  :   UnivariatePolynomialCategory R
  L   ==> List

  C == with
       reduction: (BP,R) -> BP
         ++ reduction(p,prime) reduces the polynomial p modulo prime of R.
         ++ Note that this function is exported only because it's conditional.
       compBound: (BP,L BP) -> NNI
         ++ compBound(p,lp)
         ++ computes a bound for the coefficients of the solution
         ++ polynomials.
         ++ Given a polynomial right hand side p,
         ++ and a list lp of left hand side polynomials.
         ++ Exported because it depends on the valuation.
       tablePow :    (NNI,R,L BP)     -> Union(Vector(L BP),"failed")
         ++ tablePow(maxdeg,prime,lpol) constructs the table with the
         ++ coefficients of the Extended Euclidean Algorithm for lpol.
         ++ Here the right side is \spad{x**k}, for k less or equal to maxdeg.
         ++ The operation returns "failed" when the elements
         ++ are not coprime modulo prime.
       solveid  : (BP,R,Vector L BP) -> Union(L BP,"failed")
         ++ solveid(h,table) computes the coefficients of the
         ++ extended euclidean algorithm for a list of polynomials
         ++ whose tablePow is table and with right side h.
```

```
      testModulus : (R, L BP)    -> Boolean
        ++ testModulus(p,lp) returns true if the the prime p
        ++ is valid for the list of polynomials lp, i.e. preserves
        ++ the degree and they remain relatively prime.

 T == add
   if R has multiplicativeValuation then
     compBound(m:BP,listpolys:L BP) : NNI ==
       ldeg:=[degree f for f in listpolys]
       n:NNI:= (+/[df for df in ldeg])
       normlist:=[ +/[euclideanSize(u)**2 for u in coefficients f]
                       for f in listpolys]
       nm:= +/[euclideanSize(u)**2 for u in coefficients m]
       normprod := */[g**((n-df)::NNI) for g in normlist for df in ldeg]
       2*(approxSqrt(normprod * nm)$IntegerRoots(Integer))::NNI
   else if R has additiveValuation then
     -- a fairly crude Hadamard-style bound for the solution
     -- based on regarding the problem as a system of linear equations.
     compBound(m:BP,listpolys:L BP) : NNI ==
       "max"/[euclideanSize u for u in coefficients m] +
         +/["max"/[euclideanSize u for u in coefficients p]
            for p in listpolys]
   else
     compBound(m:BP,listpolys:L BP) : NNI ==
       error "attempt to use compBound without a well-understood valuation"
   if R has IntegerNumberSystem then
     reduction(u:BP,p:R):BP ==
       p = 0 => u
       map(x +-> symmetricRemainder(x,p),u)
   else reduction(u:BP,p:R):BP ==
       p = 0 => u
       map(x +-> x rem p,u)

   merge(p:R,q:R):Union(R,"failed") ==
        p = q => p
        p = 0 => q
        q = 0 => p
        "failed"

   modInverse(c:R,p:R):R ==
       (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

   exactquo(u:BP,v:BP,p:R):Union(BP,"failed") ==
       invlcv:=modInverse(leadingCoefficient v,p)
       r:=monicDivide(u,reduction(invlcv*v,p))
       reduction(r.remainder,p) ^=0 => "failed"
       reduction(invlcv*r.quotient,p)

   FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)
```

```
--make table global variable!
table:Vector L BP
import GeneralHenselPackage(R,BP)

                  --local  functions
makeProducts :    L BP   -> L BP
liftSol: (L BP,BP,R,R,Vector L BP,BP,NNI) -> Union(L BP,"failed")

reduceList(lp:L BP,lmod:R): L FP ==[reduce(ff,lmod) for ff in lp]

coerceLFP(lf:L FP):L BP == [fm::BP for fm in lf]

liftSol(oldsol:L BP,err:BP,lmod:R,lmodk:R,
       table:Vector L BP,m:BP,bound:NNI):Union(L BP,"failed") ==
  euclideanSize(lmodk) > bound => "failed"
  d:=degree err
  ftab:Vector L FP :=
    map(x +-> reduceList(x,lmod),table)$VectorFunctions2(List BP,List FP)
  sln:L FP:=[0$FP for xx in ftab.1 ]
  for i in 0 .. d |(cc:=coefficient(err,i)) ^=0 repeat
    sln:=[slp+reduce(cc::BP,lmod)*pp
          for pp in ftab.(i+1) for slp in sln]
  nsol:=[f-lmodk*reduction(g::BP,lmod) for f in oldsol for g in sln]
  lmodk1:=lmod*lmodk
  nsol:=[reduction(slp,lmodk1) for slp in nsol]
  lpolys:L BP:=table.(#table)
  (fs:=+/[f*g for f in lpolys for g in nsol]) = m => nsol
  a:BP:=((fs-m) exquo lmodk1)::BP
  liftSol(nsol,a,lmod,lmodk1,table,m,bound)

makeProducts(listPol:L BP):L BP ==
  #listPol < 2 => listPol
  #listPol = 2 => reverse listPol
  f:= first listPol
  ll := rest listPol
  [*/ll,:[f*g for g in makeProducts ll]]

testModulus(pmod, listPol) ==
     redListPol := reduceList(listPol, pmod)
     for pol in listPol for rpol in redListPol repeat
          degree(pol) ^= degree(rpol::BP) => return false
     while not empty? redListPol repeat
         rpol := first redListPol
         redListPol := rest redListPol
         for rpol2 in redListPol repeat
            gcd(rpol, rpol2) ^= 1 => return false
     true

if R has Field then
  tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
```

```
    multiE:=multiEuclidean(listPol,1$BP)
    multiE case "failed" => "failed"
    ptable:Vector L BP :=new(mdeg+1,[])
    ptable.1:=multiE
    x:BP:=monomial(1,1)
    for i in 2..mdeg repeat ptable.i:=
        [tpol*x rem fpol for tpol in ptable.(i-1) for fpol in listPol]
    ptable.(mdeg+1):=makeProducts listPol
    ptable

  solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
        -- Actually, there's no possibility of failure
    d:=degree m
    sln:L BP:=[0$BP for xx in table.1]
    for i in 0 .. d | coefficient(m,i)^=0 repeat
      sln:=[slp+coefficient(m,i)*pp
            for pp in table.(i+1) for slp in sln]
    sln

else

  tablePow(mdeg:NNI,pmod:R,listPol:L BP) ==
    listP:L FP:= [reduce(pol,pmod) for pol in listPol]
    multiE:=multiEuclidean(listP,1$FP)
    multiE case "failed" => "failed"
    ftable:Vector L FP :=new(mdeg+1,[])
    fl:L FP:= [ff::FP for ff in multiE]
    ftable.1:=[fpol for fpol in fl]
    x:FP:=reduce(monomial(1,1),pmod)
    for i in 2..mdeg repeat ftable.i:=
        [tpol*x rem fpol for tpol in ftable.(i-1) for fpol in listP]
    ptable:= map(coerceLFP,ftable)$VectorFunctions2(List FP,List BP)
    ptable.(mdeg+1):=makeProducts listPol
    ptable

  solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
    d:=degree m
    ftab:Vector L FP:=
      map(x+->reduceList(x,pmod),table)$VectorFunctions2(List BP,List FP)
    lpolys:L BP:=table.(#table)
    sln:L FP:=[0$FP for xx in ftab.1]
    for i in 0 .. d | coefficient(m,i)^=0 repeat
      sln:=[slp+reduce(coefficient(m,i)::BP,pmod)*pp
            for pp in ftab.(i+1) for slp in sln]
    soln:=[slp::BP for slp in sln]
    (fs:=+/[f*g for f in lpolys for g in soln]) = m=> soln
    -- Compute bound
    bound:=compBound(m,lpolys)
    a:BP:=((fs-m) exquo pmod)::BP
    liftSol(soln,a,pmod,pmod,table,m,bound)
```

———-

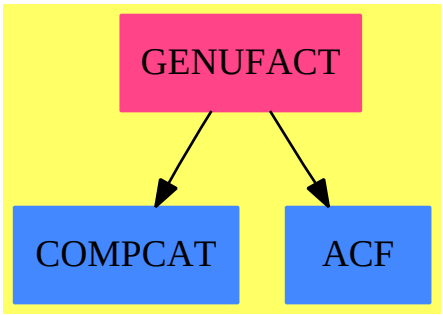
— GENEEZ.dotabb —

```
"GENEEZ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENEEZ"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GENEEZ" -> "PFECAT"
```

———-

8.12 package GENUFACT GenUFactorize

8.12.1 GenUFactorize (GENUFACT)



Exports:
factor

— package GENUFACT GenUFactorize —

```
)abbrev package GENUFACT GenUFactorize
++ Description:
++ This package provides operations for the factorization
++ of univariate polynomials with integer
++ coefficients. The factorization is done by "lifting" the
++ finite "berlekamp's" factorization

GenUFactorize(R) : public == private where
  R    :    EuclideanDomain
  PR   ==>  SparseUnivariatePolynomial R  -- with factor
            -- should be UnivariatePolynomialCategory
  NNI  ==>  NonNegativeInteger
```

```
  SUP  ==>  SparseUnivariatePolynomial


  public == with
     factor      :           PR  -> Factored PR
       ++ factor(p) returns the factorisation of p

  private == add

    -- Factorisation currently fails when algebraic extensions have multiple
    -- generators.
    factorWarning(f:OutputForm):Void ==
      import AnyFunctions1(String)
      import AnyFunctions1(OutputForm)
      outputList(["WARNING (genufact): No known algorithm to factor "::Any, _
              f::Any, _
              ", trying square-free."::Any])$OutputPackage

    factor(f:PR) : Factored PR ==
      R is Integer => (factor f)$GaloisGroupFactorizer(PR)

      R is Fraction Integer  =>
                                (factor f)$RationalFactorize(PR)

--      R has Field and R has Finite =>
      R has FiniteFieldCategory =>
                                (factor f)$DistinctDegreeFactorize(R,PR)

      R is (Complex Integer) => (factor f)$ComplexFactorization(Integer,PR)

      R is (Complex Fraction Integer) =>
                           (factor f)$ComplexFactorization(Fraction Integer,PR)

      R is AlgebraicNumber =>   (factor f)$AlgFactor(PR)

   -- following is to handle SAE
      R has generator : () -> R =>
        var := symbol(convert(generator()::OutputForm)@InputForm)
        up:=UnivariatePolynomial(var,Fraction Integer)
        R has MonogenicAlgebra(Fraction Integer, up) =>
           factor(f)$SimpleAlgebraicExtensionAlgFactor(up, R, PR)
        upp:=UnivariatePolynomial(var,Fraction Polynomial Integer)
        R has MonogenicAlgebra(Fraction Polynomial Integer, upp) =>
           factor(f)$SAERationalFunctionAlgFactor(upp, R, PR)
        factorWarning(f::OutputForm)
        squareFree f
      factorWarning(f::OutputForm)
      squareFree f
```

———-

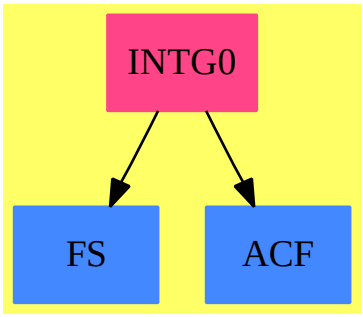
— GENUFACT.dotabb —

```
"GENUFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENUFACT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"GENUFACT" -> "COMPCAT"
"GENUFACT" -> "ACF"
```

———-

8.13 package INTG0 GenusZeroIntegration

8.13.1 GenusZeroIntegration (INTG0)



Exports:

lift	multivariate	palgLODE0	palgRDE0
palgextint0	palgint0	palglimint0	univariate

— package INTG0 GenusZeroIntegration —

```
)abbrev package INTG0 GenusZeroIntegration
++ Author: Manuel Bronstein
++ Date Created: 11 October 1988
++ Date Last Updated: 24 June 1994
++ Description:
++ Rationalization of several types of genus 0 integrands;
++ This internal package rationalises integrands on curves of the form:\br
++ \tab{5}\spad{y\^2 = a x\^2 + b x + c}\br
++ \tab{5}\spad{y\^2 = (a x + b) / (c x + d)}\br
++ \tab{5}\spad{f(x, y) = 0} where f has degree 1 in x\br
++ The rationalization is done for integration, limited integration,
```

```
++ extended integration and the risch differential equation;

GenusZeroIntegration(R, F, L): Exports == Implementation where
  R: Join(GcdDomain, RetractableTo Integer, OrderedSet, CharacteristicZero,
          LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, AlgebraicallyClosedField,
          TranscendentalFunctionCategory)
  L: SetCategory

  SY  ==> Symbol
  Q   ==> Fraction Integer
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
  UPUP ==> SparseUnivariatePolynomial RF
  IR  ==> IntegrationResult F
  LOG ==> Record(coeff:F, logand:F)
  U1  ==> Union(F, "failed")
  U2  ==> Union(Record(ratpart:F, coeff:F),"failed")
  U3  ==> Union(Record(mainpart:F, limitedlogs:List LOG), "failed")
  REC ==> Record(coeff:F, var:List K, val:List F)
  ODE ==> Record(particular: Union(F, "failed"), basis: List F)
  LODO==> LinearOrdinaryDifferentialOperator1 RF

  Exports ==> with
    palgint0   : (F, K, K, F, UP)    -> IR
      ++ palgint0(f, x, y, d, p) returns the integral of \spad{f(x,y)dx}
      ++ where y is an algebraic function of x satisfying
      ++ \spad{d(x)\^2 y(x)\^2 = P(x)}.
    palgint0   : (F, K, K, K, F, RF) -> IR
      ++ palgint0(f, x, y, z, t, c) returns the integral of \spad{f(x,y)dx}
      ++ where y is an algebraic function of x satisfying
      ++ \spad{f(x,y)dx = c f(t,y) dy}; c and t are rational functions of y.
      ++ Argument z is a dummy variable not appearing in \spad{f(x,y)}.
    palgextint0: (F, K, K, F, F, UP) -> U2
      ++ palgextint0(f, x, y, g, d, p) returns functions \spad{[h, c]} such
      ++ that \spad{dh/dx = f(x,y) - c g}, where y is an algebraic function
      ++ of x satisfying \spad{d(x)\^2 y(x)\^2 = P(x)},
      ++ or "failed" if no such functions exist.
    palgextint0: (F, K, K, F, K, F, RF) -> U2
      ++ palgextint0(f, x, y, g, z, t, c) returns functions \spad{[h, d]} such
      ++ that \spad{dh/dx = f(x,y) - d g}, where y is an algebraic function
      ++ of x satisfying \spad{f(x,y)dx = c f(t,y) dy}, and c and t are
      ++ rational functions of y.
      ++ Argument z is a dummy variable not appearing in \spad{f(x,y)}.
      ++ The operation returns "failed" if no such functions exist.
    palglimint0: (F, K, K, List F, F, UP) -> U3
      ++ palglimint0(f, x, y, [u1,...,un], d, p) returns functions
      ++ \spad{[h,[[ci, ui]]]} such that the ui's are among \spad{[u1,...,un]}
```

```
      ++ and \spad{d(h + sum(ci log(ui)))/dx = f(x,y)} if such functions exist,
      ++ and "failed" otherwise.
      ++ Argument y is an algebraic function of x satisfying
      ++ \spad{d(x)\^2y(x)\^2 = P(x)}.
    palglimint0: (F, K, K, List F, K, F, RF) -> U3
      ++ palglimint0(f, x, y, [u1,...,un], z, t, c) returns functions
      ++ \spad{[h,[[ci, ui]]]} such that the ui's are among \spad{[u1,...,un]}
      ++ and \spad{d(h + sum(ci log(ui)))/dx = f(x,y)} if such functions exist,
      ++ and "failed" otherwise.
      ++ Argument y is an algebraic function of x satisfying
      ++ \spad{f(x,y)dx = c f(t,y) dy}; c and t are rational functions of y.
    palgRDE0   : (F, F, K, K, (F, F, SY) -> U1, F, UP) -> U1
      ++ palgRDE0(f, g, x, y, foo, d, p) returns a function \spad{z(x,y)}
      ++ such that \spad{dz/dx + n * df/dx z(x,y) = g(x,y)} if such a z exists,
      ++ and "failed" otherwise.
      ++ Argument y is an algebraic function of x satisfying
      ++ \spad{d(x)\^2y(x)\^2 = P(x)}.
      ++ Argument foo, called by \spad{foo(a, b, x)}, is a function that solves
      ++ \spad{du/dx + n * da/dx u(x) = u(x)}
      ++ for an unknown \spad{u(x)} not involving y.
    palgRDE0   : (F, F, K, K, (F, F, SY) -> U1, K, F, RF) -> U1
      ++ palgRDE0(f, g, x, y, foo, t, c) returns a function \spad{z(x,y)}
      ++ such that \spad{dz/dx + n * df/dx z(x,y) = g(x,y)} if such a z exists,
      ++ and "failed" otherwise.
      ++ Argument y is an algebraic function of x satisfying
      ++ \spad{f(x,y)dx = c f(t,y) dy}; c and t are rational functions of y.
      ++ Argument \spad{foo}, called by \spad{foo(a, b, x)}, is a function that
      ++ solves \spad{du/dx + n * da/dx u(x) = u(x)}
      ++ for an unknown \spad{u(x)} not involving y.
    univariate: (F, K, K, UP) -> UPUP
      ++ univariate(f,k,k,p) \undocumented
    multivariate: (UPUP, K, F) -> F
      ++ multivariate(u,k,f) \undocumented
    lift: (UP, K) -> UPUP
      ++ lift(u,k) \undocumented
    if L has LinearOrdinaryDifferentialOperatorCategory F then
      palgLODE0  : (L, F, K, K, F, UP) -> ODE
        ++ palgLODE0(op, g, x, y, d, p) returns the solution of \spad{op f = g}.
        ++ Argument y is an algebraic function of x satisfying
        ++ \spad{d(x)\^2y(x)\^2 = P(x)}.
      palgLODE0  : (L, F, K, K, K, F, RF) -> ODE
        ++ palgLODE0(op,g,x,y,z,t,c) returns the solution of \spad{op f = g}
        ++ Argument y is an algebraic function of x satisfying
        ++ \spad{f(x,y)dx = c f(t,y) dy}; c and t are rational functions of y.

  Implementation ==> add
    import RationalIntegration(F, UP)
    import AlgebraicManipulations(R, F)
    import IntegrationResultFunctions2(RF, F)
    import ElementaryFunctionStructurePackage(R, F)
```

```
    import SparseUnivariatePolynomialFunctions2(F, RF)
    import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                                    K, R, P, F)

    mkRat    : (F, REC, List K) -> RF
    mkRatlx  : (F, K, K, F, K, RF) -> RF
    quadsubst: (K, K, F, UP) -> Record(diff:F, subs:REC, newk:List K)
    kerdiff  : (F, F) -> List K
    checkroot: (F, List K) -> F
    univ     : (F, List K, K) -> RF

    dummy := kernel(new()$SY)@K

    kerdiff(sa, a)         == setDifference(kernels sa, kernels a)
    checkroot(f, l)        == (empty? l => f; rootNormalize(f, first l))
    univ(c, l, x)          == univariate(checkroot(c, l), x)
    univariate(f, x, y, p) == lift(univariate(f, y, p), x)
    lift(p, k)             == map(x1+->univariate(x1, k), p)

    palgint0(f, x, y, den, radi) ==
      -- y is a square root so write f as f1 y + f0 and integrate separately
      ff := univariate(f, x, y, minPoly y)
      f0 := reductum ff
      pr := quadsubst(x, y, den, radi)
      map(f1+->f1(x::F), integrate(retract(f0)@RF)) +
        map(f1+->f1(pr.diff),
            integrate
              mkRat(multivariate(leadingMonomial ff,x,y::F), pr.subs, pr.newk))

-- the algebraic relation is (den * y)**2 = p  where p is a * x**2 + b * x + c
-- if p is squarefree, then parametrize in the following form:
--     u  = y - x \sqrt{a}
--     x  = (u^2 - c) / (b - 2 u \sqrt{a}) = h(u)
--     dx = h'(u) du
--     y  = (u + a h(u)) / den = g(u)
-- if a is a perfect square,
--     u  = (y - \sqrt{c}) / x
--     x  = (b - 2 u \sqrt{c}) / (u^2 - a) = h(u)
--     dx = h'(u) du
--     y  = (u h(u) + \sqrt{c}) / den = g(u)
-- otherwise.
-- if p is a square p = a t^2, then we choose only one branch for now:
--     u  = x
--     x  = u = h(u)
--     dx = du
--     y  = t \sqrt{a} / den = g(u)
-- returns [u(x,y), [h'(u), [x,y], [h(u), g(u)], l] in both cases,
-- where l is empty if no new square root was needed,
-- l := [k] if k is the new square root kernel that was created.
    quadsubst(x, y, den, p) ==
```

```
      u   := dummy::F
      b   := coefficient(p, 1)
      c   := coefficient(p, 0)
      sa  := rootSimp sqrt(a := coefficient(p, 2))
      zero?(b * b - 4 * a * c) =>    -- case where p = a (x + b/(2a))^2
        [x::F, [1, [x, y], [u, sa * (u + b / (2*a)) / eval(den,x,u)]], empty()]
      empty? kerdiff(sa, a) =>
        bm2u := b - 2 * u * sa
        q    := eval(den, x, xx := (u**2 - c) / bm2u)
        yy   := (ua := u + xx * sa) / q
        [y::F - x::F * sa, [2 * ua / bm2u, [x, y], [xx, yy]], empty()]
      u2ma:= u**2 - a
      sc  := rootSimp sqrt c
      q   := eval(den, x, xx := (b - 2 * u * sc) / u2ma)
      yy  := (ux := xx * u + sc) / q
      [(y::F - sc) / x::F, [- 2 * ux / u2ma, [x ,y], [xx, yy]], kerdiff(sc, c)]

    mkRatlx(f,x,y,t,z,dx) ==
      rat := univariate(eval(f, [x, y], [t, z::F]), z) * dx
      numer(rat) / denom(rat)

    mkRat(f, rec, l) ==
      rat:=univariate(checkroot(rec.coeff * eval(f,rec.var,rec.val), l), dummy)
      numer(rat) / denom(rat)

    palgint0(f, x, y, z, xx, dx) ==
      map(x1+->multivariate(x1, y), integrate mkRatlx(f, x, y, xx, z, dx))

    palgextint0(f, x, y, g, z, xx, dx) ==
      map(x1+->multivariate(x1, y),
            extendedint(mkRatlx(f,x,y,xx,z,dx), mkRatlx(g,x,y,xx,z,dx)))

    palglimint0(f, x, y, lu, z, xx, dx) ==
      map(x1+->multivariate(x1, y), limitedint(mkRatlx(f, x, y, xx, z, dx),
                           [mkRatlx(u, x, y, xx, z, dx) for u in lu]))

    palgRDE0(f, g, x, y, rischde, z, xx, dx) ==
      (u := rischde(eval(f, [x, y], [xx, z::F]),
                    multivariate(dx, z) * eval(g, [x, y], [xx, z::F]),
                        symbolIfCan(z)::SY)) case "failed" => "failed"
      eval(u::F, z, y::F)

-- given p = sum_i a_i(X) Y^i, returns  sum_i a_i(x) y^i
    multivariate(p, x, y) ==
      (map((x1:RF):F+->multivariate(x1, x),
           p)$SparseUnivariatePolynomialFunctions2(RF, F))
              (y)

    palgextint0(f, x, y, g, den, radi) ==
      pr := quadsubst(x, y, den, radi)
```

```
    map(f1+->f1(pr.diff),
        extendedint(mkRat(f, pr.subs, pr.newk), mkRat(g, pr.subs, pr.newk)))

  palglimint0(f, x, y, lu, den, radi) ==
    pr := quadsubst(x, y, den, radi)
    map(f1+->f1(pr.diff),
       limitedint(mkRat(f, pr.subs, pr.newk),
                  [mkRat(u, pr.subs, pr.newk) for u in lu]))

  palgRDE0(f, g, x, y, rischde, den, radi) ==
    pr := quadsubst(x, y, den, radi)
    (u := rischde(checkroot(eval(f, pr.subs.var, pr.subs.val), pr.newk),
                  checkroot(pr.subs.coeff * eval(g, pr.subs.var, pr.subs.val),
                            pr.newk), symbolIfCan(dummy)::SY)) case "failed"
                                  => "failed"
    eval(u::F, dummy, pr.diff)

  if L has LinearOrdinaryDifferentialOperatorCategory F then
    import RationalLODE(F, UP)

    palgLODE0(eq, g, x, y, den, radi) ==
      pr := quadsubst(x, y, den, radi)
      d := monomial(univ(inv(pr.subs.coeff), pr.newk, dummy), 1)$LODO
      di:LODO := 1                  -- will accumulate the powers of d
      op:LODO := 0                  -- will accumulate the new LODO
      for i in 0..degree eq repeat
        op := op + univ(eval(coefficient(eq, i), pr.subs.var, pr.subs.val),
                        pr.newk, dummy) * di
        di := d * di
      rec := ratDsolve(op,univ(eval(g,pr.subs.var,pr.subs.val),pr.newk,dummy))
      bas:List(F) := [b(pr.diff) for b in rec.basis]
      rec.particular case "failed" => ["failed", bas]
      [((rec.particular)::RF) (pr.diff), bas]

    palgLODE0(eq, g, x, y, kz, xx, dx) ==
      d := monomial(univariate(inv multivariate(dx, kz), kz), 1)$LODO
      di:LODO := 1                  -- will accumulate the powers of d
      op:LODO := 0                  -- will accumulate the new LODO
      lk:List(K) := [x, y]
      lv:List(F) := [xx, kz::F]
      for i in 0..degree eq repeat
        op := op + univariate(eval(coefficient(eq, i), lk, lv), kz) * di
        di := d * di
      rec := ratDsolve(op, univariate(eval(g, lk, lv), kz))
      bas:List(F) := [multivariate(b, y) for b in rec.basis]
      rec.particular case "failed" => ["failed", bas]
      [multivariate((rec.particular)::RF, y), bas]
```

———-

— **INTG0.dotabb** —

```
"INTG0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTG0"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"INTG0" -> "FS"
"INTG0" -> "ACF"
```

———-

8.14 package GDRAW GnuDraw

— **GnuDraw.input** —

```
)set break resume
)sys rm -f GnuDraw.output
)spool GnuDraw.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show GnuDraw
--R
--R GnuDraw  is a package constructor
--R Abbreviation for GnuDraw is GDRAW
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GDRAW
--R
--R------------------------------ Operations --------------------------------
--R gnuDraw : (Expression(Float),SegmentBinding(Float),String,List(DrawOption)) -> Void
--R gnuDraw : (Expression(Float),SegmentBinding(Float),String) -> Void
--R gnuDraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String,List(DrawOpt
--R gnuDraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String) -> Void
--R
--E 1

)spool
)lisp (bye)
```

———-

— **GnuDraw.help** —

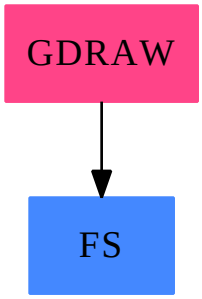
```
=======================================================================
```

```
GnuDraw examples
================================================================================

See Also:
o )show GnuDraw
```

———-

8.14.1 GnuDraw (GDRAW)



Exports:
GospersMethod

— **package GDRAW GnuDraw** —

```
)abbrev package GDRAW GnuDraw
++ Author: Bill Page and David Cyganski
++ Date: June 25, 2008
++ Description:
++ This package provides support for gnuplot. These routines
++ generate output files contain gnuplot scripts that may be
++ processed directly by gnuplot. This is especially convenient
++ in the axiom-wiki environment where gnuplot is called from
++ LaTeX via gnuplottex.
GnuDraw(): Exports == Implementation where

 EF    ==> Expression Float
 SBF   ==> SegmentBinding Float
 DROP  ==> DrawOption
 DROP0 ==> DrawOptionFunctions0
 STR   ==> String

 Exports ==> with
```

```
 gnuDraw:(EF, SBF, STR, List DROP)->Void
   ++ \spad{gnuDraw} provides 2d plotting with options
   ++
   ++X gnuDraw(D(cos(exp(z))/exp(z^2),z),z=-5..5,"out2d.dat",title=="out2d")
   ++X )sys gnuplot -persist out2d.dat

 gnuDraw:(EF, SBF, STR)->Void
   ++ \spad{gnuDraw} provides 2d plotting, default options
   ++
   ++X gnuDraw(D(cos(exp(z))/exp(z^2),z),z=-5..5,"out2d.dat")
   ++X )sys gnuplot -persist out2d.dat

 gnuDraw:(EF, SBF, SBF, STR, List DROP)->Void
   ++ \spad{gnuDraw} provides 3d surface plotting with options
   ++
   ++X gnuDraw(sin(x)*cos(y),x=-6..4,y=-4..6,"out3d.dat",title=="out3d")
   ++X )sys gnuplot -persist out3d.dat

 gnuDraw:(EF, SBF, SBF, STR)->Void
   ++ \spad{gnuDraw} provides 3d surface plotting, default options
   ++
   ++X gnuDraw(sin(x)*cos(y),x=-6..4,y=-4..6,"out3d.dat")
   ++X )sys gnuplot -persist out3d.dat

Implementation ==> add
 -- 2-d plotting
 gnuDraw(f:EF,segbind:SBF,filename:STR,opts:List DROP):Void ==
   import TwoDimensionalViewport, GraphImage, TopLevelDrawFunctions EF
   f1:TextFile:=open(filename::FileName,"output")
   -- handle optional parameters
   writeLine!(f1,concat(["set title _"",title(opts,"")$DROPO,"_""]))
   writeLine!(f1,"plot '-' title '' lw 3 with lines")
   -- extract data as List List Point DoubleFloat
   p2:=pointLists(getGraph(draw(f, segbind),1));
   for p1 in p2 repeat
     for p in p1 repeat
       writeLine!(f1,concat([unparse(convert(p.1)@InputForm)," ",
                             unparse(convert(p.2)@InputForm)]))
     writeLine!(f1); -- blank line need to mark a "branch"
   close! f1

 -- default title is ""
 gnuDraw(f:EF,segbind:SBF,filename:STR):Void ==
   gnuDraw(f,segbind,filename,[title("")$DROP])

 -- 3-d plotting
 gnuDraw(f:EF,segbind1:SBF,segbind2:SBF,filename:STR,opts:List DROP):Void ==
   import SubSpace, ThreeSpace DoubleFloat, TopLevelDrawFunctions EF
   f1:TextFile:=open(filename::FileName,"output")
```

```
    -- process optional parameters
    writeLine!(f1,concat(["set title _"",title(opts,"")$DROP0,"_""]))
    writeLine!(f1,"splot '-' title '' with pm3d")
    -- extract data as List List Point DoubleFloat
    p2:=mesh(subspace(draw(f, segbind1, segbind2)));
    for p1 in p2 repeat
      for p in p1 repeat
        writeLine!(f1,concat([unparse(convert(p.1)@InputForm)," ",
                              unparse(convert(p.2)@InputForm)," ",
                              unparse(convert(p.3)@InputForm)]))
      writeLine!(f1); -- blank line need to mark a "branch"
    close! f1

  -- default title is ""
  gnuDraw(f:EF,segbind1:SBF, segbind2:SBF, filename:STR):Void ==
    gnuDraw(f,segbind1,segbind2,filename,[title("")$DROP])
```

———-

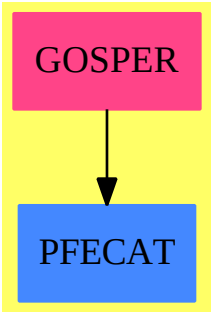
— **GDRAW.dotabb** —

```
"GDRAW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GDRAW"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"GDRAW" -> "FS"
```

———-

8.15 package GOSPER GosperSummationMethod

8.15.1 GosperSummationMethod (GOSPER)



Exports:
GospersMethod

— package GOSPER GosperSummationMethod —

```
)abbrev package GOSPER GosperSummationMethod
++ Author: SMW
++ Date Created: ???
++ Date Last Updated: 19 August 1991
++ Description:
++ Gosper's summation algorithm.

GosperSummationMethod(E, V, R, P, Q): Exports == Impl where
    E: OrderedAbelianMonoidSup
    V: OrderedSet
    R: IntegralDomain
    P: PolynomialCategory(R, E, V)
    Q: Join(RetractableTo Fraction Integer, Field with
              (coerce: P -> %; numer : % -> P; denom : % -> P))

    I   ==> Integer
    RN  ==> Fraction I
    PQ  ==> SparseMultivariatePolynomial(RN, V)
    RQ  ==> Fraction PQ

    Exports ==> with
        GospersMethod: (Q, V, () -> V) -> Union(Q, "failed")
            ++ GospersMethod(b, n, new) returns a rational function
            ++ \spad{rf(n)} such that \spad{a(n) * rf(n)} is the indefinite
            ++ sum of \spad{a(n)}
            ++ with respect to upward difference on \spad{n}, i.e.
            ++ \spad{a(n+1) * rf(n+1) - a(n) * rf(n) = a(n)},
            ++ where \spad{b(n) = a(n)/a(n-1)} is a rational function.
            ++ Returns "failed" if no such rational function \spad{rf(n)}
            ++ exists.
            ++ Note that \spad{new} is a nullary function returning a new
            ++ V every time.
            ++ The condition on \spad{a(n)} is that \spad{a(n)/a(n-1)}
            ++ is a rational function of \spad{n}.
            --++  \spad{sum(a(n), n) = rf(n) * a(n)}.

    Impl ==> add
      import PolynomialCategoryQuotientFunctions(E, V, R, P, Q)
      import LinearSystemMatrixPackage(RQ,Vector RQ,Vector RQ,Matrix RQ)

      InnerGospersMethod: (RQ, V, () -> V) -> Union(RQ, "failed")
      GosperPQR:   (PQ, PQ, V, () -> V)       -> List PQ
      GosperDegBd: (PQ, PQ, PQ, V, () -> V)    -> I
      GosperF:     (I, PQ, PQ, PQ, V, () -> V) -> Union(RQ, "failed")
      linearAndNNIntRoot: (PQ, V) -> Union(I, "failed")
      deg0:    (PQ, V) -> I       -- degree with deg 0 = -1.
      pCoef:   (PQ, PQ) -> PQ  -- pCoef(p, a*b**2)
```

```
RF2QIfCan: Q -> Union(RQ, "failed")
UP2QIfCan: P -> Union(PQ,"failed")
RFQ2R    : RQ -> Q
PQ2R     : PQ -> Q
rat?     : R  -> Boolean

deg0(p, v) == (zero? p => -1; degree(p, v))
rat? x     == retractIfCan(x::P::Q)@Union(RN, "failed") case RN
RFQ2R f    == PQ2R(numer f) / PQ2R(denom f)

PQ2R p ==
  map(x+->x::P::Q, y+->y::Q, p)$PolynomialCategoryLifting(
                                  IndexedExponents V, V, RN, PQ, Q)

GospersMethod(aquo, n, newV) ==
  ((q := RF2QIfCan aquo) case "failed") or
    ((u := InnerGospersMethod(q::RQ, n, newV)) case "failed") =>
       "failed"
  RFQ2R(u::RQ)

RF2QIfCan f ==
  (n := UP2QIfCan numer f) case "failed" => "failed"
  (d := UP2QIfCan denom f) case "failed" => "failed"
  n::PQ / d::PQ

UP2QIfCan p ==
  every?(rat?, coefficients p) =>
    map(x +-> x::PQ,
        y +-> (retractIfCan(y::P::Q)@Union(RN, "failed"))::RN::PQ,p)_
         $PolynomialCategoryLifting(E, V, R, P, PQ)
  "failed"

InnerGospersMethod(aquo, n, newV) ==
      -- 1. Define coprime polys an,anm1 such that
      --      an/anm1=a(n)/a(n-1)
      an   := numer aquo
      anm1 := denom aquo

      -- 2. Define p,q,r such that
      --      a(n)/a(n-1) = (p(n)/p(n-1)) * (q(n)/r(n))
      --    and
      --      gcd(q(n), r(n+j)) = 1, for all j: NNI.
      pqr:= GosperPQR(an, anm1, n, newV)
      pn := first pqr; qn := second pqr; rn := third pqr

      -- 3. If the sum is a rational fn, there is a poly f with
      --      sum(a(n), n) = q(n+1)/p(n) * a(n) * f(n).

      -- 4. Bound the degree of f(n).
      (k := GosperDegBd(pn, qn, rn, n, newV)) < 0 => "failed"
```

```
      -- 5. Find a polynomial f of degree at most k, satisfying
      --      p(n) = q(n+1)*f(n) - r(n)*f(n-1)
      (ufn := GosperF(k, pn, qn, rn, n, newV)) case "failed" =>
        "failed"
      fn  := ufn::RQ

      -- 6. The sum is q(n-1)/p(n)*f(n) * a(n). We leave out a(n).
      --qnm1 := eval(qn,n,n::PQ - 1)
      --qnm1/pn * fn
      qn1 := eval(qn,n,n::PQ + 1)
      qn1/pn * fn

GosperF(k, pn, qn, rn, n, newV) ==
      mv := newV(); mp := mv::PQ; np := n::PQ
      fn:       PQ := +/[mp**(i+1) * np**i for i in 0..k]
      fnminus1: PQ := eval(fn, n, np-1)
      qnplus1        := eval(qn, n, np+1)
      zro  := qnplus1 * fn - rn * fnminus1 - pn
      zron := univariate(zro, n)
      dz  := degree zron
      mat: Matrix RQ := zero(dz+1, (k+1)::NonNegativeInteger)
      vec: Vector RQ := new(dz+1, 0)
      while zron ^= 0 repeat
          cz := leadingCoefficient zron
          dz := degree zron
          zron := reductum zron
          mz := univariate(cz, mv)
          while mz ^= 0 repeat
              cmz := leadingCoefficient(mz)::RQ
              dmz := degree mz
              mz := reductum mz
              dmz = 0 => vec(dz + minIndex vec) := -cmz
              qsetelt_!(mat, dz + minRowIndex mat,
                           dmz + minColIndex(mat) - 1, cmz)
      (soln := particularSolution(mat, vec)) case "failed" => "failed"
      vec := soln::Vector RQ
      (+/[np**i * vec(i + minIndex vec) for i in 0..k])@RQ

GosperPQR(an, anm1, n, newV) ==
      np := n::PQ   -- polynomial version of n
      -- Initial guess.
      pn: PQ := 1
      qn: PQ := an
      rn: PQ := anm1
      -- Find all j: NNI giving common factors to q(n) and r(n+j).
      j     := newV()
      rnj   := eval(rn, n, np + j::PQ)
      res   := resultant(qn, rnj, n)
      fres  := factor(res)$MRationalFactorize(IndexedExponents V,
```

```
                                              V, I, PQ)
      js    := [rt::I for fe in factors fres
                 | (rt := linearAndNNIntRoot(fe.factor,j)) case I]
      -- For each such j, change variables to remove the gcd.
      for rt in js repeat
          rtp:= rt::PQ  -- polynomial version of rt
          gn := gcd(qn, eval(rn,n,np+rtp))
          qn := (qn exquo gn)::PQ
          rn := (rn exquo eval(gn, n, np-rtp))::PQ
          pn := pn * */[eval(gn, n, np-i::PQ) for i in 0..rt-1]
      [pn, qn, rn]

  -- Find a degree bound for the polynomial f(n) which satisfies
  --   p(n) = q(n+1)*f(n) - r(n)*f(n-1).
GosperDegBd(pn, qn, rn, n, newV) ==
      np := n::PQ
      qnplus1  := eval(qn, n, np+1)
      lplus  := deg0(qnplus1 + rn,  n)
      lminus := deg0(qnplus1 - rn, n)
      degp   := deg0(pn, n)
      k := degp - max(lplus-1, lminus)
      lplus <= lminus => k
      -- Find L(k), such that
      --   p(n) = L(k)*c[k]*n**(k + lplus - 1) + ...
      -- To do this, write f(n) and f(n-1) symbolically.
      --   f(n)  = c[k]*n**k + c[k-1]*n**(k-1) +O(n**(k-2))
      --   f(n-1)=c[k]*n**k + (c[k-1]-k*c[k])*n**(k-1)+O(n**(k-2))
      kk := newV()::PQ
      ck := newV()::PQ
      ckm1 := newV()::PQ
      nkm1:= newV()::PQ
      nk := np*nkm1
      headfn   := ck*nk +         ckm1*nkm1
      headfnm1 := ck*nk + (ckm1-kk*ck)*nkm1
      -- Then p(n) = q(n+1)*f(n) - r(n)*f(n-1) gives L(k).
      pk   := qnplus1 * headfn - rn * headfnm1
      lcpk := pCoef(pk, ck*np*nkm1)
      -- The degree bd is now given by k, and the root of L.
      k0 := linearAndNNIntRoot(lcpk, mainVariable(kk)::V)
      k0 case "failed" => k
      max(k0::I, k)

pCoef(p, nom) ==
      not monomial? nom =>
        error "pCoef requires a monomial 2nd arg"
      vlist := variables nom
      for v in vlist while p ^= 0 repeat
          unom:= univariate(nom,v)
          pow:=degree unom
          nom:=leadingCoefficient unom
```

```
            up  := univariate(p, v)
            p   := coefficient(up, pow)
        p

    linearAndNNIntRoot(mp, v) ==
        p := univariate(mp, v)
        degree p ^= 1 => "failed"
        (p1 := retractIfCan(coefficient(p, 1))@Union(RN,"failed"))
         case "failed" or
          (p0 := retractIfCan(coefficient(p, 0))@Union(RN,"failed"))
           case "failed" => "failed"
        rt := -(p0::RN)/(p1::RN)
        rt < 0 or denom rt ^= 1 => "failed"
        numer rt
```

———-

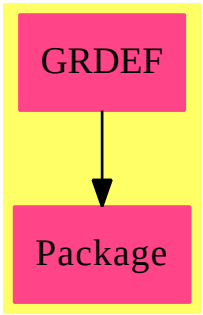
— **GOSPER.dotabb** —

```
"GOSPER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GOSPER"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GOSPER" -> "PFECAT"
```

———-

8.16 package GRDEF GraphicsDefaults

8.16.1 GraphicsDefaults (GRDEF)



Exports:
adaptive clipPointsDefault drawToScale maxPoints minPoints screenResolution

— **package GRDEF GraphicsDefaults** —

```
)abbrev package GRDEF GraphicsDefaults
++ Author: Clifton J. Williamson
++ Date Created: 8 January 1990
++ Date Last Updated: 8 January 1990
++ Basic Operations: clipPointsDefault, drawToScale, adaptive, maxPoints,
++ minPoints, screenResolution
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ TwoDimensionalPlotSettings sets global flags and constants
++ for 2-dimensional plotting.

GraphicsDefaults(): Exports == Implementation where
  B  ==> Boolean
  I  ==> Integer
  SF ==> DoubleFloat
  maxWidth  ==> 1000
  maxHeight ==> 1000

  Exports ==> with
    clipPointsDefault: () -> B
      ++ clipPointsDefault() determines whether or not automatic clipping is
      ++ to be done.
    drawToScale: () -> B
      ++ drawToScale() determines whether or not plots are to be drawn to scale.

    clipPointsDefault: B -> B
      ++ clipPointsDefault(true) turns on automatic clipping;
      ++ \spad{clipPointsDefault(false)} turns off automatic clipping.
      ++ The default setting is true.
    drawToScale: B -> B
      ++ drawToScale(true) causes plots to be drawn to scale.
      ++ \spad{drawToScale(false)} causes plots to be drawn so that they
      ++ fill up the viewport window.
      ++ The default setting is false.

--% settings from the two-dimensional plot package

    adaptive: () -> B
      ++ adaptive() determines whether plotting will be done adaptively.
    maxPoints: () -> I
      ++ maxPoints() returns the maximum number of points in a plot.
    minPoints: () -> I
      ++ minPoints() returns the minimum number of points in a plot.
    screenResolution: () -> I
      ++ screenResolution() returns the screen resolution n.
```

```
    adaptive: B -> B
      ++ adaptive(true) turns adaptive plotting on;
      ++ \spad{adaptive(false)} turns adaptive plotting off.
    maxPoints: I -> I
      ++ maxPoints() sets the maximum number of points in a plot.
    minPoints: I -> I
      ++ minPoints() sets the minimum number of points in a plot.
    screenResolution: I -> I
      ++ screenResolution(n) sets the screen resolution to n.

  Implementation ==> add

--% global flags and constants

    CLIPPOINTSDEFAULT : B := true
    TOSCALE  : B := false

--% functions

    clipPointsDefault()     == CLIPPOINTSDEFAULT
    drawToScale()  == TOSCALE

    clipPointsDefault b    == CLIPPOINTSDEFAULT := b
    drawToScale b == TOSCALE := b

--% settings from the two-dimensional plot package

    adaptive() == adaptive?()$Plot
    minPoints() == minPoints()$Plot
    maxPoints() == maxPoints()$Plot
    screenResolution() == screenResolution()$Plot

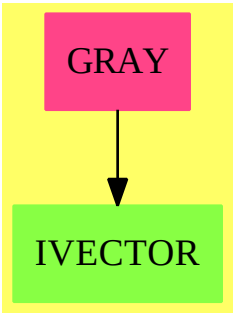
    adaptive b == setAdaptive(b)$Plot
    minPoints n == setMinPoints(n)$Plot
    maxPoints n == setMaxPoints(n)$Plot
    screenResolution n == setScreenResolution(n)$Plot
```

— GRDEF.dotabb —

```
"GRDEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GRDEF"]
"Package" [color="#FF4488"]
"GRDEF" -> "Package"
```

8.17 package GRAY GrayCode

8.17.1 GrayCode (GRAY)



Exports:
firstSubsetGray nextSubsetGray

— package GRAY GrayCode —

```
)abbrev package GRAY GrayCode
++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ Basic Operations: nextSubsetGray
++ Related Constructors: Permanent
++ Also See: SymmetricGroupCombinatoric Functions
++ AMS Classifications:
++ Keywords: gray code, subsets of finite sets
++ References:
++  Henryk Minc: Evaluation of Permanents,
++    Proc. of the Edinburgh Math. Soc.(1979), 22/1 pp 27-32.
++  Nijenhuis and Wilf : Combinatorical Algorithms, Academic
++    Press, New York 1978.
++   S.G.Williamson, Combinatorics for Computer Science,
++    Computer Science Press, 1985.
++ Description:
++ GrayCode provides a function for efficiently running
++ through all subsets of a finite set, only changing one element
++ by another one.

GrayCode: public == private where

  PI ==> PositiveInteger
  I  ==> Integer
  V  ==> Vector
```

```
public ==> with

  nextSubsetGray: (V V I,PI) -> V V I
    ++ nextSubsetGray(ww,n) returns a vector vv whose components
    ++ have the following meanings:\br
    ++ vv.1: a vector of length n whose entries are 0 or 1. This
    ++ can be interpreted as a code for a subset of the set 1,...,n;
    ++ vv.1 differs from ww.1 by exactly one entry;\br
    ++ vv.2.1 is the number of the entry of vv.1 which
    ++ will be changed next time;\br
    ++ vv.2.1 = n+1 means that vv.1 is the last subset;
    ++ trying to compute nextSubsetGray(vv) if vv.2.1 = n+1
    ++ will produce an error!\br
    ++
    ++ The other components of vv.2 are needed to compute
    ++ nextSubsetGray efficiently.
    ++ Note that this is an implementation of [Williamson, Topic II, 3.54,
    ++ p. 112] for the special case r1 = r2 = ... = rn = 2;
    ++ Note that nextSubsetGray produces a side-effect, i.e.
    ++ nextSubsetGray(vv) and vv := nextSubsetGray(vv)
    ++ will have the same effect.

  firstSubsetGray: PI -> V V I
    ++ firstSubsetGray(n) creates the first vector ww to start a
    ++ loop using nextSubsetGray(ww,n)

private ==> add

  firstSubsetGray(n : PI) ==
    vv : V V I := new(2,[])
    vv.1 := new(n,0) : V I
    vv.2 := new(n+1,1) : V I
    for i in 1..(n+1) repeat
      vv.2.i := i
    vv

  nextSubsetGray(vv : V V I,n : PI) ==
    subs : V I := vv.1    -- subset
    lab : V I := vv.2     -- labels
    c : I := lab(1)    -- element which is to be changed next
    lab(1):= 1
    if subs.c = 0 then subs.c := 1
    else subs.c := 0
    lab.c := lab(c+1)
    lab(c+1) := c+1
    vv
```

— **GRAY.dotabb** —

```
"GRAY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GRAY"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"GRAY" -> "IVECTOR"
```

———-

8.18 package GBF GroebnerFactorizationPackage

— **GroebnerFactorizationPackage.input** —

```
)set break resume
)sys rm -f GroebnerFactorizationPackage.output
)spool GroebnerFactorizationPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 3
mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], [1,8/3
--R
--R
--R        +0  1  1  1  1  1+
--R        |                 |
--R        |         8     8|
--R        |1  0  1  -  x  -|
--R        |         3     3|
--R        |                 |
--R        |            8    |
--R        |1  1  0  1  -  y|
--R        |            3    |
--R        |                 |
--R   (1)  |   8           8|
--R        |1  -  1  0  1  -|
--R        |   3           3|
--R        |                 |
--R        |      8          |
--R        |1  x  -  1  0  1|
--R        |      3          |
--R        |                 |
--R        |   8     8       |
--R        |1  -  y  -  1  0|
--R        +   3     3      +
--RType: SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],Fraction(Integer)))
--E 1
```

```
--S 2 of 3
eq := determinant mfzn
--R
--R
--R   (2)
--R      2 2   22  2    25  2   22    2   388       250     25  2   250     14575
--R   - x y  + -- x y - -- x  + -- x y  - --- x y - --- x - -- y  - --- y + -----
--R            3        9       3         9         27      9       27       81
--R           Type: DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))
--E 2

--S 3 of 3
groebnerFactorize [eq,eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])]
--R
--R
--R   (3)
--R   [
--R                  22           22     22     121
--R     [x y + x z - -- x + y z - -- y - -- z + ---,
--R                  3            3      3      3
--R         2   22       25        2   22       25     22  2   388     250
--R      x z  - -- x z + -- x + y z  - -- y z + -- y - -- z  + --- z + ---,
--R            3        9             3        9      3       9       27
--R       2 2   22  2    25  2   22    2   388       250     25  2   250     14575
--R      y z  - -- y z + -- y  - -- y z  + --- y z + --- y + -- z  + --- z - -----]
--R            3        9       3         9         27      9       27       81
--R     ,
--R             21994  2   21994     4427     463
--R    [x + y - -----,y  - ----- y + ----,z - ---],
--R             5625       5625      675      87
--R      2   1       11     5     265        2   38     265
--R    [x  - - x z - -- x - - z + ---,y - z,z  - -- z + ---],
--R          2        2     6      18             3      9
--R         25     11     11        11     11     11        5     5     5
--R    [x - --,y - --,z - --], [x - --,y - --,z - --], [x + -,y + -,z + -],
--R          9      3      3         3      3      3        3     3     3
--R         19     5     5
--R    [x - --,y + -,z + -]]
--R          3     3     3
--RType: List(List(DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))))
--E 3
)spool
)lisp (bye)
```

———-

— GroebnerFactorizationPackage.help —

```
=====================================================================
GroebnerFactorizationPackage examples
=====================================================================

Solving systems of polynomial equations with the Groebner basis
algorithm can often be very time consuming because, in general, the
algorithm has exponential run-time.  These systems, which often come
from concrete applications, frequently have symmetries which are not
taken advantage of by the algorithm.  However, it often happens in
this case that the polynomials which occur during the Groebner
calculations are reducible.  Since Axiom has an excellent polynomial
factorization algorithm, it is very natural to combine the Groebner
and factorization algorithms.

GroebnerFactorizationPackage exports the groebnerFactorize operation
which implements a modified Groebner basis algorithm.  In this
algorithm, each polynomial that is to be put into the partial list of
the basis is first factored.  The remaining calculation is split into
as many parts as there are irreducible factors.  Call these factors
p1,...,pN. In the branches corresponding to p2,...,pN, the factor p1
can be divided out, and so on.  This package also contains operations
that allow you to specify the polynomials that are not zero on the
common roots of the final Groebner basis.

Here is an example from chemistry.  In a theoretical model of the
cyclohexan C6H12, the six carbon atoms each sit in the center of
gravity of a tetrahedron that has two hydrogen atoms and two carbon
atoms at its corners.  We first normalize and set the length of each
edge to 1.  Hence, the distances of one fixed carbon atom to each of
its immediate neighbours is 1.  We will denote the distances to the
other three carbon atoms by x, y and z.

A. Dress developed a theory to decide whether a set of points
and distances between them can be realized in an n-dimensional space.
Here, of course, we have n = 3.

  mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := _
   [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], _
     [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
        +0  1  1  1  1  1+
        |                 |
        |         8     8 |
        |1  0  1  -  x  -|
        |         3     3 |
        |                 |
        |            8    |
        |1  1  0  1  -  y|
        |            3    |
        |                 |
        |   8           8 |
```

```
       |1  -  1  0  1  -|
       |   3          3|
       |               |
       |      8        |
       |1  x  -  1  0  1|
       |      3        |
       |               |
       |   8     8     |
       |1  -  y  -  1  0|
       +   3     3     +
Type: SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],
                    Fraction Integer))
```

For the cyclohexan, the distances have to satisfy this equation.

```
  eq := determinant mfzn
      2 2   22  2    25  2   22    2   388       250     25  2   250     14575
   - x y  + -- x y - -- x  + -- x y  - --- x y - --- x - -- y  - --- y + -----
            3        9       3         9         27      9       27       81
          Type: DistributedMultivariatePolynomial([x,y,z],Fraction Integer)
```

They also must satisfy the equations given by cyclic shifts of the indeterminates.

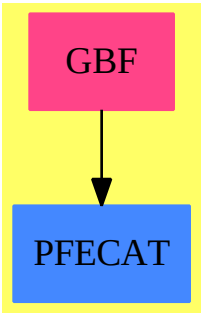
```
  groebnerFactorize [eq,eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])]
   [
                22           22     22     121
    [x y + x z - -- x + y z - -- y - -- z + ---,
                 3            3      3      3
        2   22       25        2   22       25     22  2   388     250
     x z  - -- x z + -- x + y z  - -- y z + -- y - -- z  + --- z + ---,
             3        9             3        9      3       9       27
      2 2   22  2    25  2   22    2   388       250     25  2   250     14575
     y z  - -- y z + -- y  - -- y z  + --- y z + --- y + -- z  + --- z - -----]
             3        9       3         9         27      9       27       81
    ,
           21994  2   21994     4427     463
   [x + y - -----,y  - ----- y + ----,z - ---],
           5625       5625      675      87
     2   1       11     5     265        2   38     265
   [x  - - x z - -- x - - z + ---,y - z,z  - -- z + ---],
        2         2     6      18             3      9
       25     11     11        11     11     11        5     5     5
   [x - --,y - --,z - --], [x - --,y - --,z - --], [x + -,y + -,z + -],
        9      3      3         3      3      3        3     3     3
       19     5     5
   [x - --,y + -,z + -]]
        3     3     3
  Type: List List DistributedMultivariatePolynomial([x,y,z],Fraction Integer)
```

```
The union of the solutions of this list is the solution of our original
problem.  If we impose positivity conditions, we get two relevant ideals.
One ideal is zero-dimensional, namely x = y = z =11/3, and this determines
the "boat" form of the cyclohexan.  The other ideal is one-dimensional,
which means that we have a solution space given by one parameter.  This
gives the "chair" form of the cyclohexan.  The parameter describes the
angle of the "back of the chair."

groebnerFactorize has an optional Boolean-valued second argument.
When it is true partial results are displayed, since it may happen
that the calculation does not terminate in a reasonable time.  See the
source code for GroebnerFactorizationPackage in groebf.spad.pamphlet
for more details about the algorithms used.

See Also:
o )display operations groebnerFactorize
o )show GroebnerFactorizationPackage
o) show GroebnerPackage
o )show EuclideanGroebnerBasisPackage
```

8.18.1 GroebnerFactorizationPackage (GBF)



Exports:
factorGroebnerBasis groebnerFactorize

— package GBF GroebnerFactorizationPackage —

```
)abbrev package GBF GroebnerFactorizationPackage
++ Author: H. Michael Moeller, Johannes Grabmeier
++ Date Created: 24 August 1989
++ Date Last Updated: 01 January 1992
```

```
++ Basic Operations: groebnerFactorize factorGroebnerBasis
++ Related Constructors:
++ Also See: GroebnerPackage, Ideal, IdealDecompositionPackage
++ AMS Classifications:
++ Keywords: groebner basis, groebner factorization, ideal decomposition
++ References:
++ Description:
++ \spadtype{GroebnerFactorizationPackage} provides the function
++ groebnerFactor" which uses the factorization routines of \Language{} to
++ factor each polynomial under consideration while doing the groebner basis
++ algorithm. Then it writes the ideal as an intersection of ideals
++ determined by the irreducible factors. Note that the whole ring may
++ occur as well as other redundancies. We also use the fact, that from the
++ second factor on we can assume that the preceding factors are
++ not equal to 0 and we divide all polynomials under considerations
++ by the elements of this list of "nonZeroRestrictions".
++ The result is a list of groebner bases, whose union of solutions
++ of the corresponding systems of equations is the solution of
++ the system of equation corresponding to the input list.
++ The term ordering is determined by the polynomial type used.
++ Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

GroebnerFactorizationPackage(Dom, Expon, VarSet, Dpol): T == C where

 Dom :    Join(EuclideanDomain,CharacteristicZero)
 Expon :  OrderedAbelianMonoidSup
 VarSet : OrderedSet
 Dpol: PolynomialCategory(Dom, Expon, VarSet)
 MF       ==>   MultivariateFactorize(VarSet,Expon,Dom,Dpol)
 sugarPol ==>   Record(totdeg: NonNegativeInteger, pol : Dpol)
 critPair ==>   Record(lcmfij: Expon,totdeg: NonNegativeInteger, poli: Dpol, polj: Dpol )
 L        ==>   List
 B        ==>   Boolean
 NNI      ==>   NonNegativeInteger
 OUT      ==>   OutputForm

 T ==> with

   factorGroebnerBasis : L Dpol -> L L Dpol
     ++ factorGroebnerBasis(basis) checks whether the basis contains
     ++ reducible polynomials and uses these to split the basis.
   factorGroebnerBasis : (L Dpol, Boolean) -> L L Dpol
     ++ factorGroebnerBasis(basis,info) checks whether the basis contains
     ++ reducible polynomials and uses these to split the basis.
     ++ If argument info is true, information is printed about
     ++ partial results.
   groebnerFactorize : (L Dpol, L Dpol) -> L L Dpol
```

```
    ++ groebnerFactorize(listOfPolys, nonZeroRestrictions) returns
    ++ a list of groebner basis. The union of their solutions
    ++ is the solution of the system of equations given by listOfPolys
    ++ under the restriction that the polynomials of nonZeroRestrictions
    ++ don't vanish.
    ++ At each stage the polynomial p under consideration (either from
    ++ the given basis or obtained from a reduction of the next S-polynomial)
    ++ is factorized. For each irreducible factors of p, a
    ++ new createGroebnerBasis is started
    ++ doing the usual updates with the factor
    ++ in place of p.
  groebnerFactorize : (L Dpol, L Dpol, Boolean) -> L L Dpol
    ++ groebnerFactorize(listOfPolys, nonZeroRestrictions, info) returns
    ++ a list of groebner basis. The union of their solutions
    ++ is the solution of the system of equations given by listOfPolys
    ++ under the restriction that the polynomials of nonZeroRestrictions
    ++ don't vanish.
    ++ At each stage the polynomial p under consideration (either from
    ++ the given basis or obtained from a reduction of the next S-polynomial)
    ++ is factorized. For each irreducible factors of p a
    ++ new createGroebnerBasis is started
    ++ doing the usual updates with the factor in place of p.
    ++ If argument info is true, information is printed about
    ++ partial results.
  groebnerFactorize : L Dpol  -> L L Dpol
    ++ groebnerFactorize(listOfPolys) returns
    ++ a list of groebner bases. The union of their solutions
    ++ is the solution of the system of equations given by listOfPolys.
    ++ At each stage the polynomial p under consideration (either from
    ++ the given basis or obtained from a reduction of the next S-polynomial)
    ++ is factorized. For each irreducible factors of p, a
    ++ new createGroebnerBasis is started
    ++ doing the usual updates with the factor
    ++ in place of p.
    ++
    ++X mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := _
    ++X   [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], _
    ++X   [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
    ++X eq := determinant mfzn
    ++X groebnerFactorize _
    ++X   [eq,eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])]
  groebnerFactorize : (L Dpol, Boolean)  -> L L Dpol
    ++ groebnerFactorize(listOfPolys, info) returns
    ++ a list of groebner bases. The union of their solutions
    ++ is the solution of the system of equations given by listOfPolys.
    ++ At each stage the polynomial p under consideration (either from
    ++ the given basis or obtained from a reduction of the next S-polynomial)
    ++ is factorized. For each irreducible factors of p, a
    ++ new createGroebnerBasis is started
    ++ doing the usual updates with the factor
```

```
    ++ in place of p.
    ++ If info is true, information is printed about partial results.

C ==> add

  import GroebnerInternalPackage(Dom,Expon,VarSet,Dpol)
  -- next to help compiler to choose correct signatures:
  info: Boolean
  -- signatures of local functions

  newPairs : (L sugarPol, Dpol) -> L critPair
    ++ newPairs(lp, p) constructs list of critical pairs from the list of
    ++ lp of input polynomials and a given further one p.
    ++ It uses criteria M and T to reduce the list.
  updateCritPairs : (L critPair, L critPair, Dpol) -> L critPair
    ++ updateCritPairs(lcP1,lcP2,p) applies criterion B to lcP1 using
    ++ p. Then this list is merged with lcP2.
  updateBasis : (L sugarPol, Dpol, NNI) -> L sugarPol
    ++ updateBasis(li,p,deg) every polynomial in li is dropped if
    ++ its leading term is a multiple of the leading term of p.
    ++ The result is this list enlarged by p.
  createGroebnerBases : (L sugarPol, L Dpol, L Dpol, L Dpol, L critPair,_
                         L L Dpol, Boolean) -> L L Dpol
    ++ createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,
    ++   lcP,listOfBases): This function is used to be called from
    ++ groebnerFactorize.
    ++ basis: part of a Groebner basis, computed so far
    ++ redPols: Polynomials from the ideal to be used for reducing,
    ++   we don't throw away polynomials
    ++ nonZeroRestrictions: polynomials not zero in the common zeros
    ++   of the polynomials in the final (Groebner) basis
    ++ inputPolys: assumed to be in descending order
    ++ lcP: list of critical pairs built from polynomials of the
    ++   actual basis
    ++ listOfBases: Collects the (Groebner) bases constructed by this
    ++   recursive algorithm at different stages.
    ++   we print info messages if info is true
  createAllFactors: Dpol -> L Dpol
    ++ factor reduced critpair polynomial

  -- implementation of local functions


  createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,_
      lcP, listOfBases, info) ==
    doSplitting? : B := false
    terminateWithBasis : B := false
    allReducedFactors : L Dpol := []
    nP : Dpol  -- actual polynomial under consideration
    p :  Dpol  -- next polynomial from input list
```

```
h :  Dpol  -- next polynomial from critical pairs
stopDividing : Boolean
--    STEP 1   do the next polynomials until a splitting is possible
-- In the first step we take the first polynomial of "inputPolys"
-- if empty, from list of critical pairs "lcP" and do the following:
-- Divide it, if possible, by the polynomials from "nonZeroRestrictions".
-- We factorize it and reduce each irreducible  factor with respect to
-- "basis". If 0$Dpol occurs in the list we update the list and continue
-- with next polynomial.
-- If there are at least two (irreducible) factors
-- in the list of factors we finish STEP 1 and set a boolean variable
-- to continue with STEP 2, the splitting step.
-- If there is just one of it, we do the following:
-- If it is 1$Dpol we stop the whole calculation and put
-- [1$Dpol] into the listOfBases
-- Otherwise we update the "basis" and the other lists and continue
-- with next polynomial.

while (not doSplitting?) and (not terminateWithBasis) repeat
  terminateWithBasis := (null inputPolys and null lcP)
  not terminateWithBasis =>  -- still polynomials left
    -- determine next polynomial "nP"
    nP :=
      not null inputPolys =>
        p := first inputPolys
        inputPolys := rest inputPolys
        -- we know that p is not equal to 0 or 1, but, although,
        -- the inputPolys and the basis are ordered, we cannot assume
        -- that p is reduced w.r.t. basis, as the ordering is only quasi
        -- and we could have equal leading terms, and due to factorization
        -- polynomials of smaller leading terms, hence reduce p first:
        hMonic redPol(p,redPols)
      -- now we have inputPolys empty and hence lcP is not empty:
      -- create S-Polynomial from first critical pair:
      h := sPol first lcP
      lcP := rest lcP
      hMonic redPol(h,redPols)

    nP = 1$Dpol =>
      basis := [[0,1$Dpol]$sugarPol]
      terminateWithBasis := true

    -- if "nP" ^= 0, then  we continue, otherwise we determine next "nP"
    nP ^= 0$Dpol =>
      -- now we divide "nP", if possible, by the polynomials
      -- from "nonZeroRestrictions"
      for q in nonZeroRestrictions repeat
        stopDividing := false
        until stopDividing repeat
          nPq := nP exquo q
```

```
            stopDividing := (nPq case "failed")
            if not stopDividing then nP := autoCoerce nPq
            stopDividing := stopDividing or zero? degree nP

          zero? degree nP =>
            basis := [[0,1$Dpol]$sugarPol]
            terminateWithBasis := true  -- doSplitting? is still false

          -- a careful analysis has to be done, when and whether the
          -- following reduction and case nP=1 is necessary

          nP := hMonic redPol(nP,redPols)
          zero? degree nP =>
            basis := [[0,1$Dpol]$sugarPol]
            terminateWithBasis := true  -- doSplitting? is still false

          -- if "nP" ^= 0, then  we continue, otherwise we determine next "nP"
          nP ^= 0$Dpol =>
            -- now we factorize "nP", which is not constant
            irreducibleFactors : L Dpol := createAllFactors(nP)
            -- if there are more than 1 factors we reduce them and split
            (doSplitting? := not null rest irreducibleFactors) =>
              -- and reduce and normalize the factors
              for fnP in irreducibleFactors repeat
                fnP := hMonic redPol(fnP,redPols)
                -- no factor reduces to 0, as then "fP" would have been
                -- reduced to zero,
                -- but 1 may occur, which we will drop in a later version.
                allReducedFactors := cons(fnP, allReducedFactors)
              -- end of "for fnP in irreducibleFactors repeat"

              -- we want that the smaller factors are dealt with first
              allReducedFactors := reverse allReducedFactors
            -- now the case of exactly 1 factor, but certainly not
            -- further reducible with respect to "redPols"
            nP := first irreducibleFactors
            -- put "nP" into "basis" and update "lcP" and "redPols":
            lcP : L critPair := updateCritPairs(lcP,newPairs(basis,nP),nP)
            basis := updateBasis(basis,nP,virtualDegree nP)
            redPols := concat(redPols,nP)
      -- end of "while not doSplitting? and not terminateWithBasis repeat"

      --    STEP 2  splitting step

      doSplitting? =>
        for fnP in allReducedFactors repeat
          if fnP ^= 1$Dpol
            then
              newInputPolys : L Dpol  := _
                sort((x,y) +-> degree x > degree y ,cons(fnP,inputPolys))
```

```
            listOfBases := createGroebnerBases(basis, redPols, _
              nonZeroRestrictions,newInputPolys,lcP,listOfBases,info)
            -- update "nonZeroRestrictions"
            nonZeroRestrictions := cons(fnP,nonZeroRestrictions)
          else
            if info then
              messagePrint("we terminated with [1]")$OUT
            listOfBases := cons([1$Dpol],listOfBases)

      -- we finished with all the branches on one level and hence
      -- finished this call of createGroebnerBasis. Therefore
      -- we terminate with the actual "listOfBasis" as
      -- everything is done in the recursions
      listOfBases
    -- end of "doSplitting? =>"

    --    STEP 3 termination step

    --  we found a groebner basis and put it into the list "listOfBases"
    --  (auto)reduce each basis element modulo the others
    newBasis :=
      minGbasis(sort((x,y)+->degree x > degree y,[p.pol for p in basis]))
    -- now check whether the normalized basis again has reducible
    -- polynomials, in this case continue splitting!
    if info then
      messagePrint("we found a groebner basis and check whether it ")$OUT
      messagePrint("contains reducible polynomials")$OUT
      print(newBasis::OUT)$OUT
      -- here we should create an output form which is reusable by the system
      -- print(convert(newBasis::OUT)$InputForm :: OUT)$OUT
    removeDuplicates append(factorGroebnerBasis(newBasis, info), listOfBases)

  createAllFactors(p: Dpol) ==
    loF : L Dpol := [el.fctr for el in factorList factor(p)$MF]
    sort((x,y) +-> degree x < degree y, loF)
  newPairs(lp : L sugarPol,p : Dpol) ==
    totdegreeOfp : NNI := virtualDegree p
    -- next list lcP contains all critPair constructed from
    -- p and and the polynomials q in lp
    lcP: L critPair := _
      --[[sup(degree q, degreeOfp), q, p]$critPair for q in lp]
      [makeCrit(q, p, totdegreeOfp) for q in lp]
    -- application of the criteria to reduce the list lcP
    critMTonD1 sort(critpOrder,lcP)
  updateCritPairs(oldListOfcritPairs, newListOfcritPairs, p)==
    updatD (newListOfcritPairs, critBonD(p,oldListOfcritPairs))
  updateBasis(lp, p, deg) == updatF(p,deg,lp)

  -- exported functions
```

```
  factorGroebnerBasis basis == factorGroebnerBasis(basis, false)

  factorGroebnerBasis (basis, info) ==
    foundAReducible : Boolean := false
    for p in basis while not foundAReducible repeat
      -- we use fact that polynomials have content 1
      foundAReducible := 1 < #[el.fctr for el in factorList factor(p)$MF]
    not foundAReducible =>
      if info then  messagePrint("factorGroebnerBasis: no reducible polynomials in this basis")
      [basis]
    -- improve! Use the fact that the irreducible ones already
    -- build part of the basis, use the done factorizations, etc.
    if info then  messagePrint("factorGroebnerBasis:_
       we found reducible polynomials and continue splitting")$OUT
    createGroebnerBases([],[],[],basis,[],[],info)

  groebnerFactorize(basis, nonZeroRestrictions) ==
    groebnerFactorize(basis, nonZeroRestrictions, false)

  groebnerFactorize(basis, nonZeroRestrictions, info) ==
    basis = [] => [basis]
    basis := remove((x:Dpol):Boolean +->(x = 0$Dpol),basis)
    basis = [] => [[0$Dpol]]
    -- normalize all input polynomial
    basis := [hMonic p for p in basis]
    member?(1$Dpol,basis) => [[1$Dpol]]
    basis :=  sort((x,y) +-> degree x > degree y, basis)
    createGroebnerBases([],[],nonZeroRestrictions,basis,[],[],info)

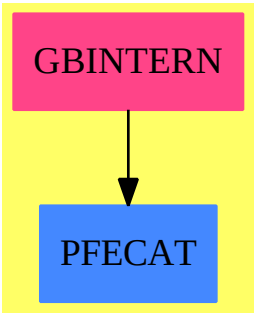
  groebnerFactorize(basis) == groebnerFactorize(basis, [], false)
  groebnerFactorize(basis,info) == groebnerFactorize(basis, [], info)
```

— GBF.dotabb —

```
"GBF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GBF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GBF" -> "PFECAT"
```

8.19 package GBINTERN GroebnerInternalPackage

8.19.1 GroebnerInternalPackage (GBINTERN)



Exports:

credPol	critB	critBonD	critM	critMTonD1
critMonD1	critpOrder	critT	fprindINFO	gbasis
hMonic	lepol	makeCrit	minGbasis	prinb
prindINFO	prinpolINFO	prinshINFO	redPo	redPol
sPol	updatD	updatF	virtualDegree	

— package GBINTERN GroebnerInternalPackage —

```
)abbrev package GBINTERN GroebnerInternalPackage
++ Author:
++ Date Created:
++ Date Last Updated:
++ Keywords:
++ Description:
++ This package provides low level tools for Groebner basis computations

GroebnerInternalPackage(Dom, Expon, VarSet, Dpol): T == C where
 Dom:   GcdDomain
 Expon: OrderedAbelianMonoidSup
 VarSet: OrderedSet
 Dpol:  PolynomialCategory(Dom, Expon, VarSet)
 NNI    ==> NonNegativeInteger
   ------  Definition of Record critPair and Prinp

 critPair ==> Record( lcmfij: Expon, totdeg: NonNegativeInteger,
                      poli: Dpol, polj: Dpol )
 sugarPol ==> Record( totdeg: NonNegativeInteger, pol : Dpol)
 Prinp    ==> Record( ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol,
                tc:Integer,rc:Dpol,trc:Integer,tF:Integer,tD:Integer)
 Prinpp   ==> Record( ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol,
                tc:Integer,rc:Dpol,trc:Integer,tF:Integer,tDD:Integer,
```

```
                tDF:Integer)
 T== with

     credPol:  (Dpol, List(Dpol))  -> Dpol
        ++ credPol \undocumented
     redPol:   (Dpol, List(Dpol))  -> Dpol
        ++ redPol \undocumented
     gbasis:  (List(Dpol), Integer, Integer) -> List(Dpol)
        ++ gbasis \undocumented
     critT:  critPair   -> Boolean
        ++ critT \undocumented
     critM:  (Expon, Expon) -> Boolean
        ++ critM \undocumented
     critB:  (Expon, Expon, Expon, Expon) -> Boolean
        ++ critB \undocumented
     critBonD:  (Dpol, List(critPair)) -> List(critPair)
        ++ critBonD \undocumented
     critMTonD1: (List(critPair)) -> List(critPair)
        ++ critMTonD1 \undocumented
     critMonD1:  (Expon, List(critPair)) -> List(critPair)
        ++ critMonD1 \undocumented
     redPo: (Dpol, List(Dpol) )  ->  Record(poly:Dpol, mult:Dom)
        ++ redPo \undocumented
     hMonic:  Dpol  -> Dpol
        ++ hMonic \undocumented
     updatF: (Dpol, NNI, List(sugarPol) ) -> List(sugarPol)
        ++ updatF \undocumented
     sPol:  critPair  -> Dpol
        ++ sPol \undocumented
     updatD: (List(critPair), List(critPair)) -> List(critPair)
        ++ updatD \undocumented
     minGbasis: List(Dpol) -> List(Dpol)
        ++ minGbasis \undocumented
     lepol: Dpol -> Integer
        ++ lepol \undocumented
     prinshINFO : Dpol -> Void
        ++ prinshINFO \undocumented
     prindINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
        ++ prindINFO \undocumented
     fprindINFO: (critPair, Dpol, Dpol, Integer,Integer,Integer
                 ,Integer) ->  Integer
        ++ fprindINFO \undocumented
     prinpolINFO: List(Dpol) -> Void
        ++ prinpolINFO \undocumented
     prinb: Integer-> Void
        ++ prinb \undocumented
     critpOrder: (critPair, critPair) -> Boolean
        ++ critpOrder \undocumented
     makeCrit: (sugarPol, Dpol, NonNegativeInteger) -> critPair
        ++ makeCrit \undocumented
```

```
    virtualDegree : Dpol -> NonNegativeInteger
       ++ virtualDegree \undocumented

C== add
  Ex ==> OutputForm
  import OutputForm

  ------  Definition of intermediate functions
  if Dpol has totalDegree: Dpol -> NonNegativeInteger then
    virtualDegree p == totalDegree p
  else
    virtualDegree p == 0

  ------  ordering of critpairs

  critpOrder(cp1,cp2) ==
    cp1.totdeg < cp2.totdeg => true
    cp2.totdeg < cp1.totdeg => false
    cp1.lcmfij < cp2.lcmfij

  ------    creating a critical pair

  makeCrit(sp1, p2, totdeg2) ==
    p1 := sp1.pol
    deg := sup(degree(p1), degree(p2))
    e1 := subtractIfCan(deg, degree(p1))::Expon
    e2 := subtractIfCan(deg, degree(p2))::Expon
    tdeg := max(sp1.totdeg + virtualDegree(monomial(1,e1)),
                totdeg2 + virtualDegree(monomial(1,e2)))
    [deg, tdeg, p1, p2]$critPair

  ------    calculate basis

  gbasis(Pol: List(Dpol), xx1: Integer, xx2: Integer ) ==
    D, D1: List(critPair)
    ---------   create D and Pol

    Pol1:= sort((z1,z2) +-> degree z1 > degree z2, Pol)
    basPols:= updatF(hMonic(first Pol1),virtualDegree(first Pol1),[])
    Pol1:= rest(Pol1)
    D:= nil
    while _^ null Pol1 repeat
       h:= hMonic(first(Pol1))
       Pol1:= rest(Pol1)
       toth := virtualDegree h
       D1:= [makeCrit(x,h,toth) for x in basPols]
       D:= updatD(critMTonD1(sort(critpOrder, D1)),
                  critBonD(h,D))
       basPols:= updatF(h,toth,basPols)
    D:= sort(critpOrder, D)
```

```
    xx:= xx2
    --------  loop

    redPols := [x.pol for x in basPols]
    while _^ null D repeat
        D0:= first D
        s:= hMonic(sPol(D0))
        D:= rest(D)
        h:= hMonic(redPol(s,redPols))
        if xx1 = 1  then
             prinshINFO(h)
        h = 0  =>
         if xx2 = 1 then
          prindINFO(D0,s,h,# basPols, # D,xx)
          xx:= 2
         " go to top of while "
        degree(h) = 0 =>
          D:= nil
          if xx2 = 1 then
           prindINFO(D0,s,h,# basPols, # D,xx)
           xx:= 2
          basPols:= updatF(h,0,[])
          leave "out of while"
        D1:= [makeCrit(x,h,D0.totdeg) for x in basPols]
        D:= updatD(critMTonD1(sort(critpOrder, D1)),
                   critBonD(h,D))
        basPols:= updatF(h,D0.totdeg,basPols)
        redPols := concat(redPols,h)
        if xx2 = 1 then
           prindINFO(D0,s,h,# basPols, # D,xx)
           xx:= 2
    Pol := [x.pol for x in basPols]
    if xx2 = 1 then
      prinpolINFO(Pol)
      messagePrint("    THE GROEBNER BASIS POLYNOMIALS")
    if xx1 = 1 and xx2 ^= 1 then
      messagePrint("    THE GROEBNER BASIS POLYNOMIALS")
    Pol

            ---------------------------------------

            --- erase multiple of e in D2 using crit M

  critMonD1(e: Expon, D2: List(critPair))==
     null D2 => nil
     x:= first(D2)
     critM(e, x.lcmfij) => critMonD1(e, rest(D2))
     cons(x, critMonD1(e, rest(D2)))

            ---------------------------
```

```
          --- reduce D1 using crit T and crit M

critMTonD1(D1: List(critPair))==
        null D1 => nil
        f1:= first(D1)
        s1:= #(D1)
        cT1:= critT(f1)
        s1= 1 and cT1 => nil
        s1= 1 => D1
        e1:= f1.lcmfij
        r1:= rest(D1)
        e1 = (first r1).lcmfij  =>
           cT1 =>   critMTonD1(cons(f1, rest(r1)))
           critMTonD1(r1)
        D1 := critMonD1(e1, r1)
        cT1 => critMTonD1(D1)
        cons(f1, critMTonD1(D1))

          ------------------------------

          --- erase elements in D fullfilling crit B

critBonD(h:Dpol, D: List(critPair))==
      null D => nil
      x:= first(D)
      critB(degree(h), x.lcmfij, degree(x.poli), degree(x.polj)) =>
        critBonD(h, rest(D))
      cons(x, critBonD(h, rest(D)))

          ------------------------------

          --- concat F and h and erase multiples of h in F

updatF(h: Dpol, deg:NNI, F: List(sugarPol)) ==
    null F => [[deg,h]]
    f1:= first(F)
    critM(degree(h), degree(f1.pol))  => updatF(h, deg, rest(F))
    cons(f1, updatF(h, deg, rest(F)))

          ------------------------------

          --- concat ordered critical pair lists D1 and D2

updatD(D1: List(critPair), D2: List(critPair)) ==
   null D1 => D2
   null D2 => D1
   dl1:= first(D1)
   dl2:= first(D2)
   critpOrder(dl1,dl2) => cons(dl1, updatD(D1.rest, D2))
```

```
      cons(dl2, updatD(D1, D2.rest))

            ----------------------------

            --- remove gcd from pair of coefficients

   gcdCo(c1:Dom, c2:Dom):Record(co1:Dom,co2:Dom) ==
      d:=gcd(c1,c2)
      [(c1 exquo d)::Dom, (c2 exquo d)::Dom]

            --- calculate S-polynomial of a critical pair

   sPol(p:critPair)==
      Tij := p.lcmfij
      fi := p.poli
      fj := p.polj
      cc := gcdCo(leadingCoefficient fi, leadingCoefficient fj)
      reductum(fi)*monomial(cc.co2,subtractIfCan(Tij, degree fi)::Expon) -
        reductum(fj)*monomial(cc.co1,subtractIfCan(Tij, degree fj)::Expon)

            ----------------------------

            --- reduce critpair polynomial mod F
            --- iterative version

   redPo(s: Dpol, F: List(Dpol)) ==
      m:Dom := 1
      Fh := F
      while _^ ( s = 0 or null F ) repeat
        f1:= first(F)
        s1:= degree(s)
        e: Union(Expon, "failed")
        (e:= subtractIfCan(s1, degree(f1))) case Expon  =>
           cc:=gcdCo(leadingCoefficient f1, leadingCoefficient s)
           s:=cc.co1*reductum(s) - monomial(cc.co2,e)*reductum(f1)
           m := m*cc.co1
           F:= Fh
        F:= rest F
      [s,m]

   redPol(s: Dpol, F: List(Dpol)) ==  credPol(redPo(s,F).poly,F)

            ----------------------------

            --- crit T  true, if e1 and e2 are disjoint

   critT(p: critPair) == p.lcmfij =  (degree(p.poli) + degree(p.polj))

            ----------------------------
```

```
          --- crit M - true, if lcm#2 multiple of lcm#1

   critM(e1: Expon, e2: Expon) ==
         en: Union(Expon, "failed")
         (en:=subtractIfCan(e2, e1)) case Expon

            ----------------------------

            --- crit B - true, if eik is a multiple of eh and eik ^equal
            ---          lcm(eh,ei) and eik ^equal lcm(eh,ek)

   critB(eh:Expon, eik:Expon, ei:Expon, ek:Expon) ==
       critM(eh, eik) and (eik ^= sup(eh, ei)) and (eik ^= sup(eh, ek))

            ----------------------------

            ---  make polynomial monic case Domain a Field

   hMonic(p: Dpol) ==
        p= 0 => p
        -- inv(leadingCoefficient(p))*p
        primitivePart p

            ----------------------------

            ---  reduce all terms of h mod F  (iterative version )

   credPol(h: Dpol, F: List(Dpol) ) ==
        null F => h
        h0:Dpol:= monomial(leadingCoefficient h, degree h)
        while (h:=reductum h) ^= 0 repeat
           hred:= redPo(h, F)
           h := hred.poly
           h0:=(hred.mult)*h0 + monomial(leadingCoefficient(h),degree h)
        h0

            ------------------------------

            ----  calculate minimal basis for ordered F

   minGbasis(F: List(Dpol)) ==
        null F => nil
        newbas := minGbasis rest F
        cons(hMonic credPol( first(F), newbas),newbas)

            ------------------------------

            ----  calculate number of terms of polynomial

   lepol(p1:Dpol)==
```

```
      n: Integer
      n:= 0
      while p1 ^= 0 repeat
         n:= n + 1
         p1:= reductum(p1)
      n

            ----  print blanc lines

   prinb(n: Integer)==
      for x in 1..n repeat
         messagePrint("    ")

            ----  print reduced critpair polynom

   prinshINFO(h: Dpol)==
           prinb(2)
           messagePrint(" reduced Critpair - Polynom :")
           prinb(2)
           print(h::Ex)
           prinb(2)

            --------------------------------

            ----  print info string

   prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer,
             i2:Integer, n:Integer) ==
       ll: List Prinp
       a: Dom
       cpi:= cp.poli
       cpj:= cp.polj
       if n = 1 then
        prinb(1)
        messagePrint("you choose option  -info-  ")
        messagePrint("abbrev. for the following information strings are")
        messagePrint("  ci  =>  Leading monomial  for critpair calculation")
        messagePrint("  tci =>  Number of terms of polynomial i")
        messagePrint("  cj  =>  Leading monomial  for critpair calculation")
        messagePrint("  tcj =>  Number of terms of polynomial j")
        messagePrint("  c   =>  Leading monomial of critpair polynomial")
        messagePrint("  tc  =>  Number of terms of critpair polynomial")
        messagePrint("  rc  =>  Leading monomial of redcritpair polynomial")
        messagePrint("  trc =>  Number of terms of redcritpair polynomial")
        messagePrint("  tF  =>  Number of polynomials in reduction list F")
        messagePrint("  tD  =>  Number of critpairs still to do")
        prinb(4)
        n:= 2
       prinb(1)
       a:= 1
```

```
      ph = 0  =>
         ps = 0 =>
           ll:= [[monomial(a,degree(cpi)),lepol(cpi),
                 monomial(a,degree(cpj)),
                  lepol(cpj),ps,0,ph,0,i1,i2]$Prinp]
           print(ll::Ex)
           prinb(1)
           n
         ll:= [[monomial(a,degree(cpi)),lepol(cpi),
            monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
             lepol(ps), ph,0,i1,i2]$Prinp]
         print(ll::Ex)
         prinb(1)
         n
      ll:= [[monomial(a,degree(cpi)),lepol(cpi),
           monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
            lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2]$Prinp]
      print(ll::Ex)
      prinb(1)
      n

           -------------------------------

           ----  print the groebner basis polynomials

  prinpolINFO(pl: List(Dpol))==
      n:Integer
      n:= # pl
      prinb(1)
      n = 1 =>
        messagePrint("  There is 1  Groebner Basis Polynomial ")
        prinb(2)
      messagePrint("  There are ")
      prinb(1)
      print(n::Ex)
      prinb(1)
      messagePrint("  Groebner Basis Polynomials. ")
      prinb(2)

  fprindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer,
            i2:Integer, i3:Integer, n: Integer) ==
      ll: List Prinpp
      a: Dom
      cpi:= cp.poli
      cpj:= cp.polj
      if n = 1 then
       prinb(1)
       messagePrint("you choose option  -info-  ")
       messagePrint("abbrev. for the following information strings are")
       messagePrint("  ci  =>  Leading monomial  for critpair calculation")
```

```
 messagePrint("  tci =>  Number of terms of polynomial i")
 messagePrint("  cj  =>  Leading monomial  for critpair calculation")
 messagePrint("  tcj =>  Number of terms of polynomial j")
 messagePrint("  c   =>  Leading monomial of critpair polynomial")
 messagePrint("  tc  =>  Number of terms of critpair polynomial")
 messagePrint("  rc  =>  Leading monomial of redcritpair polynomial")
 messagePrint("  trc =>  Number of terms of redcritpair polynomial")
 messagePrint("  tF  =>  Number of polynomials in reduction list F")
 messagePrint("  tD  =>  Number of critpairs still to do")
 messagePrint("  tDF =>  Number of subproblems still to do")
 prinb(4)
 n:= 2
prinb(1)
a:= 1
ph = 0  =>
   ps = 0 =>
     ll:= [[monomial(a,degree(cpi)),lepol(cpi),
       monomial(a,degree(cpj)),
        lepol(cpj),ps,0,ph,0,i1,i2,i3]$Prinpp]
     print(ll::Ex)
     prinb(1)
     n
   ll:= [[monomial(a,degree(cpi)),lepol(cpi),
     monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
      lepol(ps), ph,0,i1,i2,i3]$Prinpp]
   print(ll::Ex)
   prinb(1)
   n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
     monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
      lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2,i3]$Prinpp]
print(ll::Ex)
prinb(1)
n
```

———-

— GBINTERN.dotabb —

```
"GBINTERN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GBINTERN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GBINTERN" -> "PFECAT"
```

———-

8.20 package GB GroebnerPackage

— GroebnerPackage.input —

```
)set break resume
)sys rm -f GroebnerPackage.output
)spool GroebnerPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 24
s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
--R
--R   (1)  45p + 35s - 165b - 36
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 1

--S 2 of 24
s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
--R
--R   (2)  35p + 40z + 25t - 27s
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 2

--S 3 of 24
s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
--R
--R                                      2
--R   (3)  15w + 25p s + 30z - 18t - 165b
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 3

--S 4 of 24
s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
--R
--R   (4)  - 9w + 15p t + 20z s
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 4

--S 5 of 24
s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
--R
--R                         3
--R   (5)  w p + 2z t - 11b
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 5

--S 6 of 24
```

```
s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
--R
--R                         2
--R   (6)  99w - 11s b + 3b
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 6

--S 7 of 24
s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
--R
--R         2   33      2673
--R   (7)  b  + -- b + -----
--R             50     10000
--R     Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 7

--S 8 of 24
sn7:=[s1,s2,s3,s4,s5,s6,s7]
--R
--R   (8)
--R   [45p + 35s - 165b - 36, 35p + 40z + 25t - 27s,
--R                                    2                                      3
--R    15w + 25p s + 30z - 18t - 165b , - 9w + 15p t + 20z s, w p + 2z t - 11b ,
--R                      2   2   33      2673
--R    99w - 11s b + 3b , b  + -- b + -----]
--R                             50     10000
--RType: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 8

--S 9 of 24
groebner(sn7)
--R
--R   (9)
--R         19      1323      31     153      49     1143      37      27
--R   [w + --- b + -----, p - -- b - ---, z + -- b + ----, t - -- b + ---,
--R        120     20000      18     200      36     2000      15     250
--R        5      9    2   33      2673
--R    s - - b - ---, b  + -- b + -----]
--R        2     200       50     10000
--RType: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 9

--S 10 of 24
groebner(sn7,"redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5     61     77      7
```

```
--R   z + - t - -- s + -- b + --
--R       8     45     24     10
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         66        603     278  2   11       672     2277     415881
--R   t s - -- t b + ---- t - --- s  + -- s b - --- s - ---- b - ------
--R         29       1450     435      29       725     7250     725000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       100  2   160       104      37      79
--R   t + --- s  - --- s b - --- s - --- b - ---
--R       189       63        63     105     125
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R    3   1026  2    5424  2   2529       1326807     12717      660717
--R   s  - ---- s b - ---- s  - ---- s b - ------- s + ----- b + -------
--R         145       3625       725        362500      6250     3625000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R      2     91248294  2   6550614        7087292937     20020838931
--R     s b + --------- s  - ------- s b + ----------- s - ----------- b
--R           128176525      5127061       12817652500     12817652500
--R   +
--R       37595502243
--R     - -----------
--R       51270610000
--R
--R
--R
--R
```

```
--R    reduced Critpair - Polynom :
--R
--R
--R      2   4746183626079988       1015195815329760     30723564870033201
--R     s  - ---------------- s b - ---------------- s - ----------------- b
--R           987357073521193        987357073521193     24683926838029825
--R   +
--R       3696123458901625353
--R     - -------------------
--R       2468392683802982500
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R           16827373608076633182513471     12627931635816456985349 64
--R     s b + -------------------------- s - ------------------------- b
--R           23063714246644859914108300     57659285616612149785270 75
--R   +
--R      915943452059811196524360 33
--R     --------------------------
--R     144148214041530374463176875
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5      9
--R   s - - b - ---
--R       2     200
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
```

```
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (10)
--R         19      1323      31     153      49     1143      37      27
--R   [w + --- b + -----, p - -- b - ---, z + -- b + ----, t - -- b + ---,
--R        120     20000      18     200      36     2000      15     250
--R        5      9    2   33      2673
--R    s - - b - ---, b  + -- b + -----]
--R        2     200       50     10000
--RType: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 10

--S 11 of 24
groebner(sn7,"info")
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R   [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 3]]
--R
--R
--R   [[ci= w,tci= 3,cj= w,tcj= 5,c= p t,tc= 6,rc= t s,trc= 8,tF= 5,tD= 2]]
--R
--R
--R   [[ci= w,tci= 3,cj= w,tcj= 3,c= p t,tc= 4,rc= t,trc= 6,tF= 5,tD= 2]]
--R
```

```
--R
--R                                                    3
--R   [[ci= t s,tci= 8,cj= t,tcj= 6,c= t b,tc= 9,rc= s ,trc= 7,tF= 6,tD= 1]]
--R
--R
--R                                                     2
--R   [[ci= w p,tci= 3,cj= w,tcj= 3,c= p s b,tc= 4,rc= s b,trc= 6,tF= 7,tD= 2]]
--R
--R
--R          2             2             2             2
--R   [[ci= b ,tci= 3,cj= s b,tcj= 6,c= s b,tc= 6,rc= s ,trc= 5,tF= 6,tD= 2]]
--R
--R
--R          2              2            2
--R   [[ci= s b,tci= 6,cj= s ,tcj= 5,c= s ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R          3             2            2
--R   [[ci= s ,tci= 7,cj= s ,tcj= 5,c= s b,tc= 6,rc= s b,trc= 4,tF= 7,tD= 2]]
--R
--R
--R          2
--R   [[ci= b ,tci= 3,cj= s b,tcj= 4,c= s b,tc= 4,rc= s,trc= 3,tF= 6,tD= 2]]
--R
--R
--R   [[ci= s b,tci= 4,cj= s,tcj= 3,c= s,tc= 4,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R          2
--R   [[ci= s ,tci= 5,cj= s,tcj= 3,c= s b,tc= 4,rc= 0,trc= 0,tF= 6,tD= 0]]
--R
--R
--R     There are
--R
--R   6
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (11)
--R         19      1323      31     153      49     1143      37      27
--R   [w + --- b + -----, p - -- b - ---, z + -- b + ----, t - -- b + ---,
--R        120     20000      18     200      36     2000      15     250
--R        5      9    2   33      2673
--R    s - - b - ---, b  + -- b + -----]
--R        2     200       50     10000
--RType: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 11
```

```
--S 12 of 24
groebner(sn7,"redcrit","info")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5     61     77      7
--R   z + - t - -- s + -- b + --
--R       8     45     24     10
--R
--R
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R   [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         66        603     278  2   11       672     2277     415881
--R   t s - -- t b + ---- t - --- s  + -- s b - --- s - ---- b - ------
--R         29       1450     435      29       725     7250     725000
--R
--R
--R
--R   [[ci= w,tci= 3,cj= w,tcj= 5,c= p t,tc= 6,rc= t s,trc= 8,tF= 5,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
```

```
--R
--R       100  2   160       104      37      79
--R   t + --- s  - --- s b - --- s - --- b - ---
--R       189       63        63     105     125
--R
--R
--R
--R   [[ci= w,tci= 3,cj= w,tcj= 3,c= p t,tc= 4,rc= t,trc= 6,tF= 5,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R    3   1026  2    5424  2   2529       1326807     12717      660717
--R   s  - ---- s b - ---- s  - ---- s b - ------- s + ----- b + -------
--R         145       3625       725        362500      6250     3625000
--R
--R
--R
--R                                                        3
--R   [[ci= t s,tci= 8,cj= t,tcj= 6,c= t b,tc= 9,rc= s ,trc= 7,tF= 6,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R      2     91248294  2   6550614        7087292937     20020838931
--R     s b + --------- s  - ------- s b + ----------- s - ----------- b
--R           128176525      5127061       12817652500     12817652500
--R   +
--R       37595502243
--R     - -----------
--R       51270610000
--R
--R
--R
--R                                                          2
--R   [[ci= w p,tci= 3,cj= w,tcj= 3,c= p s b,tc= 4,rc= s b,trc= 6,tF= 7,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R      2   4746183626079988       1015195815329760     30723564870033201
--R     s  - ---------------- s b - ---------------- s - ----------------- b
--R           987357073521193        987357073521193     24683926838029825
--R   +
```

```
--R       3696123458901625353
--R     - -------------------
--R       2468392683802982500
--R
--R
--R
--R         2             2             2             2
--R   [[ci= b ,tci= 3,cj= s b,tcj= 6,c= s b,tc= 6,rc= s ,trc= 5,tF= 6,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R         2              2            2
--R   [[ci= s b,tci= 6,cj= s ,tcj= 5,c= s ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R           16827373608076633182513471     12627931635816456985349 64
--R     s b + -------------------------- s - ------------------------- b
--R           23063714246644859914108300     57659285616612149785270 75
--R   +
--R      9159434520598111965243603 3
--R     ---------------------------
--R     14414821404153037446317687 5
--R
--R
--R
--R         3             2            2
--R   [[ci= s ,tci= 7,cj= s ,tcj= 5,c= s b,tc= 6,rc= s b,trc= 4,tF= 7,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5      9
--R   s - - b - ---
--R       2     200
--R
--R
--R
```

```
--R          2
--R   [[ci= b ,tci= 3,cj= s b,tcj= 4,c= s b,tc= 4,rc= s,trc= 3,tF= 6,tD= 2]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R   [[ci= s b,tci= 4,cj= s,tcj= 3,c= s,tc= 4,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R          2
--R   [[ci= s ,tci= 5,cj= s,tcj= 3,c= s b,tc= 4,rc= 0,trc= 0,tF= 6,tD= 0]]
--R
--R
--R     There are
--R
--R   6
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (12)
--R         19      1323      31     153      49     1143      37      27
--R   [w + --- b + -----, p - -- b - ---, z + -- b + ----, t - -- b + ---,
--R        120     20000      18     200      36     2000      15     250
--R        5      9    2   33      2673
--R    s - - b - ---, b  + -- b + -----]
--R        2     200       50     10000
--RType: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 12

--S 13 of 24
hs1:HDMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
--R
--R   (13)  45p + 35s - 165b - 36
```

```
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 13

--S 14 of 24
hs2:HDMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
--R
--R   (14)  35p + 40z + 25t - 27s
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 14

--S 15 of 24
hs3:HDMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
--R                     2
--R   (15)  25p s - 165b  + 15w + 30z - 18t
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 15

--S 16 of 24
hs4:HDMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
--R
--R   (16)  15p t + 20z s - 9w
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 16

--S 17 of 24
hs5:HDMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
--R
--R              3
--R   (17)  - 11b  + w p + 2z t
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 17

--S 18 of 24
hs6:HDMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
--R
--R                     2
--R   (18)  - 11s b + 3b  + 99w
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 18

--S 19 of 24
hs7:HDMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
--R
--R          2   33      2673
--R   (19)  b  + -- b + -----
--R              50     10000
--RType: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 19

--S 20 of 24
```

```
hsn7:=[hs1,hs2,hs3,hs4,hs5,hs6,hs7]
--R
--R   (20)
--R   [45p + 35s - 165b - 36, 35p + 40z + 25t - 27s,
--R               2                                          3
--R    25p s - 165b  + 15w + 30z - 18t, 15p t + 20z s - 9w, - 11b  + w p + 2z t,
--R               2         2   33      2673
--R    - 11s b + 3b  + 99w, b  + -- b + -----]
--R                             50     10000
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 20

--S 21 of 24
groebner(hsn7)
--R
--R   (21)
--R     2   33      2673       19      1323      31     153      49     1143
--R   [b  + -- b + -----, w + --- b + -----, p - -- b - ---, z + -- b + ----,
--R         50     10000      120     20000      18     200      36     2000
--R        37      27      5      9
--R    t - -- b + ---, s - - b - ---]
--R        15     250      2     200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 21

--S 22 of 24
groebner(hsn7,"redcrit")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5     61     77      7
--R   z + - t - -- s + -- b + --
--R       8     45     24     10
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R    2   216     189     78      99     10557
--R   s  - --- w + --- t - -- s + --- b - -----
--R         5      100     25     500     12500
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
```

```
--R
--R
--R         66       17541     5886     10588      9273     8272413
--R   t s - -- t b - ----- w + ---- t - ----- s - ----- b - -------
--R         29        725      3625      3625     36250     7250000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R      2   28       44       143       962712     420652     5166944
--R     t  + -- w s - -- w b + --- t b - ------ w + ------ t - ------- s
--R          45       15       725        18125      90625      815625
--R   +
--R     5036339     83580953
--R     ------- b - --------
--R     5437500     90625000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         33      297        81
--R   w b + -- w + ----- s - ----- b
--R         50     10000     10000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          21        33      6723      2031      104247
--R   w s + --- t b - --- w + ----- s - ----- b + -------
--R         100       250     50000     25000     5000000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2373       41563      17253      578853      258751      11330361
--R   w t + ---- t b - ----- w + ------ t + ------- s - ------- b + ---------
--R         7250       36250     290000     7250000     3625000     362500000
--R
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         51061712      91248294     1516761889      481096937      5789482077
--R   t b - -------- w + --------- t - ---------- s + ---------- b + -----------
--R          5127061     128176525     1922647875     1281765250     51270610000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2962071220563579     1229379913128787     4524811449715289
--R     w + ---------------- t - ---------------- s + ---------------- b
--R          98138188260880       36801820597830       490690941304400
--R   +
--R     59240140318722273
--R     -----------------
--R     12267273532610000
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       172832706542351932      47302810289036749      2736061156820726
--R   t - ------------------ s + ------------------ b + ----------------
--R       155991468675747195     155991468675747195     17332385408416355
--R
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5      9
--R   s - - b - ---
--R       2     200
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (22)
--R     2   33      2673       19      1323      31     153      49     1143
--R   [b  + -- b + -----, w + --- b + -----, p - -- b - ---, z + -- b + ----,
--R         50     10000      120     20000      18     200      36     2000
--R        37      27      5      9
--R    t - -- b + ---, s - - b - ---]
--R        15     250      2     200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 22

--S 23 of 24
groebner(hsn7,"info")
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
```

```
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R   [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 5]]
--R
--R
--R                                                      2
--R   [[ci= p s,tci= 5,cj= p,tcj= 4,c= z s,tc= 7,rc= s ,trc= 6,tF= 5,tD= 5]]
--R
--R
--R   [[ci= p t,tci= 3,cj= p,tcj= 4,c= z t,tc= 5,rc= t s,trc= 7,tF= 6,tD= 6]]
--R
--R
--R         3            2                          2
--R   [[ci= b ,tci= 3,cj= b ,tcj= 3,c= w p,tc= 4,rc= t ,trc= 9,tF= 7,tD= 6]]
--R
--R
--R                       2           3
--R   [[ci= s b,tci= 3,cj= b ,tcj= 3,c= b ,tc= 4,rc= w b,trc= 4,tF= 8,tD= 7]]
--R
--R
--R                       2             2
--R   [[ci= s b,tci= 3,cj= s ,tcj= 6,c= s b ,tc= 7,rc= w s,trc= 6,tF= 9,tD= 9]]
--R
--R
--R                                      2
--R   [[ci= s b,tci= 3,cj= t s,tcj= 7,c= t b ,tc= 7,rc= w t,trc= 7,tF= 10,tD= 11]]
--R
--R
--R                                      2
--R   [[ci= p s,tci= 5,cj= s b,tcj= 3,c= p b ,tc= 6,rc= 0,trc= 0,tF= 10,tD= 10]]
--R
--R
--R         2
--R   [[ci= s ,tci= 6,cj= t s,tcj= 7,c= t s b,tc= 10,rc= t b,trc= 6,tF= 11,tD= 13]]
--R
--R
--R         2
```

```
--R   [[ci= b ,tci= 3,cj= t b,tcj= 6,c= w b,tc= 6,rc= w,trc= 5,tF= 9,tD= 14]]
--R
--R
--R          2
--R   [[ci= b ,tci= 3,cj= w b,tcj= 4,c= s b,tc= 3,rc= 0,trc= 0,tF= 9,tD= 13]]
--R
--R
--R                                        2
--R   [[ci= s b,tci= 3,cj= t b,tcj= 6,c= t b ,tc= 7,rc= t,trc= 4,tF= 7,tD= 11]]
--R
--R
--R                                       2
--R   [[ci= s b,tci= 3,cj= w b,tcj= 4,c= w b ,tc= 5,rc= s,trc= 3,tF= 6,tD= 9]]
--R
--R
--R                                     2
--R   [[ci= w b,tci= 4,cj= t b,tcj= 6,c= w ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 8]]
--R
--R
--R                                   2
--R   [[ci= s b,tci= 3,cj= s,tcj= 3,c= b ,tc= 3,rc= 0,trc= 0,tF= 6,tD= 7]]
--R
--R
--R   [[ci= t b,tci= 6,cj= t,tcj= 4,c= s b,tc= 7,rc= 0,trc= 0,tF= 6,tD= 6]]
--R
--R
--R   [[ci= w b,tci= 4,cj= w,tcj= 5,c= t b,tc= 6,rc= 0,trc= 0,tF= 6,tD= 5]]
--R
--R
--R          2
--R   [[ci= s ,tci= 6,cj= s,tcj= 3,c= s b,tc= 6,rc= 0,trc= 0,tF= 6,tD= 4]]
--R
--R
--R                                   2
--R   [[ci= t s,tci= 7,cj= t,tcj= 4,c= s ,tc= 8,rc= 0,trc= 0,tF= 6,tD= 3]]
--R
--R
--R   [[ci= w s,tci= 6,cj= w,tcj= 5,c= t s,tc= 8,rc= 0,trc= 0,tF= 6,tD= 2]]
--R
--R
--R          2
--R   [[ci= t ,tci= 9,cj= t,tcj= 4,c= w s,tc= 9,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R                                   2
--R   [[ci= w t,tci= 7,cj= w,tcj= 5,c= t ,tc= 8,rc= 0,trc= 0,tF= 6,tD= 0]]
--R
--R
--R     There are
--R
```

```
--R   6
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (23)
--R     2   33      2673       19      1323      31     153      49     1143
--R   [b  + -- b + -----, w + --- b + -----, p - -- b - ---, z + -- b + ----,
--R         50     10000      120     20000      18     200      36     2000
--R        37      27      5      9
--R    t - -- b + ---, s - - b - ---]
--R        15     250      2     200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 23

--S 24 of 24
groebner(hsn7,"redcrit","info")
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5     61     77      7
--R   z + - t - -- s + -- b + --
--R       8     45     24     10
--R
--R
--R
--R   you choose option  -info-
--R   abbrev. for the following information strings are
--R     ci  =>  Leading monomial  for critpair calculation
--R     tci =>  Number of terms of polynomial i
--R     cj  =>  Leading monomial  for critpair calculation
--R     tcj =>  Number of terms of polynomial j
--R     c   =>  Leading monomial of critpair polynomial
--R     tc  =>  Number of terms of critpair polynomial
--R     rc  =>  Leading monomial of redcritpair polynomial
--R     trc =>  Number of terms of redcritpair polynomial
--R     tF  =>  Number of polynomials in reduction list F
--R     tD  =>  Number of critpairs still to do
--R
--R
--R
--R
--R
--R   [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 5]]
--R
--R
```

```
--R
--R    reduced Critpair - Polynom :
--R
--R
--R    2   216     189     78      99     10557
--R   s  - --- w + --- t - -- s + --- b - -----
--R         5      100     25     500     12500
--R
--R
--R
--R                                                 2
--R   [[ci= p s,tci= 5,cj= p,tcj= 4,c= z s,tc= 7,rc= s ,trc= 6,tF= 5,tD= 5]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         66       17541     5886     10588      9273     8272413
--R   t s - -- t b - ----- w + ---- t - ----- s - ----- b - -------
--R         29        725      3625      3625     36250     7250000
--R
--R
--R
--R   [[ci= p t,tci= 3,cj= p,tcj= 4,c= z t,tc= 5,rc= t s,trc= 7,tF= 6,tD= 6]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R      2   28       44       143       962712     420652     5166944
--R     t  + -- w s - -- w b + --- t b - ------ w + ------ t - ------- s
--R          45       15       725        18125      90625      815625
--R   +
--R     5036339     83580953
--R     ------- b - --------
--R     5437500     90625000
--R
--R
--R
--R          3             2                          2
--R   [[ci= b ,tci= 3,cj= b ,tcj= 3,c= w p,tc= 4,rc= t ,trc= 9,tF= 7,tD= 6]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         33      297        81
```

```
--R   w b + -- w + ----- s - ----- b
--R        50     10000     10000
--R
--R
--R
--R                          2            3
--R   [[ci= s b,tci= 3,cj= b ,tcj= 3,c= b ,tc= 4,rc= w b,trc= 4,tF= 8,tD= 7]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R          21        33      6723      2031      104247
--R   w s + --- t b - --- w + ----- s - ----- b + -------
--R        100       250     50000     25000     5000000
--R
--R
--R
--R                          2              2
--R   [[ci= s b,tci= 3,cj= s ,tcj= 6,c= s b ,tc= 7,rc= w s,trc= 6,tF= 9,tD= 9]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2373       41563      17253      578853      258751      11330361
--R   w t + ---- t b - ----- w + ------ t + ------- s - ------- b + ---------
--R         7250       36250     290000     7250000     3625000     362500000
--R
--R
--R
--R                                        2
--R   [[ci= s b,tci= 3,cj= t s,tcj= 7,c= t b ,tc= 7,rc= w t,trc= 7,tF= 10,tD= 11]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                                        2
--R   [[ci= p s,tci= 5,cj= s b,tcj= 3,c= p b ,tc= 6,rc= 0,trc= 0,tF= 10,tD= 10]]
--R
--R
--R
```

```
--R    reduced Critpair - Polynom :
--R
--R
--R         51061712      91248294     1516761889      481096937      5789482077
--R   t b - -------- w + --------- t - ---------- s + ---------- b + -----------
--R          5127061     128176525     1922647875     1281765250     51270610000
--R
--R
--R
--R          2
--R   [[ci= s ,tci= 6,cj= t s,tcj= 7,c= t s b,tc= 10,rc= t b,trc= 6,tF= 11,tD= 13]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R         2962071220563579     1229379913128787     4524811449715289
--R     w + ---------------- t - ---------------- s + ---------------- b
--R          98138188260880       36801820597830       490690941304400
--R   +
--R     59240140318722273
--R     -----------------
--R     12267273532610000
--R
--R
--R
--R          2
--R   [[ci= b ,tci= 3,cj= t b,tcj= 6,c= w b,tc= 6,rc= w,trc= 5,tF= 9,tD= 14]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R          2
--R   [[ci= b ,tci= 3,cj= w b,tcj= 4,c= s b,tc= 3,rc= 0,trc= 0,tF= 9,tD= 13]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R        172832706542351932      47302810289036749      2736061156820726
--R   t - ------------------ s + ------------------ b + ----------------
--R        155991468675747195     155991468675747195     17332385408416355
```

```
--R
--R
--R
--R                                        2
--R   [[ci= s b,tci= 3,cj= t b,tcj= 6,c= t b ,tc= 7,rc= t,trc= 4,tF= 7,tD= 11]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R       5      9
--R   s - - b - ---
--R       2     200
--R
--R
--R
--R                                         2
--R   [[ci= s b,tci= 3,cj= w b,tcj= 4,c= w b ,tc= 5,rc= s,trc= 3,tF= 6,tD= 9]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                                       2
--R   [[ci= w b,tci= 4,cj= t b,tcj= 6,c= w ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 8]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                                     2
--R   [[ci= s b,tci= 3,cj= s,tcj= 3,c= b ,tc= 3,rc= 0,trc= 0,tF= 6,tD= 7]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
```

```
--R
--R
--R
--R   [[ci= t b,tci= 6,cj= t,tcj= 4,c= s b,tc= 7,rc= 0,trc= 0,tF= 6,tD= 6]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R   [[ci= w b,tci= 4,cj= w,tcj= 5,c= t b,tc= 6,rc= 0,trc= 0,tF= 6,tD= 5]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R          2
--R   [[ci= s ,tci= 6,cj= s,tcj= 3,c= s b,tc= 6,rc= 0,trc= 0,tF= 6,tD= 4]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                                      2
--R   [[ci= t s,tci= 7,cj= t,tcj= 4,c= s ,tc= 8,rc= 0,trc= 0,tF= 6,tD= 3]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R   [[ci= w s,tci= 6,cj= w,tcj= 5,c= t s,tc= 8,rc= 0,trc= 0,tF= 6,tD= 2]]
```

```
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R          2
--R   [[ci= t ,tci= 9,cj= t,tcj= 4,c= w s,tc= 9,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R
--R    reduced Critpair - Polynom :
--R
--R
--R   0
--R
--R
--R
--R                                    2
--R   [[ci= w t,tci= 7,cj= w,tcj= 5,c= t ,tc= 8,rc= 0,trc= 0,tF= 6,tD= 0]]
--R
--R
--R     There are
--R
--R   6
--R
--R     Groebner Basis Polynomials.
--R
--R
--R       THE GROEBNER BASIS POLYNOMIALS
--R
--R   (24)
--R     2   33      2673       19      1323      31     153      49     1143
--R   [b  + -- b + -----, w + --- b + -----, p - -- b - ---, z + -- b + ----,
--R         50     10000      120     20000      18     200      36     2000
--R        37      27      5      9
--R    t - -- b + ---, s - - b - ---]
--R        15     250      2     200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 24

)spool
)lisp (bye)
```

— GroebnerPackage.help —

```
=================================================================
groebner examples
=================================================================

Example to call groebner:

  s1:DMP[w,p,z,t,s,b]RN:= 45*p + 35*s - 165*b - 36
  s2:DMP[w,p,z,t,s,b]RN:= 35*p + 40*z + 25*t - 27*s
  s3:DMP[w,p,z,t,s,b]RN:= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
  s4:DMP[w,p,z,t,s,b]RN:= -9*w + 15*p*t + 20*z*s
  s5:DMP[w,p,z,t,s,b]RN:= w*p + 2*z*t - 11*b**3
  s6:DMP[w,p,z,t,s,b]RN:= 99*w - 11*b*s + 3*b**2
  s7:DMP[w,p,z,t,s,b]RN:= b**2 + 33/50*b + 2673/10000

  sn7:=[s1,s2,s3,s4,s5,s6,s7]

  groebner(sn7,info)

groebner calculates a minimal Groebner Basis
all reductions are TOTAL reductions

To get the reduced critical pairs do:

  groebner(sn7,"redcrit")

You can get other information by calling:

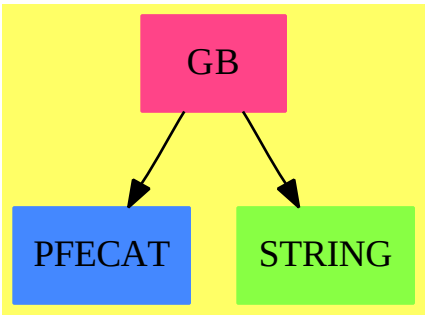
  groebner(sn7,"info")

which returns:
      ci  =>  Leading monomial  for critpair calculation
      tci =>  Number of terms of polynomial i
      cj  =>  Leading monomial  for critpair calculation
      tcj =>  Number of terms of polynomial j
      c   =>  Leading monomial of critpair polynomial
      tc  =>  Number of terms of critpair polynomial
      rc  =>  Leading monomial of redcritpair polynomial
      trc =>  Number of terms of redcritpair polynomial
      tF  =>  Number of polynomials in reduction list F
      tD  =>  Number of critpairs still to do

See Also:
o )display operations groebner
o )show GroebnerPackage
o )show DistributedMultivariatePolynomial
o )show HomogeneousDistributedMultivariatePolynomial
o )show EuclideanGroebnerBasisPackage
```

———-

8.20.1 GroebnerPackage (GB)



Exports:
groebner normalForm

— package GB GroebnerPackage —

```
)abbrev package GB GroebnerPackage
++ Authors: Gebauer, Trager
++ Date Created: 12-1-86
++ Date Last Updated: 2-28-91
++ Basic Functions: groebner normalForm
++ Related Constructors: Ideal, IdealDecompositionPackage
++ Also See:
++ AMS Classifications:
++ Keywords: groebner basis, polynomial ideal
++ References:
++ Description:
++ \spadtype{GroebnerPackage} computes groebner
++ bases for polynomial ideals. The basic computation provides a distinguished
++ set of generators for polynomial ideals over fields. This basis allows an
++ easy test for membership: the operation \spadfun{normalForm}
++ returns zero on ideal members. When the provided coefficient domain, Dom,
++ is not a field, the result is equivalent to considering the extended
++ ideal with \spadtype{Fraction(Dom)} as coefficients, but considerably more
++ efficient since all calculations are performed in Dom. Additional
++ argument "info" and "redcrit" can be given to provide incremental
++ information during computation. Argument "info" produces a computational
++ summary for each s-polynomial.
++ Argument "redcrit" prints out the reduced critical pairs. The term ordering
```

```
++ is determined by the polynomial type used. Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

GroebnerPackage(Dom, Expon, VarSet, Dpol): T == C where

 Dom:   GcdDomain
 Expon: OrderedAbelianMonoidSup
 VarSet: OrderedSet
 Dpol:  PolynomialCategory(Dom, Expon, VarSet)

 T== with

  groebner: List(Dpol) -> List(Dpol)
   ++ groebner(lp) computes a groebner basis for a polynomial ideal
   ++ generated by the list of polynomials lp.
   ++
   ++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
   ++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
   ++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
   ++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
   ++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
   ++X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
   ++X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
   ++X sn7:=[s1,s2,s3,s4,s5,s6,s7]
   ++X groebner(sn7)

  groebner: ( List(Dpol), String ) -> List(Dpol)
   ++ groebner(lp, infoflag) computes a groebner basis
   ++ for a polynomial ideal
   ++ generated by the list of polynomials lp.
   ++ Argument infoflag is used to get information on the computation.
   ++ If infoflag is "info", then summary information
   ++ is displayed for each s-polynomial generated.
   ++ If infoflag is "redcrit", the reduced critical pairs are displayed.
   ++ If infoflag is any other string,
   ++ no information is printed during computation.
   ++
   ++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
   ++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
   ++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
   ++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
   ++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
   ++X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
   ++X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
   ++X sn7:=[s1,s2,s3,s4,s5,s6,s7]
   ++X groebner(sn7,"info")
   ++X groebner(sn7,"redcrit")
```

```
  groebner: ( List(Dpol), String, String ) -> List(Dpol)
   ++ groebner(lp, "info", "redcrit") computes a groebner basis
   ++ for a polynomial ideal generated by the list of polynomials lp,
   ++ displaying both a summary of the critical pairs considered ("info")
   ++ and the result of reducing each critical pair ("redcrit").
   ++ If the second or third arguments have any other string value,
   ++ the indicated information is suppressed.
   ++
   ++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
   ++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
   ++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
   ++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
   ++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
   ++X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
   ++X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
   ++X sn7:=[s1,s2,s3,s4,s5,s6,s7]
   ++X groebner(sn7,"info","redcrit")

  if Dom has Field then
    normalForm: (Dpol, List(Dpol))  -> Dpol
      ++ normalForm(poly,gb) reduces the polynomial poly modulo the
      ++ precomputed groebner basis gb giving a canonical representative
      ++ of the residue class.
 C== add
   import OutputForm
   import GroebnerInternalPackage(Dom,Expon,VarSet,Dpol)

   if Dom has Field then
     monicize(p: Dpol):Dpol ==
--       one?(lc := leadingCoefficient p) => p
       ((lc := leadingCoefficient p) = 1) => p
       inv(lc)*p

     normalForm(p : Dpol, l : List(Dpol)) : Dpol ==
       redPol(p,map(monicize,l))

   ------    MAIN ALGORITHM GROEBNER ------------------------

   groebner( Pol: List(Dpol) ) ==
     Pol=[] => Pol
     Pol:=[x for x in Pol | x ^= 0]
     Pol=[] => [0]
     minGbasis(sort((x,y) +->  degree x > degree y, gbasis(Pol,0,0)))

   groebner( Pol: List(Dpol), xx1: String) ==
     Pol=[] => Pol
     Pol:=[x for x in Pol | x ^= 0]
     Pol=[] => [0]
     xx1 = "redcrit" =>
       minGbasis(sort((x,y) +->  degree x > degree y, gbasis(Pol,1,0)))
```

```
    xx1 = "info" =>
      minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,2,1)))
    messagePrint("   ")
    messagePrint("WARNING: options are - redcrit and/or info - ")
    messagePrint("         you didn't type them correct")
    messagePrint("         please try again")
    messagePrint("   ")
    []

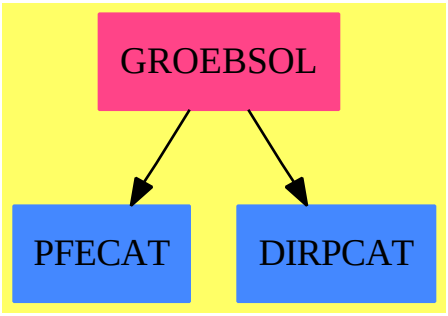
  groebner( Pol: List(Dpol), xx1: String, xx2: String) ==
    Pol=[] => Pol
    Pol:=[x for x in Pol | x ^= 0]
    Pol=[] => [0]
    (xx1 = "redcrit" and xx2 = "info") or
     (xx1 = "info" and xx2 = "redcrit")   =>
      minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,1)))
    xx1 = "redcrit" and xx2 = "redcrit" =>
      minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,0)))
    xx1 = "info" and xx2 = "info" =>
      minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,2,1)))
    messagePrint("   ")
    messagePrint("WARNING:  options are - redcrit and/or info - ")
    messagePrint("          you didn't type them correctly")
    messagePrint("          please try again ")
    messagePrint("   ")
    []
```

— GB.dotabb —

```
"GB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GB"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"GB" -> "PFECAT"
"GB" -> "STRING"
```

8.21 package GROEBSOL GroebnerSolve

8.21.1 GroebnerSolve (GROEBSOL)



Exports:
genericPosition groebSolve testDim

— package GROEBSOL GroebnerSolve —

```
)abbrev package GROEBSOL GroebnerSolve
++ Author : P.Gianni, Summer '88, revised November '89
++ Description:
++ Solve systems of polynomial equations using Groebner bases
++ Total order Groebner bases are computed and then converted to lex ones
++ This package is mostly intended for internal use.

GroebnerSolve(lv,F,R) : C == T

  where
   R      :   GcdDomain
   F      :   GcdDomain
   lv     :   List Symbol

   NNI    ==>  NonNegativeInteger
   I      ==>  Integer
   S      ==>  Symbol

   OV     ==>  OrderedVariableList(lv)
   IES    ==>  IndexedExponents Symbol

   DP     ==>  DirectProduct(#lv,NonNegativeInteger)
   DPoly  ==>  DistributedMultivariatePolynomial(lv,F)

   HDP    ==>  HomogeneousDirectProduct(#lv,NonNegativeInteger)
   HDPoly ==>  HomogeneousDistributedMultivariatePolynomial(lv,F)
```

```
SUP    ==>  SparseUnivariatePolynomial(DPoly)
L      ==>  List
P      ==>  Polynomial

C == with
   groebSolve   : (L DPoly,L OV)  -> L L DPoly
     ++ groebSolve(lp,lv) reduces the polynomial system lp in variables lv
     ++ to triangular form. Algorithm based on groebner bases algorithm
     ++ with linear algebra for change of ordering.
     ++ Preprocessing for the general solver.
     ++ The polynomials in input are of type \spadtype{DMP}.

   testDim     : (L HDPoly,L OV)  -> Union(L HDPoly,"failed")
     ++ testDim(lp,lv) tests if the polynomial system lp
     ++ in variables lv is zero dimensional.

   genericPosition : (L DPoly, L OV) -> Record(dpolys:L DPoly, coords: L I)
     ++ genericPosition(lp,lv) puts a radical zero dimensional ideal
     ++ in general position, for system lp in variables lv.

T == add
  import PolToPol(lv,F)
  import GroebnerPackage(F,DP,OV,DPoly)
  import GroebnerInternalPackage(F,DP,OV,DPoly)
  import GroebnerPackage(F,HDP,OV,HDPoly)
  import LinGroebnerPackage(lv,F)

  nv:NNI:=#lv

       ---- test if f is power of a linear mod (rad lpol) ----
                  ----  f is monic  ----
  testPower(uf:SUP,x:OV,lpol:L DPoly) : Union(DPoly,"failed") ==
    df:=degree(uf)
    trailp:DPoly := coefficient(uf,(df-1)::NNI)
    (testquo := trailp exquo (df::F)) case "failed" => "failed"
    trailp := testquo::DPoly
    gg:=gcd(lc:=leadingCoefficient(uf),trailp)
    trailp := (trailp exquo gg)::DPoly
    lc := (lc exquo gg)::DPoly
    linp:SUP:=monomial(lc,1$NNI)$SUP + monomial(trailp,0$NNI)$SUP
    g:DPoly:=multivariate(uf-linp**df,x)
    redPol(g,lpol) ^= 0 => "failed"
    multivariate(linp,x)

         -- is the 0-dimensional ideal I in general position ?  --
                  ----  internal function  ----
  testGenPos(lpol:L DPoly,lvar:L OV):Union(L DPoly,"failed") ==
    rlpol:=reverse lpol
    f:=rlpol.first
    #lvar=1 => [f]
```

```
  rlvar:=rest reverse lvar
  newlpol:List(DPoly):=[f]
  for f in rlpol.rest repeat
    x:=first rlvar
    fi:= univariate(f,x)
    if (mainVariable leadingCoefficient fi case "failed") then
      if ((g:= testPower(fi,x,newlpol)) case "failed")
      then return "failed"
      newlpol :=concat(redPol(g::DPoly,newlpol),newlpol)
      rlvar:=rest rlvar
    else if redPol(f,newlpol)^=0 then return"failed"
  newlpol


   -- change coordinates and out the ideal in general position  ----
genPos(lp:L DPoly,lvar:L OV): Record(polys:L HDPoly, lpolys:L DPoly,
                                     coord:L I, univp:HDPoly) ==
      rlvar:=reverse lvar
      lnp:=[dmpToHdmp(f) for f in lp]
      x := first rlvar;rlvar:=rest rlvar
      testfail:=true
      for count in 1.. while testfail repeat
        ranvals:L I:=[1+(random()$I rem (count*(# lvar))) for vv in rlvar]
        val:=+/[rv*(vv::HDPoly)
                   for vv in rlvar for rv in ranvals]
        val:=val+x::HDPoly
        gb:L HDPoly:= [elt(univariate(p,x),val) for p in lnp]
        gb:=groebner gb
        gbt:=totolex gb
        (gb1:=testGenPos(gbt,lvar)) case "failed"=>"try again"
        testfail:=false
      [gb,gbt,ranvals,dmpToHdmp(last (gb1::L DPoly))]

genericPosition(lp:L DPoly,lvar:L OV) ==
   nans:=genPos(lp,lvar)
   [nans.lpolys, nans.coord]

   ---- select  the univariate factors
select(lup:L L HDPoly) : L L HDPoly ==
  lup=[] => list []
  [:[cons(f,lsel) for lsel in select lup.rest] for f in lup.first]

   ---- in the non generic case, we compute the prime ideals ----
      ---- associated to leq, basis is the algebra basis  ----
findCompon(leq:L HDPoly,lvar:L OV):L L DPoly ==
  teq:=totolex(leq)
  #teq = #lvar => [teq]
 -- ^((teq1:=testGenPos(teq,lvar)) case "failed") => [teq1::L DPoly]
  gp:=genPos(teq,lvar)
  lgp:= gp.polys
```

```
  g:HDPoly:=gp.univp
  fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDP,R,F,HDPoly)
  lfact:=[ff.factor for ff in factors(fg::Factored(HDPoly))]
  result: L L HDPoly := []
  #lfact=1 => [teq]
  for tfact in lfact repeat
    tlfact:=concat(tfact,lgp)
    result:=concat(tlfact,result)
  ranvals:L I:=gp.coord
  rlvar:=reverse lvar
  x:=first rlvar
  rlvar:=rest rlvar
  val:=+/[rv*(vv::HDPoly) for vv in rlvar for rv in ranvals]
  val:=(x::HDPoly)-val
  ans:=[totolex groebner [elt(univariate(p,x),val) for p in lp]
                      for lp in result]
  [ll for ll in ans | ll^=[1]]

zeroDim?(lp: List HDPoly,lvar:L OV) : Boolean ==
  empty? lp => false
  n:NNI := #lvar
  #lp < n => false
  lvint1 := lvar
  for f in lp while not empty?(lvint1) repeat
     g:= f - reductum f
     x:=mainVariable(g)::OV
     if ground?(leadingCoefficient(univariate(g,x))) then
          lvint1 := remove(x, lvint1)
  empty? lvint1

-- general solve, gives an error if the system not 0-dimensional
groebSolve(leq: L DPoly,lvar:L OV) : L L DPoly ==
  lnp:=[dmpToHdmp(f) for f in leq]
  leq1:=groebner lnp
  #(leq1) = 1 and first(leq1) = 1 => list empty()
  ^(zeroDim?(leq1,lvar)) =>
    error "system does not have a finite number of solutions"
  -- add computation of dimension, for a more useful error
  basis:=computeBasis(leq1)
  lup:L HDPoly:=[]
  llfact:L Factored(HDPoly):=[]
  for x in lvar repeat
    g:=minPol(leq1,basis,x)
    fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDP,R,F,HDPoly)
    llfact:=concat(fg::Factored(HDPoly),llfact)
    if degree(g,x) = #basis then leave "stop factoring"
  result: L L DPoly := []
  -- selecting a factor from the lists of the univariate factors
  lfact:=select [[ff.factor for ff in factors llf]
                  for llf in llfact]
```

```
      for tfact in lfact repeat
        tfact:=groebner concat(tfact,leq1)
        tfact=[1] => "next value"
        result:=concat(result,findCompon(tfact,lvar))
      result

    -- test if the system is zero dimensional
    testDim(leq : L HDPoly,lvar : L OV) : Union(L HDPoly,"failed") ==
      leq1:=groebner leq
      #(leq1) = 1 and first(leq1) = 1 => empty()
      ^(zeroDim?(leq1,lvar)) => "failed"
      leq1
```

———-

— GROEBSOL.dotabb —

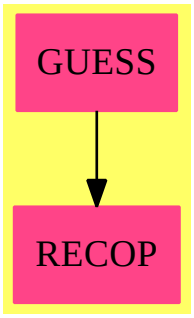
```
"GROEBSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GROEBSOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"GROEBSOL" -> "PFECAT"
"GROEBSOL" -> "DIRPCAT"
```

———-

8.22 package GUESS Guess

The packages defined in this file enable Axiom to guess formulas for sequences of, for example, rational numbers or rational functions, given the first few terms. It extends and complements Christian Krattenthaler's program Rate and the relevant parts of Bruno Salvy and Paul Zimmermann's GFUN.

8.22.1 Guess (GUESS)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

The original code would not compile. This is a temporary replacement with changes marked with my initials. I will pick up the latest version sometime in the future and hope it compiles. (Tim Daly)

— **package GUESS Guess** —

```
)abbrev package GUESS Guess
++ Author: Martin Rubey, Timothy Daly
++ Description:
++ This package implements guessing of sequences. Packages for the
++ most common cases are provided as \spadtype{GuessInteger},
++ \spadtype{GuessPolynomial}, etc.

Guess(F, S, EXPRR, R, retract, coerce): Exports == Implementation where
    F: Field                                  -- zB.: FRAC POLY PF 5
-- in F we interpolate und check

    S: GcdDomain

-- in guessExpRat I would like to determine the roots of polynomials in F. When
-- F is a quotientfield, I can get rid of the denominator.  In this case F is
-- roughly QFCAT S

    R: Join(OrderedSet, IntegralDomain)      -- zB.: FRAC POLY INT

-- results are given as elements of EXPRR
--    EXPRR: Join(ExpressionSpace, IntegralDomain,
    EXPRR: Join(FunctionSpace Integer, IntegralDomain,
                RetractableTo R, RetractableTo Symbol,
                RetractableTo Integer, CombinatorialOpsCategory,
                PartialDifferentialRing Symbol) with
```

```
            _* : (%,%) -> %
            _/ : (%,%) -> %
            _*_* : (%,%) -> %
            numerator : % -> %
            denominator : % -> %
            ground? : % -> Boolean

                                              -- zB.: EXPR INT
-- EXPR FRAC POLY INT is forbidden. Thus i cannot just use EXPR R

-- EXPRR exists, in case at some point there is support for EXPR PF 5.


-- the following I really would like to get rid of

    retract: R -> F                           -- zB.: i+->i
    coerce: F -> EXPRR                        -- zB.: i+->i
-- attention: EXPRR ~= EXPR R

    LGOPT ==> List GuessOption
    GOPT0 ==> GuessOptionFunctions0

    NNI ==> NonNegativeInteger
    PI ==> PositiveInteger
    EXPRI ==> Expression Integer
    GUESSRESULT ==> List Record(function: EXPRR, order: NNI)

    UFPSF ==> UnivariateFormalPowerSeries F
    UFPS1 ==> UnivariateFormalPowerSeriesFunctions

    UFPSS ==> UnivariateFormalPowerSeries S

    SUP ==> SparseUnivariatePolynomial

    UFPSSUPF ==> UnivariateFormalPowerSeries SUP F

    FFFG ==> FractionFreeFastGaussian
    FFFGF ==> FractionFreeFastGaussianFractions

-- CoeffAction
    DIFFSPECA ==> (NNI, NNI, SUP S) -> S

    DIFFSPECAF ==> (NNI, NNI, UFPSSUPF) -> SUP F

    DIFFSPECAX ==> (NNI, Symbol, EXPRR) -> EXPRR

-- the diagonal of the C-matrix
    DIFFSPECC ==> NNI -> List S
```

```
    HPSPEC ==> Record(guessStream:  UFPSF -> Stream UFPSF,
                      degreeStream: Stream NNI,
                      testStream:   UFPSSUPF -> Stream UFPSSUPF,
                      exprStream:   (EXPRR, Symbol) -> Stream EXPRR,
                      A:  DIFFSPECA,
                      AF: DIFFSPECAF,
                      AX: DIFFSPECAX,
                      C:  DIFFSPECC)

-- note that empty?(guessStream.o) has to return always. In other words, if the
-- stream is finite, empty? should recognize it.

    DIFFSPECN ==> EXPRR -> EXPRR             -- eg.: i+->q^i

    GUESSER ==> (List F, LGOPT) -> GUESSRESULT

    FSUPS ==> Fraction SUP S
    FSUPF ==> Fraction SUP F

    V ==> OrderedVariableList(['a1, 'A])
    POLYF ==> SparseMultivariatePolynomial(F, V)
    FPOLYF ==> Fraction POLYF
    FSUPFPOLYF ==> Fraction SUP FPOLYF
    POLYS ==> SparseMultivariatePolynomial(S, V)
    FPOLYS ==> Fraction POLYS
    FSUPFPOLYS ==> Fraction SUP FPOLYS

    --@<<implementation: Guess - Hermite-Pade - Types for Operators>>
    -- EXT ==> (Integer, V, V) -> FPOLYS
    -- EXTEXPR ==> (Symbol, F, F) -> EXPRR
    Exports == with

        guess: List F -> GUESSRESULT
          ++ \spad{guess l} applies recursively \spadfun{guessRec} and
          ++ \spadfun{guessADE} to the successive differences and quotients of
          ++ the list. Default options as described in
          ++ \spadtype{GuessOptionFunctions0} are used.

        guess: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guess(l, options)} applies recursively \spadfun{guessRec}
          ++ and \spadfun{guessADE} to the successive differences and quotients
          ++ of the list. The given options are used.

        guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
          ++ \spad{guess(l, guessers, ops)} applies recursively the given
          ++ guessers to the successive differences if ops contains the symbol
          ++ guessSum and quotients if ops contains the symbol guessProduct to
          ++ the list. Default options as described in
          ++ \spadtype{GuessOptionFunctions0} are used.
```

```
guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT
  ++ \spad{guess(l, guessers, ops)} applies recursively the given
  ++ guessers to the successive differences if ops contains the symbol
  ++ \spad{guessSum} and quotients if ops contains the symbol
  ++ \spad{guessProduct} to the list. The given options are used.

guessExpRat: List F -> GUESSRESULT
  ++ \spad{guessExpRat l} tries to find a function of the form
  ++ n+->(a+b n)^n r(n), where r(n) is a rational function, that fits
  ++ l.

guessExpRat: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessExpRat(l, options)} tries to find a function of the
  ++ form n+->(a+b n)^n r(n), where r(n) is a rational function, that
  ++ fits l.

guessBinRat: List F -> GUESSRESULT
  ++ \spad{guessBinRat(l, options)} tries to find a function of the
  ++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
  ++ function, that fits l.

guessBinRat: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessBinRat(l, options)} tries to find a function of the
  ++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
  ++ function, that fits l.

if F has RetractableTo Symbol and S has RetractableTo Symbol then

    guessExpRat: Symbol -> GUESSER
      ++ \spad{guessExpRat q} returns a guesser that tries to find a
      ++ function of the form n+->(a+b q^n)^n r(q^n), where r(q^n) is a
      ++ q-rational function, that fits l.

    guessBinRat: Symbol -> GUESSER
      ++ \spad{guessBinRat q} returns a guesser that tries to find a
      ++ function of the form n+->qbinomial(a+b n, n) r(n), where r(q^n) is a
      ++ q-rational function, that fits l.

guessHP: (LGOPT -> HPSPEC) -> GUESSER
  ++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
  ++ approximation to the series generated by the given function f.

guessADE: List F -> GUESSRESULT
  ++ \spad{guessADE l} tries to find an algebraic differential equation
  ++ for a generating function whose first Taylor coefficients are
  ++ given by l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}.

guessADE: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessADE(l, options)} tries to find an algebraic
```

```
  ++ differential equation for a generating function whose first Taylor
  ++ coefficients are given by l, using the given options.

guessAlg: List F -> GUESSRESULT
  ++ \spad{guessAlg l} tries to find an algebraic equation for a
  ++ generating function whose first Taylor coefficients are given by
  ++ l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessADE}(l, maxDerivative == 0).

guessAlg: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessAlg(l, options)} tries to find an algebraic equation
  ++ for a generating function whose first Taylor coefficients are
  ++ given by l, using the given options. It is equivalent to
  ++ \spadfun{guessADE}(l, options) with \spad{maxDerivative == 0}.

guessHolo: List F -> GUESSRESULT
  ++ \spad{guessHolo l} tries to find an ordinary linear differential
  ++ equation for a generating function whose first Taylor coefficients
  ++ are given by l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessADE}\spad{(l, maxPower == 1)}.

guessHolo: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessHolo(l, options)} tries to find an ordinary linear
  ++ differential equation for a generating function whose first Taylor
  ++ coefficients are given by l, using the given options. It is
  ++ equivalent to \spadfun{guessADE}\spad{(l, options)} with
  ++ \spad{maxPower == 1}.

guessPade: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessPade(l, options)} tries to find a rational function
  ++ whose first Taylor coefficients are given by l, using the given
  ++ options. It is equivalent to \spadfun{guessADE}\spad{(l,
  ++ maxDerivative == 0, maxPower == 1, allDegrees == true)}.

guessPade: List F -> GUESSRESULT
  ++ \spad{guessPade(l, options)} tries to find a rational function
  ++ whose first Taylor coefficients are given by l, using the default
  ++ options described in \spadtype{GuessOptionFunctions0}. It is
  ++ equivalent to \spadfun{guessADE}\spad{(l, options)} with
  ++ \spad{maxDerivative == 0, maxPower == 1, allDegrees == true}.

guessRec: List F -> GUESSRESULT
  ++ \spad{guessRec l} tries to find an ordinary difference equation
  ++ whose first values are given by l, using the default options
  ++ described in \spadtype{GuessOptionFunctions0}.

guessRec: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessRec(l, options)} tries to find an ordinary difference
```

```
  ++ equation whose first values are given by l, using the given
  ++ options.

guessPRec: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessPRec(l, options)} tries to find a linear recurrence
  ++ with polynomial coefficients whose first values are given by l,
  ++ using the given options. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, options)} with \spad{maxPower == 1}.

guessPRec: List F -> GUESSRESULT
  ++ \spad{guessPRec l} tries to find a linear recurrence with
  ++ polynomial coefficients whose first values are given by l, using
  ++ the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, maxPower == 1)}.

guessRat: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessRat(l, options)} tries to find a rational function
  ++ whose first values are given by l, using the given options. It is
  ++ equivalent to \spadfun{guessRec}\spad{(l, maxShift == 0, maxPower
  ++ == 1, allDegrees == true)}.

guessRat: List F -> GUESSRESULT
  ++ \spad{guessRat l} tries to find a rational function whose first
  ++ values are given by l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, maxShift == 0, maxPower == 1,
  ++ allDegrees == true)}.

diffHP: LGOPT -> HPSPEC
  ++ \spad{diffHP options} returns a specification for Hermite-Pade
  ++ approximation with the differential operator

shiftHP: LGOPT -> HPSPEC
  ++ \spad{shiftHP options} returns a specification for Hermite-Pade
  ++ approximation with the shift operator

if F has RetractableTo Symbol and S has RetractableTo Symbol then

    shiftHP: Symbol -> (LGOPT -> HPSPEC)
      ++ \spad{shiftHP options} returns a specification for
      ++ Hermite-Pade approximation with the $q$-shift operator

    diffHP: Symbol -> (LGOPT -> HPSPEC)
      ++ \spad{diffHP options} returns a specification for Hermite-Pade
      ++ approximation with the  $q$-dilation operator

    guessRec: Symbol -> GUESSER
      ++ \spad{guessRec q} returns a guesser that finds an ordinary
      ++ q-difference equation whose first values are given by l, using
```

```
            ++ the given options.

          guessPRec: Symbol -> GUESSER
            ++ \spad{guessPRec q} returns a guesser that tries to find
            ++ a linear q-recurrence with polynomial coefficients whose first
            ++ values are given by l, using the given options. It is
            ++ equivalent to \spadfun{guessRec}\spad{(q)} with
            ++ \spad{maxPower == 1}.

          guessRat: Symbol -> GUESSER
            ++ \spad{guessRat q} returns a guesser that tries to find a
            ++ q-rational function whose first values are given by l, using
            ++ the given options. It is equivalent to \spadfun{guessRec} with
            ++ \spad{(l, maxShift == 0, maxPower == 1, allDegrees == true)}.

          guessADE: Symbol -> GUESSER
            ++ \spad{guessADE q} returns a guesser that tries to find an
            ++ algebraic differential equation for a generating function whose
            ++ first Taylor coefficients are given by l, using the given
            ++ options.

        --@<<debug exports: Guess>>
    Implementation == add

-- We have to put this chunk at the beginning, because otherwise it will take
-- very long to compile.

      ord1(x: List Integer, i: Integer): Integer ==
          n := #x - 3 - i
          x.(n+1)*reduce(_+, [x.j for j in 1..n], 0) + _
              2*reduce(_+, [reduce(_+, [x.k*x.j for k in 1..j-1], 0) _
                      for j in 1..n], 0)

      ord2(x: List Integer, i: Integer): Integer ==
          if zero? i then
              n := #x - 3 - i
              ord1(x, i) + reduce(_+, [x.j for j in 1..n], 0)*(x.(n+2)-x.(n+1))
          else
              ord1(x, i)

      deg1(x: List Integer, i: Integer): Integer ==
          m := #x - 3
          (x.(m+3)+x.(m+1)+x.(1+i))*reduce(_+, [x.j for j in 2+i..m], 0) + _
              x.(m+3)*x.(m+1) + _
              2*reduce(_+, [reduce(_+, [x.k*x.j for k in 2+i..j-1], 0) _
                            for j in 2+i..m], 0)

      deg2(x: List Integer, i: Integer): Integer ==
          m := #x - 3
          deg1(x, i) + _
```

```
        (x.(m+3) + reduce(_+, [x.j for j in 2+i..m], 0)) * _
        (x.(m+2)-x.(m+1))


checkResult(res: EXPRR, n: Symbol, l: Integer, list: List F,
            options: LGOPT): NNI ==
    for i in l..1 by -1 repeat
        den := eval(denominator res, n::EXPRR, (i-1)::EXPRR)
        if den = 0 then return i::NNI
        num := eval(numerator res, n::EXPRR, (i-1)::EXPRR)
        if list.i ~= retract(retract(num/den)@R)
        then return i::NNI
    0$NNI

SUPS2SUPF(p: SUP S): SUP F ==
  if F is S then
    p pretend SUP(F)
  else if F is Fraction S then
    map(coerce(#1)$Fraction(S), p)
      $SparseUnivariatePolynomialFunctions2(S, F)
  else error "Type parameter F should be either equal to S or equal _
              to Fraction S"

F2FPOLYS(p: F): FPOLYS ==
  if F is S then
    p::POLYF::FPOLYF pretend FPOLYS
  else if F is Fraction S then
    numer(p)$Fraction(S)::POLYS/denom(p)$Fraction(S)::POLYS
  else error "Type parameter F should be either equal to S or equal _
               to Fraction S"

MPCSF ==> MPolyCatFunctions2(V, IndexedExponents V,
                                IndexedExponents V, S, F,
                                POLYS, POLYF)

SUPF2EXPRR(xx: Symbol, p: SUP F): EXPRR ==
  zero? p => 0
  (coerce(leadingCoefficient p))::EXPRR * (xx::EXPRR)**degree p
     + SUPF2EXPRR(xx, reductum p)

FSUPF2EXPRR(xx: Symbol, p: FSUPF): EXPRR ==
  (SUPF2EXPRR(xx, numer p)) / (SUPF2EXPRR(xx, denom p))


POLYS2POLYF(p: POLYS): POLYF ==
  if F is S then
    p pretend POLYF
  else if F is Fraction S then
    map(coerce(#1)$Fraction(S), p)$MPCSF
  else error "Type parameter F should be either equal to S or equal _
```

```
                to Fraction S"

      SUPPOLYS2SUPF(p: SUP POLYS, a1v: F, Av: F): SUP F ==
        zero? p => 0
        lc: POLYF := POLYS2POLYF leadingCoefficient(p)
        monomial(retract(eval(lc, [index(1)$V, index(2)$V]::List V,
                                  [a1v, Av])),
                 degree p)
          + SUPPOLYS2SUPF(reductum p, a1v, Av)


      SUPFPOLYS2FSUPPOLYS(p: SUP FPOLYS): Fraction SUP POLYS  ==
        cden := splitDenominator(p)
               $UnivariatePolynomialCommonDenominator(POLYS, FPOLYS,SUP FPOLYS)

        pnum: SUP POLYS
             := map(retract(#1 * cden.den)$FPOLYS, p)
                  $SparseUnivariatePolynomialFunctions2(FPOLYS, POLYS)
        pden: SUP POLYS := (cden.den)::SUP POLYS

        pnum/pden


      POLYF2EXPRR(p: POLYF): EXPRR ==
        map(convert(#1)@Symbol::EXPRR, coerce(#1)@EXPRR, p)
           $PolynomialCategoryLifting(IndexedExponents V, V,
                                      F, POLYF, EXPRR)

-- this needs documentation. In particular, why is V appearing here?
      GF ==> GeneralizedMultivariateFactorize(SingletonAsOrderedSet,
                                              IndexedExponents V, F, F,
                                              SUP F)

-- does not work:
--                                                    6
--   WARNING (genufact): No known algorithm to factor ? , trying square-free.

-- GF ==> GenUFactorize F
      defaultD: DIFFSPECN
      defaultD(expr: EXPRR): EXPRR == expr

      -- applies n+->q^n or whatever DN is to i
      DN2DL: (DIFFSPECN, Integer) -> F
      DN2DL(DN, i) == retract(retract(DN(i::EXPRR))@R)

      evalResultant(p1: POLYS, p2: POLYS, o: Integer, d: Integer, va1: V, vA: V)_
      : List S ==
          res: List S := []
          d1 := degree(p1, va1)
          d2 := degree(p2, va1)
```

```
        lead: S
        for k in 1..d-o+1 repeat
            p1atk := univariate eval(p1, vA, k::S)
            p2atk := univariate eval(p2, vA, k::S)

            d1atk := degree p1atk
            d2atk := degree p2atk

--          output("k: " string(k))$OutputPackage
--          output("d1: " string(d1) " d1atk: " string(d1atk))$OutputPackage
--          output("d2: " string(d2) " d2atk: " string(d2atk))$OutputPackage


            if d2atk < d2 then
                if  d1atk < d1
                then lead := 0$S
                else lead := (leadingCoefficient p1atk)**((d2-d2atk)::NNI)
            else
                if  d1atk < d1
                then lead := (-1$S)**d2 * (leadingCoefficient p2atk)**((d1-d1atk)::NNI)
                else lead := 1$S

            if zero? lead
            then res := cons(0, res)
            else res := cons(lead * (resultant(p1atk, p2atk)$SUP(S) exquo _
                                    (k::S)**(o::NNI))::S,
                             res)

        reverse res

    p(xm: Integer, i: Integer, va1: V, vA: V, basis: DIFFSPECN): FPOLYS ==
        vA::POLYS::FPOLYS + va1::POLYS::FPOLYS _
                           * F2FPOLYS(DN2DL(basis, i) - DN2DL(basis, xm))

    p2(xm: Integer, i: Symbol, a1v: F, Av: F, basis: DIFFSPECN): EXPRR ==
        coerce(Av) + coerce(a1v)*(basis(i::EXPRR) - basis(xm::EXPRR))

    guessExpRatAux(xx: Symbol, list: List F, basis: DIFFSPECN,
                   xValues: List Integer, options: LGOPT): List EXPRR ==

        a1: V := index(1)$V
        A: V := index(2)$V

        len: NNI := #list
        if len < 4 then return []
                   else len := (len-3)::NNI

        xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]
        x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))
        x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))
```

```
        x3 := F2FPOLYS DN2DL(basis, xValues.(len+3))

        y: NNI -> FPOLYS :=
            F2FPOLYS(list.#1) * _
            p(last xValues, (xValues.#1)::Integer, a1, A, basis)**_
                (-(xValues.#1)::Integer)

        ylist: List FPOLYS := [y i for i in 1..len]

        y1 := y(len+1)
        y2 := y(len+2)
        y3 := y(len+3)

        res := []::List EXPRR
    -- tpd: this is undefined since maxDegree is always nonnegative
    --     if maxDegree(options)$GOPT0 = -1
    --     then maxDeg := len-1
    --     else maxDeg := min(maxDegree(options)$GOPT0, len-1)
    --     maxDeg := min(maxDegree(options)$GOPT0, len-1)
        tpd:Integer := (maxDegree(options)$GOPT0)::NNI::Integer
        maxDeg:Integer:=min(tpd,len-1)

        for i in 0..maxDeg repeat
            if debug(options)$GOPT0 then
                output(hconcat("degree ExpRat "::OutputForm, i::OutputForm))
                    $OutputPackage

            if debug(options)$GOPT0 then
                systemCommand("sys date +%s")$MoreSystemCommands
                output("interpolating..."::OutputForm)$OutputPackage

            ri: FSUPFPOLYS
               := interpolate(xlist, ylist, (len-1-i)::NNI) _
                             $FFFG(FPOLYS, SUP FPOLYS)

     -- for experimental fraction free interpolation
     --       ri: Fraction SUP POLYS
     --          := interpolate(xlist, ylist, (len-1-i)::NNI) _
     --                        $FFFG(POLYS, SUP POLYS)

            if debug(options)$GOPT0 then
     --           output(hconcat("xlist: ", xlist::OutputForm))$OutputPackage
     --           output(hconcat("ylist: ", ylist::OutputForm))$OutputPackage
     --           output(hconcat("ri: ", ri::OutputForm))$OutputPackage
                systemCommand("sys date +%s")$MoreSystemCommands
                output("polynomials..."::OutputForm)$OutputPackage

            poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)
            poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)
            poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)
```

```
-- for experimental fraction free interpolation
--        ri2: FSUPFPOLYS := map(#1::FPOLYS, numer ri) _
--               $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)_
--                 /map(#1::FPOLYS, denom ri)  _
--                   $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)
--
--        poly1: POLYS := numer(elt(ri2, x1)$SUP(FPOLYS) - y1)
--        poly2: POLYS := numer(elt(ri2, x2)$SUP(FPOLYS) - y2)
--        poly3: POLYS := numer(elt(ri2, x3)$SUP(FPOLYS) - y3)

        n:Integer := len - i
        o1: Integer := ord1(xValues, i)
        d1: Integer := deg1(xValues, i)
        o2: Integer := ord2(xValues, i)
        d2: Integer := deg2(xValues, i)

-- another compiler bug: using i as iterator here makes the loop break

        if debug(options)$GOPT0 then
            systemCommand("sys date +%s")$MoreSystemCommands
            output("interpolating resultants..."::OutputForm)$OutputPackage

        res1: SUP S
             := newton(evalResultant(poly1, poly3, o1, d1, a1, A))
                      $NewtonInterpolation(S)

        res2: SUP S
             := newton(evalResultant(poly2, poly3, o2, d2, a1, A))
                      $NewtonInterpolation(S)

        if debug(options)$GOPT0 then
--             res1: SUP S := univariate(resultant(poly1, poly3, a1))
--             res2: SUP S := univariate(resultant(poly2, poly3, a1))
--             if res1 ~= res1res or res2 ~= res2res then
--             output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
--                 output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
--             output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
--                 output(hconcat("res1 ", res1::OutputForm))$OutputPackage
--                 output(hconcat("res2 ", res2::OutputForm))$OutputPackage
            output("n/i: " string(n) " " string(i))$OutputPackage
            output("res1 ord: " string(o1) " " string(minimumDegree res1))
                  $OutputPackage
            output("res1 deg: " string(d1) " " string(degree res1))
                  $OutputPackage
            output("res2 ord: " string(o2) " " string(minimumDegree res2))
                  $OutputPackage
            output("res2 deg: " string(d2) " " string(degree res2))
                  $OutputPackage
```

```
        if debug(options)$GOPT0 then
            systemCommand("sys date +%s")$MoreSystemCommands
            output("computing gcd..."::OutputForm)$OutputPackage

-- we want to solve over F
-- for polynomial domains S this seems to be very costly!
        res3: SUP F := SUPS2SUPF(primitivePart(gcd(res1, res2)))

        if debug(options)$GOPT0 then
            systemCommand("sys date +%s")$MoreSystemCommands
            output("solving..."::OutputForm)$OutputPackage

-- res3 is a polynomial in A=a0+(len+3)*a1
-- now we have to find the roots of res3

        for f in factors factor(res3)$GF | degree f.factor = 1 repeat
-- we are only interested in the linear factors
            if debug(options)$GOPT0 then
                output(hconcat("f: ", f::OutputForm))$OutputPackage

            Av: F := -coefficient(f.factor, 0)
                    / leadingCoefficient f.factor

-- FIXME: in an earlier version, we disregarded vanishing Av
--        maybe we intended to disregard vanishing a1v? Either doesn't really
--        make sense to me right now.

            evalPoly := eval(POLYS2POLYF poly3, A, Av)
            if zero? evalPoly
            then evalPoly := eval(POLYS2POLYF poly1, A, Av)
-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([1,1,1,1]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

            for g in factors factor(univariate evalPoly)$GF
                     | degree g.factor = 1 repeat
                if debug(options)$GOPT0 then
                    output(hconcat("g: ", g::OutputForm))$OutputPackage

                a1v: F := -coefficient(g.factor, 0)
                         / leadingCoefficient g.factor

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.

                t1 := eval(POLYS2POLYF poly1, [a1, A]::List V,
                                              [a1v, Av]::List F)
```

```
                    if zero? t1 then
                        t2 := eval(POLYS2POLYF poly2, [a1, A]::List V,
                                                      [a1v, Av]::List F)
                        if zero? t2 then

                            ri1: Fraction SUP POLYS
                                := SUPFPOLYS2FSUPPOLYS(numer ri)
                                 / SUPFPOLYS2FSUPPOLYS(denom ri)

    -- for experimental fraction free interpolation
    --                          ri1: Fraction SUP POLYS := ri

                            numr: SUP F := SUPPOLYS2SUPF(numer ri1, a1v, Av)
                            denr: SUP F := SUPPOLYS2SUPF(denom ri1, a1v, Av)

                            if not zero? denr then
                                res4: EXPRR := eval(FSUPF2EXPRR(xx, numr/denr),
                                                    kernel(xx),
                                                    basis(xx::EXPRR))
                                                *p2(last xValues, _
                                                    xx, a1v, Av, basis)_
                                                 **xx::EXPRR
                                res := cons(res4, res)
                            else if zero? numr and debug(options)$GOPT0 then
                                output("numerator and denominator vanish!")
                                      $OutputPackage

    -- If we are only interested in one solution, we do not try other degrees if we
    -- have found already some solutions. I.e., the indentation here is correct.

            if not null(res) and one(options)$GOPT0 then return res

        res

    guessExpRatAux0(list: List F, basis: DIFFSPECN, options: LGOPT): GUESSRESULT ==
        if zero? safety(options)$GOPT0 then
            error "Guess: guessExpRat does not support zero safety"
    -- guesses Functions of the Form (a1*n+a0)^n*rat(n)
        xx := indexName(options)$GOPT0

    -- restrict to safety

        len: Integer := #list
        if len-safety(options)$GOPT0+1 < 0 then return []

        shortlist: List F := first(list, (len-safety(options)$GOPT0+1)::NNI)

    -- remove zeros from list

        zeros: EXPRR := 1
```

```
        newlist: List F
        xValues: List Integer

        i: Integer := -1
        for x in shortlist repeat
            i := i+1
            if x = 0 then
                zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))

        i := -1
        for x in shortlist repeat
            i := i+1
            if x ~= 0 then
                newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR,
                                                       i::EXPRR))@R),
                                newlist)
                xValues := cons(i, xValues)

        newlist := reverse newlist
        xValues := reverse xValues

        res: List EXPRR
            := [eval(zeros * f, xx::EXPRR, xx::EXPRR) _
                for f in guessExpRatAux(xx, newlist, basis, xValues, options)]

        reslist := map([#1, checkResult(#1, xx, len, list, options)], res)
                     $ListFunctions2(EXPRR, Record(function: EXPRR, order: NNI))

        select(#1.order < len-safety(options)$GOPT0, reslist)

    guessExpRat(list : List F): GUESSRESULT ==
        guessExpRatAux0(list, defaultD, [])

    guessExpRat(list: List F, options: LGOPT): GUESSRESULT ==
        guessExpRatAux0(list, defaultD, options)

    if F has RetractableTo Symbol and S has RetractableTo Symbol then

        guessExpRat(q: Symbol): GUESSER ==
      guessExpRatAux0(#1, q::EXPRR**#1, #2)


    EXT ==> (Integer, V, V) -> FPOLYS
    EXTEXPR ==> (Symbol, F, F) -> EXPRR

    binExt: EXT
    binExt(i: Integer, va1: V, vA: V): FPOLYS ==
        numl: List POLYS := [(vA::POLYS) + i * (va1::POLYS) - (l::POLYS) _
                             for l in 0..i-1]
        num: POLYS := reduce(_*, numl, 1)
```

```
        num/(factorial(i)::POLYS)

    binExtEXPR: EXTEXPR
    binExtEXPR(i: Symbol, a1v: F, Av: F): EXPRR ==
        binomial(coerce Av + coerce a1v * (i::EXPRR), i::EXPRR)


    guessBinRatAux(xx: Symbol, list: List F,
                   basis: DIFFSPECN, ext: EXT, extEXPR: EXTEXPR,
                   xValues: List Integer, options: LGOPT): List EXPRR ==

        a1: V := index(1)$V
        A: V := index(2)$V

        len: NNI := #list
        if len < 4 then return []
                   else len := (len-3)::NNI

        xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]
        x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))
        x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))
        x3 := F2FPOLYS DN2DL(basis, xValues.(len+3))

        y: NNI -> FPOLYS :=
            F2FPOLYS(list.#1) / _
            ext((xValues.#1)::Integer, a1, A)

        ylist: List FPOLYS := [y i for i in 1..len]

        y1 := y(len+1)
        y2 := y(len+2)
        y3 := y(len+3)

        res := []::List EXPRR
    -- tpd: this is undefined since maxDegree is always nonnegative
    --     if maxDegree(options)$GOPT0 = -1
    --     then maxDeg := len-1
    --     else maxDeg := min(maxDegree(options)$GOPT0, len-1)
    --     maxDeg := min(maxDegree(options)$GOPT0, len-1)
        tpd:Integer := (maxDegree(options)$GOPT0)::NNI::Integer
        maxDeg:Integer:=min(tpd,len-1)
--          maxDeg:Integer := (maxDegree(options)$GOPT0)::NNI::Integer

        for i in 0..maxDeg repeat
     --        if debug(options)$GOPT0 then
     --            output(hconcat("degree BinRat "::OutputForm, i::OutputForm))
     --                $OutputPackage

     --        if debug(options)$GOPT0 then
```

```
--            output("interpolating..."::OutputForm)$OutputPackage

        ri: FSUPFPOLYS
           := interpolate(xlist, ylist, (len-1-i)::NNI) _
                        $FFFG(FPOLYS, SUP FPOLYS)

--        if debug(options)$GOPT0 then
--            output(hconcat("ri ", ri::OutputForm))$OutputPackage

        poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)
        poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)
        poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)

--        if debug(options)$GOPT0 then
--            output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
--            output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
--            output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage


        n:Integer := len - i
        res1: SUP S := univariate(resultant(poly1, poly3, a1))
        res2: SUP S := univariate(resultant(poly2, poly3, a1))
        if debug(options)$GOPT0 then
--            output(hconcat("res1 ", res1::OutputForm))$OutputPackage
--            output(hconcat("res2 ", res2::OutputForm))$OutputPackage

--            if res1 ~= res1res or res2 ~= res2res then
--            output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
--                output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
--            output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
--                output(hconcat("res1 ", res1::OutputForm))$OutputPackage
--                output(hconcat("res2 ", res2::OutputForm))$OutputPackage
--            output("n/i: " string(n) " " string(i))$OutputPackage
              output("res1 ord: " string(minimumDegree res1))
                    $OutputPackage
              output("res1 deg: " string(degree res1))
                    $OutputPackage
              output("res2 ord: " string(minimumDegree res2))
                    $OutputPackage
              output("res2 deg: " string(degree res2))
                    $OutputPackage

        if debug(options)$GOPT0 then
            output("computing gcd..."::OutputForm)$OutputPackage

-- we want to solve over F
        res3: SUP F := SUPS2SUPF(primitivePart(gcd(res1, res2)))

--        if debug(options)$GOPT0 then
--            output(hconcat("res3 ", res3::OutputForm))$OutputPackage
```

```
-- res3 is a polynomial in A=a0+(len+3)*a1
-- now we have to find the roots of res3

       for f in factors factor(res3)$GF | degree f.factor = 1 repeat
-- we are only interested in the linear factors
--             if debug(options)$GOPT0 then
--                 output(hconcat("f: ", f::OutputForm))$OutputPackage

            Av: F := -coefficient(f.factor, 0)
                    / leadingCoefficient f.factor

--             if debug(options)$GOPT0 then
--                 output(hconcat("Av: ", Av::OutputForm))$OutputPackage

-- FIXME: in an earlier version, we disregarded vanishing Av
--        maybe we intended to disregard vanishing a1v? Either doesn't really
--        make sense to me right now.

            evalPoly := eval(POLYS2POLYF poly3, A, Av)
            if zero? evalPoly
            then evalPoly := eval(POLYS2POLYF poly1, A, Av)
-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([1,1,1,1]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

            for g in factors factor(univariate evalPoly)$GF
                     | degree g.factor = 1 repeat
--                 if debug(options)$GOPT0 then
--                     output(hconcat("g: ", g::OutputForm))$OutputPackage

                a1v: F := -coefficient(g.factor, 0)
                         / leadingCoefficient g.factor

--                 if debug(options)$GOPT0 then
--                     output(hconcat("a1v: ", a1v::OutputForm))$OutputPackage

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.

                t1 := eval(POLYS2POLYF poly1, [a1, A]::List V,
                                              [a1v, Av]::List F)

--                 if debug(options)$GOPT0 then
--                     output(hconcat("t1: ", t1::OutputForm))$OutputPackage

                if zero? t1 then
```

```
                        t2 := eval(POLYS2POLYF poly2, [a1, A]::List V,
                                                      [a1v, Av]::List F)

--                        if debug(options)$GOPT0 then
--                            output(hconcat("t2: ", t2::OutputForm))$OutputPackage

                        if zero? t2 then

                            ri1: Fraction SUP POLYS
                                := SUPFPOLYS2FSUPPOLYS(numer ri)
                                 / SUPFPOLYS2FSUPPOLYS(denom ri)

--                            if debug(options)$GOPT0 then
--                                output(hconcat("ri1: ", ri1::OutputForm))$OutputPackage

                            numr: SUP F := SUPPOLYS2SUPF(numer ri1, a1v, Av)
                            denr: SUP F := SUPPOLYS2SUPF(denom ri1, a1v, Av)

--                            if debug(options)$GOPT0 then
--                                output(hconcat("numr: ", numr::OutputForm))$OutputPackage
--                                output(hconcat("denr: ", denr::OutputForm))$OutputPackage

                            if not zero? denr then
                                res4: EXPRR := eval(FSUPF2EXPRR(xx, numr/denr),
                                                    kernel(xx),
                                                    basis(xx::EXPRR))
                                               * extEXPR(xx, a1v, Av)

--                                  if debug(options)$GOPT0 then
--                                      output(hconcat("res4: ", res4::OutputForm))$OutputPacka

                                res := cons(res4, res)
                            else if zero? numr and debug(options)$GOPT0 then
                                output("numerator and denominator vanish!")
                                      $OutputPackage

        -- If we are only interested in one solution, we do not try other degrees if we
        -- have found already some solutions. I.e., the indentation here is correct.

                if not null(res) and one(options)$GOPT0 then return res

            res

        guessBinRatAux0(list: List F,
                        basis: DIFFSPECN, ext: EXT, extEXPR: EXTEXPR,
                        options: LGOPT): GUESSRESULT ==

            if zero? safety(options)$GOPT0 then
                error "Guess: guessBinRat does not support zero safety"
        -- guesses Functions of the form binomial(a+b*n, n)*rat(n)
```

```
        xx := indexName(options)$GOPT0

    -- restrict to safety

        len: Integer := #list
        if len-safety(options)$GOPT0+1 < 0 then return []

        shortlist: List F := first(list, (len-safety(options)$GOPT0+1)::NNI)

    -- remove zeros from list

        zeros: EXPRR := 1
        newlist: List F
        xValues: List Integer

        i: Integer := -1
        for x in shortlist repeat
            i := i+1
            if x = 0 then
                zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))

        i := -1
        for x in shortlist repeat
            i := i+1
            if x ~= 0 then
                newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR,
                                                       i::EXPRR))@R),
                                newlist)
                xValues := cons(i, xValues)

        newlist := reverse newlist
        xValues := reverse xValues

        res: List EXPRR
            := [eval(zeros * f, xx::EXPRR, xx::EXPRR) _
                for f in guessBinRatAux(xx, newlist, basis, ext, extEXPR, xValues, _
                                        options)]

        reslist := map([#1, checkResult(#1, xx, len, list, options)], res)
                     $ListFunctions2(EXPRR, Record(function: EXPRR, order: NNI))

        select(#1.order < len-safety(options)$GOPT0, reslist)

    guessBinRat(list : List F): GUESSRESULT ==
        guessBinRatAux0(list, defaultD, binExt, binExtEXPR, [])

    guessBinRat(list: List F, options: LGOPT): GUESSRESULT ==
        guessBinRatAux0(list, defaultD, binExt, binExtEXPR, options)
```

```
if F has RetractableTo Symbol and S has RetractableTo Symbol then

    qD: Symbol -> DIFFSPECN
    qD q == (q::EXPRR)**#1


    qBinExtAux(q: Symbol, i: Integer, va1: V, vA: V): FPOLYS ==
        fl: List FPOLYS
             := [(1$FPOLYS - _
                  va1::POLYS::FPOLYS * (vA::POLYS::FPOLYS)**(i-1) * _
                  F2FPOLYS(q::F)**l) / (1-F2FPOLYS(q::F)**l) _
                 for l in 1..i]
        reduce(_*, fl, 1)

    qBinExt: Symbol -> EXT
    qBinExt q == qBinExtAux(q, #1, #2, #3)

    qBinExtEXPRaux(q: Symbol, i: Symbol, a1v: F, Av: F): EXPRR ==
        l: Symbol := 'l
        product((1$EXPRR - _
                 coerce a1v * (coerce Av) ** (coerce i - 1$EXPRR) * _
                 (q::EXPRR) ** coerce(l)) / _
                (1$EXPRR - (q::EXPRR) ** coerce(l)), _
                equation(l, 1$EXPRR..i::EXPRR))

    qBinExtEXPR: Symbol -> EXTEXPR
    qBinExtEXPR q == qBinExtEXPRaux(q, #1, #2, #3)

    guessBinRat(q: Symbol): GUESSER ==
        guessBinRatAux0(#1, qD q, qBinExt q, qBinExtEXPR q, #2)_

-- some useful types for Ore operators that work on series

-- the differentiation operator
DIFFSPECX ==> (EXPRR, Symbol, NonNegativeInteger) -> EXPRR
                                                -- eg.: f(x)+->f(q*x)
                                                --      f(x)+->D(f, x)
DIFFSPECS ==> (UFPSF, NonNegativeInteger) -> UFPSF
                                                -- eg.: f(x)+->f(q*x)

DIFFSPECSF ==> (UFPSSUPF, NonNegativeInteger) -> UFPSSUPF
                                                -- eg.: f(x)+->f(q*x)

-- the constant term for the inhomogeneous case

DIFFSPEC1 ==> UFPSF

DIFFSPEC1F ==> UFPSSUPF

DIFFSPEC1X ==> Symbol -> EXPRR
```

```
termAsEXPRR(f: EXPRR, xx: Symbol, l: List Integer,
            DX: DIFFSPECX, D1X: DIFFSPEC1X): EXPRR ==
    if empty? l then D1X(xx)
    else
        ll: List List Integer := powers(l)$Partition

        fl: List EXPRR := [DX(f, xx, (first part-1)::NonNegativeInteger)
                           ** second(part)::NNI for part in ll]
        reduce(_*, fl)

termAsUFPSF(f: UFPSF, l: List Integer, DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF ==
    if empty? l then D1
    else
        ll: List List Integer := powers(l)$Partition

-- first of each element of ll is the derivative, second is the power

        fl: List UFPSF := [DS(f, (first part -1)::NonNegativeInteger) _
                           ** second(part)::NNI for part in ll]

        reduce(_*, fl)

-- returns \prod f^(l.i), but using the Hadamard product
termAsUFPSF2(f: UFPSF, l: List Integer,
             DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF ==
    if empty? l then D1
    else
        ll: List List Integer := powers(l)$Partition

-- first of each element of ll is the derivative, second is the power

        fl: List UFPSF
            := [map(#1** second(part)::NNI, DS(f, (first part -1)::NNI)) _
                for part in ll]

        reduce(hadamard$UFPS1(F), fl)


termAsUFPSSUPF(f: UFPSSUPF, l: List Integer,
                    DSF: DIFFSPECSF, D1F: DIFFSPEC1F): UFPSSUPF ==
    if empty? l then D1F
    else
        ll: List List Integer := powers(l)$Partition

-- first of each element of ll is the derivative, second is the power

        fl: List UFPSSUPF
           := [DSF(f, (first part -1)::NonNegativeInteger)
               ** second(part)::NNI for part in ll]
```

```
        reduce(_*, fl)


-- returns \prod f^(l.i), but using the Hadamard product
termAsUFPSSUPF2(f: UFPSSUPF, l: List Integer,
                DSF: DIFFSPECSF, D1F: DIFFSPEC1F): UFPSSUPF ==
    if empty? l then D1F
    else
        ll: List List Integer := powers(l)$Partition

-- first of each element of ll is the derivative, second is the power

        fl: List UFPSSUPF
           := [map(#1 ** second(part)::NNI, DSF(f, (first part -1)::NNI)) _
               for part in ll]

        reduce(hadamard$UFPS1(SUP F), fl)

FilteredPartitionStream(options: LGOPT): Stream List Integer ==
--tpd: must force types to NNI and Integer
    maxD:Integer := 1+maxDerivative(options)$GOPT0::NNI::Integer
    maxP:Integer := maxPower(options)$GOPT0::NNI::Integer

    if maxD > 0 and maxP > -1 then
        s := partitions(maxD, maxP)$PartitionsAndPermutations
    else
        s1: Stream Integer := generate(inc, 1)$Stream(Integer)
        s2: Stream Stream List Integer
           := map(partitions(#1)$PartitionsAndPermutations, s1)
                 $StreamFunctions2(Integer, Stream List Integer)
        s3: Stream List Integer
           := concat(s2)$StreamFunctions1(List Integer)

--        s := cons([],
--                  select(((maxD = 0) or (first #1 <= maxD)) _
--                     and ((maxP = -1) or (# #1 <= maxP)), s3))

        s := cons([],
                  select(((maxD = 0) or (# #1 <= maxD)) _
                     and ((maxP = -1) or (first #1 <= maxP)), s3))

    s := conjugates(s)$PartitionsAndPermutations
    --tpd: force the Boolean branch
    tpd2:Boolean:=homogeneous(options)$GOPT0::Boolean
    if tpd2 then rest s else s

-- for functions
ADEguessStream(f: UFPSF, partitions: Stream List Integer,
               DS: DIFFSPECS, D1: DIFFSPEC1): Stream UFPSF ==
```

```
    map(termAsUFPSF(f, #1, DS, D1), partitions)
       $StreamFunctions2(List Integer, UFPSF)

-- for coefficients, i.e., using the Hadamard product
ADEguessStream2(f: UFPSF, partitions: Stream List Integer,
                DS: DIFFSPECS, D1: DIFFSPEC1): Stream UFPSF ==
    map(termAsUFPSF2(f, #1, DS, D1), partitions)
       $StreamFunctions2(List Integer, UFPSF)

ADEdegreeStream(partitions: Stream List Integer): Stream NNI ==
    scan(0, max((if empty? #1 then 0 else (first #1 - 1)::NNI), #2),
         partitions)$StreamFunctions2(List Integer, NNI)

ADEtestStream(f: UFPSSUPF, partitions: Stream List Integer,
              DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
    map(termAsUFPSSUPF(f, #1, DSF, D1F), partitions)
       $StreamFunctions2(List Integer, UFPSSUPF)

ADEtestStream2(f: UFPSSUPF, partitions: Stream List Integer,
              DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
    map(termAsUFPSSUPF2(f, #1, DSF, D1F), partitions)
       $StreamFunctions2(List Integer, UFPSSUPF)

ADEEXPRRStream(f: EXPRR, xx: Symbol, partitions: Stream List Integer,
               DX: DIFFSPECX, D1X: DIFFSPEC1X): Stream EXPRR ==
    map(termAsEXPRR(f, xx, #1, DX, D1X), partitions)
       $StreamFunctions2(List Integer, EXPRR)

diffDX: DIFFSPECX
diffDX(expr, x, n) == D(expr, x, n)

diffDS: DIFFSPECS
diffDS(s, n) == D(s, n)

diffDSF: DIFFSPECSF
diffDSF(s, n) ==
-- I have to help the compiler here a little to choose the right signature...
    if SUP F has _*: (NonNegativeInteger, SUP F) -> SUP F
    then D(s, n)

diffAX: DIFFSPECAX
diffAX(l: NNI, x: Symbol, f: EXPRR): EXPRR ==
    (x::EXPRR)**l * f

diffA: DIFFSPECA
diffA(k: NNI, l: NNI, f: SUP S): S ==
    DiffAction(k, l, f)$FFFG(S, SUP S)

diffAF: DIFFSPECAF
diffAF(k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
```

```
        DiffAction(k, l, f)$FFFG(SUP F, UFPSSUPF)

    diffC: DIFFSPECC
    diffC(total: NNI): List S == DiffC(total)$FFFG(S, SUP S)

    diff1X: DIFFSPEC1X
    diff1X(x: Symbol)== 1$EXPRR

    diffHP options ==
        if displayAsGF(options)$GOPT0 then
            partitions := FilteredPartitionStream options
            [ADEguessStream(#1, partitions, diffDS, 1$UFPSF), _
             ADEdegreeStream partitions, _
             ADEtestStream(#1, partitions, diffDSF, 1$UFPSSUPF), _
             ADEEXPRRStream(#1, #2, partitions, diffDX, diff1X), _
             diffA, diffAF, diffAX, diffC]$HPSPEC
        else
            error "Guess: guessADE supports only displayAsGF"

    if F has RetractableTo Symbol and S has RetractableTo Symbol then

        qDiffDX(q: Symbol, expr: EXPRR, x: Symbol, n: NonNegativeInteger): EXPRR ==
            eval(expr, x::EXPRR, (q::EXPRR)**n*x::EXPRR)

        qDiffDS(q: Symbol, s: UFPSF, n: NonNegativeInteger): UFPSF ==
            multiplyCoefficients((q::F)**((n*#1)::NonNegativeInteger), s)

        qDiffDSF(q: Symbol, s: UFPSSUPF, n: NonNegativeInteger): UFPSSUPF ==
            multiplyCoefficients((q::F::SUP F)**((n*#1)::NonNegativeInteger), s)

        diffHP(q: Symbol): (LGOPT -> HPSPEC) ==
            if displayAsGF(#1)$GOPT0 then
                partitions := FilteredPartitionStream #1
                [ADEguessStream(#1, partitions, qDiffDS(q, #1, #2), 1$UFPSF), _
                 repeating([0$NNI])$Stream(NNI), _
                 ADEtestStream(#1, partitions, qDiffDSF(q, #1, #2), 1$UFPSSUPF), _
                 ADEEXPRRStream(#1, #2, partitions, qDiffDX(q, #1, #2, #3), diff1X), _
                 diffA, diffAF, diffAX, diffC]$HPSPEC
            else
                error "Guess: guessADE supports only displayAsGF"

    ShiftSX(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==
        eval(expr, x::EXPRR, x::EXPRR+n::EXPRR)

    ShiftSXGF(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==
        if zero? n then expr
        else
            l := [eval(D(expr, x, i)/factorial(i)::EXPRR, x::EXPRR, 0$EXPRR)_
                  *(x::EXPRR)**i for i in 0..n-1]
            (expr-reduce(_+, l))/(x::EXPRR**n)
```

```
ShiftSS(s:UFPSF, n:NNI): UFPSF ==
    ((quoByVar #1)**n)$MappingPackage1(UFPSF) (s)

ShiftSF(s:UFPSSUPF, n: NNI):UFPSSUPF ==
    ((quoByVar #1)**n)$MappingPackage1(UFPSSUPF) (s)

ShiftAX(l: NNI, n: Symbol, f: EXPRR): EXPRR ==
    n::EXPRR**l * f

ShiftAXGF(l: NNI, x: Symbol, f: EXPRR): EXPRR ==
-- I need to help the compiler here, unfortunately
      if zero? l then f
      else
          s := [stirling2(l, i)$IntegerCombinatoricFunctions(Integer)::EXPRR _
                * (x::EXPRR)**i*D(f, x, i) for i in 1..l]
          reduce(_+, s)

ShiftA(k: NNI, l: NNI, f: SUP S): S ==
    ShiftAction(k, l, f)$FFFG(S, SUP S)

ShiftAF(k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
    ShiftAction(k, l, f)$FFFG(SUP F, UFPSSUPF)

ShiftC(total: NNI): List S ==
    ShiftC(total)$FFFG(S, SUP S)

shiftHP options ==
    partitions := FilteredPartitionStream options
    if displayAsGF(options)$GOPT0 then
        if maxPower(options)$GOPT0 = 1 then
            [ADEguessStream(#1, partitions, ShiftSS, (1-monomial(1,1))**(-1)),_
             ADEdegreeStream partitions, _
             ADEtestStream(#1, partitions, ShiftSF, (1-monomial(1,1))**(-1)), _
             ADEEXPRRStream(#1, #2, partitions, ShiftSXGF, 1/(1-#1::EXPRR)), _
             ShiftA, ShiftAF, ShiftAXGF, ShiftC]$HPSPEC
       else
            error "Guess: no support for the Shift operator with displayAsGF _
                   and maxPower>1"
    else
        [ADEguessStream2(#1, partitions, ShiftSS, (1-monomial(1,1))**(-1)), _
         ADEdegreeStream partitions, _
         ADEtestStream2(#1, partitions, ShiftSF, (1-monomial(1,1))**(-1)), _
         ADEEXPRRStream(#1, #2, partitions, ShiftSX, diff1X), _
         ShiftA, ShiftAF, ShiftAX, ShiftC]$HPSPEC

if F has RetractableTo Symbol and S has RetractableTo Symbol then

    qShiftAX(q: Symbol, l: NNI, n: Symbol, f: EXPRR): EXPRR ==
        (q::EXPRR)**(l*n::EXPRR) * f
```

```
        qShiftA(q: Symbol, k: NNI, l: NNI, f: SUP S): S ==
            qShiftAction(q::S, k, l, f)$FFFG(S, SUP S)

        qShiftAF(q: Symbol, k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
            qShiftAction(q::F::SUP(F), k, l, f)$FFFG(SUP F, UFPSSUPF)

        qShiftC(q: Symbol, total: NNI): List S ==
            qShiftC(q::S, total)$FFFG(S, SUP S)

        shiftHP(q: Symbol): (LGOPT -> HPSPEC) ==
            partitions := FilteredPartitionStream #1
            if displayAsGF(#1)$GOPT0 then
                error "Guess: no support for the qShift operator with displayAsGF"
            else
                [ADEguessStream2(#1, partitions, ShiftSS, _
                                 (1-monomial(1,1))**(-1)), _
                 ADEdegreeStream partitions, _
                 ADEtestStream2(#1, partitions, ShiftSF, _
                               (1-monomial(1,1))**(-1)), _
                 ADEEXPRRStream(#1, #2, partitions, ShiftSX, diff1X), _
                 qShiftA(q, #1, #2, #3), qShiftAF(q, #1, #2, #3), _
                 qShiftAX(q, #1, #2, #3), qShiftC(q, #1)]$HPSPEC

    makeEXPRR(DAX: DIFFSPECAX, x: Symbol, p: SUP F, expr: EXPRR): EXPRR ==
        if zero? p then 0$EXPRR
        else
            coerce(leadingCoefficient p)::EXPRR * DAX(degree p, x, expr) _
            + makeEXPRR(DAX, x, reductum p, expr)

    list2UFPSF(list: List F): UFPSF == series(list::Stream F)$UFPSF

    list2UFPSSUPF(list: List F): UFPSSUPF ==
        l := [e::SUP(F) for e in list for i in 0..]::Stream SUP F
        series(l)$UFPSSUPF + monomial(monomial(1,1)$SUP(F), #list)$UFPSSUPF

    SUPF2SUPSUPF(p: SUP F): SUP SUP F ==
        map(#1::SUP F, p)$SparseUnivariatePolynomialFunctions2(F, SUP F)

    UFPSF2SUPF(f: UFPSF, deg: NNI): SUP F ==
        makeSUP univariatePolynomial(f, deg)

    getListSUPF(s: Stream UFPSF, o: NNI, deg: NNI): List SUP F ==
        map(UFPSF2SUPF(#1, deg), entries complete first(s, o))
           $ListFunctions2(UFPSF, SUP F)

    S2EXPRR(s: S): EXPRR ==
        if F is S then
            coerce(s pretend F)@EXPRR
        else if F is Fraction S then
```

```
        coerce(s::Fraction(S))@EXPRR
    else error "Type parameter F should be either equal to S or equal _
                to Fraction S"

guessInterpolate(guessList: List SUP F, eta: List NNI, D: HPSPEC)
                : Matrix SUP S ==
    if F is S then
        vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
        generalInterpolation((D.C)(reduce(_+, eta)), D.A,
                             vguessList, eta)$FFFG(S, SUP S)
    else if F is Fraction S then
        vguessListF: Vector SUP F := vector(guessList)
        generalInterpolation((D.C)(reduce(_+, eta)), D.A,
                             vguessListF, eta)$FFFGF(S, SUP S, SUP F)

    else error "Type parameter F should be either equal to S or equal _
                to Fraction S"

guessInterpolate2(guessList: List SUP F,
                  sumEta: NNI, maxEta: NNI,
                  D: HPSPEC): Stream Matrix SUP S ==
    if F is S then
        vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
        generalInterpolation((D.C)(sumEta), D.A,
                             vguessList, sumEta, maxEta)
                            $FFFG(S, SUP S)
    else if F is Fraction S then
        vguessListF: Vector SUP F := vector(guessList)
        generalInterpolation((D.C)(sumEta), D.A,
                             vguessListF, sumEta, maxEta)
                            $FFFGF(S, SUP S, SUP F)

    else error "Type parameter F should be either equal to S or equal _
                to Fraction S"
testInterpolant(resi: List SUP S,
                list: List F,
                testList: List UFPSSUPF,
                exprList: List EXPRR,
                initials: List EXPRR,
                guessDegree: NNI,
                D: HPSPEC,
                dummy: Symbol, op: BasicOperator, options: LGOPT)
               : Union("failed", Record(function: EXPRR, order: NNI)) ==
  --tpd: maxDegree is defined to be nonnegative
  --  ((maxDegree(options)$GOPT0 = -1) and
    ((allDegrees(options)$GOPT0 = false) and
     zero?(last resi))
     => return "failed"
    nonZeroCoefficient: Integer := 0
```

```
        for i in 1..#resi repeat
            if not zero? resi.i then
                if zero? nonZeroCoefficient then
                    nonZeroCoefficient := i
                else
                    nonZeroCoefficient := 0
                    break
        if not zero? nonZeroCoefficient then
            (freeOf?(exprList.nonZeroCoefficient, name op)) => return "failed"

            for e in list repeat
                if not zero? e then return "failed"
        else
            resiSUPF := map(SUPF2SUPSUPF SUPS2SUPF #1, resi)
                        $ListFunctions2(SUP S, SUP SUP F)

            iterate? := true;
            for d in guessDegree+1.. repeat
                c: SUP F := generalCoefficient(D.AF, vector testList,
                                               d, vector resiSUPF)
                                              $FFFG(SUP F, UFPSSUPF)

                if not zero? c then
                    iterate? := ground? c
                    break

            iterate? => return "failed"
        g: SUP S
        if S has Field
        then g := leadingCoefficient(find(not zero? #1, reverse resi)::SUP(S))::SUP(S)
        else g := gcd resi
        resiF := map(SUPS2SUPF((#1 exquo g)::SUP(S)), resi)
                    $ListFunctions2(SUP S, SUP F)


        if debug(options)$GOPT0 then
            output(hconcat("trying possible solution ", resiF::OutputForm))
                  $OutputPackage

    -- transform each term into an expression

        ex: List EXPRR := [makeEXPRR(D.AX, dummy, p, e) _
                           for p in resiF for e in exprList]

    -- transform the list of expressions into a sum of expressions

        res: EXPRR
        if displayAsGF(options)$GOPT0 then
            res := evalADE(op, dummy, variableName(options)$GOPT0::EXPRR,
                           indexName(options)$GOPT0::EXPRR,
```

```
                        numerator reduce(_+, ex),
                        reverse initials)
                       $RecurrenceOperator(Integer, EXPRR)
        ord: NNI := 0
-- FIXME: checkResult doesn't really work yet for generating functions
    else
        res := evalRec(op, dummy, indexName(options)$GOPT0::EXPRR,
                        indexName(options)$GOPT0::EXPRR,
                        numerator reduce(_+, ex),
                        reverse initials)
                       $RecurrenceOperator(Integer, EXPRR)
        ord: NNI := checkResult(res, indexName(options)$GOPT0, _
                                #list, list, options)


    [res, ord]$Record(function: EXPRR, order: NNI)

guessHPaux(list: List F, D: HPSPEC, options: LGOPT): GUESSRESULT ==
    reslist: GUESSRESULT := []

    listDegree := #list-1-safety(options)$GOPT0
    if listDegree < 0 then return reslist
    a := functionName(options)$GOPT0
    op := operator a
    x := variableName(options)$GOPT0
    dummy := new$Symbol

    initials: List EXPRR := [coerce(e)@EXPRR for e in list]

    guessS  := (D.guessStream)(list2UFPSF list)
    degreeS := D.degreeStream
    testS   := (D.testStream)(list2UFPSSUPF list)
    exprS   := (D.exprStream)(op(dummy::EXPRR)::EXPRR, dummy)
    iterate?: Boolean := false -- this is necessary because the compiler
                               -- doesn't understand => "iterate" properly
                               -- the latter just leaves the current block, it
                               -- seems
    for o in 2.. repeat
        empty? rest(guessS, (o-1)::NNI) => break
        guessDegree: Integer := listDegree-(degreeS.o)::Integer
        guessDegree < 0 => break
        if debug(options)$GOPT0 then
            output(hconcat("Trying order ", o::OutputForm))$OutputPackage
            output(hconcat("guessDegree is ", guessDegree::OutputForm))
                  $OutputPackage
        if allDegrees(options)$GOPT0 then
            (o > guessDegree+2) => return reslist
            --tpd: force NNI and Integer
            maxEta: Integer := 1+maxDegree(options)$GOPT0::NNI::Integer
            if maxEta = 0
```

```
        then maxEta := guessDegree+1
    else
        maxParams := divide(guessDegree::NNI+1, o)
        if debug(options)$GOPT0
        then output(hconcat("maxParams: ", maxParams::OutputForm))
                  $OutputPackage
        if maxParams.quotient = 0 and maxParams.remainder < o-1
        then return reslist
    --tpd: maxDegree is defined to be nonnegative
    --      if ((maxDegree(options)$GOPT0 ~= -1) and
        if ((maxDegree(options)$GOPT0::NNI::Integer < maxParams.quotient)) and
            not (empty? rest(guessS, o) or
                 ((newGuessDegree := listDegree-(degreeS.(o+1))::Integer)
                      < 0) or
                  (((newMaxParams := divide(newGuessDegree::NNI+1, o+1))
                      .quotient = 0) and
                   (newMaxParams.remainder < o)))
        then iterate? := true
     --tpd:maxDegree is defined to be nonnegative
     -- else if ((maxDegree(options)$GOPT0 ~= -1) and
        if (maxParams.quotient > maxDegree(options)$GOPT0::NNI::Integer)
             then
     --tpd:maxDegree is defined to be nonnegative
                guessDegree := o*(1+maxDegree(options)$GOPT0::NNI::Integer)-2
                eta: List NNI
                    := [(if i < o    _
                          then maxDegree(options)$GOPT0::NNI + 1   _
                          else maxDegree(options)$GOPT0::NNI) _
                       for i in 1..o]
             else eta: List NNI
                      := [(if i <= maxParams.remainder   _
                          then maxParams.quotient + 1   _
                          else maxParams.quotient)::NNI for i in 1..o]
    if iterate?
    then
        iterate? := false
        if debug(options)$GOPT0 then output("iterating")$OutputPackage
    else
        guessList: List SUP F    := getListSUPF(guessS, o, guessDegree::NNI)
        testList:  List UFPSSUPF := entries complete first(testS, o)
        exprList:  List EXPRR    := entries complete first(exprS, o)

        if debug(options)$GOPT0 then
            output("The list of expressions is")$OutputPackage
            output(exprList::OutputForm)$OutputPackage

        if allDegrees(options)$GOPT0 then
            MS: Stream Matrix SUP S := guessInterpolate2(guessList,
                                                         guessDegree::NNI+1,
                                                         maxEta::NNI, D)
```

```
                repeat
                    (empty? MS) => break
                    M := first MS

                    for i in 1..o repeat
                        res := testInterpolant(entries column(M, i),
                                               list,
                                               testList,
                                               exprList,
                                               initials,
                                               guessDegree::NNI,
                                               D, dummy, op, options)

                        (res case "failed") => "iterate"

                        if not member?(res, reslist)
                        then reslist := cons(res, reslist)

                        if one(options)$GOPT0 then return reslist

                    MS := rest MS
            else
                M: Matrix SUP S := guessInterpolate(guessList, eta, D)

                for i in 1..o repeat
                    res := testInterpolant(entries column(M, i),
                                           list,
                                           testList,
                                           exprList,
                                           initials,
                                           guessDegree::NNI,
                                           D, dummy, op, options)
                    (res case "failed") => "iterate"

                    if not member?(res, reslist)
                    then reslist := cons(res, reslist)

                    if one(options)$GOPT0 then return reslist

    reslist

guessHP(D: LGOPT -> HPSPEC): GUESSER == guessHPaux(#1, D #2, #2)

guessADE(list: List F, options: LGOPT): GUESSRESULT ==
    opts: LGOPT := cons(displayAsGF(true)$GuessOption, options)
    guessHPaux(list, diffHP opts, opts)

guessADE(list: List F): GUESSRESULT == guessADE(list, [])

guessAlg(list: List F, options: LGOPT) ==
```

```
        guessADE(list, cons(maxDerivative(0)$GuessOption, options))

    guessAlg(list: List F): GUESSRESULT == guessAlg(list, [])

    guessHolo(list: List F, options: LGOPT): GUESSRESULT ==
        guessADE(list, cons(maxPower(1)$GuessOption, options))

    guessHolo(list: List F): GUESSRESULT == guessHolo(list, [])

    guessPade(list: List F, options: LGOPT): GUESSRESULT ==
        opts := append(options, [maxDerivative(0)$GuessOption,
                                 maxPower(1)$GuessOption,
                                 allDegrees(true)$GuessOption])
        guessADE(list, opts)

    guessPade(list: List F): GUESSRESULT == guessPade(list, [])

    if F has RetractableTo Symbol and S has RetractableTo Symbol then

        guessADE(q: Symbol): GUESSER ==
            opts: LGOPT := cons(displayAsGF(true)$GuessOption, #2)
            guessHPaux(#1, (diffHP q)(opts), opts)

    guessRec(list: List F, options: LGOPT): GUESSRESULT ==
          opts: LGOPT := cons(displayAsGF(false)$GuessOption, options)
          guessHPaux(list, shiftHP opts, opts)

    guessRec(list: List F): GUESSRESULT == guessRec(list, [])

    guessPRec(list: List F, options: LGOPT): GUESSRESULT ==
          guessRec(list, cons(maxPower(1)$GuessOption, options))

    guessPRec(list: List F): GUESSRESULT == guessPRec(list, [])

    guessRat(list: List F, options: LGOPT): GUESSRESULT ==
          opts := append(options, [maxShift(0)$GuessOption,
                                   maxPower(1)$GuessOption,
                                   allDegrees(true)$GuessOption])
          guessRec(list, opts)

    guessRat(list: List F): GUESSRESULT == guessRat(list, [])

    if F has RetractableTo Symbol and S has RetractableTo Symbol then

        guessRec(q: Symbol): GUESSER ==
            opts: LGOPT := cons(displayAsGF(false)$GuessOption, #2)
            guessHPaux(#1, (shiftHP q)(opts), opts)

        guessPRec(q: Symbol): GUESSER ==
            opts: LGOPT := append([displayAsGF(false)$GuessOption,
```

```
                                maxPower(1)$GuessOption], #2)
            guessHPaux(#1, (shiftHP q)(opts), opts)

        guessRat(q: Symbol): GUESSER ==
            opts := append(#2, [displayAsGF(false)$GuessOption,
                                maxShift(0)$GuessOption,
                                maxPower(1)$GuessOption,
                                allDegrees(true)$GuessOption])
            guessHPaux(#1, (shiftHP q)(opts), opts)

    guess(list: List F, guessers: List GUESSER, ops: List Symbol,
          options: LGOPT): GUESSRESULT ==
        maxLevel := maxLevel(options)$GOPT0
        xx := indexName(options)$GOPT0
        if debug(options)$GOPT0 then
            output(hconcat("guessing level ", maxLevel::OutputForm))
                  $OutputPackage
            output(hconcat("guessing ", list::OutputForm))$OutputPackage
        res: GUESSRESULT := []
        len := #list :: PositiveInteger
        if len <= 1 then return res

        for guesser in guessers repeat
            res := append(guesser(list, options), res)

            if debug(options)$GOPT0 then
                output(hconcat("res ", res::OutputForm))$OutputPackage

            if one(options)$GOPT0 and not empty? res then return res

        if (maxLevel = 0) then return res

        if member?('guessProduct, ops) and not member?(0$F, list) then
            prodList: List F := [(list.(i+1))/(list.i) for i in 1..(len-1)]

        -- tpd: maxLevel is NNI
            if not every?(one?, prodList) then
                var: Symbol := subscript('p, [len::OutputForm])
                prodGuess :=
                    [[coerce(list.(guess.order+1))
                      * product(guess.function, _
                                equation(var, _
                                         (guess.order)::EXPRR..xx::EXPRR-1)), _
                     guess.order] _
                    for guess in guess(prodList, guessers, ops,  options)$%]
-- tpd: this is broken
--                             append([(indexName(var)$GuessOption)::Symbol,_
--                                     (maxLevel(maxLevel-1)$GuessOption)::NNI],_
--                                    options))$%]
```

```
                if debug(options)$GOPT0 then
                    output(hconcat("prodGuess "::OutputForm,
                                   prodGuess::OutputForm))
                         $OutputPackage

                for guess in prodGuess
                        | not any?(guess.function = #1.function, res) repeat
                    res := cons(guess, res)

        if one(options)$GOPT0 and not empty? res then return res

        if member?('guessSum, ops) then
            sumList: List F := [(list.(i+1))-(list.i) for i in 1..(len-1)]
        -- tpd:maxLevel is NNI
            if not every?(#1=sumList.1, sumList) then
                var: Symbol := subscript('s, [len::OutputForm])
                sumGuess :=
                    [[coerce(list.(guess.order+1)) _
                      + summation(guess.function, _
                                  equation(var, _
                                           (guess.order)::EXPRR..xx::EXPRR-1)),_
                      guess.order] _
                     for guess in guess(sumList, guessers, ops,  options)$%]
--tpd: this is broken
--                      for guess in guess(sumList, guessers, ops,_
--                           append([(indexName(var)$GuessOption)::Symbol,_
--                                   (maxLevel(maxLevel-1)$GuessOption)::NNI],_
--                                               options))$%]

                for guess in sumGuess
                        | not any?(guess.function = #1.function, res) repeat
                    res := cons(guess, res)

            res

    guess(list: List F): GUESSRESULT ==
        guess(list, [guessADE$%, guessRec$%], ['guessProduct, 'guessSum], [])

    guess(list: List F, options: LGOPT): GUESSRESULT ==
        guess(list, [guessADE$%, guessRat$%], ['guessProduct, 'guessSum],
              options)

    guess(list: List F, guessers: List GUESSER, ops: List Symbol)
         : GUESSRESULT ==
        guess(list, guessers, ops, [])
```

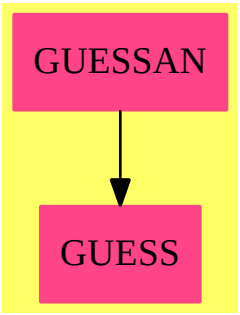
— **GUESS.dotabb** —

```
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"RECOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RECOP"]
"GUESS" -> "RECOP"
```

———-

8.23 package GUESSAN GuessAlgebraicNumber

8.23.1 GuessAlgebraicNumber (GUESSAN)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

— **package GUESSAN GuessAlgebraicNumber** —

```
)abbrev package GUESSAN GuessAlgebraicNumber
++ Description:
++ This package exports guessing of sequences of rational functions

GuessAlgebraicNumber() == Guess(AlgebraicNumber, AlgebraicNumber,
                           Expression Integer,
                           AlgebraicNumber,
                           id$MappingPackage1(AlgebraicNumber),
                           coerce$Expression(Integer))
```

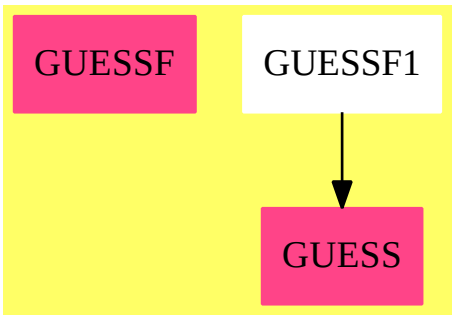
———-

— **GUESSAN.dotabb** —

```
"GUESSAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSAN"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSAN" -> "GUESS"
```

8.24 package GUESSF GuessFinite

8.24.1 GuessFinite (GUESSF)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

— package GUESSF GuessFinite —

```
)abbrev package GUESSF GuessFinite
++ Description:
++ This package exports guessing of sequences of numbers in a finite field

GuessFinite(F:Join(FiniteFieldCategory, ConvertibleTo Integer)) ==
  Guess(F, F, Expression Integer, Integer, coerce$F,
        F2EXPRR$GuessFiniteFunctions(F))
```

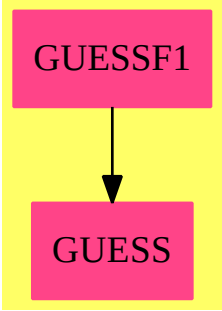
— GUESSF.dotabb —

```
"GUESSF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSF"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSF1" -> "GUESS"
```

———-

8.25 package GUESSF1 GuessFiniteFunctions

8.25.1 GuessFiniteFunctions (GUESSF1)



Exports:
F2EXPRR

— **package GUESSF1 GuessFiniteFunctions** —

```
)abbrev package GUESSF1 GuessFiniteFunctions
++ Description:
++ This package exports guessing of sequences of numbers in a finite field

GuessFiniteFunctions(F:Join(FiniteFieldCategory, ConvertibleTo Integer)):
  Exports == Implementation where

    EXPRR ==> Expression Integer

    Exports == with

      F2EXPRR: F -> EXPRR

    Implementation == add

      F2EXPRR(p: F): EXPRR == convert(p)@Integer::EXPRR
```

———-

— **GUESSF1.dotabb** —

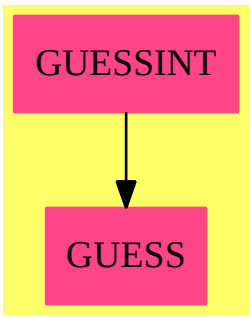
```
"GUESSF1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSF1"]
```

```
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSF1" -> "GUESS"
```

———-

8.26 package GUESSINT GuessInteger

8.26.1 GuessInteger (GUESSINT)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

— **package GUESSINT GuessInteger** —

```
-- concerning algebraic functions, it may make sense to look at A.J.van der
-- Poorten, Power Series Representing Algebraic Functions, Section 6.
)abbrev package GUESSINT GuessInteger
++ Description:
++ This package exports guessing of sequences of rational numbers

GuessInteger() == Guess(Fraction Integer, Integer, Expression Integer,
                    Fraction Integer,
                    id$MappingPackage1(Fraction Integer),
                    coerce$Expression(Integer))
```

———-

— **GUESSINT.dotabb** —

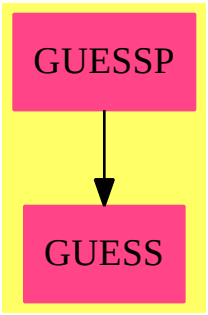
```
"GUESSINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSINT"]
```

```
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSINT" -> "GUESS"
```

———-

8.27 package GUESSP GuessPolynomial

8.27.1 GuessPolynomial (GUESSP)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

— **package GUESSP GuessPolynomial** —

```
)abbrev package GUESSP GuessPolynomial
++ Description:
++ This package exports guessing of sequences of rational functions

GuessPolynomial() == Guess(Fraction Polynomial Integer, Polynomial Integer,
                           Expression Integer,
                           Fraction Polynomial Integer,
                           id$MappingPackage1(Fraction Polynomial Integer),
                           coerce$Expression(Integer))
```

———-

— **GUESSP.dotabb** —

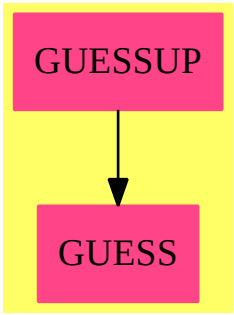
```
"GUESSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSP"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
```

```
"GUESSP" -> "GUESS"
```

———-

8.28 package GUESSUP GuessUnivariatePolynomial

8.28.1 GuessUnivariatePolynomial (GUESSUP)



Exports:

diffHP	guess	guessADE	guessAlg	guessBinRat
guessExpRat	guessHP	guessHolo	guessPRec	guessPade
guessRat	guessRec	shiftHP		

— **package GUESSUP GuessUnivariatePolynomial** —

```
)abbrev package GUESSUP GuessUnivariatePolynomial
++ Description:
++ This package exports guessing of sequences of univariate rational functions

GuessUnivariatePolynomial(q: Symbol): Exports == Implementation where

    UP ==> MyUnivariatePolynomial(q, Integer)
    EXPRR ==> MyExpression(q, Integer)
    F ==> Fraction UP
    S ==> UP
    NNI ==> NonNegativeInteger
    LGOPT ==> List GuessOption
    GUESSRESULT ==> List Record(function: EXPRR, order: NNI)
    GUESSER ==> (List F, LGOPT) -> GUESSRESULT

    Exports == with

        guess: List F -> GUESSRESULT
          ++ \spad{guess l} applies recursively \spadfun{guessRec} and
```

```
          ++ \spadfun{guessADE} to the successive differences and quotients of
          ++ the list. Default options as described in
          ++ \spadtype{GuessOptionFunctions0} are used.

        guess: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guess(l, options)} applies recursively \spadfun{guessRec}
          ++ and \spadfun{guessADE} to the successive differences and quotients
          ++ of the list. The given options are used.

        guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
          ++ \spad{guess(l, guessers, ops)} applies recursively the given
          ++ guessers to the successive differences if ops contains the symbol
          ++ guessSum and quotients if ops contains the symbol guessProduct to
          ++ the list. Default options as described in
          ++ \spadtype{GuessOptionFunctions0} are used.

        guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT
          ++ \spad{guess(l, guessers, ops)} applies recursively the given
          ++ guessers to the successive differences if ops contains the symbol
          ++ \spad{guessSum} and quotients if ops contains the symbol
          ++ \spad{guessProduct} to the list. The given options are used.

        guessExpRat: List F -> GUESSRESULT
          ++ \spad{guessExpRat l} tries to find a function of the form
          ++ n+->(a+b n)^n r(n), where r(n) is a rational function, that fits
          ++ l.

        guessExpRat: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessExpRat(l, options)} tries to find a function of the
          ++ form n+->(a+b n)^n r(n), where r(n) is a rational function, that
          ++ fits l.

        guessBinRat: List F -> GUESSRESULT
          ++ \spad{guessBinRat(l, options)} tries to find a function of the
          ++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
          ++ function, that fits l.

        guessBinRat: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessBinRat(l, options)} tries to find a function of the
          ++ form n+->binomial(a+b n, n) r(n), where r(n) is a rational
          ++ function, that fits l.

--        if F has RetractableTo Symbol and S has RetractableTo Symbol then

        guessExpRat: Symbol -> GUESSER
          ++ \spad{guessExpRat q} returns a guesser that tries to find a
          ++ function of the form n+->(a+b q^n)^n r(q^n), where r(q^n) is a
          ++ q-rational function, that fits l.

        guessBinRat: Symbol -> GUESSER
```

```
          ++ \spad{guessBinRat q} returns a guesser that tries to find a
          ++ function of the form n+->qbinomial(a+b n, n) r(n), where r(q^n) is a
          ++ q-rational function, that fits l.

--------------------------------------------------------------------------------

        guessHP: (LGOPT -> HPSPEC) -> GUESSER
          ++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
          ++ approximation to the series generated by the given function f.

        guessADE: List F -> GUESSRESULT
          ++ \spad{guessADE l} tries to find an algebraic differential equation
          ++ for a generating function whose first Taylor coefficients are
          ++ given by l, using the default options described in
          ++ \spadtype{GuessOptionFunctions0}.

        guessADE: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessADE(l, options)} tries to find an algebraic
          ++ differential equation for a generating function whose first Taylor
          ++ coefficients are given by l, using the given options.

        guessAlg: List F -> GUESSRESULT
          ++ \spad{guessAlg l} tries to find an algebraic equation for a
          ++ generating function whose first Taylor coefficients are given by
          ++ l, using the default options described in
          ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
          ++ \spadfun{guessADE}(l, maxDerivative == 0).

        guessAlg: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessAlg(l, options)} tries to find an algebraic equation
          ++ for a generating function whose first Taylor coefficients are
          ++ given by l, using the given options. It is equivalent to
          ++ \spadfun{guessADE}(l, options) with \spad{maxDerivative == 0}.

        guessHolo: List F -> GUESSRESULT
          ++ \spad{guessHolo l} tries to find an ordinary linear differential
          ++ equation for a generating function whose first Taylor coefficients
          ++ are given by l, using the default options described in
          ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
          ++ \spadfun{guessADE}\spad{(l, maxPower == 1)}.

        guessHolo: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessHolo(l, options)} tries to find an ordinary linear
          ++ differential equation for a generating function whose first Taylor
          ++ coefficients are given by l, using the given options. It is
          ++ equivalent to \spadfun{guessADE}\spad{(l, options)} with
          ++ \spad{maxPower == 1}.

        guessPade: (List F, LGOPT) -> GUESSRESULT
          ++ \spad{guessPade(l, options)} tries to find a rational function
```

```
  ++ whose first Taylor coefficients are given by l, using the given
  ++ options. It is equivalent to \spadfun{guessADE}\spad{(l,
  ++ maxDerivative == 0, maxPower == 1, allDegrees == true)}.

guessPade: List F -> GUESSRESULT
  ++ \spad{guessPade(l, options)} tries to find a rational function
  ++ whose first Taylor coefficients are given by l, using the default
  ++ options described in \spadtype{GuessOptionFunctions0}. It is
  ++ equivalent to \spadfun{guessADE}\spad{(l, options)} with
  ++ \spad{maxDerivative == 0, maxPower == 1, allDegrees == true}.

guessRec: List F -> GUESSRESULT
  ++ \spad{guessRec l} tries to find an ordinary difference equation
  ++ whose first values are given by l, using the default options
  ++ described in \spadtype{GuessOptionFunctions0}.

guessRec: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessRec(l, options)} tries to find an ordinary difference
  ++ equation whose first values are given by l, using the given
  ++ options.

guessPRec: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessPRec(l, options)} tries to find a linear recurrence
  ++ with polynomial coefficients whose first values are given by l,
  ++ using the given options. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, options)} with \spad{maxPower == 1}.

guessPRec: List F -> GUESSRESULT
  ++ \spad{guessPRec l} tries to find a linear recurrence with
  ++ polynomial coefficients whose first values are given by l, using
  ++ the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, maxPower == 1)}.

guessRat: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guessRat(l, options)} tries to find a rational function
  ++ whose first values are given by l, using the given options. It is
  ++ equivalent to \spadfun{guessRec}\spad{(l, maxShift == 0, maxPower
  ++ == 1, allDegrees == true)}.

guessRat: List F -> GUESSRESULT
  ++ \spad{guessRat l} tries to find a rational function whose first
  ++ values are given by l, using the default options described in
  ++ \spadtype{GuessOptionFunctions0}. It is equivalent to
  ++ \spadfun{guessRec}\spad{(l, maxShift == 0, maxPower == 1,
  ++ allDegrees == true)}.

diffHP: LGOPT -> HPSPEC
  ++ \spad{diffHP options} returns a specification for Hermite-Pade
  ++ approximation with the differential operator
```

```
        shiftHP: LGOPT -> HPSPEC
          ++ \spad{shiftHP options} returns a specification for Hermite-Pade
          ++ approximation with the shift operator

--        if F has RetractableTo Symbol and S has RetractableTo Symbol then

        shiftHP: Symbol -> (LGOPT -> HPSPEC)
          ++ \spad{shiftHP options} returns a specification for
          ++ Hermite-Pade approximation with the $q$-shift operator

        diffHP: Symbol -> (LGOPT -> HPSPEC)
          ++ \spad{diffHP options} returns a specification for Hermite-Pade
          ++ approximation with the  $q$-dilation operator

        guessRec: Symbol -> GUESSER
          ++ \spad{guessRec q} returns a guesser that finds an ordinary
          ++ q-difference equation whose first values are given by l, using
          ++ the given options.

        guessPRec: Symbol -> GUESSER
          ++ \spad{guessPRec q} returns a guesser that tries to find
          ++ a linear q-recurrence with polynomial coefficients whose first
          ++ values are given by l, using the given options. It is
          ++ equivalent to \spadfun{guessRec}\spad{(q)} with
          ++ \spad{maxPower == 1}.

        guessRat: Symbol -> GUESSER
          ++ \spad{guessRat q} returns a guesser that tries to find a
          ++ q-rational function whose first values are given by l, using
          ++ the given options. It is equivalent to \spadfun{guessRec} with
          ++ \spad{(l, maxShift == 0, maxPower == 1, allDegrees == true)}.

        guessADE: Symbol -> GUESSER
          ++ \spad{guessADE q} returns a guesser that tries to find an
          ++ algebraic differential equation for a generating function whose
          ++ first Taylor coefficients are given by l, using the given
          ++ options.

------------------------------------------------------------------------------

    Implementation == Guess(Fraction UP, UP,
                            MyExpression(q, Integer),
                            Fraction UP,
                            id$MappingPackage1(Fraction UP),
                            coerce$MyExpression(q, Integer))
```

— **GUESSUP.dotabb** —

```
"GUESSUP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSUP"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSUP" -> "GUESS"
```

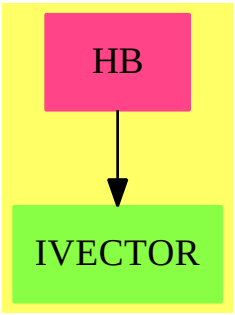
———-

Chapter 9

Chapter H

9.1 package HB HallBasis

9.1.1 HallBasis (HB)



Exports:
generate inHallBasis? lfunc

— package HB HallBasis —

```
)abbrev package HB HallBasis
++ Author : Larry Lambe
++ Date Created : August  1988
++ Date Last Updated : March 9 1990
++ Related Constructors: OrderedSetInts, Commutator, FreeNilpotentLie
++ AMS Classification: Primary 17B05, 17B30; Secondary 17A50
++ Keywords: free Lie algebra, Hall basis, basic commutators
++ Description:
++ Generate a basis for the free Lie algebra on n
```

```
++ generators over a ring R with identity up to basic commutators
++ of length c using the algorithm of P. Hall as given in Serre's
++ book Lie Groups -- Lie Algebras

HallBasis() : Export == Implement where
   B   ==> Boolean
   I   ==> Integer
   NNI ==> NonNegativeInteger
   VI  ==> Vector Integer
   VLI ==> Vector List Integer

   Export  ==> with
     lfunc : (I,I) -> I
       ++ lfunc(d,n) computes the rank of the nth factor in the
       ++ lower central series of the free d-generated free Lie
       ++ algebra;  This rank is d if n = 1 and binom(d,2) if
       ++ n = 2
     inHallBasis? : (I,I,I,I) -> B
       ++ inHallBasis?(numberOfGens, leftCandidate, rightCandidate, left)
       ++ tests to see if a new element should be added to the P. Hall
       ++ basis being constructed.
       ++ The list \spad{[leftCandidate,wt,rightCandidate]}
       ++ is included in the basis if in the unique factorization of
       ++ rightCandidate, we have left factor leftOfRight, and
       ++ leftOfRight <= leftCandidate
     generate : (NNI,NNI) -> VLI
       ++ generate(numberOfGens, maximalWeight) generates a vector of
       ++ elements of the form [left,weight,right] which represents a
       ++ P. Hall basis element for the free lie algebra on numberOfGens
       ++ generators.  We only generate those basis elements of weight
       ++ less than or equal to maximalWeight

   Implement ==> add

     lfunc(d,n) ==
        n < 0 => 0
        n = 0 => 1
        n = 1 => d
        sum:I := 0
        m:I
        for m in 1..(n-1) repeat
          if n rem m = 0 then
             sum := sum + m * lfunc(d,m)
        res := (d**(n::NNI) - sum) quo n

     inHallBasis?(n,i,j,l) ==
        i >= j => false
        j <= n => true
        l <= i => true
        false
```

```
generate(n:NNI,c:NNI) ==
   gens:=n
   maxweight:=c
   siz:I := 0
   for i in 1 .. maxweight repeat siz := siz + lfunc(gens,i)
   v:VLI:= new(siz::NNI,[])
   for i in 1..gens repeat v(i) := [0, 1, i]
   firstindex:VI := new(maxweight::NNI,0)
   wt:I := 1
   firstindex(1) := 1
   numComms:I := gens
   newNumComms:I := numComms
   done:B := false
   while not done repeat
     wt := wt + 1
     if wt > maxweight then done := true
     else
       firstindex(wt) := newNumComms + 1
       leftIndex := 1
       -- cW == complimentaryWeight
       cW:I := wt - 1
       while (leftIndex <= numComms) and (v(leftIndex).2 <= cW) repeat
         for rightIndex in firstindex(cW)..(firstindex(cW+1) - 1) repeat
           if inHallBasis?(gens,leftIndex,rightIndex,v(rightIndex).1) then
             newNumComms := newNumComms + 1
             v(newNumComms) := [leftIndex,wt,rightIndex]
         leftIndex := leftIndex + 1
         cW := wt - v(leftIndex).2
       numComms := newNumComms
   v
```

———-

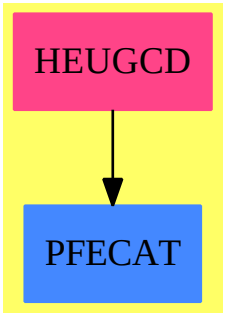
— **HB.dotabb** —

```
"HB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=HB"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"HB" -> "IVECTOR"
```

———-

9.2 package HEUGCD HeuGcd

9.2.1 HeuGcd (HEUGCD)



Exports:
content gcd gcdcofact gcdcofactprim gcdprim lintgcd

— package HEUGCD HeuGcd —

```
)abbrev package HEUGCD HeuGcd
++ Author: P.Gianni
++ Date Created:
++ Date Last Updated: 13 September 94
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides the functions for the heuristic integer gcd.
++ Geddes's algorithm,for univariate polynomials with integer coefficients

HeuGcd (BP):C == T
 where
  BP       :   UnivariatePolynomialCategory Integer
  Z        ==> Integer
  ContPrim ==> Record(cont:Z,prim:BP)


  C == with
     gcd          : List BP  -> BP
       ++ gcd([f1,..,fk]) = gcd of the polynomials fi.
       ++
       ++X gcd([671*671*x^2-1,671*671*x^2+2*671*x+1])
       ++X gcd([7*x^2+1,(7*x^2+1)^2])
```

```
    gcdprim      : List BP  -> BP
      ++ gcdprim([f1,..,fk]) = gcd of k PRIMITIVE univariate polynomials
    gcdcofact    : List BP  -> List BP
      ++ gcdcofact([f1,..fk]) = gcd and cofactors of k univariate polynomials.
    gcdcofactprim: List BP  -> List BP
      ++ gcdcofactprim([f1,..fk]) = gcd and cofactors of k
      ++ primitive polynomials.
    content      : List BP  -> List Z
      ++ content([f1,..,fk]) = content of a list of univariate polynonials
    lintgcd      : List  Z  -> Z
      ++ lintgcd([a1,..,ak]) = gcd of a list of integers

T == add

  PI    ==> PositiveInteger
  NNI   ==> NonNegativeInteger
  Cases ==> Union("gcdprim","gcd","gcdcofactprim","gcdcofact")
  import ModularDistinctDegreeFactorizer BP

  --local functions
  localgcd     :        List BP       -> List BP
  constNotZero :           BP         -> Boolean
  height       :           BP         -> PI
  genpoly      :         (Z,PI)       -> BP
  negShiftz    :         (Z,PI)       -> Z
  internal     :     (Cases,List BP ) -> List BP
  constcase    : (List NNI ,List BP ) -> List BP
  lincase      : (List NNI ,List BP ) -> List BP
  myNextPrime  :        ( Z , NNI )   -> Z

  bigPrime:= prevPrime(2**26)$IntegerPrimesPackage(Integer)

  myNextPrime(val:Z,bound:NNI) : Z == nextPrime(val)$IntegerPrimesPackage(Z)

  constNotZero(f : BP ) : Boolean == (degree f = 0) and ^(zero? f)

  negShiftz(n:Z,Modulus:PI):Z ==
    n < 0 => n:= n+Modulus
    n > (Modulus quo 2) => n-Modulus
    n

  --compute the height of a polynomial
  height(f:BP):PI ==
    k:PI:=1
    while f^=0 repeat
         k:=max(k,abs(leadingCoefficient(f)@Z)::PI)
         f:=reductum f
    k
```

```
--reconstruct the polynomial from the value-adic representation of
--dval.
genpoly(dval:Z,value:PI):BP ==
  d:=0$BP
  val:=dval
  for i in 0..  while (val^=0) repeat
    val1:=negShiftz(val rem value,value)
    d:= d+monomial(val1,i)
    val:=(val-val1) quo value
  d

--gcd of a list of integers
lintgcd(lval:List(Z)):Z ==
  empty? lval => 0$Z
  member?(1,lval) => 1$Z
  lval:=sort((z1,z2) +-> z1<z2,lval)
  val:=lval.first
  for val1 in lval.rest while ^(val=1) repeat val:=gcd(val,val1)
  val

--content for a list of univariate polynomials
content(listf:List BP ):List(Z) ==
  [lintgcd coefficients f for f in listf]

--content of a list of polynomials with the relative primitive parts
contprim(listf:List BP ):List(ContPrim) ==
   [[c:=lintgcd coefficients f,(f exquo c)::BP]$ContPrim  for f in listf]

-- one polynomial is constant, remark that they are primitive
-- but listf can contain the zero polynomial
constcase(listdeg:List NNI ,listf:List BP ): List BP  ==
  lind:=select(constNotZero,listf)
  empty? lind =>
    member?(1,listdeg) => lincase(listdeg,listf)
    localgcd listf
  or/[n>0 for n in listdeg] => cons(1$BP,listf)
  lclistf:List(Z):= [leadingCoefficient f for f in listf]
  d:=lintgcd(lclistf)
  d=1 =>  cons(1$BP,listf)
  cons(d::BP,[(lcf quo d)::BP for lcf in lclistf])

testDivide(listf: List BP, g:BP):Union(List BP, "failed") ==
  result:List BP := []
  for f in listf repeat
    if (f1:=f exquo g) case "failed" then return "failed"
    result := cons(f1::BP,result)
  reverse!(result)

--one polynomial is linear, remark that they are primitive
lincase(listdeg:List NNI ,listf:List BP ):List BP  ==
```

```
    n:= position(1,listdeg)
    g:=listf.n
    result:=[g]
    for f in listf repeat
      if (f1:=f exquo g) case "failed" then return cons(1$BP,listf)
      result := cons(f1::BP,result)
    reverse(result)

  IMG := InnerModularGcd(Z,BP,67108859,myNextPrime)

  mindegpol(f:BP, g:BP):BP ==
    degree(g) < degree (f) => g
    f

  --local function for the gcd among n PRIMITIVE univariate polynomials
  localgcd(listf:List BP ):List BP  ==
    hgt:="min"/[height(f) for f in listf|^zero? f]
    answr:=2+2*hgt
    minf := "mindegpol"/[f for f in listf|^zero? f]
    (result := testDivide(listf, minf)) case List(BP) =>
         cons(minf, result::List BP)
    if degree minf < 100 then for k in 1..10 repeat
      listval:=[f answr for f in listf]
      dval:=lintgcd(listval)
      dd:=genpoly(dval,answr)
      contd:=content(dd)
      d:=(dd exquo contd)::BP
      result:List BP :=[d]
      flag : Boolean := true
      for f in listf while flag repeat
        (f1:=f exquo d) case "failed" => flag:=false
        result := cons (f1::BP,result)
      if flag then return reverse(result)
      nvalue:= answr*832040 quo 317811
      if ((nvalue + answr) rem 2) = 0 then nvalue:=nvalue+1
      answr:=nvalue::PI
    gg:=modularGcdPrimitive(listf)$IMG
    cons(gg,[(f exquo gg) :: BP for f in listf])

  --internal function:it evaluates the gcd and avoids duplication of
  --code.
  internal(flag:Cases,listf:List BP ):List BP  ==
    --special cases
    listf=[] => [1$BP]
    (nlf:=#listf)=1 => [first listf,1$BP]
    minpol:=1$BP
    -- extract a monomial gcd
    mdeg:= "min"/[minimumDegree f for f in listf]
    if mdeg>0 then
      minpol1:= monomial(1,mdeg)
```

```
        listf:= [(f exquo minpol1)::BP for f in listf]
        minpol:=minpol*minpol1
      -- make the polynomials primitive
      Cgcd:List(Z):=[]
      contgcd : Z := 1
      if (flag case "gcd") or (flag case "gcdcofact") then
        contlistf:List(ContPrim):=contprim(listf)
        Cgcd:= [term.cont for term in contlistf]
        contgcd:=lintgcd(Cgcd)
        listf:List BP :=[term.prim for term in contlistf]
        minpol:=contgcd*minpol
      listdeg:=[degree f for f in listf ]
      f:= first listf
      if positiveRemainder(leadingCoefficient(f), bigPrime) ~= 0 then
        for g in rest listf repeat
          lcg := leadingCoefficient(g)
          if positiveRemainder(lcg, bigPrime) = 0 then
            leave
          f:=gcd(f,g,bigPrime)
          if degree f = 0 then return cons(minpol,listf)
      ans:List BP :=
         --one polynomial is constant
         member?(0,listdeg) => constcase(listdeg,listf)
         --one polynomial is linear
         member?(1,listdeg) => lincase(listdeg,listf)
         localgcd(listf)
      (result,ans):=(first ans*minpol,rest ans)
      if (flag case "gcdcofact") then
        ans:= [(p quo contgcd)*q for p in Cgcd for q in ans]
      cons(result,ans)

    --gcd among n PRIMITIVE univariate polynomials
    gcdprim (listf:List BP ):BP == first internal("gcdprim",listf)

    --gcd and cofactors for n PRIMITIVE univariate polynomials
    gcdcofactprim(listf:List BP ):List BP  == internal("gcdcofactprim",listf)

    --gcd for n generic univariate polynomials.
    gcd(listf:List BP ): BP  ==  first internal("gcd",listf)

    --gcd and cofactors for n generic univariate polynomials.
    gcdcofact (listf:List BP ):List BP == internal("gcdcofact",listf)
```

———-

— HEUGCD.dotabb —

```
"HEUGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=HEUGCD"]
```

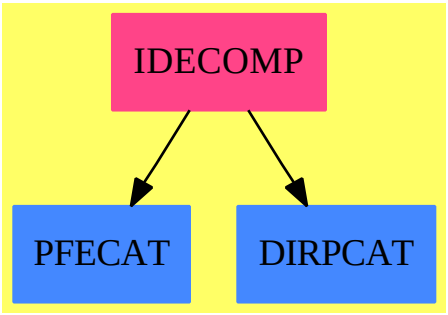
```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"HEUGCD" -> "PFECAT"
```

Chapter 10

Chapter I

10.1 package IDECOMP IdealDecompositionPackage

10.1.1 IdealDecompositionPackage (IDECOMP)



Exports:
contract primaryDecomp prime? radical zeroDimPrimary? zeroDimPrime?

— package IDECOMP IdealDecompositionPackage —

```
)abbrev package IDECOMP IdealDecompositionPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: PolynomialIdeals
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ This package provides functions for the primary decomposition of
++ polynomial ideals over the rational numbers. The ideals are members
++ of the \spadtype{PolynomialIdeals} domain, and the polynomial generators are
++ required to be from the \spadtype{DistributedMultivariatePolynomial} domain.

IdealDecompositionPackage(vl,nv) : C == T -- take away nv, now doesn't
                                          -- compile if it isn't there
 where
   vl      :  List Symbol
   nv      :  NonNegativeInteger
   Z       ==>  Integer  -- substitute with PFE cat
   Q       ==>  Fraction Z
   F       ==>  Fraction P
   P       ==>  Polynomial Z
   UP      ==>  SparseUnivariatePolynomial P
   Expon   ==>  DirectProduct(nv,NNI)
   OV      ==>  OrderedVariableList(vl)
   SE      ==>  Symbol
   SUP     ==>  SparseUnivariatePolynomial(DPoly)

   DPoly1 ==>  DistributedMultivariatePolynomial(vl,Q)
   DPoly  ==>  DistributedMultivariatePolynomial(vl,F)
   NNI    ==>  NonNegativeInteger

   Ideal  ==  PolynomialIdeals(Q,Expon,OV,DPoly1)
   FIdeal ==  PolynomialIdeals(F,Expon,OV,DPoly)
   Fun0   ==  Union("zeroPrimDecomp","zeroRadComp")
   GenPos ==  Record(changeval:List Z,genideal:FIdeal)

   C == with


     zeroDimPrime?       :        Ideal         -> Boolean
       ++ zeroDimPrime?(I) tests if the ideal I is a 0-dimensional prime.

     zeroDimPrimary?     :        Ideal         -> Boolean
       ++ zeroDimPrimary?(I) tests if the ideal I is 0-dimensional primary.
     prime?              :        Ideal         -> Boolean
       ++ prime?(I) tests if the ideal I is prime.
     radical             :        Ideal         -> Ideal
       ++ radical(I) returns the radical of the ideal I.
     primaryDecomp       :        Ideal         -> List(Ideal)
       ++ primaryDecomp(I) returns a list of primary ideals such that their
       ++ intersection is the ideal I.

     contract        : (Ideal,List OV   )       -> Ideal
       ++ contract(I,lvar) contracts the ideal I to the polynomial ring
       ++ \spad{F[lvar]}.
```

```
T  == add

  import MPolyCatRationalFunctionFactorizer(Expon,OV,Z,DPoly)
  import GroebnerPackage(F,Expon,OV,DPoly)
  import GroebnerPackage(Q,Expon,OV,DPoly1)

              ----  Local  Functions  -----
  genPosLastVar       :    (FIdeal,List OV)     -> GenPos
  zeroPrimDecomp      :    (FIdeal,List OV)     -> List(FIdeal)
  zeroRadComp         :    (FIdeal,List OV)     -> FIdeal
  zerodimcase         :    (FIdeal,List OV)     -> Boolean
  is0dimprimary       :    (FIdeal,List OV)     -> Boolean
  backGenPos          : (FIdeal,List Z,List OV) -> FIdeal
  reduceDim           : (Fun0,FIdeal,List OV)   -> List FIdeal
  findvar             :   (FIdeal,List OV)      -> OV
  testPower           :    (SUP,OV,FIdeal)      -> Boolean
  goodPower           :     (DPoly,FIdeal)  -> Record(spol:DPoly,id:FIdeal)
  pushdown            :      (DPoly,OV)        -> DPoly
  pushdterm           :     (DPoly,OV,Z)       -> DPoly
  pushup              :      (DPoly,OV)        -> DPoly
  pushuterm           :    (DPoly,SE,OV)       -> DPoly
  pushucoef           :       (UP,OV)          -> DPoly
  trueden             :        (P,SE)          -> P
  rearrange           :       (List OV)        -> List OV
  deleteunit          :      List FIdeal        -> List FIdeal
  ismonic             :      (DPoly,OV)        -> Boolean


  MPCFQF ==> MPolyCatFunctions2(OV,Expon,Expon,Q,F,DPoly1,DPoly)
  MPCFFQ ==> MPolyCatFunctions2(OV,Expon,Expon,F,Q,DPoly,DPoly1)

  convertQF(a:Q) : F == ((numer a):: F)/((denom a)::F)
  convertFQ(a:F) : Q == (ground numer a)/(ground denom a)

  internalForm(I:Ideal) : FIdeal ==
    Id:=generators I
    nId:=[map(convertQF,poly)$MPCFQF for poly in Id]
    groebner? I => groebnerIdeal nId
    ideal nId

  externalForm(I:FIdeal) : Ideal ==
    Id:=generators I
    nId:=[map(convertFQ,poly)$MPCFFQ for poly in Id]
    groebner? I => groebnerIdeal nId
    ideal nId

  lvint:=[variable(xx)::OV for xx in vl]
  nvint1:=(#lvint-1)::NNI
```

```
deleteunit(lI: List FIdeal) : List FIdeal ==
  [I for I in lI | _^ element?(1$DPoly,I)]

rearrange(vlist:List OV) :List OV ==
  vlist=[] => vlist
  sort((z1,z2)+->z1>z2,setDifference(lvint,setDifference(lvint,vlist)))

       ---- radical of a 0-dimensional ideal  ----
zeroRadComp(I:FIdeal,truelist:List OV) : FIdeal ==
  truelist=[] => I
  Id:=generators I
  x:OV:=truelist.last
  #Id=1 =>
    f:=Id.first
    g:= (f exquo (gcd (f,differentiate(f,x))))::DPoly
    groebnerIdeal([g])
  y:=truelist.first
  px:DPoly:=x::DPoly
  py:DPoly:=y::DPoly
  f:=Id.last
  g:= (f exquo (gcd (f,differentiate(f,x))))::DPoly
  Id:=groebner(cons(g,remove(f,Id)))
  lf:=Id.first
  pv:DPoly:=0
  pw:DPoly:=0
  while degree(lf,y)^=1 repeat
    val:=random()$Z rem 23
    pv:=px+val*py
    pw:=px-val*py
    Id:=groebner([(univariate(h,x)).pv for h in Id])
    lf:=Id.first
  ris:= generators(zeroRadComp(groebnerIdeal(Id.rest),truelist.rest))
  ris:=cons(lf,ris)
  if pv^=0 then
    ris:=[(univariate(h,x)).pw for h in ris]
  groebnerIdeal(groebner ris)

     ----  find the power that stabilizes (I:s)  ----
goodPower(s:DPoly,I:FIdeal) : Record(spol:DPoly,id:FIdeal) ==
  f:DPoly:=s
  I:=groebner I
  J:=generators(JJ:= (saturate(I,s)))
  while _^ in?(ideal([f*g for g in J]),I) repeat f:=s*f
  [f,JJ]

        ----  is the ideal zerodimensional?  ----
  ----     the "true variables" are  in truelist         ----
zerodimcase(J:FIdeal,truelist:List OV) : Boolean ==
  element?(1,J) => true
  truelist=[] => true
```

```
  n:=#truelist
  Jd:=groebner generators J
  for x in truelist while Jd^=[] repeat
    f := Jd.first
    Jd:=Jd.rest
    if ((y:=mainVariable f) case "failed") or (y::OV ^=x )
         or _^ (ismonic (f,x)) then return false
    while Jd^=[] and (mainVariable Jd.first)::OV=x repeat Jd:=Jd.rest
    if Jd=[] and position(x,truelist)<n then return false
  true

    ----  choose the variable for the reduction step  ----
              --- J groebnerner in gen pos  ---
findvar(J:FIdeal,truelist:List OV) : OV ==
  lmonicvar:List OV :=[]
  for f in generators J repeat
    t:=f - reductum f
    vt:List OV :=variables t
    if #vt=1 then lmonicvar:=setUnion(vt,lmonicvar)
  badvar:=setDifference(truelist,lmonicvar)
  badvar.first

      ---- function for the "reduction step  ----
reduceDim(flag:Fun0,J:FIdeal,truelist:List OV) : List(FIdeal) ==
  element?(1,J) => [J]
  zerodimcase(J,truelist) =>
    (flag case "zeroPrimDecomp") => zeroPrimDecomp(J,truelist)
    (flag case "zeroRadComp") => [zeroRadComp(J,truelist)]
  x:OV:=findvar(J,truelist)
  Jnew:=[pushdown(f,x) for f in generators J]
  Jc: List FIdeal :=[]
  Jc:=reduceDim(flag,groebnerIdeal Jnew,remove(x,truelist))
  res1:=[ideal([pushup(f,x) for f in generators idp]) for idp in Jc]
  s:=pushup((_*/[leadingCoefficient f for f in Jnew])::DPoly,x)
  degree(s,x)=0 => res1
  res1:=[saturate(II,s) for II in res1]
  good:=goodPower(s,J)
  sideal := groebnerIdeal(groebner(cons(good.spol,generators J)))
  in?(good.id, sideal) => res1
  sresult:=reduceDim(flag,sideal,truelist)
  for JJ in sresult repeat
     if not(in?(good.id,JJ)) then res1:=cons(JJ,res1)
  res1

 ----  Primary Decomposition for 0-dimensional ideals  ----
zeroPrimDecomp(I:FIdeal,truelist:List OV): List(FIdeal) ==
  truelist=[] => list I
  newJ:=genPosLastVar(I,truelist);lval:=newJ.changeval;
  J:=groebner newJ.genideal
  x:=truelist.last
```

```
      Jd:=generators J
      g:=Jd.last
      lfact:= factors factor(g)
      ris:List FIdeal:=[]
      for ef in lfact repeat
        g:DPoly:=(ef.factor)**(ef.exponent::NNI)
        J1:= groebnerIdeal(groebner cons(g,Jd))
        if _^ (is0dimprimary (J1,truelist)) then
                                  return zeroPrimDecomp(I,truelist)
        ris:=cons(groebner backGenPos(J1,lval,truelist),ris)
      ris

            ----  radical of an Ideal  ----
    radical(I:Ideal) : Ideal ==
      J:=groebner(internalForm I)
      truelist:=rearrange("setUnion"/[variables f for f in generators J])
      truelist=[] => externalForm J
      externalForm("intersect"/reduceDim("zeroRadComp",J,truelist))


-- the following functions are used to "push" x in the coefficient ring -

       ----  push x in the coefficient domain for a polynomial ----
    pushdown(g:DPoly,x:OV) : DPoly ==
      rf:DPoly:=0$DPoly
      i:=position(x,lvint)
      while g^=0 repeat
        g1:=reductum g
        rf:=rf+pushdterm(g-g1,x,i)
        g := g1
      rf

     ----  push x in the coefficient domain for a term ----
    pushdterm(t:DPoly,x:OV,i:Z):DPoly ==
      n:=degree(t,x)
      xp:=convert(x)@SE
      cf:=monomial(1,xp,n)$P :: F
      newt := t exquo monomial(1,x,n)$DPoly
      cf * newt::DPoly

             ----  push back the variable  ----
    pushup(f:DPoly,x:OV) :DPoly ==
      h:=1$P
      rf:DPoly:=0$DPoly
      g := f
      xp := convert(x)@SE
      while g^=0 repeat
        h:=lcm(trueden(denom leadingCoefficient g,xp),h)
        g:=reductum g
      f:=(h::F)*f
```

```
  while f^=0 repeat
    g:=reductum f
    rf:=rf+pushuterm(f-g,xp,x)
    f:=g
  rf

trueden(c:P,x:SE) : P ==
  degree(c,x) = 0 => 1
  c

 ----  push x back from the coefficient domain for a term ----
pushuterm(t:DPoly,xp:SE,x:OV):DPoly ==
  pushucoef((univariate(numer leadingCoefficient t,xp)$P), x)*
    monomial(inv((denom leadingCoefficient t)::F),degree t)$DPoly


pushucoef(c:UP,x:OV):DPoly ==
  c = 0 => 0
  monomial((leadingCoefficient c)::F::DPoly,x,degree c) +
     pushucoef(reductum c,x)

      -- is the 0-dimensional ideal I primary ?  --
          ----  internal function  ----
is0dimprimary(J:FIdeal,truelist:List OV) : Boolean ==
  element?(1,J) => true
  Jd:=generators(groebner J)
  #(factors factor Jd.last)^=1 => return false
  i:=subtractIfCan(#truelist,1)
  (i case "failed") => return true
  JR:=(reverse Jd);JM:=groebnerIdeal([JR.first]);JP:List(DPoly):=[]
  for f in JR.rest repeat
    if _^ ismonic(f,truelist.i) then
      if _^ inRadical?(f,JM) then return false
      JP:=cons(f,JP)
     else
      x:=truelist.i
      i:=(i-1)::NNI
      if _^ testPower(univariate(f,x),x,JM) then return false
      JM :=groebnerIdeal(append(cons(f,JP),generators JM))
  true

    ---- Functions for the General Position step  ----

  ----  put the ideal in general position  ----
genPosLastVar(J:FIdeal,truelist:List OV):GenPos ==
  x := last truelist ;lv1:List OV :=remove(x,truelist)
  ranvals:List(Z):=[(random()$Z rem 23) for vv in lv1]
  val:=_+/[rv*(vv::DPoly)  for vv in lv1 for rv in ranvals]
  val:=val+(x::DPoly)
  [ranvals,groebnerIdeal(groebner([(univariate(p,x)).val
```

```
                        for p in generators J]))]$GenPos


       ----  convert back the ideal  ----
backGenPos(I:FIdeal,lval:List Z,truelist:List OV) : FIdeal ==
  lval=[] => I
  x := last truelist ;lv1:List OV:=remove(x,truelist)
  val:=-(_+/[rv*(vv::DPoly) for vv in lv1 for rv in lval])
  val:=val+(x::DPoly)
  groebnerIdeal
    (groebner([(univariate(p,x)).val for p in generators I ]))

ismonic(f:DPoly,x:OV) : Boolean ==
  ground? leadingCoefficient(univariate(f,x))

    ---- test if f is power of a linear mod (rad J) ----
              ----  f is monic  ----
testPower(uf:SUP,x:OV,J:FIdeal) : Boolean ==
  df:=degree(uf)
  trailp:DPoly := inv(df:Z ::F) *coefficient(uf,(df-1)::NNI)
  linp:SUP:=(monomial(1$DPoly,1$NNI)$SUP +
            monomial(trailp,0$NNI)$SUP)**df
  g:DPoly:=multivariate(uf-linp,x)
  inRadical?(g,J)


              ----  Exported Functions  ----

      -- is the 0-dimensional ideal I prime ?  --
zeroDimPrime?(I:Ideal) : Boolean ==
  J:=groebner((genPosLastVar(internalForm I,lvint)).genideal)
  element?(1,J) => true
  n:NNI:=#vl;i:NNI:=1
  Jd:=generators J
  #Jd^=n => false
  for f in Jd repeat
    if _^ ismonic(f,lvint.i) then return false
    if i<n and (degree univariate(f,lvint.i))^=1 then return false
    i:=i+1
  g:=Jd.n
  #(lfact:=factors(factor g)) >1 => false
  lfact.1.exponent =1


      -- is the 0-dimensional ideal I primary ?  --
zeroDimPrimary?(J:Ideal):Boolean ==
  is0dimprimary(internalForm J,lvint)

       ----  Primary Decomposition of I  -----
```

```
    primaryDecomp(I:Ideal) : List(Ideal) ==
      J:=groebner(internalForm I)
      truelist:=rearrange("setUnion"/[variables f for f in generators J])
      truelist=[] => [externalForm J]
      [externalForm II for II in reduceDim("zeroPrimDecomp",J,truelist)]

         ----  contract I to the ring with lvar variables  ----
    contract(I:Ideal,lvar: List OV) : Ideal ==
      Id:= generators(groebner I)
      empty?(Id) => I
      fullVars:= "setUnion"/[variables g for g in Id]
      fullVars = lvar => I
      n:= # lvar
      #fullVars < n  => error "wrong vars"
      n=0 => I
      newVars:=
        append([vv for vv in fullVars| ^member?(vv,lvar)]$List(OV),lvar)
      subsVars := [monomial(1,vv,1)$DPoly1 for vv in newVars]
      lJ:= [eval(g,fullVars,subsVars) for g in Id]
      J := groebner(lJ)
      J=[1] => groebnerIdeal J
      J=[0] => groebnerIdeal empty()
      J:=[f for f in J| member?(mainVariable(f)::OV,newVars)]
      fullPol :=[monomial(1,vv,1)$DPoly1 for vv in fullVars]
      groebnerIdeal([eval(gg,newVars,fullPol) for gg in J])
```

———-

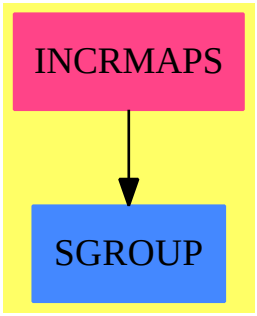
— IDECOMP.dotabb —

```
"IDECOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IDECOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"IDECOMP" -> "PFECAT"
"IDECOMP" -> "DIRPCAT"
```

———-

10.2 package INCRMAPS IncrementingMaps

10.2.1 IncrementingMaps (INCRMAPS)



Exports:
increment incrementBy

— package INCRMAPS IncrementingMaps —

```
)abbrev package INCRMAPS IncrementingMaps
++ Author:
++ Date Created:
++ Date Last Updated: June 4, 1991
++ Basic Operations:
++ Related Domains: UniversalSegment
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ This package provides operations to create incrementing functions.

IncrementingMaps(R:Join(Monoid, AbelianSemiGroup)): with
    increment: () -> (R -> R)
        ++ increment() produces a function which adds \spad{1} to whatever
        ++ argument it is given.  For example, if {f := increment()} then
        ++ \spad{f x} is \spad{x+1}.
    incrementBy: R -> (R -> R)
        ++ incrementBy(n) produces a function which adds \spad{n} to whatever
        ++ argument it is given.  For example, if {f := increment(n)} then
        ++ \spad{f x} is \spad{x+n}.
  == add
    increment()   == x +-> 1 + x
    incrementBy n == x +-> n + x
```

———-

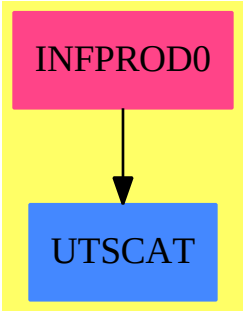
— **INCRMAPS.dotabb** —

```
"INCRMAPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INCRMAPS"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"INCRMAPS" -> "SGROUP"
```

———-

10.3 package INFPROD0 InfiniteProductCharacteristicZero

10.3.1 InfiniteProductCharacteristicZero (INFPROD0)



Exports:
evenInfiniteProduct infiniteProduct oddInfiniteProduct generalInfiniteProduct

— **package INFPROD0 InfiniteProductCharacteristicZero** —

```
)abbrev package INFPROD0 InfiniteProductCharacteristicZero
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Basic Operations: infiniteProduct, evenInfiniteProduct, oddInfiniteProduct,
++   generalInfiniteProduct
++ Related Domains: UnivariateTaylorSeriesCategory
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, infinite product
++ Examples:
++ References:
++ Description:
++ This package computes infinite products of univariate Taylor series
```

```
++ over an integral domain of characteristic 0.

InfiniteProductCharacteristicZero(Coef,UTS):_
 Exports == Implementation where
  Coef : Join(IntegralDomain,CharacteristicZero)
  UTS  : UnivariateTaylorSeriesCategory Coef
  I  ==> Integer

  Exports ==> with

    infiniteProduct: UTS -> UTS
      ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    evenInfiniteProduct: UTS -> UTS
      ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    oddInfiniteProduct: UTS -> UTS
      ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    generalInfiniteProduct: (UTS,I,I) -> UTS
      ++ generalInfiniteProduct(f(x),a,d) computes
      ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.

  Implementation ==> add

    import StreamInfiniteProduct Coef

    infiniteProduct x     == series infiniteProduct coefficients x
    evenInfiniteProduct x == series evenInfiniteProduct coefficients x
    oddInfiniteProduct x  == series oddInfiniteProduct coefficients x

    generalInfiniteProduct(x,a,d) ==
      series generalInfiniteProduct(coefficients x,a,d)
```

———-

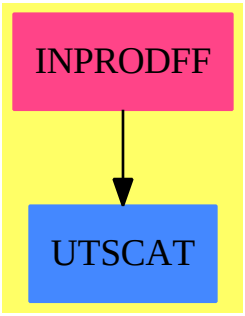
— INFPROD0.dotabb —

```
"INFPROD0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFPROD0"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"INFPROD0" -> "UTSCAT"
```

———-

10.4 package INPRODFF InfiniteProductFiniteField

10.4.1 InfiniteProductFiniteField (INPRODFF)



Exports:
evenInfiniteProduct generalInfiniteProduct infiniteProduct oddInfiniteProduct

— package INPRODFF InfiniteProductFiniteField —

```
)abbrev package INPRODFF InfiniteProductFiniteField
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Basic Operations: infiniteProduct, evenInfiniteProduct, oddInfiniteProduct,
++   generalInfiniteProduct
++ Related Domains: UnivariateTaylorSeriesCategory
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, infinite product
++ Examples:
++ References:
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over an arbitrary finite field.

InfiniteProductFiniteField(K,UP,Coef,UTS):_
 Exports == Implementation where
  K    :  Join(Field,Finite,ConvertibleTo Integer)
  UP   :  UnivariatePolynomialCategory K
  Coef :  MonogenicAlgebra(K,UP)
  UTS  :  UnivariateTaylorSeriesCategory Coef
  I   ==> Integer
  RN  ==> Fraction Integer
  SAE ==> SimpleAlgebraicExtension
  ST  ==> Stream
  STF ==> StreamTranscendentalFunctions
```

```
STT ==> StreamTaylorSeriesOperations
ST2 ==> StreamFunctions2
SUP ==> SparseUnivariatePolynomial

Exports ==> with

  infiniteProduct: UTS -> UTS
    ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
    ++ The series \spad{f(x)} should have constant coefficient 1.
  evenInfiniteProduct: UTS -> UTS
    ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
    ++ The series \spad{f(x)} should have constant coefficient 1.
  oddInfiniteProduct: UTS -> UTS
    ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
    ++ The series \spad{f(x)} should have constant coefficient 1.
  generalInfiniteProduct: (UTS,I,I) -> UTS
    ++ generalInfiniteProduct(f(x),a,d) computes
    ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
    ++ The series \spad{f(x)} should have constant coefficient 1.

Implementation ==> add

  liftPoly: UP -> SUP RN
  liftPoly poly ==
    -- lift coefficients of 'poly' to integers
    ans : SUP RN := 0
    while not zero? poly repeat
      coef := convert(leadingCoefficient poly)@I :: RN
      ans := ans + monomial(coef,degree poly)
      poly := reductum poly
    ans

  reducePoly: SUP RN -> UP
  reducePoly poly ==
    -- reduce coefficients of 'poly' to elements of K
    ans : UP := 0
    while not zero? poly repeat
      coef := numer(leadingCoefficient(poly)) :: K
      ans := ans + monomial(coef,degree poly)
      poly := reductum poly
    ans

  POLY := liftPoly definingPolynomial()$Coef
  ALG  := SAE(RN,SUP RN,POLY)

  infiniteProduct x ==
    stUP := map(lift,coefficients x)$ST2(Coef,UP)
    stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
    stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
    stALG := exp(lambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
```

```
      stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
      stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
      series map(reduce,stUP)$ST2(UP,Coef)

    evenInfiniteProduct x ==
      stUP := map(lift,coefficients x)$ST2(Coef,UP)
      stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
      stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
      stALG := exp(evenlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
      stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
      stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
      series map(reduce,stUP)$ST2(UP,Coef)

    oddInfiniteProduct x ==
      stUP := map(lift,coefficients x)$ST2(Coef,UP)
      stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
      stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
      stALG := exp(oddlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
      stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
      stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
      series map(reduce,stUP)$ST2(UP,Coef)

    generalInfiniteProduct(x,a,d) ==
      stUP := map(lift,coefficients x)$ST2(Coef,UP)
      stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
      stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
      stALG := generalLambert(log(stALG)$STF(ALG),a,d)$STT(ALG)
      stALG := exp(stALG)$STF(ALG)
      stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
      stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
      series map(reduce,stUP)$ST2(UP,Coef)
```

———-

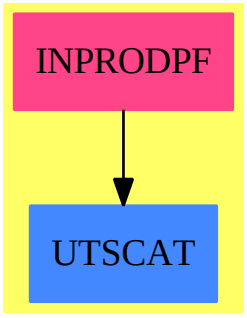
— INPRODFF.dotabb —

```
"INPRODFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPRODFF"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"INPRODFF" -> "UTSCAT"
```

———-

10.5 package INPRODPF InfiniteProductPrimeField

10.5.1 InfiniteProductPrimeField (INPRODPF)



Exports:
evenInfiniteProduct generalInfiniteProduct infiniteProduct oddInfiniteProduct

— package INPRODPF InfiniteProductPrimeField —

```
)abbrev package INPRODPF InfiniteProductPrimeField
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Basic Operations: infiniteProduct, evenInfiniteProduct, oddInfiniteProduct,
++   generalInfiniteProduct
++ Related Domains: UnivariateTaylorSeriesCategory
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, infinite product
++ Examples:
++ References:
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over a field of prime order.

InfiniteProductPrimeField(Coef,UTS): Exports == Implementation where
  Coef : Join(Field,Finite,ConvertibleTo Integer)
  UTS  : UnivariateTaylorSeriesCategory Coef
  I  ==> Integer
  ST ==> Stream

  Exports ==> with

    infiniteProduct: UTS -> UTS
      ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
```

```
    evenInfiniteProduct: UTS -> UTS
      ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    oddInfiniteProduct: UTS -> UTS
      ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    generalInfiniteProduct: (UTS,I,I) -> UTS
      ++ generalInfiniteProduct(f(x),a,d) computes
      ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.

  Implementation ==> add

    import StreamInfiniteProduct Integer

    applyOverZ:(ST I -> ST I,ST Coef) -> ST Coef
    applyOverZ(f,st) ==
      stZ := map(z1 +-> convert(z1)@Integer,st)$StreamFunctions2(Coef,I)
      map(z1 +-> z1 :: Coef,f stZ)$StreamFunctions2(I,Coef)

    infiniteProduct x ==
      series applyOverZ(infiniteProduct,coefficients x)
    evenInfiniteProduct x ==
      series applyOverZ(evenInfiniteProduct,coefficients x)
    oddInfiniteProduct x ==
      series applyOverZ(oddInfiniteProduct,coefficients x)
    generalInfiniteProduct(x,a,d) ==
      series
       applyOverZ(
        (z1:ST(I)):ST(I) +-> generalInfiniteProduct(z1,a,d),coefficients x)
```

———-

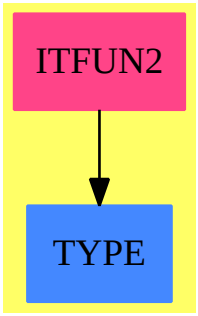
— **INPRODPF.dotabb** —

```
"INPRODPF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPRODPF"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"INPRODPF" -> "UTSCAT"
```

———-

10.6 package ITFUN2 InfiniteTupleFunctions2

10.6.1 InfiniteTupleFunctions2 (ITFUN2)



Exports:
map

— **package ITFUN2 InfiniteTupleFunctions2** —

```
)abbrev package ITFUN2 InfiniteTupleFunctions2
++ Description:
++ Functions defined on streams with entries in two sets.

InfiniteTupleFunctions2(A:Type,B:Type): Exports == Implementation where
  IT   ==> InfiniteTuple

  Exports ==> with
    map: ((A -> B),IT A) -> IT B
      ++ \spad{map(f,[x0,x1,x2,...])} returns \spad{[f(x0),f(x1),f(x2),..]}.

  Implementation ==> add

    map(f,x) ==
      map(f,x pretend Stream(A))$StreamFunctions2(A,B) pretend IT(B)
```

———-

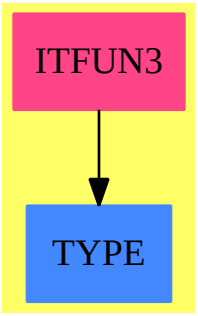
— **ITFUN2.dotabb** —

```
"ITFUN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITFUN2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITFUN2" -> "TYPE"
```

———-

10.7 package ITFUN3 InfiniteTupleFunctions3

10.7.1 InfiniteTupleFunctions3 (ITFUN3)



Exports:
map

— package ITFUN3 InfiniteTupleFunctions3 —

```
)abbrev package ITFUN3 InfiniteTupleFunctions3
++ Description:
++ Functions defined on streams with entries in two sets.

InfiniteTupleFunctions3(A:Type, B:Type,C:Type): Exports
 == Implementation where
   IT   ==> InfiniteTuple
   ST   ==> Stream
   SF3  ==> StreamFunctions3(A,B,C)
   FUN  ==> ((A,B)->C)
   Exports ==> with
     map: (((A,B)->C), IT A, IT B) -> IT C
       ++ map(f,a,b) \undocumented
     map: (((A,B)->C), ST A, IT B) -> ST C
       ++ map(f,a,b) \undocumented
     map: (((A,B)->C), IT A, ST B) -> ST C
       ++ map(f,a,b) \undocumented

   Implementation ==> add

     map(f:FUN, s1:IT A, s2:IT B):IT C ==
       map(f, s1 pretend Stream(A), s2 pretend Stream(B))$SF3 pretend IT(C)
     map(f:FUN, s1:ST A, s2:IT B):ST C ==
       map(f, s1, s2 pretend Stream(B))$SF3
     map(f:FUN, s1:IT A, s2:ST B):ST C ==
       map(f, s1 pretend Stream(A), s2)$SF3
```

———-

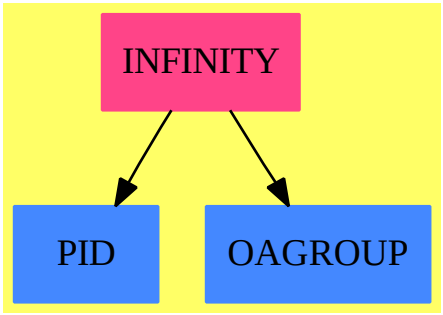
— **ITFUN3.dotabb** —

```
"ITFUN3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITFUN3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITFUN3" -> "TYPE"
```

———-

10.8 package INFINITY Infinity

10.8.1 Infinity (INFINITY)



Exports:
infinity minusInfinity plusInfinity

— **package INFINITY Infinity** —

```
)abbrev package INFINITY Infinity
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Top-level infinity
++ Default infinity signatures for the interpreter;

Infinity(): with
  infinity     : () -> OnePointCompletion Integer
    ++ infinity() returns infinity.
  plusInfinity : () -> OrderedCompletion  Integer
    ++ plusInfinity() returns plusIinfinity.
  minusInfinity: () -> OrderedCompletion  Integer
```

```
    ++ minusInfinity() returns minusInfinity.
 == add
  infinity()      == infinity()$OnePointCompletion(Integer)
  plusInfinity()  == plusInfinity()$OrderedCompletion(Integer)
  minusInfinity() == minusInfinity()$OrderedCompletion(Integer)
```

———-

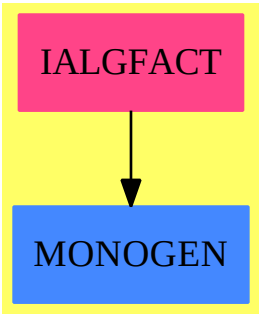
— **INFINITY.dotabb** —

```
"INFINITY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFINITY"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"INFINITY" -> "PID"
"INFINITY" -> "OAGROUP"
```

———-

10.9 package IALGFACT InnerAlgFactor

10.9.1 InnerAlgFactor (IALGFACT)



Exports:
factor

— **package IALGFACT InnerAlgFactor** —

```
)abbrev package IALGFACT InnerAlgFactor
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: 20 Jul 1988
++ Keywords: factorization, algebraic extension, univariate polynomial
```

```
++ Description:
++ Factorisation in a simple algebraic extension
++ Factorization of univariate polynomials with coefficients in an
++ algebraic extension of a field over which we can factor UP's;

InnerAlgFactor(F, UP, AlExt, AlPol): Exports == Implementation where
  F    : Field
  UP   : UnivariatePolynomialCategory F
  AlPol: UnivariatePolynomialCategory AlExt
  AlExt : Join(Field, CharacteristicZero, MonogenicAlgebra(F,UP))
  NUP   ==> SparseUnivariatePolynomial UP
  N     ==> NonNegativeInteger
  Z     ==> Integer
  FR    ==> Factored UP
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2


  Exports ==> with
    factor: (AlPol, UP -> FR)  ->  Factored AlPol
      ++ factor(p, f) returns a prime factorisation of p;
      ++ f is a factorisation map for elements of UP;

  Implementation ==> add
    pnorm        : AlPol -> UP
    convrt       : AlPol -> NUP
    change       : UP    -> AlPol
    perturbfactor: (AlPol, Z, UP -> FR) -> List AlPol
    irrfactor    : (AlPol, Z, UP -> FR) -> List AlPol


    perturbfactor(f, k, fact) ==
      pol   := monomial(1$AlExt,1)-
               monomial(reduce monomial(k::F,1)$UP ,0)
      newf  := elt(f, pol)
      lsols := irrfactor(newf, k, fact)
      pol   := monomial(1, 1) +
               monomial(reduce monomial(k::F,1)$UP,0)
      [elt(pp, pol) for pp in lsols]

    ---  factorize the square-free parts of f  ---
    irrfactor(f, k, fact) ==
      degree(f) =$N 1 => [f]
      newf := f
      nn   := pnorm f
      --newval:RN:=1
      --pert:=false
      --if ^ SqFr? nn then
      --  pert:=true
      --  newterm:=perturb(f)
      --  newf:=newterm.ppol
```

```
      --  newval:=newterm.pval
      --  nn:=newterm.nnorm
      listfact := factors fact nn
      #listfact =$N 1 =>
        first(listfact).exponent =$Z 1 => [f]
        perturbfactor(f, k + 1, fact)
      listerm:List(AlPol):= []
      for pelt in listfact repeat
        g    := gcd(change(pelt.factor), newf)
        newf := (newf exquo g)::AlPol
        listerm :=
          pelt.exponent =$Z 1 => cons(g, listerm)
          append(perturbfactor(g, k + 1, fact), listerm)
      listerm

    factor(f, fact) ==
      sqf := squareFree f
      unit(sqf) * _*/[_*/[primeFactor(pol, sqterm.exponent)
                          for pol in irrfactor(sqterm.factor, 0, fact)]
                                        for sqterm in factors sqf]

    p := definingPolynomial()$AlExt
    newp := map(x +-> x::UP, p)$UPCF2(F, UP, UP, NUP)

    pnorm  q == resultant(convrt q, newp)
    change q == map(coerce, q)$UPCF2(F,UP,AlExt,AlPol)

    convrt q ==
      swap(map(lift, q)$UPCF2(AlExt, AlPol,
           UP, NUP))$CommuteUnivariatePolynomialCategory(F, UP, NUP)
```

———-

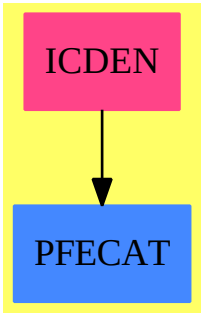
— IALGFACT.dotabb —

```
"IALGFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IALGFACT"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"IALGFACT" -> "MONOGEN"
```

———-

10.10 package ICDEN InnerCommonDenominator

10.10.1 InnerCommonDenominator (ICDEN)



Exports:
clearDenominator commonDenominator splitDenominator

— package ICDEN InnerCommonDenominator —

```
)abbrev package ICDEN InnerCommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Keywords: gcd, quotient, common, denominator.
++ Description:
++ InnerCommonDenominator provides functions to compute
++ the common denominator of a finite linear aggregate of elements
++ of the quotient field of an integral domain.

InnerCommonDenominator(R, Q, A, B): Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R
  A: FiniteLinearAggregate R
  B: FiniteLinearAggregate Q

  Exports ==> with
    commonDenominator: B -> R
      ++ commonDenominator([q1,...,qn]) returns a common denominator
      ++ d for q1,...,qn.
    clearDenominator : B -> A
      ++ clearDenominator([q1,...,qn]) returns \spad{[p1,...,pn]} such that
      ++ \spad{qi = pi/d} where d is a common denominator for the qi's.
    splitDenominator : B -> Record(num: A, den: R)
      ++ splitDenominator([q1,...,qn]) returns
      ++ \spad{[[p1,...,pn], d]} such that
      ++ \spad{qi = pi/d} and d is a common denominator for the qi's.
```

```
  Implementation ==> add
    import FiniteLinearAggregateFunctions2(Q, B, R, A)

    clearDenominator l ==
      d := commonDenominator l
      map(x +-> numer(d*x), l)

    splitDenominator l ==
      d := commonDenominator l
      [map(x +-> numer(d*x), l), d]

    if R has GcdDomain then
      commonDenominator l == reduce(lcm, map(denom, l),1)
    else
      commonDenominator l == reduce("*", map(denom, l), 1)
```

———-

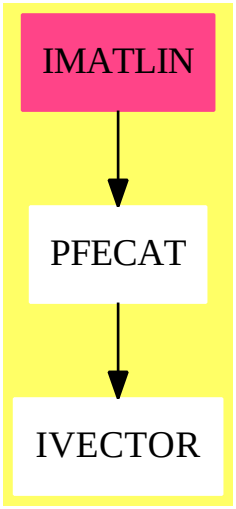
— ICDEN.dotabb —

```
"ICDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ICDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ICDEN" -> "PFECAT"
```

———-

10.11 package IMATLIN InnerMatrixLinearAlgebraFunctions

10.11.1 InnerMatrixLinearAlgebraFunctions (IMATLIN)



Exports:

— package IMATLIN InnerMatrixLinearAlgebraFunctions —

```
)abbrev package IMATLIN InnerMatrixLinearAlgebraFunctions
++ Author: Clifton J. Williamson, P.Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: September 1993
++ Basic Operations:
++ Related Domains: IndexedMatrix(R,minRow,minCol), Matrix(R),
++    RectangularMatrix(n,m,R), SquareMatrix(n,R)
++ Also See:
++ AMS Classifications:
++ Keywords: matrix, canonical forms, linear algebra
++ Examples:
++ References:
++ Description:
++ \spadtype{InnerMatrixLinearAlgebraFunctions} is an internal package
++ which provides standard linear algebra functions on domains in
++ \spad{MatrixCategory}

InnerMatrixLinearAlgebraFunctions(R,Row,Col,M):_
             Exports == Implementation where
```

```
R   : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M   : MatrixCategory(R,Row,Col)
I ==> Integer

Exports ==> with
  rowEchelon: M -> M
    ++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.
  rank: M -> NonNegativeInteger
    ++ \spad{rank(m)} returns the rank of the matrix m.
  nullity: M -> NonNegativeInteger
    ++ \spad{nullity(m)} returns the mullity of the matrix m. This is the
    ++ dimension of the null space of the matrix m.
  if Col has shallowlyMutable then
    nullSpace: M -> List Col
      ++ \spad{nullSpace(m)} returns a basis for the null space of the
      ++ matrix m.
  determinant: M -> R
    ++ \spad{determinant(m)} returns the determinant of the matrix m.
    ++ an error message is returned if the matrix is not square.
  generalizedInverse: M -> M
    ++ \spad{generalizedInverse(m)} returns the generalized (Moore--Penrose)
    ++ inverse of the matrix m, i.e. the matrix h such that
    ++ m*h*m=h, h*m*h=m, m*h and h*m are both symmetric matrices.
  inverse: M -> Union(M,"failed")
    ++ \spad{inverse(m)} returns the inverse of the matrix m.
    ++ If the matrix is not invertible, "failed" is returned.
    ++ Error: if the matrix is not square.

Implementation ==> add

  rowAllZeroes?: (M,I) -> Boolean
  rowAllZeroes?(x,i) ==
    -- determines if the ith row of x consists only of zeroes
    -- internal function: no check on index i
    for j in minColIndex(x)..maxColIndex(x) repeat
      qelt(x,i,j) ^= 0 => return false
    true

  colAllZeroes?: (M,I) -> Boolean
  colAllZeroes?(x,j) ==
    -- determines if the ith column of x consists only of zeroes
    -- internal function: no check on index j
    for i in minRowIndex(x)..maxRowIndex(x) repeat
      qelt(x,i,j) ^= 0 => return false
    true

  rowEchelon y ==
    -- row echelon form via Gaussian elimination
```

```
    x := copy y
    minR := minRowIndex x; maxR := maxRowIndex x
    minC := minColIndex x; maxC := maxColIndex x
    i := minR
    n: I := minR - 1
    for j in minC..maxC repeat
      i > maxR => return x
      n := minR - 1
      -- n = smallest k such that k >= i and x(k,j) ^= 0
      for k in i..maxR repeat
        if qelt(x,k,j) ^= 0 then leave (n := k)
      n = minR - 1 => "no non-zeroes"
      -- put nth row in ith position
      if i ^= n then swapRows_!(x,i,n)
      -- divide ith row by its first non-zero entry
      b := inv qelt(x,i,j)
      qsetelt_!(x,i,j,1)
      for k in (j+1)..maxC repeat qsetelt_!(x,i,k,b * qelt(x,i,k))
      -- perform row operations so that jth column has only one 1
      for k in minR..maxR repeat
        if k ^= i and qelt(x,k,j) ^= 0 then
          for k1 in (j+1)..maxC repeat
            qsetelt_!(x,k,k1,qelt(x,k,k1) - qelt(x,k,j) * qelt(x,i,k1))
          qsetelt_!(x,k,j,0)
      -- increment i
      i := i + 1
    x

  rank x ==
    y :=
      (rk := nrows x) > (rh := ncols x) =>
        rk := rh
        transpose x
      copy x
    y := rowEchelon y; i := maxRowIndex y
    while rk > 0 and rowAllZeroes?(y,i) repeat
      i := i - 1
      rk := (rk - 1) :: NonNegativeInteger
    rk :: NonNegativeInteger

  nullity x == (ncols x - rank x) :: NonNegativeInteger

  if Col has shallowlyMutable then

    nullSpace y ==
      x := rowEchelon y
      minR := minRowIndex x; maxR := maxRowIndex x
      minC := minColIndex x; maxC := maxColIndex x
      nrow := nrows x; ncol := ncols x
      basis : List Col := nil()
```

```
    rk := nrow; row := maxR
    -- compute rank = # rows - # rows of all zeroes
    while rk > 0 and rowAllZeroes?(x,row) repeat
      rk := (rk - 1) :: NonNegativeInteger
      row := (row - 1) :: NonNegativeInteger
    -- if maximal rank, return zero vector
    ncol <= nrow and rk = ncol => [new(ncol,0)]
    -- if rank = 0, return standard basis vectors
    rk = 0 =>
      for j in minC..maxC repeat
        w : Col := new(ncol,0)
        qsetelt_!(w,j,1)
        basis := cons(w,basis)
      basis
    -- v contains information about initial 1's in the rows of x
    -- if the ith row has an initial 1 in the jth column, then
    -- v.j = i; v.j = minR - 1, otherwise
    v : IndexedOneDimensionalArray(I,minC) := new(ncol,minR - 1)
    for i in minR..(minR + rk - 1) repeat
      for j in minC.. while qelt(x,i,j) = 0 repeat j
      qsetelt_!(v,j,i)
    j := maxC; l := minR + ncol - 1
    while j >= minC repeat
      w : Col := new(ncol,0)
      -- if there is no row with an initial 1 in the jth column,
      -- create a basis vector with a 1 in the jth row
      if qelt(v,j) = minR - 1 then
        colAllZeroes?(x,j) =>
          qsetelt_!(w,l,1)
          basis := cons(w,basis)
        for k in minC..(j-1) for ll in minR..(l-1) repeat
          if qelt(v,k) ^= minR - 1 then
            qsetelt_!(w,ll,-qelt(x,qelt(v,k),j))
        qsetelt_!(w,l,1)
        basis := cons(w,basis)
      j := j - 1; l := l - 1
    basis

determinant y ==
  (ndim := nrows y) ^= (ncols y) =>
    error "determinant: matrix must be square"
  -- Gaussian Elimination
  ndim = 1 => qelt(y,minRowIndex y,minColIndex y)
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  ans : R := 1
  for i in minR..(maxR - 1) for j in minC..(maxC - 1) repeat
    if qelt(x,i,j) = 0 then
      rown := minR - 1
```

```
      for k in (i+1)..maxR repeat
        qelt(x,k,j) ^= 0 => leave (rown := k)
      if rown = minR - 1 then return 0
      swapRows_!(x,i,rown); ans := -ans
    ans := qelt(x,i,j) * ans; b := -inv qelt(x,i,j)
    for l in (j+1)..maxC repeat qsetelt_!(x,i,l,b * qelt(x,i,l))
    for k in (i+1)..maxR repeat
      if (b := qelt(x,k,j)) ^= 0 then
        for l in (j+1)..maxC repeat
          qsetelt_!(x,k,l,qelt(x,k,l) + b * qelt(x,i,l))
  qelt(x,maxR,maxC) * ans

generalizedInverse(x) ==
  SUP:=SparseUnivariatePolynomial R
  FSUP := Fraction SUP
  VFSUP := Vector FSUP
  MATCAT2 := MatrixCategoryFunctions2(R, Row, Col, M,
               FSUP, VFSUP, VFSUP, Matrix FSUP)
  MATCAT22 := MatrixCategoryFunctions2(FSUP, VFSUP, VFSUP, Matrix FSUP,
               R, Row, Col, M)
  y:= map((r1:R):FSUP +-> coerce(coerce(r1)$SUP)$(Fraction SUP),x)$MATCAT2
  ty:=transpose y
  yy:=ty*y
  nc:=ncols yy
  var:=monomial(1,1)$SUP ::(Fraction SUP)
  yy:=inverse(yy+scalarMatrix(ncols yy,var))::Matrix(FSUP)*ty
  map((z1:FSUP):R +-> elt(z1,0),yy)$MATCAT22

inverse x ==
  (ndim := nrows x) ^= (ncols x) =>
    error "inverse: matrix must be square"
  ndim = 2 =>
     ans2 : M := zero(ndim, ndim)
     zero?(det :=  x(1,1)*x(2,2)-x(1,2)*x(2,1)) => "failed"
     detinv := inv det
     ans2(1,1) := x(2,2)*detinv
     ans2(1,2) := -x(1,2)*detinv
     ans2(2,1) := -x(2,1)*detinv
     ans2(2,2) := x(1,1)*detinv
     ans2
  AB : M := zero(ndim,ndim + ndim)
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  kmin := minRowIndex AB; kmax := kmin + ndim - 1
  lmin := minColIndex AB; lmax := lmin + ndim - 1
  for i in minR..maxR for k in kmin..kmax repeat
    for j in minC..maxC for l in lmin..lmax repeat
      qsetelt_!(AB,k,l,qelt(x,i,j))
    qsetelt_!(AB,k,lmin + ndim + k - kmin,1)
  AB := rowEchelon AB
```

```
      elt(AB,kmax,lmax) = 0 => "failed"
      subMatrix(AB,kmin,kmax,lmin + ndim,lmax + ndim)
```

———-

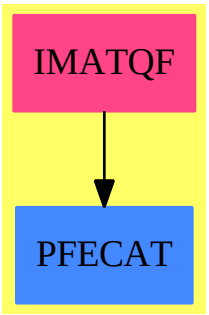
— **IMATLIN.dotabb** —

```
"IMATLIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IMATLIN"]
"PFECAT" -> "IVECTOR"
"IMATLIN" -> "PFECAT"
```

———-

10.12 package IMATQF InnerMatrixQuotientFieldFunctions

10.12.1 InnerMatrixQuotientFieldFunctions (IMATQF)



Exports:
inverse rowEchelon nullSpace

— **package IMATQF InnerMatrixQuotientFieldFunctions** —

```
)abbrev package IMATQF InnerMatrixQuotientFieldFunctions
++ Author: Clifton J. Williamson
++ Date Created: 22 November 1989
++ Date Last Updated: 22 November 1989
++ Basic Operations:
++ Related Domains: IndexedMatrix(R,minRow,minCol), Matrix(R), RectangularMatrix(n,m,R), SquareMatrix(n,R)
++ Also See:
++ AMS Classifications:
```

```
++ Keywords: matrix, inverse, integral domain
++ Examples:
++ References:
++ Description:
++ \spadtype{InnerMatrixQuotientFieldFunctions} provides functions on matrices
++ over an integral domain which involve the quotient field of that integral
++ domain.  The functions rowEchelon and inverse return matrices with
++ entries in the quotient field.

InnerMatrixQuotientFieldFunctions(R,Row,Col,M,QF,Row2,Col2,M2):_
         Exports == Implementation where
  R    : IntegralDomain
  Row  : FiniteLinearAggregate R
  Col  : FiniteLinearAggregate R
  M    : MatrixCategory(R,Row,Col)
  QF   : QuotientFieldCategory R
  Row2 : FiniteLinearAggregate QF
  Col2 : FiniteLinearAggregate QF
  M2   : MatrixCategory(QF,Row2,Col2)
  IMATLIN ==> InnerMatrixLinearAlgebraFunctions(QF,Row2,Col2,M2)
  MATCAT2 ==> MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Col2,M2)
  CDEN    ==> InnerCommonDenominator(R,QF,Col,Col2)

  Exports ==> with
      rowEchelon: M -> M2
        ++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.
        ++ the result will have entries in the quotient field.
      inverse: M -> Union(M2,"failed")
        ++ \spad{inverse(m)} returns the inverse of the matrix m.
        ++ If the matrix is not invertible, "failed" is returned.
        ++ Error: if the matrix is not square.
        ++ Note that the result will have entries in the quotient field.
      if Col2 has shallowlyMutable then
        nullSpace : M -> List Col
          ++ \spad{nullSpace(m)} returns a basis for the null space of the
          ++ matrix m.
  Implementation ==> add

    qfMat: M -> M2
    qfMat m == map((r1:R):QF +-> r1::QF,m)$MATCAT2

    rowEchelon m == rowEchelon(qfMat m)$IMATLIN
    inverse m ==
      (inv := inverse(qfMat m)$IMATLIN) case "failed" => "failed"
      inv :: M2

    if Col2 has shallowlyMutable then
      nullSpace m ==
        [clearDenominator(v)$CDEN for v in nullSpace(qfMat m)$IMATLIN]
```

———-

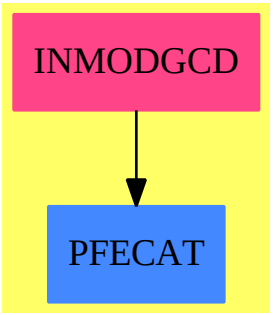
— IMATQF.dotabb —

```
"IMATQF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IMATQF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"IMATQF" -> "PFECAT"
```

———-

10.13 package INMODGCD InnerModularGcd

10.13.1 InnerModularGcd (INMODGCD)



Exports:
modularGcd modularGcdPrimitive reduction

— package INMODGCD InnerModularGcd —

```
)abbrev package INMODGCD InnerModularGcd
++ Author: J.H. Davenport and P. Gianni
++ Date Created: July 1990
++ Date Last Updated: November 1991
++ Description:
++ This file contains the functions for modular gcd algorithm
++ for univariate polynomials with coefficients in a
++ non-trivial euclidean domain (i.e. not a field).
++ The package parametrised by the coefficient domain,
++ the polynomial domain, a prime, and a function for choosing the next prime


InnerModularGcd(R,BP,pMod,nextMod):C == T
 where
```

```
Z    ==> Integer
NNI  ==> NonNegativeInteger
R       :  EuclideanDomain
BP      :  UnivariatePolynomialCategory(R)
pMod    :  R
nextMod :  (R,NNI) -> R

C == with
   modularGcdPrimitive    : List BP  -> BP
       ++ modularGcdPrimitive(f1,f2) computes the gcd of the two polynomials
       ++ f1 and f2 by modular methods.
   modularGcd    : List BP  -> BP
       ++ modularGcd(listf) computes the gcd of the list of polynomials
       ++ listf  by modular methods.
   reduction :       (BP,R)              ->   BP
       ++ reduction(f,p) reduces the coefficients of the polynomial f
       ++ modulo the prime p.

T == add

                 -- local functions --
  height    :         BP               ->   NNI
  mbound    :       (BP,BP)            ->   NNI
  modGcdPrimitive    :   (BP,BP)       ->   BP
  test         :     (BP,BP,BP)        ->   Boolean
  merge        :        (R,R)          ->   Union(R,"failed")
  modInverse   :      (R,R)            ->   R
  exactquo     :       (BP,BP,R)       ->   Union(BP,"failed")
  constNotZero :           BP          ->   Boolean
  constcase    : (List NNI ,List BP )  ->   BP
  lincase      : (List NNI ,List BP )  ->   BP


  if R has IntegerNumberSystem then
      reduction(u:BP,p:R):BP ==
          p = 0 => u
          map((r1:R):R +-> symmetricRemainder(r1,p),u)
    else
      reduction(u:BP,p:R):BP ==
          p = 0 => u
          map((r1:R):R +-> r1 rem p,u)

  FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)
  zeroChar : Boolean := R has CharacteristicZero

              --  exported functions --

  -- modular Gcd for a list of primitive polynomials
  modularGcdPrimitive(listf : List BP) :BP ==
    empty? listf => 0$BP
```

```
    g := first listf
    for f in rest listf | ^zero? f  while degree g > 0 repeat
      g:=modGcdPrimitive(g,f)
    g

  -- gcd for univariate polynomials
  modularGcd(listf : List BP): BP  ==
    listf:=remove!(0$BP,listf)
    empty? listf => 0$BP
    # listf = 1 => first listf
    minpol:=1$BP
    -- extract a monomial gcd
    mdeg:= "min"/[minimumDegree f for f in listf]
    if mdeg>0 then
      minpol1:= monomial(1,mdeg)
      listf:= [(f exquo minpol1)::BP for f in listf]
      minpol:=minpol*minpol1
    listdeg:=[degree f for f in listf ]
  -- make the polynomials primitive
    listCont := [content f for f in listf]
    contgcd:= gcd listCont
    -- make the polynomials primitive
    listf :=[(f exquo cf)::BP for f in listf for cf in listCont]
    minpol:=contgcd*minpol
    ans:BP :=
       --one polynomial is constant
       member?(1,listf) => 1
       --one polynomial is linear
       member?(1,listdeg) => lincase(listdeg,listf)
       modularGcdPrimitive listf
    minpol*ans

                --  local functions --

  --one polynomial is linear, remark that they are primitive
  lincase(listdeg:List NNI ,listf:List BP ): BP  ==
    n:= position(1,listdeg)
    g:=listf.n
    for f in listf repeat
      if (f1:=f exquo g) case "failed" then return 1$BP
    g

  -- test if d is the gcd
  test(f:BP,g:BP,d:BP):Boolean ==
    d0:=coefficient(d,0)
    coefficient(f,0) exquo d0 case "failed" => false
    coefficient(g,0) exquo d0 case "failed" => false
    f exquo d case "failed" => false
    g exquo d case "failed" => false
    true
```

```
-- gcd and cofactors for PRIMITIVE univariate polynomials
-- also assumes that constant terms are non zero
modGcdPrimitive(f:BP,g:BP): BP ==
  df:=degree f
  dg:=degree g
  dp:FP
  lcf:=leadingCoefficient f
  lcg:=leadingCoefficient g
  testdeg:NNI
  lcd:R:=gcd(lcf,lcg)
  prime:=pMod
  bound:=mbound(f,g)
  while zero? (lcd rem prime) repeat
    prime := nextMod(prime,bound)
  soFar:=gcd(reduce(f,prime),reduce(g,prime))::BP
  testdeg:=degree soFar
  zero? testdeg => return 1$BP
  ldp:FP:=
    ((lcdp:=leadingCoefficient(soFar::BP)) = 1) =>
                                  reduce(lcd::BP,prime)
    reduce((modInverse(lcdp,prime)*lcd)::BP,prime)
  soFar:=reduce(ldp::BP *soFar,prime)::BP
  soFarModulus:=prime
  -- choose the prime
  while true repeat
    prime := nextMod(prime,bound)
    lcd rem prime =0 => "next prime"
    fp:=reduce(f,prime)
    gp:=reduce(g,prime)
    dp:=gcd(fp,gp)
    dgp :=euclideanSize dp
    if dgp =0 then return 1$BP
    if dgp=dg and ^(f exquo g case "failed") then return g
    if dgp=df and ^(g exquo f case "failed") then return f
    dgp > testdeg => "next prime"
    ldp:FP:=
      ((lcdp:=leadingCoefficient(dp::BP)) = 1) =>
                                  reduce(lcd::BP,prime)
      reduce((modInverse(lcdp,prime)*lcd)::BP,prime)
    dp:=ldp *dp
    dgp=testdeg  =>
       correction:=reduce(dp::BP-soFar,prime)::BP
       zero? correction =>
          ans:=reduce(lcd::BP*soFar,soFarModulus)::BP
          cont:=content ans
          ans:=(ans exquo cont)::BP
          test(f,g,ans) => return ans
          soFarModulus:=soFarModulus*prime
       correctionFactor:=modInverse(soFarModulus rem prime,prime)
```

```
                          -- the initial rem is just for efficiency
          soFar:=soFar+soFarModulus*(correctionFactor*correction)
          soFarModulus:=soFarModulus*prime
          soFar:=reduce(soFar,soFarModulus)::BP
        dgp<testdeg =>
          soFarModulus:=prime
          soFar:=dp::BP
          testdeg:=dgp
        if ^zeroChar and euclideanSize(prime)>1 then
           result:=dp::BP
           test(f,g,result) => return result
        -- this is based on the assumption that the caller of this package,
        -- in non-zero characteristic, will use primes of the form
        -- x-alpha as long as possible, but, if these are exhausted,
        -- will switch to a prime of degree larger than the answer
        -- so the result can be used directly.

    merge(p:R,q:R):Union(R,"failed") ==
         p = q => p
         p = 0 => q
         q = 0 => p
         "failed"

    modInverse(c:R,p:R):R ==
        (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

    exactquo(u:BP,v:BP,p:R):Union(BP,"failed") ==
        invlcv:=modInverse(leadingCoefficient v,p)
        r:=monicDivide(u,reduction(invlcv*v,p))
        reduction(r.remainder,p) ^=0 => "failed"
        reduction(invlcv*r.quotient,p)


    -- compute the height of a polynomial --
    height(f:BP):NNI ==
      degf:=degree f
      "max"/[euclideanSize cc for cc in coefficients f]

    -- compute the bound
    mbound(f:BP,g:BP):NNI ==
      hf:=height f
      hg:=height g
      2*min(hf,hg)


            ———————-

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\section{package FOMOGCD ForModularGcd}
-- ForModularGcd(R,BP) : C == T
```

```
--  where
--   R          :   EuclideanDomain  -- characteristic 0
--   BP         :   UnivariatePolynomialCategory(R)
--
--   C == with
--     nextMod :  (R,NNI) -> R
--
--   T == add
--     nextMod(val:R,bound:NNI) : R  ==
--       ival:Z:= val pretend Z
--       (nextPrime(ival)$IntegerPrimesPackage(Z))::R
--
-- ForTwoGcd(F) : C == T
--  where
--   F          :   Join(Finite,Field)
--   SUP       ==>  SparseUnivariatePolynomial
--   R         ==>  SUP F
--   P         ==>  SUP R
--   UPCF2     ==>  UnivariatePolynomialCategoryFunctions2
--
--   C == with
--     nextMod :  (R,NNI) -> R
--
--   T == add
--     nextMod(val:R,bound:NNI) : R ==
--       ris:R:= nextItem(val) :: R
--       euclideanSize ris < 2 => ris
--       generateIrredPoly(
--             (bound+1)::PositiveInteger)$IrredPolyOverFiniteField(F)
--
--
-- ModularGcd(R,BP) == T
--  where
--   R  : EuclideanDomain -- characteristic 0
--   BP : UnivariatePolynomialCategory(R)
--   T ==> InnerModularGcd(R,BP,67108859::R,nextMod$ForModularGcd(R,BP))
--
-- TwoGcd(F) : C == T
--  where
--   F          :   Join(Finite,Field)
--   SUP       ==>  SparseUnivariatePolynomial
--   R         ==>  SUP F
--   P         ==>  SUP R
--
--   T ==> InnerModularGcd(R,P,nextMod(monomial(1,1)$R)$ForTwoGcd(F),
--                         nextMod$ForTwoGcd(F))
```

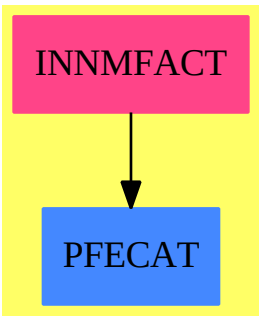
— INMODGCD.dotabb —

```
"INMODGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INMODGCD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INMODGCD" -> "PFECAT"
```

———-

10.14 package INNMFACT InnerMultFact

10.14.1 InnerMultFact (INNMFACT)



Exports:
factor

— package INNMFACT InnerMultFact —

```
)abbrev package INNMFACT InnerMultFact
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Additional Comments: JHD Aug 1997
++ Basic Functions:
++ Related Constructors: MultivariateFactorize, AlgebraicMultFact
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is an inner package for factoring multivariate polynomials
++ over various coefficient domains in characteristic 0.
++ The univariate factor operation is passed as a parameter.
++ Multivariate hensel lifting is used to lift the univariate
++ factorization

-- Both exposed functions call mFactor. This deals with issues such as
```

```
-- monomial factors, contents, square-freeness etc., then calls intfact.
-- This uses intChoose to find a "good" evaluation and factorise the
-- corresponding univariate, and then uses MultivariateLifting to find
-- the multivariate factors.

InnerMultFact(OV,E,R,P) : C == T
 where
  R          :   Join(EuclideanDomain, CharacteristicZero)
                     -- with factor on R[x]
  OV         :   OrderedSet
  E          :   OrderedAbelianMonoidSup
  P          :   PolynomialCategory(R,E,OV)
  BP         ==> SparseUnivariatePolynomial R
  UFactor    ==> (BP -> Factored BP)
  Z            ==> Integer
  MParFact     ==> Record(irr:P,pow:Z)
  USP          ==> SparseUnivariatePolynomial P
  SUParFact    ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact   ==> Record(contp:R,factors:List MParFact)

               --  contp   =  content,
               --  factors =  List of irreducible factors with exponent
  L          ==> List

  C == with
    factor      :      (P,UFactor)    ->  Factored P
      ++ factor(p,ufact) factors the multivariate polynomial p
      ++ by specializing variables and calling the univariate
      ++ factorizer ufact.
    factor      :      (USP,UFactor)    ->  Factored USP
      ++ factor(p,ufact) factors the multivariate polynomial p
      ++ by specializing variables and calling the univariate
      ++ factorizer ufact. p is represented as a univariate
      ++ polynomial with multivariate coefficients.

  T == add

    NNI       ==> NonNegativeInteger

    LeadFact  ==> Record(polfac:L P,correct:R,corrfact:L BP)
    ContPrim  ==> Record(cont:P,prim:P)
    ParFact   ==> Record(irr:BP,pow:Z)
    FinalFact ==> Record(contp:R,factors:L ParFact)
    NewOrd    ==> Record(npol:USP,nvar:L OV,newdeg:L NNI)
    pmod:R   :=  (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

    import GenExEuclid(R,BP)
    import MultivariateLifting(E,OV,R,P)
    import FactoringUtilities(E,OV,R,P)
```

```
import LeadingCoefDetermination(OV,E,R,P)
Valuf ==> Record(inval:L L R,unvfact:L BP,lu:R,complead:L R)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2

              ----  Local Functions  ----
mFactor   :            (P,UFactor)           ->  MFinalFact
supFactor :           (USP,UFactor)          ->  SUPFinalFact
mfconst   :      (USP,L OV,L NNI,UFactor)    -> L USP
mfpol     :      (USP,L OV,L NNI,UFactor)    -> L USP
monicMfpol:      (USP,L OV,L NNI,UFactor)    -> L USP
varChoose :           (P,L OV,L NNI)         -> NewOrd
simplify  :       (P,L OV,L NNI,UFactor)     -> MFinalFact
intChoose :  (USP,L OV,R,L P,L L R,UFactor)  -> Union(Valuf,"failed")
intfact   : (USP,L OV,L NNI,MFinalFact,L L R,UFactor) -> L USP
pretest   :         (P,NNI,L OV,L R)         -> FinalFact
checkzero :            (USP,BP)              -> Boolean
localNorm :               L BP               -> Z

convertPUP(lfg:MFinalFact): SUPFinalFact ==
  [lfg.contp,[[lff.irr ::USP,lff.pow]$SUParFact
               for lff in lfg.factors]]$SUPFinalFact

-- intermediate routine if an SUP was passed in.
supFactor(um:USP,ufactor:UFactor) : SUPFinalFact ==
  ground?(um) => convertPUP(mFactor(ground um,ufactor))
  lvar:L OV:= "setUnion"/[variables cf for cf in coefficients um]
  empty? lvar => -- the polynomial is univariate
    umv:= map(ground,um)$UPCF2(P,USP,R,BP)
    lfact:=ufactor umv
    [retract unit lfact,[[map(coerce,ff.factor)$UPCF2(R,BP,P,USP),
       ff.exponent] for ff in factors lfact]]$SUPFinalFact
  lcont:P
  lf:L USP
  flead : SUPFinalFact:=[0,empty()]$SUPFinalFact
  factorlist:L SUParFact :=empty()

  mdeg :=minimumDegree um     ---- is the Mindeg > 0? ----
  if mdeg>0 then
    f1:USP:=monomial(1,mdeg)
    um:=(um exquo f1)::USP
    factorlist:=cons([monomial(1,1),mdeg],factorlist)
    if degree um=0 then return
      lfg:=convertPUP mFactor(ground um, ufactor)
      [lfg.contp,append(factorlist,lfg.factors)]
  uum:=unitNormal um
  um :=uum.canonical
  sqfacs := squareFree(um)$MultivariateSquareFree(E,OV,R,P)
  lcont :=  ground(uum.unit * unit sqfacs)
                               ----  Factorize the content  ----
  flead:=convertPUP mFactor(lcont,ufactor)
```

```
  factorlist:=append(flead.factors,factorlist)
                          ----  Make the polynomial square-free  ----
  sqqfact:=factors sqfacs
                   ---  Factorize the primitive square-free terms ---
  for fact in sqqfact repeat
    ffactor:USP:=fact.factor
    ffexp:=fact.exponent
    zero? degree ffactor =>
      lfg:=mFactor(ground ffactor,ufactor)
      lcont:=lfg.contp * lcont
      factorlist := append(factorlist,
         [[lff.irr ::USP,lff.pow * ffexp]$SUParFact
                  for lff in lfg.factors])
    coefs := coefficients ffactor
    ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]
    lf :=
      ground?(leadingCoefficient ffactor) =>
         mfconst(ffactor,lvar,ldeg,ufactor)
      mfpol(ffactor,lvar,ldeg,ufactor)
    auxfl:=[[lfp,ffexp]$SUParFact  for lfp in lf]
    factorlist:=append(factorlist,auxfl)
  lcfacs := */[leadingCoefficient leadingCoefficient(f.irr)**((f.pow)::NNI)
                      for f in factorlist]
  [(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R,
                factorlist]$SUPFinalFact

factor(um:USP,ufactor:UFactor):Factored USP ==
  flist := supFactor(um,ufactor)
  (flist.contp):: P :: USP *
    (*/[primeFactor(u.irr,u.pow) for u in flist.factors])

checkzero(u:USP,um:BP) : Boolean ==
  u=0 => um =0
  um = 0 => false
  degree u = degree um => checkzero(reductum u, reductum um)
  false
          ---  Choose the variable of less degree  ---
varChoose(m:P,lvar:L OV,ldeg:L NNI) : NewOrd ==
  k:="min"/[d for d in ldeg]
  k=degree(m,first lvar) =>
                         [univariate(m,first lvar),lvar,ldeg]$NewOrd
  i:=position(k,ldeg)
  x:OV:=lvar.i
  ldeg:=cons(k,delete(ldeg,i))
  lvar:=cons(x,delete(lvar,i))
  [univariate(m,x),lvar,ldeg]$NewOrd

localNorm(lum: L BP): Z ==
  R is AlgebraicNumber =>
    "max"/[numberOfMonomials ff for ff in lum]
```

```
    "max"/[+/[euclideanSize cc for i in 0..degree ff|
              (cc:= coefficient(ff,i))^=0] for ff in lum]

        ---  Choose the integer to reduce to univariate case  ---
  intChoose(um:USP,lvar:L OV,clc:R,plist:L P,ltry:L L R,
                               ufactor:UFactor) : Union(Valuf,"failed") ==
    -- declarations
    degum:NNI := degree um
    nvar1:=#lvar
    range:NNI:=5
    unifact:L BP
    ctf1 : R := 1
    testp:Boolean :=             -- polynomial leading coefficient
      empty? plist => false
      true
    leadcomp,leadcomp1 : L R
    leadcomp:=leadcomp1:=empty()
    nfatt:NNI := degum+1
    lffc:R:=1
    lffc1:=lffc
    newunifact : L BP:=empty()
    leadtest:=true --- the lc test with polCase has to be performed
    int:L R:=empty()

  --  New sets of integers are chosen to reduce the multivariate problem to
  --  a univariate one, until we find twice the
  --  same (and minimal) number of "univariate" factors:
  --  the set smaller in modulo is chosen.
  --  Note that there is no guarantee that this is the truth:
  --  merely the closest approximation we have found!

    while true repeat
     testp and #ltry>10 => return "failed"
     lval := [ ran(range) for i in 1..nvar1]
     member?(lval,ltry) => range:=2*range
     ltry := cons(lval,ltry)
     leadcomp1:=[retract eval(pol,lvar,lval) for pol in plist]
     testp and or/[unit? epl for epl in leadcomp1] => range:=2*range
     newm:BP:=completeEval(um,lvar,lval)
     degum ^= degree newm or minimumDegree newm ^=0 => range:=2*range
     lffc1:=content newm
     newm:=(newm exquo lffc1)::BP
     testp and leadtest and ^ polCase(lffc1*clc,#plist,leadcomp1)
                           => range:=2*range
     degree(gcd [newm,differentiate(newm)])^=0 => range:=2*range
     luniv:=ufactor(newm)
     lunivf:= factors luniv
     lffc1:R:=retract(unit luniv)@R * lffc1
     nf:= #lunivf
```

```
    nf=0 or nf>nfatt => "next values"      ---  pretest failed ---

                    --- the univariate polynomial is irreducible ---
    if nf=1 then leave (unifact:=[newm])

--  the new integer give the same number of factors
    nfatt = nf =>
    -- if this is the first univariate factorization with polCase=true
    -- or if the last factorization has smaller norm and satisfies
    -- polCase
      if leadtest or
        ((localNorm unifact > localNorm [ff.factor for ff in lunivf])
          and (^testp or polCase(lffc1*clc,#plist,leadcomp1))) then
             unifact:=[uf.factor for uf in lunivf]
             int:=lval
             lffc:=lffc1
             if testp then leadcomp:=leadcomp1
      leave "foundit"

--  the first univariate factorization, inizialize
    nfatt > degum =>
      unifact:=[uf.factor for uf in lunivf]
      lffc:=lffc1
      if testp then leadcomp:=leadcomp1
      int:=lval
      leadtest := false
      nfatt := nf

    nfatt>nf =>  -- for the previous values there were more factors
      if testp then leadtest:=^polCase(lffc*clc,#plist,leadcomp)
      else leadtest:= false
      -- if polCase=true we can consider the univariate decomposition
      if ^leadtest then
        unifact:=[uf.factor for uf in lunivf]
        lffc:=lffc1
        if testp then leadcomp:=leadcomp1
        int:=lval
      nfatt := nf
   [cons(int,ltry),unifact,lffc,leadcomp]$Valuf


              ----  The polynomial has mindeg>0   ----

 simplify(m:P,lvar:L OV,lmdeg:L NNI,ufactor:UFactor):MFinalFact ==
   factorlist:L MParFact:=[]
   pol1:P:= 1$P
   for x in lvar repeat
     i := lmdeg.(position(x,lvar))
     i=0 => "next value"
```

```
        pol1:=pol1*monomial(1$P,x,i)
        factorlist:=cons([x::P,i]$MParFact,factorlist)
      m := (m exquo pol1)::P
      ground? m => [retract m,factorlist]$MFinalFact
      flead:=mFactor(m,ufactor)
      flead.factors:=append(factorlist,flead.factors)
      flead

    -- This is the key internal function
    -- We now know that the polynomial is square-free etc.,
    -- We use intChoose to find a set of integer values to reduce the
    -- problem to univariate (and for efficiency, intChoose returns
    -- the univariate factors).
    -- In the case of a polynomial leading coefficient, we check that this
    -- is consistent with leading coefficient determination (else try again)
    -- We then lift the univariate factors to multivariate factors, and
    -- return the result
    intfact(um:USP,lvar: L OV,ldeg:L NNI,tleadpol:MFinalFact,
                                  ltry:L L R,ufactor:UFactor) :  L USP ==
      polcase:Boolean:=(not empty? tleadpol.factors)
      vfchoo:Valuf:=
        polcase =>
          leadpol:L P:=[ff.irr for ff in tleadpol.factors]
          check:=intChoose(um,lvar,tleadpol.contp,leadpol,ltry,ufactor)
          check case "failed" => return monicMfpol(um,lvar,ldeg,ufactor)
          check::Valuf
        intChoose(um,lvar,1,empty(),empty(),ufactor)::Valuf
      unifact:List BP := vfchoo.unvfact
      nfact:NNI := #unifact
      nfact=1 => [um]
      ltry:L L R:= vfchoo.inval
      lval:L R:=first ltry
      dd:= vfchoo.lu
      leadval:L R:=empty()
      lpol:List P:=empty()
      if polcase then
        leadval := vfchoo.complead
        distf := distFact(vfchoo.lu,unifact,tleadpol,leadval,lvar,lval)
        distf case "failed" =>
             return intfact(um,lvar,ldeg,tleadpol,ltry,ufactor)
        dist := distf :: LeadFact
          -- check the factorization of leading coefficient
        lpol:= dist.polfac
        dd := dist.correct
        unifact:=dist.corrfact
      if dd^=1 then
--        if polcase then lpol := [unitCanonical lp for lp in lpol]
--        dd:=unitCanonical(dd)
        unifact := [dd * unif for unif in unifact]
        umd := unitNormal(dd).unit * ((dd**(nfact-1)::NNI)::P)*um
```

```
    else umd := um
    (ffin:=lifting(umd,lvar,unifact,lval,lpol,ldeg,pmod))
      case "failed" => intfact(um,lvar,ldeg,tleadpol,ltry,ufactor)
    factfin: L USP:=ffin :: L USP
    if dd^=1 then
      factfin:=[primitivePart ff for ff in factfin]
    factfin

              ----  m square-free,primitive,lc constant  ----
  mfconst(um:USP,lvar:L OV,ldeg:L NNI,ufactor:UFactor):L USP ==
    factfin:L USP:=empty()
    empty? lvar =>
      lum:=factors ufactor(map(ground,um)$UPCF2(P,USP,R,BP))
      [map(coerce,uf.factor)$UPCF2(R,BP,P,USP) for uf in lum]
    intfact(um,lvar,ldeg,[0,empty()]$MFinalFact,empty(),ufactor)

  monicize(um:USP,c:P):USP ==
    n:=degree(um)
    ans:USP := monomial(1,n)
    n:=(n-1)::NonNegativeInteger
    prod:P:=1
    while (um:=reductum(um)) ^= 0 repeat
      i := degree um
      lc := leadingCoefficient um
      prod := prod * c ** (n-(n:=i))::NonNegativeInteger
      ans := ans + monomial(prod*lc, i)
    ans

  unmonicize(m:USP,c:P):USP == primitivePart m(monomial(c,1))

            --- m is square-free,primitive,lc is a polynomial  ---
  monicMfpol(um:USP,lvar:L OV,ldeg:L NNI,ufactor:UFactor):L USP ==
    l := leadingCoefficient um
    monpol := monicize(um,l)
    nldeg := degree(monpol,lvar)
    map((z1:USP):USP +-> unmonicize(z1,l),
              mfconst(monpol,lvar,nldeg,ufactor))

  mfpol(um:USP,lvar:L OV,ldeg:L NNI,ufactor:UFactor):L USP ==
    R has Field =>
      monicMfpol(um,lvar,ldeg,ufactor)
    tleadpol:=mFactor(leadingCoefficient um,ufactor)
    intfact(um,lvar,ldeg,tleadpol,[],ufactor)

  mFactor(m:P,ufactor:UFactor) : MFinalFact ==
    ground?(m) => [retract(m),empty()]$MFinalFact
    lvar:L OV:= variables m
    lcont:P
    lf:L USP
    flead : MFinalFact:=[0,empty()]$MFinalFact
```

```
factorlist:L MParFact :=empty()

lmdeg :=minimumDegree(m,lvar)     ---- is the Mindeg > 0? ----
or/[n>0 for n in lmdeg] => simplify(m,lvar,lmdeg,ufactor)

sqfacs := squareFree m
lcont := unit sqfacs

                            ----  Factorize the content  ----
if ground? lcont then flead.contp:=retract lcont
else flead:=mFactor(lcont,ufactor)
factorlist:=flead.factors



                        ----  Make the polynomial square-free  ----
sqqfact:=factors sqfacs

                 ---  Factorize the primitive square-free terms ---
for fact in sqqfact repeat
  ffactor:P:=fact.factor
  ffexp := fact.exponent
  lvar := variables ffactor
  x:OV :=lvar.first
  ldeg:=degree(ffactor,lvar)
       ---  Is the polynomial linear in one of the variables ? ---
  member?(1,ldeg) =>
    x:OV:=lvar.position(1,ldeg)
    lcont:= gcd coefficients(univariate(ffactor,x))
    ffactor:=(ffactor exquo lcont)::P
    factorlist:=cons([ffactor,ffexp]$MParFact,factorlist)
    for lcterm in mFactor(lcont,ufactor).factors repeat
     factorlist:=cons([lcterm.irr,lcterm.pow * ffexp], factorlist)

  varch:=varChoose(ffactor,lvar,ldeg)
  um:=varch.npol

  x:=lvar.first
  ldeg:=ldeg.rest
  lvar := lvar.rest
  if varch.nvar.first ^= x then
    lvar:= varch.nvar
    x := lvar.first
    lvar := lvar.rest
  pc:= gcd coefficients um
  if pc^=1 then
      um:=(um exquo pc)::USP
      ffactor:=multivariate(um,x)
      for lcterm in mFactor(pc,ufactor).factors repeat
        factorlist:=cons([lcterm.irr,lcterm.pow*ffexp],factorlist)
```

```
        ldeg:=degree(ffactor,lvar)
        um := unitCanonical um
        if ground?(leadingCoefficient um) then
           lf:= mfconst(um,lvar,ldeg,ufactor)
        else lf:=mfpol(um,lvar,ldeg,ufactor)
        auxfl:=[[unitCanonical multivariate(lfp,x),ffexp]$MParFact  for lfp in lf]
        factorlist:=append(factorlist,auxfl)
      lcfacs := */[leadingCoefficient(f.irr)**((f.pow)::NNI) for f in factorlist]
      [(leadingCoefficient(m) exquo lcfacs):: R,factorlist]$MFinalFact

    factor(m:P,ufactor:UFactor):Factored P ==
      flist := mFactor(m,ufactor)
      (flist.contp):: P *
        (*/[primeFactor(u.irr,u.pow) for u in flist.factors])
```

———-

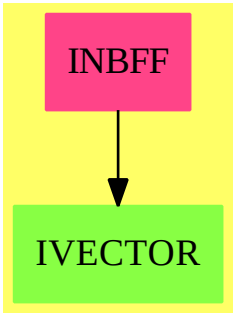
— **INNMFACT.dotabb** —

```
"INNMFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INNMFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INNMFACT" -> "PFECAT"
```

———-

10.15 package INBFF InnerNormalBasisFieldFunctions

10.15.1 InnerNormalBasisFieldFunctions (INBFF)



Exports:

basis	dAndcExp	expPot	index	inv
lookup	minimalPolynomial	norm	normalElement	normal?
pol	qPot	random	repSq	setFieldInfo
trace	xn	?*?	?**?	?/?

— package INBFF InnerNormalBasisFieldFunctions —

```
)abbrev package INBFF InnerNormalBasisFieldFunctions
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ Date Last Updated: 31 March 1991
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: finite field, normal basis
++ References:
++  R.Lidl, H.Niederreiter: Finite Field, Encyclopedia of Mathematics and
++   Its Applications, Vol. 20, Cambridge Univ. Press, 1983, ISBN 0 521 30240 4
++  D.R.Stinson: Some observations on parallel Algorithms for fast
++   exponentiation in GF(2^n), Siam J. Comp., Vol.19, No.4, pp.711-717,
++   August 1990
++  T.Itoh, S.Tsujii: A fast algorithm for computing multiplicative inverses
++   in GF(2^m) using normal bases, Inf. and Comp. 78, pp.171-177, 1988
++  J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++   AXIOM Technical Report Series, ATR/5 NP2522.
++ Description:
++ InnerNormalBasisFieldFunctions(GF) (unexposed):
++ This package has functions used by
++ every normal basis finite field extension domain.

InnerNormalBasisFieldFunctions(GF): Exports == Implementation where
  GF    : FiniteFieldCategory       -- the ground field

  PI   ==> PositiveInteger
  NNI  ==> NonNegativeInteger
  I    ==> Integer
  SI   ==> SingleInteger
  SUP  ==> SparseUnivariatePolynomial
  VGF  ==> Vector GF
  M    ==> Matrix
  V    ==> Vector
  L    ==> List
  OUT  ==> OutputForm
  TERM ==> Record(value:GF,index:SI)
  MM   ==> ModMonic(GF,SUP GF)

  Exports ==> with

      setFieldInfo: (V L TERM,GF) -> Void
```

```
        ++ setFieldInfo(m,p) initializes the field arithmetic, where m is
        ++ the multiplication table and p is the respective normal element
        ++ of the ground field GF.
      random    : PI           -> VGF
        ++ random(n) creates a vector over the ground field with random entries.
      index     : (PI,PI)      -> VGF
        ++ index(n,m) is a index function for vectors of length n over
        ++ the ground field.
      pol       : VGF          -> SUP GF
        ++ pol(v) turns the vector \spad{[v0,...,vn]} into the polynomial
        ++ \spad{v0+v1*x+ ... + vn*x**n}.
      xn         : NNI          -> SUP GF
        ++ xn(n) returns the polynomial \spad{x**n-1}.
      dAndcExp  : (VGF,NNI,SI) -> VGF
        ++ dAndcExp(v,n,k) computes \spad{v**e} interpreting v as an element of
        ++ normal basis field. A divide and conquer algorithm similar to the
        ++ one from D.R.Stinson,
        ++ "Some observations on parallel Algorithms for fast exponentiation in
        ++ GF(2^n)", Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990
        ++ is used. Argument k is a parameter of this algorithm.
      repSq     : (VGF,NNI)    -> VGF
        ++ repSq(v,e) computes \spad{v**e} by repeated squaring,
        ++ interpreting v as an element of a normal basis field.
      expPot    : (VGF,SI,SI)  -> VGF
        ++ expPot(v,e,d) returns the sum from \spad{i = 0} to
        ++ \spad{e - 1} of \spad{v**(q**i*d)}, interpreting
        ++ v as an element of a normal basis field and where q is
        ++ the size of the ground field.
        ++ Note that for a description of the algorithm, see
        ++ T.Itoh and S.Tsujii,
        ++ "A fast algorithm for computing multiplicative inverses in GF(2^m)
        ++ using normal bases",
        ++ Information and Computation 78, pp.171-177, 1988.
      qPot      : (VGF,I)      -> VGF
        ++ qPot(v,e) computes \spad{v**(q**e)}, interpreting v as an element of
        ++ normal basis field, q the size of the ground field.
        ++ This is done by a cyclic e-shift of the vector v.

-- the semantic of the following functions is obvious from the finite field
-- context, for description see category FAXF
      "**"      :(VGF,I)       -> VGF
        ++ x**n \undocumented{}
        ++ See \axiomFunFrom{**}{DivisionRing}
      "*"       :(VGF,VGF)     -> VGF
        ++ x*y \undocumented{}
        ++ See \axiomFunFrom{*}{SemiGroup}
      "/"       :(VGF,VGF)     -> VGF
        ++ x/y \undocumented{}
        ++ See \axiomFunFrom{/}{Field}
      norm      :(VGF,PI)      -> VGF
```

```
        ++ norm(x,n) \undocumented{}
        ++ See \axiomFunFrom{norm}{FiniteAlgebraicExtensionField}
      trace     :(VGF,PI)      -> VGF
        ++ trace(x,n) \undocumented{}
        ++ See \axiomFunFrom{trace}{FiniteAlgebraicExtensionField}
      inv       : VGF          -> VGF
        ++ inv x \undocumented{}
        ++ See \axiomFunFrom{inv}{DivisionRing}
      lookup    : VGF          -> PI
        ++ lookup(x) \undocumented{}
        ++ See \axiomFunFrom{lookup}{Finite}
      normal?   : VGF          -> Boolean
        ++ normal?(x) \undocumented{}
        ++ See \axiomFunFrom{normal?}{FiniteAlgebraicExtensionField}
      basis     : PI           -> V VGF
        ++ basis(n) \undocumented{}
        ++ See \axiomFunFrom{basis}{FiniteAlgebraicExtensionField}
      normalElement:PI         -> VGF
        ++ normalElement(n) \undocumented{}
        ++ See \axiomFunFrom{normalElement}{FiniteAlgebraicExtensionField}
      minimalPolynomial: VGF   -> SUP GF
        ++ minimalPolynomial(x) \undocumented{}
        ++ See \axiomFunFrom{minimalPolynomial}{FiniteAlgebraicExtensionField}

  Implementation ==> add

-- global variables ================================================

    sizeGF:NNI:=size()$GF
    -- the size of the ground field

    multTable:V L TERM:=new(1,nil()$(L TERM))$(V L TERM)
    -- global variable containing the multiplication table

    trGen:GF:=1$GF
    -- controls the imbedding of the ground field

    logq:List SI:=[0,10::SI,16::SI,20::SI,23::SI,0,28::SI,_
                             30::SI,32::SI,0,35::SI]
    -- logq.i is about 10*log2(i) for the values <12 which
    -- can match sizeGF. It's used by "**"

    expTable:L L SI:=[[],_
        [4::SI,12::SI,48::SI,160::SI,480::SI,0],_
        [8::SI,72::SI,432::SI,0],_
        [18::SI,216::SI,0],_
        [32::SI,480::SI,0],[],_
        [72::SI,0],[98::SI,0],[128::SI,0],[],[200::SI,0]]
    -- expT is used by "**" to optimize the parameter k
    -- before calling dAndcExp(..,..,k)
```

```
-- functions ===========================================================

--  computes a**(-1) = a**((q**extDeg)-2)
--  see reference of function expPot
    inv(a) ==
      b:VGF:=qPot(expPot(a,(#a-1)::NNI::SI,1::SI)$$,1)$$
      erg:VGF:=inv((a *$$ b).1 *$GF trGen)$GF *$VGF b

-- "**" decides which exponentiation algorithm will be used, in order to
-- get the fastest computation. If dAndcExp is used, it chooses the
-- optimal parameter k for that algorithm.
    a ** ex  ==
      e:NNI:=positiveRemainder(ex,sizeGF**((#a)::PI)-1)$I :: NNI
      zero?(e)$NNI => new(#a,trGen)$VGF
--      one?(e)$NNI  => copy(a)$VGF
      (e = 1)$NNI  => copy(a)$VGF
--    inGroundField?(a) => new(#a,((a.1*trGen) **$GF e))$VGF
      e1:SI:=(length(e)$I)::SI
      sizeGF >$I 11 =>
        q1:SI:=(length(sizeGF)$I)::SI
        logqe:SI:=(e1 quo$SI q1) +$SI 1$SI
        10::SI * (logqe + sizeGF-2) > 15::SI * e1 =>
--        print("repeatedSquaring"::OUT)
          repSq(a,e)
--      print("divAndConquer(a,e,1)"::OUT)
        dAndcExp(a,e,1)
      logqe:SI:=((10::SI *$SI e1) quo$SI (logq.sizeGF)) +$SI 1$SI
      k:SI:=1$SI
      expT:List SI:=expTable.sizeGF
      while (logqe >= expT.k) and not zero? expT.k repeat k:=k +$SI 1$SI
      mult:I:=(sizeGF-1) *$I sizeGF **$I ((k-1)pretend NNI) +$I_
              ((logqe +$SI k -$SI 1$SI) quo$SI k)::I -$I 2
      (10*mult) >= (15 * (e1::I)) =>
--      print("repeatedSquaring(a,e)"::OUT)
        repSq(a,e)
--    print(hconcat(["divAndConquer(a,e,"::OUT,k::OUT,")"::OUT])$OUT)
      dAndcExp(a,e,k)

-- computes a**e by repeated squaring
    repSq(b,e) ==
      a:=copy(b)$VGF
--      one? e => a
      (e = 1) => a
      odd?(e)$I => a * repSq(a*a,(e quo 2) pretend NNI)
      repSq(a*a,(e quo 2) pretend NNI)

-- computes a**e using the divide and conquer algorithm similar to the
-- one from D.R.Stinson,
-- "Some observations on parallel Algorithms for fast exponentiation in
```

```
-- GF(2^n)", Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990
    dAndcExp(a,e,k) ==
      plist:List VGF:=[copy(a)$VGF]
      qk:I:=sizeGF**(k pretend NNI)
      for j in 2..(qk-1) repeat
        if positiveRemainder(j,sizeGF)=0 then b:=qPot(plist.(j quo sizeGF),1)$$
                            else b:=a *$$ last(plist)$(List VGF)
        plist:=concat(plist,b)
      l:List NNI:=nil()
      ex:I:=e
      while not(ex = 0) repeat
        l:=concat(l,positiveRemainder(ex,qk) pretend NNI)
        ex:=ex quo qk
      if first(l)=0 then erg:VGF:=new(#a,trGen)$VGF
                    else erg:VGF:=plist.(first(l))
      i:SI:=k
      for j in rest(l) repeat
        if j^=0 then erg:=erg *$$ qPot(plist.j,i)$$
        i:=i+k
      erg

    a * b ==
      e:SI:=(#a)::SI
      erg:=zero(#a)$VGF
      for t in multTable.1 repeat
        for j in 1..e repeat
          y:=t.value  -- didn't work without defining x and y
          x:=t.index
          k:SI:=addmod(x,j::SI,e)$SI +$SI 1$SI
          erg.k:=erg.k +$GF a.j *$GF b.j *$GF y
      for i in 1..e-1 repeat
        for j in i+1..e repeat
          for t in multTable.(j-i+1) repeat
            y:=t.value   -- didn't work without defining x and y
            x:=t.index
            k:SI:=addmod(x,i::SI,e)$SI +$SI 1$SI
            erg.k:GF:=erg.k +$GF (a.i *$GF b.j +$GF a.j *$GF b.i) *$GF y
      erg

    lookup(x) ==
      erg:I:=0
      for j in (#x)..1 by -1 repeat
        erg:=(erg * sizeGF) + (lookup(x.j)$GF rem sizeGF)
      erg=0 => (sizeGF**(#x)) :: PI
      erg :: PI

--  computes the norm of a over GF**d, d must devide extdeg
--  see reference of function expPot below
    norm(a,d) ==
      dSI:=d::SI
```

```
      r:=divide((#a)::SI,dSI)
      not(r.remainder = 0) => error "norm: 2.arg must divide extdeg"
      expPot(a,r.quotient,dSI)$$

--  computes expPot(a,e,d) = sum form i=0 to e-1 over a**(q**id))
--  see T.Itoh and S.Tsujii,
--  "A fast algorithm for computing multiplicative inverses in GF(2^m)
--   using normal bases",
--  Information and Computation 78, pp.171-177, 1988
    expPot(a,e,d) ==
      deg:SI:=(#a)::SI
      e=1 => copy(a)$VGF
      k2:SI:=d
      y:=copy(a)
      if bit?(e,0) then
        erg:=copy(y)
        qpot:SI:=k2
      else
        erg:=new(#a,inv(trGen)$GF)$VGF
        qpot:SI:=0
      for k in 1..length(e) repeat
        y:= y *$$ qPot(y,k2)
        k2:=addmod(k2,k2,deg)$SI
        if bit?(e,k) then
          erg:=erg *$$ qPot(y,qpot)
          qpot:=addmod(qpot,k2,deg)$SI
      erg

-- computes qPot(a,n) = a**(q**n), q=size of GF
    qPot(e,n) ==
      ei:=(#e)::SI
      m:SI:= positiveRemainder(n::SI,ei)$SI
      zero?(m) => e
      e1:=zero(#e)$VGF
      for i in m+1..ei repeat e1.i:=e.(i-m)
      for i in 1..m    repeat e1.i:=e.(ei+i-m)
      e1

    trace(a,d) ==
      dSI:=d::SI
      r:=divide((#a)::SI,dSI)$SI
      not(r.remainder = 0) => error "trace: 2.arg must divide extdeg"
      v:=copy(a.(1..dSI))$VGF
      sSI:SI:=r.quotient
      for i in 1..dSI repeat
        for j in 1..sSI-1 repeat
          v.i:=v.i+a.(i+j::SI*dSI)
      v

    random(n) ==
```

```
      v:=zero(n)$VGF
      for i in 1..n repeat v.i:=random()$GF
      v


    xn(m) == monomial(1,m)$(SUP GF) - 1$(SUP GF)

    normal?(x) ==
      gcd(xn(#x),pol(x))$(SUP GF) = 1 => true
      false

    x:VGF / y:VGF == x *$$ inv(y)$$


    setFieldInfo(m,n) ==
      multTable:=m
      trGen:=n
      void()$Void

    minimalPolynomial(x) ==
      dx:=#x
      y:=new(#x,inv(trGen)$GF)$VGF
      m:=zero(dx,dx+1)$(M GF)
      for i in 1..dx+1 repeat
        dy:=#y
        for j in 1..dy repeat
          for k in 0..((dx quo dy)-1) repeat
            qsetelt_!(m,j+k*dy,i,y.j)$(M GF)
        y:=y *$$ x
      v:=first nullSpace(m)$(M GF)
      pol(v)$$

    basis(n) ==
      bas:(V VGF):=new(n,zero(n)$VGF)$(V VGF)
      for i in 1..n repeat
        uniti:=zero(n)$VGF
        qsetelt_!(uniti,i,1$GF)$VGF
        qsetelt_!(bas,i,uniti)$(V VGF)
      bas

    normalElement(n) ==
       v:=zero(n)$VGF
       qsetelt_!(v,1,1$GF)
       v
--    normalElement(n) == index(n,1)$$

    index(degm,n) ==
      m:I:=n rem$I (sizeGF ** degm)
      erg:=zero(degm)$VGF
      for j in 1..degm repeat
```

```
        erg.j:=index((sizeGF+(m rem sizeGF)) pretend PI)$GF
        m:=m quo sizeGF
      erg

    pol(x) ==
      +/[monomial(x.i,(i-1)::NNI)$(SUP GF) for i in 1..(#x)::I]
```

———-

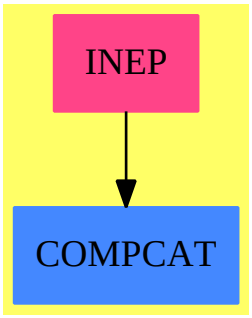
— **INBFF.dotabb** —

```
"INBFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INBFF"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"INBFF" -> "IVECTOR"
```

———-

10.16 package INEP InnerNumericEigenPackage

10.16.1 InnerNumericEigenPackage (INEP)



Exports:
charpol innerEigenvectors solve1

— **package INEP InnerNumericEigenPackage** —

```
)abbrev package INEP InnerNumericEigenPackage
++ Author:P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated:Spring 1991
++ Basic Functions:
++ Related Constructors: ModularField
```

```
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package is the inner package to be used by NumericRealEigenPackage
++ and NumericComplexEigenPackage for the computation of numeric
++ eigenvalues and eigenvectors.

InnerNumericEigenPackage(K,F,Par) : C == T
 where
   F    :   Field  -- this is the field where the answer will be
                   -- for dealing with the complex case
   K    :   Field  -- type of the  input
   Par  :   Join(Field,OrderedRing)  -- it will be NF or RN

   SE    ==> Symbol()
   RN    ==> Fraction Integer
   I     ==> Integer
   NF    ==> Float
   CF    ==> Complex Float
   GRN   ==> Complex RN
   GI    ==> Complex Integer
   PI    ==> PositiveInteger
   NNI   ==> NonNegativeInteger
   MRN   ==> Matrix RN

   MK          ==> Matrix K
   PK          ==> Polynomial K
   MF          ==> Matrix F
   SUK         ==> SparseUnivariatePolynomial K
   SUF         ==> SparseUnivariatePolynomial F
   SUP         ==> SparseUnivariatePolynomial
   MSUK        ==> Matrix SUK

   PEigenForm  ==> Record(algpol:SUK,almult:Integer,poleigen:List(MSUK))

   outForm     ==> Record(outval:F,outmult:Integer,outvect:List MF)

   IntForm     ==> Union(outForm,PEigenForm)
   UFactor     ==> (SUK -> Factored SUK)
   C == with

     charpol  :  MK   ->  SUK
       ++ charpol(m) computes the characteristic polynomial of a matrix
       ++ m with entries in K.
       ++ This function returns a polynomial
       ++ over K, while the general one (that is in EiegenPackage) returns
       ++ Fraction P K
```

```
    solve1   : (SUK, Par) -> List F
      ++ solve1(pol, eps) finds the roots of the univariate polynomial
      ++ polynomial pol to precision eps. If K is \spad{Fraction Integer}
      ++ then only the real roots are returned, if K is
      ++ \spad{Complex Fraction Integer} then all roots are found.

    innerEigenvectors    : (MK,Par,UFactor)   ->  List(outForm)
      ++ innerEigenvectors(m,eps,factor) computes explicitly
      ++ the eigenvalues and the correspondent eigenvectors
      ++ of the matrix m. The parameter eps determines the type of
      ++ the output, factor is the univariate factorizer to br used
      ++ to reduce the characteristic polynomial into irreducible factors.

  T == add

    numeric(r:K):F ==
      K is RN =>
        F is NF => convert(r)$RN
        F is RN    => r
        F is CF    => r :: RN :: CF
        F is GRN   => r::RN::GRN
      K is GRN =>
        F is GRN => r
        F is CF  => convert(convert r)
      error "unsupported coefficient type"

   ---- next functions neeeded for defining  ModularField ----

    monicize(f:SUK) : SUK ==
      (a:=leadingCoefficient f) =1 => f
      inv(a)*f

    reduction(u:SUK,p:SUK):SUK == u rem p

    merge(p:SUK,q:SUK):Union(SUK,"failed") ==
        p = q => p
        p = 0 => q
        q = 0 => p
        "failed"

    exactquo(u:SUK,v:SUK,p:SUK):Union(SUK,"failed") ==
       val:=extendedEuclidean(v,p,u)
       val case "failed" => "failed"
       val.coef1

        ----  eval a vector of F in a radical expression  ----
    evalvect(vect:MSUK,alg:F) : MF ==
      n:=nrows vect
      w:MF:=zero(n,1)$MF
      for i in 1..n repeat
```

```
    polf:=map(numeric,
      vect(i,1))$UnivariatePolynomialCategoryFunctions2(K,SUK,F,SUF)
    v:F:=elt(polf,alg)
    setelt(w,i,1,v)
  w

  ---- internal function for the computation of eigenvectors  ----
inteigen(A:MK,p:SUK,fact:UFactor) : List(IntForm) ==
  dimA:NNI:=  nrows A
  MM:=ModularField(SUK,SUK,reduction,merge,exactquo)
  AM:=Matrix(MM)
  lff:=factors fact(p)
  res: List IntForm  :=[]
  lr : List MF:=[]
  for ff in lff repeat
    pol:SUK:= ff.factor
    if (degree pol)=1 then
      alpha:K:=-coefficient(pol,0)/leadingCoefficient pol
      -- compute the eigenvectors, rational case
      B1:MK := zero(dimA,dimA)$MK
      for i in 1..dimA repeat
        for j in 1..dimA repeat B1(i,j):=A(i,j)
        B1(i,i):= B1(i,i) - alpha
      lr:=[]
      for vecr in nullSpace B1 repeat
        wf:MF:=zero(dimA,1)
        for i in 1..dimA repeat wf(i,1):=numeric vecr.i
        lr:=cons(wf,lr)
      res:=cons([numeric alpha,ff.exponent,lr]$outForm,res)
    else
      ppol:=monicize pol
      alg:MM:= reduce(monomial(1,1),ppol)
      B:AM:= zero(dimA,dimA)$AM
      for i in 1..dimA  repeat
        for j in 1..dimA repeat B(i,j):=reduce(A(i,j) ::SUK,ppol)
        B(i,i):=B(i,i) - alg
      sln2:=nullSpace B
      soln:List MSUK :=[]
      for vec in sln2 repeat
        wk:MSUK:=zero(dimA,1)
        for i in 1..dimA repeat wk(i,1):=(vec.i)::SUK
        soln:=cons(wk,soln)
      res:=cons([ff.factor,ff.exponent,soln]$PEigenForm,
                      res)
  res

if K is RN then
    solve1(up:SUK, eps:Par) : List(F) ==
      denom := "lcm"/[denom(c::RN) for c in coefficients up]
      up:=denom*up
```

```
        upi:=map(numer,up)_
          $UnivariatePolynomialCategoryFunctions2(RN,SUP RN,I,SUP I)
        innerSolve1(upi, eps)$InnerNumericFloatSolvePackage(I,F,Par)
    else if K is GRN then
        solve1(up:SUK, eps:Par) : List(F) ==
          denom := "lcm"/[lcm(denom real(c::GRN), denom imag(c::GRN))
                              for c in coefficients up]
          up:=denom*up
          upgi := map((c:GRN):GI+->complex(numer(real c), numer(imag c)),up)_
            $UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,GI,SUP GI)
          innerSolve1(upgi, eps)$InnerNumericFloatSolvePackage(GI,F,Par)
    else error "unsupported matrix type"

         ----  the real eigenvectors expressed as floats  ----

    innerEigenvectors(A:MK,eps:Par,fact:UFactor) : List outForm ==
      pol:= charpol A
      sln1:List(IntForm):=inteigen(A,pol,fact)
      n:=nrows A
      sln:List(outForm):=[]
      for lev in sln1 repeat
        lev case outForm => sln:=cons(lev,sln)
        leva:=lev::PEigenForm
        lval:List(F):= solve1(leva.algpol,eps)
        lvect:=leva.poleigen
        lmult:=leva.almult
        for alg in lval repeat
          nsl:=[alg,lmult,[evalvect(ep,alg) for ep in lvect]]$outForm
          sln:=cons(nsl,sln)
      sln

    charpol(A:MK) : SUK ==
      dimA :PI := (nrows A):PI
      dimA ^= ncols A => error " The matrix is not square"
      B:Matrix SUK :=zero(dimA,dimA)
      for i in 1..dimA repeat
        for j in 1..dimA repeat  B(i,j):=A(i,j)::SUK
        B(i,i) := B(i,i) - monomial(1,1)$SUK
      determinant B
```

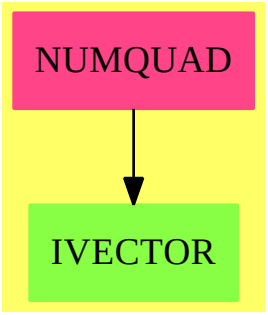
———-

— INEP.dotabb —

```
"INEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INEP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"INEP" -> "COMPCAT"
```

10.17 package INFSP InnerNumericFloatSolvePackage

10.17.1 InnerNumericFloatSolvePackage (INFSP)



Exports:
innerSolve innerSolve1 makeEq

— package INFSP InnerNumericFloatSolvePackage —

```
)abbrev package INFSP InnerNumericFloatSolvePackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is an internal package
++ for computing approximate solutions to systems of polynomial equations.
++ The parameter K specifies the coefficient field of the input polynomials
++ and must be either \spad{Fraction(Integer)} or
++ \spad{Complex(Fraction Integer)}.
++ The parameter F specifies where the solutions must lie and can
++ be one of the following: \spad{Float}, \spad{Fraction(Integer)},
++ \spad{Complex(Float)},
++ \spad{Complex(Fraction Integer)}. The last parameter specifies the type
++ of the precision operand and must be either \spad{Fraction(Integer)} or
++ \spad{Float}.
```

```
InnerNumericFloatSolvePackage(K,F,Par): Cat == Cap where
    F    :   Field  -- this is the field where the answer will be
    K    :   GcdDomain  -- type of the  input
    Par  :   Join(Field, OrderedRing ) -- it will be NF or RN

    I        ==> Integer
    NNI      ==> NonNegativeInteger
    P        ==> Polynomial
    EQ       ==> Equation
    L        ==> List
    SUP      ==> SparseUnivariatePolynomial
    RN       ==> Fraction Integer
    NF       ==> Float
    CF       ==> Complex Float
    GI       ==> Complex Integer
    GRN      ==> Complex RN
    SE       ==> Symbol
    RFI      ==> Fraction P I

    Cat == with

       innerSolve1   :  (SUP K,Par)  -> L F
          ++ innerSolve1(up,eps) returns the list of the zeros
          ++ of the univariate polynomial up with precision eps.
       innerSolve1   :  (P K,Par)  -> L F
          ++ innerSolve1(p,eps) returns the list of the zeros
          ++ of the polynomial p with precision eps.
       innerSolve    : (L P K,L P K,L SE,Par) -> L L F
          ++ innerSolve(lnum,lden,lvar,eps) returns a list of
          ++ solutions of the system of polynomials lnum, with
          ++ the side condition that none of the members of lden
          ++ vanish identically on any solution. Each solution
          ++ is expressed as a list corresponding to the list of
          ++ variables in lvar and with precision specified by eps.
       makeEq        : (L F,L SE)     -> L EQ P F
          ++ makeEq(lsol,lvar) returns a list of equations formed
          ++ by corresponding members of lvar and lsol.

    Cap == add

                ------  Local Functions  ------
       isGeneric? : (L P K,L SE) -> Boolean
       evaluate  : (P K,SE,SE,F) ->  F
       numeric   :     K          -> F
       oldCoord      : (L F,L I) -> L F
       findGenZeros  : (L P K,L SE,Par) -> L L F
       failPolSolve  : (L P K,L SE)  -> Union(L L P K,"failed")

       numeric(r:K):F ==
```

```
  K is I =>
    F is Float => r::I::Float
    F is RN    => r::I::RN
    F is CF    => r::I::CF
    F is GRN   => r::I::GRN
  K is GI =>
    gr:GI := r::GI
    F is GRN => complex(real(gr)::RN,imag(gr)::RN)$GRN
    F is CF  => convert(gr)
  error "case not handled"

-- construct the equation
makeEq(nres:L F,lv:L SE) : L EQ P F ==
    [equation(x::(P F),r::(P F)) for x in lv for r in nres]

evaluate(pol:P K,xvar:SE,zvar:SE,z:F):F ==
  rpp:=map(numeric,pol)$PolynomialFunctions2(K,F)
  rpp := eval(rpp,zvar,z)
  upol:=univariate(rpp,xvar)
  retract(-coefficient(upol,0))/retract(leadingCoefficient upol)

myConvert(eps:Par) : RN ==
  Par is RN => eps
  Par is NF => retract(eps)$NF

innerSolve1(pol:P K,eps:Par) : L F == innerSolve1(univariate pol,eps)

innerSolve1(upol:SUP K,eps:Par) : L F ==
  K is GI and (Par is RN or Par is NF) =>
      (complexZeros(upol,
                  eps)$ComplexRootPackage(SUP K,Par)) pretend L(F)
  K is I =>
    F is Float =>
      z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
      [convert((1/2)*(x.left+x.right))@Float for x in z] pretend L(F)

    F is RN =>
      z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
      [(1/2)*(x.left + x.right) for x in z] pretend L(F)
    error "improper arguments to INFSP"
  error "improper arguments to INFSP"


-- find the zeros of components in "generic" position --
findGenZeros(lp:L P K,rlvar:L SE,eps:Par) : L L F ==
  rlp:=reverse lp
  f:=rlp.first
  zvar:= rlvar.first
  rlp:=rlp.rest
  lz:=innerSolve1(f,eps)
```

```
  [reverse cons(z,[evaluate(pol,xvar,zvar,z) for pol in rlp
                  for xvar in rlvar.rest]) for z in lz]

-- convert to the old coordinates --
oldCoord(numres:L F,lval:L I) : L F ==
  rnumres:=reverse numres
  rnumres.first:= rnumres.first +
     (+/[n*nr for n in lval for nr in rnumres.rest])
  reverse rnumres

-- real zeros of a system of 2 polynomials lp (incomplete)
innerSolve2(lp:L P K,lv:L SE,eps: Par):L L F ==
   mainvar := first lv
   up1:=univariate(lp.1, mainvar)
   up2:=univariate(lp.2, mainvar)
   vec := subresultantVector(up1,up2)$SubResultantPackage(P K,SUP P K)
   p0 := primitivePart multivariate(vec.0, mainvar)
   p1 := primitivePart(multivariate(vec.1, mainvar),mainvar)
   zero? p1 or
     gcd(p0, leadingCoefficient(univariate(p1,mainvar))) ^=1 =>
       innerSolve(cons(0,lp),empty(),lv,eps)
   findGenZeros([p1, p0], reverse lv, eps)

-- real zeros of the system of polynomial lp --
innerSolve(lp:L P K,ld:L P K,lv:L SE,eps: Par) : L L F ==
   -- empty?(ld) and (#lv = 2) and (# lp = 2) => innerSolve2(lp, lv, eps)
    lnp:= [pToDmp(p)$PolToPol(lv,K) for p in lp]
    OV:=OrderedVariableList(lv)
    lvv:L OV:= [variable(vv)::OV for vv in lv]
    DP:=DirectProduct(#lv,NonNegativeInteger)
    dmp:=DistributedMultivariatePolynomial(lv,K)
    lq:L dmp:=[]
    if ld^=[] then
      lq:= [(pToDmp(q1)$PolToPol(lv,K)) pretend dmp for q1 in ld]
    partRes:=groebSolve(lnp,lvv)$GroebnerSolve(lv,K,K) pretend (L L dmp)
    partRes=list [] => []
    -- remove components where denominators vanish
    if lq^=[] then
      gb:=GroebnerInternalPackage(K,DirectProduct(#lv,NNI),OV,dmp)
      partRes:=[pr for pr in partRes|
                 and/[(redPol(fq,pr pretend List(dmp))$gb) ^=0
                   for fq in lq]]

    -- select the components in "generic" form
    rlv:=reverse lv
    rrlvv:= rest reverse lvv

    listGen:L L dmp:=[]
    for res in partRes repeat
      res1:=rest reverse res
```

```
    "and"/[("max"/degree(f,rrlvv))=1  for f in res1] =>
             listGen:=concat(res pretend (L dmp),listGen)
  result:L L F := []
  if listGen^=[] then
    listG :L L P K:=
      [[dmpToP(pf)$PolToPol(lv,K) for pf in pr] for pr in listGen]
    result:=
      "append"/[findGenZeros(res,rlv,eps) for res in listG]
    for gres in listGen repeat
       partRes:=delete(partRes,position(gres,partRes))
  -- adjust the non-generic components
  for gres in partRes repeat
      genRecord := genericPosition(gres,lvv)$GroebnerSolve(lv,K,K)
      lgen := genRecord.dpolys
      lval := genRecord.coords
      lgen1:=[dmpToP(pf)$PolToPol(lv,K) for pf in lgen]
      lris:=findGenZeros(lgen1,rlv,eps)
      result:= append([oldCoord(r,lval) for r in lris],result)
  result
```

———-

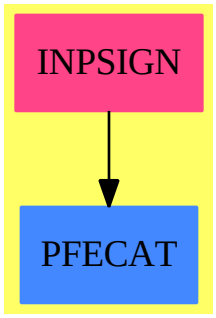
— **INFSP.dotabb** —

```
"INFSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFSP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"INFSP" -> "COMPCAT"
```

———-

10.18 package INPSIGN InnerPolySign

10.18.1 InnerPolySign (INPSIGN)



Exports:
signAround

— package INPSIGN InnerPolySign —

```
)abbrev package INPSIGN InnerPolySign
++ Author: Manuel Bronstein
++ Date Created: 23 Aug 1989
++ Date Last Updated: 19 Feb 1990
++ Description:
++ Find the sign of a polynomial around a point or infinity.

InnerPolySign(R, UP): Exports == Implementation where
  R : Ring
  UP: UnivariatePolynomialCategory R

  U ==> Union(Integer, "failed")

  Exports ==> with
    signAround: (UP,    Integer, R -> U) -> U
      ++ signAround(u,i,f) \undocumented
    signAround: (UP, R, Integer, R -> U) -> U
      ++ signAround(u,r,i,f) \undocumented
    signAround: (UP, R,          R -> U) -> U
      ++ signAround(u,r,f) \undocumented

  Implementation ==> add
    signAround(p:UP, x:R, rsign:R -> U) ==
      (ur := signAround(p, x,  1, rsign)) case "failed" => "failed"
      (ul := signAround(p, x, -1, rsign)) case "failed" => "failed"
      (ur::Integer) = (ul::Integer) => ur
      "failed"

    signAround(p, x, dir, rsign) ==
      zero? p => 0
      zero?(r := p x) =>
        (u := signAround(differentiate p, x, dir, rsign)) case "failed"
          => "failed"
        dir * u::Integer
      rsign r

    signAround(p:UP, dir:Integer, rsign:R -> U) ==
      zero? p => 0
      (u := rsign leadingCoefficient p) case "failed" => "failed"
      (dir > 0) or (even? degree p) => u::Integer
      - (u::Integer)
```

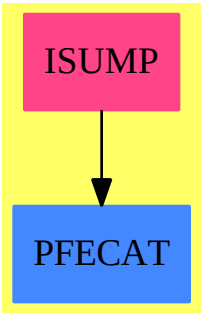
— INPSIGN.dotabb —

```
"INPSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPSIGN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INPSIGN" -> "PFECAT"
```

———-

10.19 package ISUMP InnerPolySum

10.19.1 InnerPolySum (ISUMP)



Exports:
sum

— package ISUMP InnerPolySum —

```
)abbrev package ISUMP InnerPolySum
++ Author: SMW
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for the summation packages of polynomials

InnerPolySum(E, V, R, P): Exports == Impl where
    E: OrderedAbelianMonoidSup
    V: OrderedSet
    R: IntegralDomain
    P: PolynomialCategory(R, E, V)

    Z   ==> Integer
    Q   ==> Fraction Z
```

```
    SUP ==> SparseUnivariatePolynomial

    Exports ==> with
        sum: (P, V, Segment P) -> Record(num:P, den:Z)
            ++ sum(p(n), n = a..b) returns \spad{p(a) + p(a+1) + ... + p(b)}.
        sum: (P, V) -> Record(num:P, den: Z)
            ++ sum(p(n), n) returns \spad{P(n)},
            ++ the indefinite sum of \spad{p(n)} with respect to
            ++ upward difference on n, i.e. \spad{P(n+1) - P(n) = a(n)};

    Impl ==> add
        import PolynomialNumberTheoryFunctions()
        import UnivariatePolynomialCommonDenominator(Z, Q, SUP Q)

        pmul: (P, SUP Q) -> Record(num:SUP P, den:Z)

        pmul(c, p) ==
            pn := (rec := splitDenominator p).num
            [map(x +-> numer(x) * c, pn)_
               $SparseUnivariatePolynomialFunctions2(Q, P), rec.den]

        sum(p, v, s) ==
          indef := sum(p, v)
          [eval(indef.num, v, 1 + hi s) - eval(indef.num, v, lo s),
           indef.den]

        sum(p, v) ==
            up := univariate(p, v)
            lp := nil()$List(SUP P)
            ld := nil()$List(Z)
            while up ^= 0 repeat
                ud  := degree up; uc := leadingCoefficient up
                up  := reductum up
                rec := pmul(uc, 1 / (ud+1) * bernoulli(ud+1))
                lp  := concat(rec.num, lp)
                ld  := concat(rec.den, ld)
            d := lcm ld
            vp := +/[(d exquo di)::Z * pi for di in ld for pi in lp]
            [multivariate(vp, v), d]
```

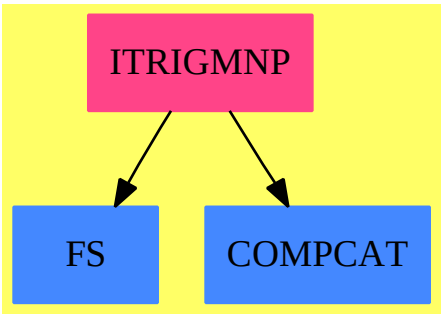
— ISUMP.dotabb —

```
"ISUMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ISUMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ISUMP" -> "PFECAT"
```

———-

10.20 package ITRIGMNP InnerTrigonometricManipulations

10.20.1 InnerTrigonometricManipulations (ITRIGMNP)



Exports:
F2FG FG2F GF2FG explogs2trigs trigs2explogs

— **package ITRIGMNP InnerTrigonometricManipulations** —

```
)abbrev package ITRIGMNP InnerTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 9 October 1993
++ Keywords: trigonometric, function, manipulation.
++ Description:
++ This package provides transformations from trigonometric functions
++ to exponentials and logarithms, and back.
++ F and FG should be the same type of function space.

InnerTrigonometricManipulations(R,F,FG): Exports == Implementation where
  R  : Join(IntegralDomain, OrderedSet)
  F  : Join(FunctionSpace R, RadicalCategory,
            TranscendentalFunctionCategory)
  FG : Join(FunctionSpace Complex R, RadicalCategory,
            TranscendentalFunctionCategory)

  Z   ==> Integer
  SY  ==> Symbol
  OP  ==> BasicOperator
  GR  ==> Complex R
  GF  ==> Complex F
```

```
  KG  ==> Kernel FG
  PG  ==> SparseMultivariatePolynomial(GR, KG)
  UP  ==> SparseUnivariatePolynomial PG
  NTHR  ==> "nthRoot"::SY

  Exports ==> with
    GF2FG        : GF -> FG
      ++ GF2FG(a + i b) returns \spad{a + i b} viewed as a function with
      ++ the \spad{i} pushed down into the coefficient domain.
    FG2F         : FG -> F
      ++ FG2F(a + i b) returns \spad{a + sqrt(-1) b}.
    F2FG         : F  -> FG
      ++ F2FG(a + sqrt(-1) b) returns \spad{a + i b}.
    explogs2trigs: FG -> GF
      ++ explogs2trigs(f) rewrites all the complex logs and
      ++ exponentials appearing in \spad{f} in terms of trigonometric
      ++ functions.
    trigs2explogs: (FG, List KG, List SY) -> FG
      ++ trigs2explogs(f, [k1,...,kn], [x1,...,xm]) rewrites
      ++ all the trigonometric functions appearing in \spad{f} and involving
      ++ one of the \spad{xi's} in terms of complex logarithms and
      ++ exponentials. A kernel of the form \spad{tan(u)} is expressed
      ++ using \spad{exp(u)**2} if it is one of the \spad{ki's}, in terms of
      ++ \spad{exp(2*u)} otherwise.

  Implementation ==> add
    ker2explogs: (KG, List KG, List SY) -> FG
    smp2explogs: (PG, List KG, List SY) -> FG
    supexp     : (UP, GF, GF, Z) -> GF
    GR2GF      : GR -> GF
    GR2F       : GR -> F
    KG2F       : KG -> F
    PG2F       : PG -> F
    ker2trigs  : (OP, List GF) -> GF
    smp2trigs  : PG -> GF
    sup2trigs  : (UP, GF) -> GF

    nth := R has RetractableTo(Integer) and F has RadicalCategory

    GR2F g        == real(g)::F + sqrt(-(1::F)) * imag(g)::F
    KG2F k        == map(FG2F, k)$ExpressionSpaceFunctions2(FG, F)
    FG2F f        == (PG2F numer f) / (PG2F denom f)
    F2FG f        == map(x +-> x::GR, f)$FunctionSpaceFunctions2(R,F,GR,FG)
    GF2FG f       == (F2FG real f) + complex(0, 1)$GR ::FG * F2FG imag f
    GR2GF gr      == complex(real(gr)::F, imag(gr)::F)

-- This expects the argument to have only tan and atans left.
-- Does a half-angle correction if k is not in the initial kernel list.
    ker2explogs(k, l, lx) ==
      empty?([v for v in variables(kf := k::FG) |
```

```
                                    member?(v, lx)]$List(SY)) => kf
  empty?(args := [trigs2explogs(a, l, lx)
                                  for a in argument k]$List(FG)) => kf
  im := complex(0, 1)$GR :: FG
  z  := first args
  is?(k, "tan"::Symbol)  =>
    e := (member?(k, l) => exp(im * z) ** 2;  exp(2 * im * z))
    - im * (e - 1) /$FG (e + 1)
  is?(k, "atan"::Symbol) =>
    im * log((1 -$FG im *$FG z)/$FG (1 +$FG im *$FG z))$FG / (2::FG)
  (operator k) args

trigs2explogs(f, l, lx) ==
  smp2explogs(numer f, l, lx) / smp2explogs(denom f, l, lx)

-- return op(arg) as f + %i g
-- op is already an operator with semantics over R, not GR
ker2trigs(op, arg) ==
  "and"/[zero? imag x for x in arg] =>
    complex(op [real x for x in arg]$List(F), 0)
  a := first arg
  is?(op, "exp"::Symbol)  => exp a
  is?(op, "log"::Symbol)  => log a
  is?(op, "sin"::Symbol)  => sin a
  is?(op, "cos"::Symbol)  => cos a
  is?(op, "tan"::Symbol)  => tan a
  is?(op, "cot"::Symbol)  => cot a
  is?(op, "sec"::Symbol)  => sec a
  is?(op, "csc"::Symbol)  => csc a
  is?(op, "asin"::Symbol)  => asin a
  is?(op, "acos"::Symbol)  => acos a
  is?(op, "atan"::Symbol)  => atan a
  is?(op, "acot"::Symbol)  => acot a
  is?(op, "asec"::Symbol)  => asec a
  is?(op, "acsc"::Symbol)  => acsc a
  is?(op, "sinh"::Symbol)  => sinh a
  is?(op, "cosh"::Symbol)  => cosh a
  is?(op, "tanh"::Symbol)  => tanh a
  is?(op, "coth"::Symbol)  => coth a
  is?(op, "sech"::Symbol)  => sech a
  is?(op, "csch"::Symbol)  => csch a
  is?(op, "asinh"::Symbol)  => asinh a
  is?(op, "acosh"::Symbol)  => acosh a
  is?(op, "atanh"::Symbol)  => atanh a
  is?(op, "acoth"::Symbol)  => acoth a
  is?(op, "asech"::Symbol)  => asech a
  is?(op, "acsch"::Symbol)  => acsch a
  is?(op, "abs"::Symbol)    => sqrt(norm a)::GF
  nth and is?(op, NTHR) => nthRoot(a, retract(second arg)@Z)
  error "ker2trigs: cannot convert kernel to gaussian function"
```

```
    sup2trigs(p, f) ==
      map(smp2trigs, p)$SparseUnivariatePolynomialFunctions2(PG, GF) f

    smp2trigs p ==
      map(x +-> explogs2trigs(x::FG),GR2GF, p)_
         $PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, GF)

    explogs2trigs f ==
      (m := mainKernel f) case "failed" =>
        GR2GF(retract(numer f)@GR) / GR2GF(retract(denom f)@GR)
      op  := operator(operator(k := m::KG))$F
      arg := [explogs2trigs x for x in argument k]
      num := univariate(numer f, k)
      den := univariate(denom f, k)
      is?(op, "exp"::Symbol) =>
        e  := exp real first arg
        y  := imag first arg
        g  := complex(e *  cos y, e * sin y)$GF
        gi := complex(cos(y) / e, - sin(y) / e)$GF
        supexp(num,g,gi,b := (degree num)::Z quo 2)/supexp(den,g,gi,b)
      sup2trigs(num, g := ker2trigs(op, arg)) / sup2trigs(den, g)

    supexp(p, f1, f2, bse) ==
      ans:GF := 0
      while p ^= 0 repeat
        g := explogs2trigs(leadingCoefficient(p)::FG)
        if ((d := degree(p)::Z - bse) >= 0) then
             ans := ans + g * f1 ** d
        else ans := ans + g * f2 ** (-d)
        p := reductum p
      ans

    PG2F p ==
      map(KG2F, GR2F, p)$PolynomialCategoryLifting(IndexedExponents KG,
                                                   KG, GR, PG, F)

    smp2explogs(p, l, lx) ==
      map(x +-> ker2explogs(x, l, lx), y +-> y::FG, p)_
       $PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, FG)
```

———-

— ITRIGMNP.dotabb —

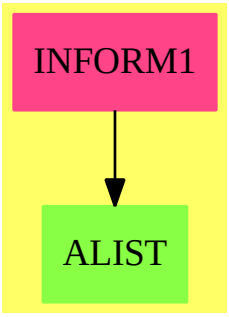
```
"ITRIGMNP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITRIGMNP"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
```

```
"ITRIGMNP" -> "FS"
"ITRIGMNP" -> "COMPCAT"
```

———-

10.21 package INFORM1 InputFormFunctions1

10.21.1 InputFormFunctions1 (INFORM1)



Exports:
interpret packageCall

— **package INFORM1 InputFormFunctions1** —

```
)abbrev package INFORM1 InputFormFunctions1
--)boot $noSubsumption := false

++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for manipulating input forms.

InputFormFunctions1(R:Type):with
  packageCall: Symbol -> InputForm
    ++ packageCall(f) returns the input form corresponding to f$R.
  interpret  : InputForm -> R
    ++ interpret(f) passes f to the interpreter, and transforms
    ++ the result into an object of type R.
 == add
  Rname := devaluate(R)$Lisp :: InputForm

  packageCall name ==
    convert([convert("$elt"::Symbol), Rname,
```

```
                              convert name]$List(InputForm))@InputForm

  interpret form ==
    retract(interpret(convert([convert("@"::Symbol), form,
          Rname]$List(InputForm))@InputForm)$InputForm)$AnyFunctions1(R)
```

———-

— **INFORM1.dotabb** —

```
"INFORM1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFORM1"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"INFORM1" -> "ALIST"
```

———-

10.22 package INTERGB InterfaceGroebnerPackage

— **InterfaceGroebnerPackage.input** —

```
)set break resume
)sys rm -f InterfaceGroebnerPackage.output
)spool InterfaceGroebnerPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show InterfaceGroebnerPackage
--R
--R InterfaceGroebnerPackage(K: Field,symb: List(Symbol),E: OrderedAbelianMonoidSup,OV: OrderedS
--R Abbreviation for InterfaceGroebnerPackage is INTERGB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTERGB
--R
--R------------------------------ Operations --------------------------------
--R groebner : List(R) -> List(R)
--R
--E 1

)spool
)lisp (bye)
```

———-

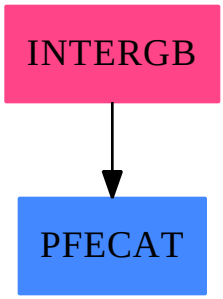
— InterfaceGroebnerPackage.help —

```
=====================================================================
InterfaceGroebnerPackage examples
=====================================================================

See Also:
o )show InterfaceGroebnerPackage
```

———-

10.22.1 InterfaceGroebnerPackage (INTERGB)



Exports:
groebner

— package INTERGB InterfaceGroebnerPackage —

```
)abbrev package INTERGB InterfaceGroebnerPackage
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: April, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
InterfaceGroebnerPackage(K,symb,E,OV,R):Exports == Implementation where
  K:Field
  symb: List Symbol
  E:OrderedAbelianMonoidSup
  OV: OrderedSet
  R    : PolynomialCategory(K,E,OV)

  Exports ==>  with
```

```
    groebner: List R -> List R

  Implementation ==>  add
    if ^(K has FiniteFieldCategory) then

      GBPackR ==> GroebnerPackage(K,E,OV,R)
      groebner(l)==groebner(l)$GBPackR

    else

      q:PositiveInteger:=(characteristic()$K pretend PositiveInteger)

      PF ==> PrimeField(q)
      DPF ==> DistributedMultivariatePolynomial(symb,PF)
      D ==> DistributedMultivariatePolynomial(symb,K)
      JCFGBPack ==> GroebnerPackage(PF,E,OV,DPF)
      GBPack ==> GroebnerPackage(K,E,OV,D)

      coerceKtoPF: K -> PF
      coerceKtoPF(a:K):PF==
        index(lookup(a)$K)$PF

      coercePFtoK: PF -> K
      coercePFtoK(a:PF):K==
        index(lookup(a)$PF)$K

      coerceRtoDwithPF: R -> DPF
      coerceRtoDwithPF(pol) ==
        map(coerceKtoPF(#1),pol)$MPolyCatFunctions2(OV,E,E,K,PF,R,DPF)

      coerceDtoRwithPF: DPF->R
      coerceDtoRwithPF(pol) ==
        map(coercePFtoK(#1),pol)$MPolyCatFunctions2(OV,E,E,PF,K,DPF,R)

      coerceRtoD: R -> D
      coerceRtoD(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,K,R,D)

      coerceDtoR: D->R
      coerceDtoR(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,K,D,R)

      gs:= size()$K
      if gs = q and (representationType()$K case "prime")  then
        groebner(l)==
  ldmp:List DPF:= [coerceRtoDwithPF(pol) for pol in l]
          gg:=groebner(ldmp)$JCFGBPack
          [coerceDtoRwithPF(pol) for pol in gg]
      else
        groebner(l)==
  ldmp:List D:= [coerceRtoD(pol) for pol in l]
          gg:=groebner(ldmp)$GBPack
```

```
[coerceDtoR(pol) for pol in gg]
```

———-

— **INTERGB.dotabb** —

```
"INTERGB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTERGB"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTERGB" -> "PFECAT"
```

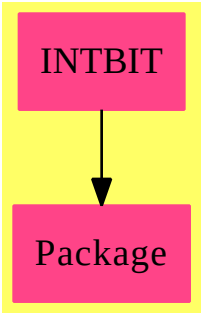
———-

10.23 package INTBIT IntegerBits

Bug! Cannot precompute params and return a function which simply computes the last call. e.g. ridHack1, below.

Functions related to the binary representation of integers. These functions directly access the bits in the big integer representation and so are much facter than using a quotient loop.

10.23.1 IntegerBits (INTBIT)



Exports:
bitLength bitCoef bitTruth

— **package INTBIT IntegerBits** —

```
)abbrev package INTBIT IntegerBits
++ Description:
++ This  package provides functions to lookup bits in integers
```

```
IntegerBits: with
    --  bitLength(n)  == # of bits to represent abs(n)
    --  bitCoef (n,i) == coef of 2**i in abs(n)
    --  bitTruth(n,i) == true if coef of 2**i in abs(n) is 1

        bitLength: Integer -> Integer
          ++ bitLength(n) returns the number of bits to represent abs(n)
        bitCoef:   (Integer, Integer) -> Integer
          ++ bitCoef(n,m) returns the coefficient of 2**m  in abs(n)
        bitTruth:  (Integer, Integer) -> Boolean
          ++ bitTruth(n,m) returns true if coefficient of 2**m in abs(n) is 1

    == add
        bitLength n   == INTEGER_-LENGTH(n)$Lisp
        bitCoef (n,i) == if INTEGER_-BIT(n,i)$Lisp then 1 else 0
        bitTruth(n,i) == INTEGER_-BIT(n,i)$Lisp
```

———-

— INTBIT.dotabb —

```
"INTBIT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTBIT"]
"Package" [color="#FF4488"]
"INTBIT" -> "Package"
```

———-

10.24 package COMBINAT IntegerCombinatoricFunctions

— IntegerCombinatoricFunctions.input —

```
)set break resume
)sys rm -f IntegerCombinatoricFunctions.output
)spool IntegerCombinatoricFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 4
)set expose add constructor OutputForm
--R
--I   OutputForm is already explicitly exposed in frame frame0
```

```
--E 1

--S 2 of 4
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
--R
--R                                                                    Type: Void
--E 2

--S 3 of 4
displayRow(n)==output center blankSeparate pascalRow(n)
--R
--R                                                                    Type: Void
--E 3

--S 4 of 4
for i in 0..7 repeat displayRow i
--R
--R   Compiling function pascalRow with type NonNegativeInteger -> List(
--R      OutputForm)
--R   Compiling function displayRow with type NonNegativeInteger -> Void
--R                                      1
--R                                    1   1
--R                                  1   2   1
--R                                1   3   3   1
--R                              1   4   6   4   1
--R                            1   5   10  10   5   1
--R                          1   6   15  20  15   6   1
--R                        1   7   21  35  35  21   7   1
--R                                                                    Type: Void
--E 4

)spool
)lisp (bye)
```

———-

— IntegerCombinatoricFunctions.help —

```
=====================================================================
IntegerCombinatoricFunctions examples
=====================================================================

The binomial(n, r) returns the number of subsets of r objects
taken among n objects, i.e. n!/(r! * (n-r)!)

The binomial coefficients are the coefficients of the series expansion
of a power of a binomial, that is

    n
```

```
   --+    / n \   k             n
   >      |     | x    =  (1 + x)
   --+    \ k /
   k= 0
```

This leads to the famous pascal triangle. First we expose the OutputForm domain, which is normally hidden, so we can use it to format the lines.

```
  )set expose add constructor OutputForm
```

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:

```
  pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
```

and now we format the whole line so that it looks centered:

```
  displayRow(n)==output center blankSeparate pascalRow(n)
```

and we compute the triangle

```
  for i in 0..7 repeat displayRow i
```

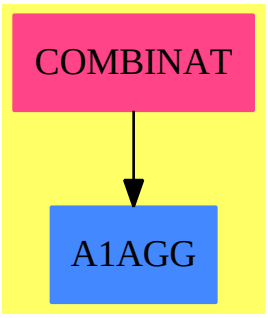
giving the pretty result:

```
   Compiling function pascalRow with type NonNegativeInteger -> List
      OutputForm
   Compiling function displayRow with type NonNegativeInteger -> Void
                                    1
                                  1   1
                                1   2   1
                              1   3   3   1
                            1   4   6   4   1
                          1   5  10  10   5   1
                        1   6  15  20  15   6   1
                      1   7  21  35  35  21   7   1
```

See Also:
o)show IntegerCombinatoricFunctions
o)d op binomial
o)show OutputForm
o)help set

———————-

10.24.1 IntegerCombinatoricFunctions (COMBINAT)



Exports:
binomial factorial multinomial partition permutation
stirling1 stirling2

— package COMBINAT IntegerCombinatoricFunctions —

```
)abbrev package COMBINAT IntegerCombinatoricFunctions
++ Authors: Martin Brock, Robert Sutor, Michael Monagan
++ Date Created: June 1987
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: integer, combinatoric function
++ Examples:
++ References:
++ Description:
++ The \spadtype{IntegerCombinatoricFunctions} package provides some
++ standard functions in combinatorics.

IntegerCombinatoricFunctions(I:IntegerNumberSystem): EE == II where
 Z   ==> Integer
 N   ==> NonNegativeInteger
 SUP ==> SparseUnivariatePolynomial
 EE ==>
  with
   binomial: (I, I) -> I
      ++ \spad{binomial(n,r)} returns the binomial coefficient
      ++ \spad{C(n,r) = n!/(r! (n-r)!)}, where \spad{n >= r >= 0}.
      ++ This is the number of combinations of n objects taken r at a time.
      ++
      ++X [binomial(5,i) for i in 0..5]
   factorial: I -> I
      ++ \spad{factorial(n)} returns \spad{n!}. this is the product of all
```

```
      ++ integers between 1 and n (inclusive).
      ++ Note that \spad{0!} is defined to be 1.
   multinomial: (I, List I) -> I
      ++ \spad{multinomial(n,[m1,m2,...,mk])} returns the multinomial
      ++ coefficient \spad{n!/(m1! m2! ... mk!)}.
   partition: I -> I
      ++ \spad{partition(n)} returns the number of partitions of the integer n.
      ++ This is the number of distinct ways that n can be written as
      ++ a sum of positive integers.
   permutation: (I, I) -> I
      ++ \spad{permutation(n)} returns \spad{!P(n,r) = n!/(n-r)!}. This is
      ++ the number of permutations of n objects taken r at a time.
   stirling1: (I, I) -> I
      ++ \spad{stirling1(n,m)} returns the Stirling number of the first kind
      ++ denoted \spad{S[n,m]}.
   stirling2: (I, I) -> I
      ++ \spad{stirling2(n,m)} returns the Stirling number of the second kind
      ++ denoted \spad{SS[n,m]}.
 II ==>
  add
   F : Record(Fn:I, Fv:I) := [0,1]
   B : Record(Bn:I, Bm:I, Bv:I) := [0,0,0]
   S : Record(Sn:I, Sp:SUP I) := [0,0]
   P : IndexedFlexibleArray(I,0) := new(1,1)$IndexedFlexibleArray(I,0)

   partition n ==
      -- This is the number of ways of expressing n as a sum of positive
      -- integers, without regard to order.  For example partition 5 = 7
      -- since 5 = 1+1+1+1+1 = 1+1+1+2 = 1+2+2 = 1+1+3 = 1+4 = 2+3 = 5 .
      -- Uses O(sqrt n) term recurrence from Abramowitz & Stegun pp. 825
      -- p(n) = sum (-1)**k p(n-j) where 0 < j := (3*k**2+-k) quo 2 <= n
      minIndex(P) ^= 0 => error "Partition: must have minIndex of 0"
      m := #P
      n < 0 => error "partition is not defined for negative integers"
      n < m::I => P(convert(n)@Z)
      concat_!(P, new((convert(n+1)@Z - m)::N,0)$IndexedFlexibleArray(I,0))
      for i in m..convert(n)@Z repeat
         s:I := 1
         t:I := 0
         for k in 1.. repeat
            l := (3*k*k-k) quo 2
            l > i => leave
            u := l+k
            t := t + s * P(convert(i-l)@Z)
            u > i => leave
            t := t + s * P(convert(i-u)@Z)
            s := -s
         P.i := t
      P(convert(n)@Z)
```

```
factorial n ==
   s,f,t : I
   n < 0 => error "factorial not defined for negative integers"
   if n <= F.Fn then s := f := 1 else (s, f) := F
   for k in convert(s+1)@Z .. convert(n)@Z by 2 repeat
      if k::I = n then t := n else t := k::I * (k+1)::I
      f := t * f
   F.Fn := n
   F.Fv := f

binomial(n, m) ==
   s,b:I
   n < 0 or m < 0 or m > n => 0
   m = 0 => 1
   n < 2*m => binomial(n, n-m)
   (s,b) := (0,1)
   if B.Bn = n then
      B.Bm = m+1 =>
         b := (B.Bv * (m+1)) quo (n-m)
         B.Bn := n
         B.Bm := m
         return(B.Bv := b)
      if m >= B.Bm then (s := B.Bm; b := B.Bv) else (s,b) := (0,1)
   for k in convert(s+1)@Z .. convert(m)@Z repeat
     b := (b*(n-k::I+1)) quo k::I
   B.Bn := n
   B.Bm := m
   B.Bv := b

multinomial(n, m) ==
   for t in m repeat t < 0 => return 0
   n < _+/m => 0
   s:I := 1
   for t in m repeat s := s * factorial t
   factorial n quo s

permutation(n, m) ==
   t:I
   m < 0 or n < m => 0
   m := n-m
   p:I := 1
   for k in convert(m+1)@Z .. convert(n)@Z by 2 repeat
      if k::I = n then t := n else t := (k*(k+1))::I
      p := p * t
   p

stirling1(n, m) ==
   -- Definition: (-1)**(n-m) S[n,m] is the number of
   -- permutations of n symbols which have m cycles.
   n < 0 or m < 1 or m > n => 0
```

```
      m = n => 1
      S.Sn = n => coefficient(S.Sp, convert(m)@Z :: N)
      x := monomial(1, 1)$SUP(I)
      S.Sn := n
      S.Sp := x
      for k in 1 .. convert(n-1)@Z repeat S.Sp := S.Sp * (x - k::SUP(I))
      coefficient(S.Sp, convert(m)@Z :: N)

   stirling2(n, m) ==
      -- definition: SS[n,m] is the number of ways of partitioning
      -- a set of n elements into m non-empty subsets
      n < 0 or m < 1 or m > n => 0
      m = 1 or n = m => 1
      s:I := if odd? m then -1 else 1
      t:I := 0
      for k in 1..convert(m)@Z repeat
         s := -s
         t := t + s * binomial(m, k::I) * k::I ** (convert(n)@Z :: N)
      t quo factorial m
```

———-

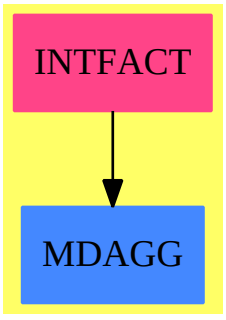
— COMBINAT.dotabb —

```
"COMBINAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMBINAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"COMBINAT" -> "A1AGG"
```

———-

10.25 package INTFACT IntegerFactorizationPackage

10.25.1 IntegerFactorizationPackage (INTFACT)



Exports:
BasicMethod factor squareFree PollardSmallFactor

— package INTFACT IntegerFactorizationPackage —

```
)abbrev package INTFACT IntegerFactorizationPackage
++ Description:
++ This Package contains basic methods for integer factorization.
++ The factor operation employs trial division up to 10,000.  It
++ then tests to see if n is a perfect power before using Pollards
++ rho method.  Because Pollards method may fail, the result
++ of factor may contain composite factors.  We should also employ
++ Lenstra's eliptic curve method.

IntegerFactorizationPackage(I): Exports == Implementation where
  I: IntegerNumberSystem

  B      ==> Boolean
  FF     ==> Factored I
  NNI    ==> NonNegativeInteger
  LMI    ==> ListMultiDictionary I
  FFE    ==> Record(flg:Union("nil","sqfr","irred","prime"),
                                                fctr:I, xpnt:Integer)

  Exports ==>  with
    factor : I -> FF
      ++ factor(n) returns the full factorization of integer n
    squareFree   : I -> FF
      ++ squareFree(n) returns the square free factorization of integer n
    BasicMethod : I -> FF
      ++ BasicMethod(n) returns the factorization
      ++ of integer n by trial division
    PollardSmallFactor: I -> Union(I,"failed")
       ++ PollardSmallFactor(n) returns a factor
       ++ of n or "failed" if no one is found

  Implementation ==> add
    import IntegerRoots(I)

    BasicSieve: (I, I) -> FF
```

———-

10.25.2 squareFree

— package INTFACT IntegerFactorizationPackage —

```
squareFree(n:I):FF ==
   u:I
   if n<0 then (m := -n; u := -1)
          else (m := n; u := 1)
   (m > 1) and ((v := perfectSqrt m) case I) =>
      for rec in (l := factorList(sv := squareFree(v::I))) repeat
        rec.xpnt := 2 * rec.xpnt
      makeFR(u * unit sv, l)
-- avoid using basic sieve when the lim is too big
-- we know the sieve constants up to sqrt(100000000)
   lim := 1 + approxSqrt(m)
   lim > (100000000::I) => makeFR(u, factorList factor m)
   x := BasicSieve(m, lim)
   y :=
     ((m:= unit x) = 1) => factorList x
     (v := perfectSqrt m) case I =>
        concat_!(factorList x, ["sqfr",v,2]$FFE)
     concat_!(factorList x, ["sqfr",m,1]$FFE)
   makeFR(u, y)
```

———-

10.25.3 PollardSmallFactor

This is Brent's[**?**] optimization of Pollard's[**?**] rho factoring. Brent's algorithm is about 24 percent faster than Pollard's. Pollard;s algorithm has complexity $O(p^{1/2})$ where p is the smallest prime factor of the composite number N.

Pollard's idea is based on the observation that two numbers x and y are congruent modulo p with probability 0.5 after $1.177 * \sqrt{p}$ numbers have been randomly chosen. If we try to factor n and p is a factor of n, then

$$1 < gcd(|x - y|, n) \leq n$$

since p divides both $|x - y|$ and n.

Given a function f which generates a pseudo-random sequence of numbers we allow x to walk the sequence in order and y to walk the sequence at twice the rate. At each cycle we compute $gcd(|x - y|, n)$. If this GCD ever equals n then $x = y$ which means that we have walked "all the way around the pseudo-random cycle" and we terminate with failure.

This algorithm returns failure on all primes but also fails on some composite numbers.

Quoting Brent's back-tracking idea:

> The best-known algorithm for finding GCDs is the Euclidean algorithm which takes $O(\log N)$ times as long as one multiplication mod N. Pollard showed that most of the GCD computations in Floyd's algorithm could be dispensed with.

> ... The idea is simple: if P_F computes $GCD(z_1, N)$, $GCD(z_2, N)$,..., then we compute
>
> $$q_i = \prod_{j=1}^{i} z_j (\text{mod } N)$$
>
> and only compute $GCD(q_i, N)$ when i is a multiple of m, where $\log N << m << N^{1/4}$. Since $q_{i+1} = q_i \times z_{i+1} (\text{mod } N)$, the work required for each GCD computation in algorithm P_F is effectively reduced to that for a multiplication mod N in the modified algorithm. The probability of the algorithm failing because $q_i = 0$ increases, so it is best not to choose m too large. This problem can be minimized by backtracking to the state after the previous GCD computation and setting $m = 1$.

Brent incorporates back-tracking, omits the random choice of u, and makes some minor modifications. His algorithm (p192-183) reads:

$y := x_0; r := 1; q := 1;$
  repeat $x := y;$
    for $i := 1$ **to** r **do** $y := f(y); k := 0;$
    repeat $ys := y;$
      for $i := 1$ **to** $min(m, r - k)$ **do**
        begin $y := f(y); q := q * |x - y| mod N$
        end;
      $G := GCD(q, N); k := k + m$
    until $(k \geq r)$ **or** $(G > 1); r := 2 * r$
  until $G > 1;$
  if $G = N$ **then**
    repeat $ys := f(ys); G := GCD(|y - yx|, N)$
    until $G > 1;$
  if $G = N$ **then** failure **else** success

Here we use the function

$$(y * y + 5 :: I) \text{ rem } n$$

as our pseudo-random sequence with a random starting value for y.

On possible optimization to explore is to keep a hash table for the computed values of the function $y_{i+1} := f(y_i)$ since we effectively walk the sequence several times. And we walk the sequence in a loop many times. But because we are generating a very large number of numbers the array can be a simple array of fixed size that captures the last n values. So if we make a fixed array F of, say 2^q elements we can store $f(y_i)$ in F[y_i mod 2^q].

One property that this algorithm assumes is that the function used to generate the numbers has a long, hopefully complete, period. It is not clear that the recommended function has that property.

— package INTFACT IntegerFactorizationPackage —

```
PollardSmallFactor(n:I):Union(I,"failed") ==
   -- Use the Brent variation
   x0 := random()$I
   m := 100::I
   y := x0 rem n
   r:I := 1
   q:I := 1
   G:I := 1
   until G > 1 repeat
      x := y
      for i in 1..convert(r)@Integer repeat
         y := (y*y+5::I) rem n
         k:I := 0
      until (k>=r) or (G>1) repeat
         ys := y
         for i in 1..convert(min(m,r-k))@Integer repeat
            y := (y*y+5::I) rem n
            q := q*abs(x-y) rem n
         G := gcd(q,n)
         k := k+m
      r := 2*r
   if G=n then
      until G>1 repeat
         ys := (ys*ys+5::I) rem n
         G := gcd(abs(x-ys),n)
   G=n => "failed"
   G
```

———-

10.25.4 BasicSieve

We create a list of prime numbers up to the limit given. The prior code used a circular list but tests of that list show that on average more than 50the required prime numbers. Overall this is a small percentage of the time needed to factor.

This loop uses three pieces of information

1. n which is the number we are testing
2. d which is the current prime to test
3. lim which is the upper limit of the primes to test

We loop d over the list of primes. If the remaining number n is smaller than the square of d then n must be prime and if it is not one, we add it to the list of primes. If the remaining

number is larger than the square of d we remove all factors of d, reducing n each time. Then we add a record of the new factor and its multiplicity, m. We continue the loop until we run out of primes.

Annoyingly enough, primes does not return an ordered list so we fix this.

The sieve works up to a given limit, reducing out the factors that it finds. If it can find all of the factors than it returns a factored result where the first element is the unit 1. If there is still a part of the number unfactored it returns the number and a list of the factors found and their multiplicity.

Basically we just loop thru the prime factors checking to see if they are a component of the number, n. If so, we remove the factor from the number n (possibly m times) and continue thru the list of primes.

— **package INTFACT IntegerFactorizationPackage** —

```
BasicSieve(n, lim) ==
   p:=primes(1::I,lim::I)$IntegerPrimesPackage(I)
   l:List(I) := append([first p],reverse rest p)
   ls := empty()$List(FFE)
   for d in l repeat
      if n<d*d then
         if n>1 then ls := concat_!(ls, ["prime",n,1]$FFE)
         return makeFR(1, ls)
      for m in 0.. while zero?(n rem d) repeat n := n quo d
      if m>0 then ls := concat_!(ls, ["prime",d,convert m]$FFE)
   makeFR(n,ls)
```

———-

10.25.5 BasicMethod

— **package INTFACT IntegerFactorizationPackage** —

```
BasicMethod n ==
   u:I
   if n<0 then (m := -n; u := -1)
          else (m := n; u := 1)
   x := BasicSieve(m, 1 + approxSqrt m)
   makeFR(u, factorList x)
```

———-

10.25.6 factor

The factor function is many orders of magnitude slower than the results of other systems. A posting on sci.math.symbolic showed that NTL could factor the final value (t6) in about 11 seconds. Axiom takes about 8 hours.

```
a1:=101
a2:=109
t1:=a1*a2
factor t1

a3:=21525175387
t2:=t1*a3
factor t2

a4:=218301576858349
t3:=t2*a4
factor t3

a5:=13731482973783137
t4:=t3*a5
factor t4

a6:=23326138687706820109
t5:=t4*a6
factor t5

a7:=432824080117318843825281371694451836916 1
t6:=t5*a7
factor t6
```

— package INTFACT IntegerFactorizationPackage —

```
factor m ==
   u:I
   zero? m => 0
   if negative? m then (n := -m; u := -1)
                  else (n := m; u := 1)
   b := BasicSieve(n, 10000::I)
   flb := factorList b
   ((n := unit b) = 1) => makeFR(u, flb)
   a:LMI := dictionary() -- numbers yet to be factored
   b:LMI := dictionary() -- prime factors found
   f:LMI := dictionary() -- number which could not be factored
   insert_!(n, a)
   while not empty? a repeat
      n := inspect a; c := count(n, a); remove_!(n, a)
      prime?(n)$IntegerPrimesPackage(I) => insert_!(n, b, c)
```

```
          -- test for a perfect power
          (s := perfectNthRoot n).exponent > 1 =>
            insert_!(s.base, a, c * s.exponent)
          -- test for a difference of square
          x:=approxSqrt n
          if (x**2<n) then x:=x+1
          (y:=perfectSqrt (x**2-n)) case I =>
                insert_!(x+y,a,c)
                insert_!(x-y,a,c)
          (d := PollardSmallFactor n) case I =>
             for m in 0.. while zero?(n rem d) repeat n := n quo d
             insert_!(d, a, m * c)
             if n > 1 then insert_!(n, a, c)
          -- an elliptic curve factorization attempt should be made here
          insert_!(n, f, c)
       -- insert prime factors found
       while not empty? b repeat
          n := inspect b; c := count(n, b); remove_!(n, b)
          flb := concat_!(flb, ["prime",n,convert c]$FFE)
       -- insert non-prime factors found
       while not empty? f repeat
          n := inspect f; c := count(n, f); remove_!(n, f)
          flb := concat_!(flb, ["nil",n,convert c]$FFE)
       makeFR(u, flb)
```

———-

— **INTFACT.dotabb** —

```
"INTFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTFACT"]
"MDAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MDAGG"]
"INTFACT" -> "MDAGG"
```

———-

10.26 package ZLINDEP IntegerLinearDependence

— **IntegerLinearDependence.input** —

```
)set break resume
)spool IntegerLinearDependence.output
)set message test on
)set message auto off
)clear all
```

```
--S 1 of 8
M := SQMATRIX(2,INT)
--R
--R
--R   (1)  SquareMatrix(2,Integer)
--R                                                              Type: Domain
--E 1

--S 2 of 8
m1: M := squareMatrix matrix [ [1, 2], [0, -1] ]
--R
--R
--R        +1   2 +
--R   (2)  |      |
--R        +0  - 1+
--R                                              Type: SquareMatrix(2,Integer)
--E 2

--S 3 of 8
m2: M := squareMatrix matrix [ [2, 3], [1, -2] ]
--R
--R
--R        +2   3 +
--R   (3)  |      |
--R        +1  - 2+
--R                                              Type: SquareMatrix(2,Integer)
--E 3

--S 4 of 8
m3: M := squareMatrix matrix [ [3, 4], [2, -3] ]
--R
--R
--R        +3   4 +
--R   (4)  |      |
--R        +2  - 3+
--R                                              Type: SquareMatrix(2,Integer)
--E 4

--S 5 of 8
linearlyDependentOverZ? vector [m1, m2, m3]
--R
--R
--R   (5)  true
--R                                                              Type: Boolean
--E 5

--S 6 of 8
c := linearDependenceOverZ vector [m1, m2, m3]
--R
```

```
--R
--R   (6)  [1,- 2,1]
--R                                                Type: Union(Vector(Integer),...)
--E 6

--S 7 of 8
c.1 * m1 + c.2 * m2 + c.3 * m3
--R
--R
--R        +0  0+
--R   (7)  |    |
--R        +0  0+
--R                                                   Type: SquareMatrix(2,Integer)
--E 7

--S 8 of 8
solveLinearlyOverQ(vector [m1, m3], m2)
--R
--R
--R         1 1
--R   (8)  [-,-]
--R         2 2
--R                                      Type: Union(Vector(Fraction(Integer)),...)
--E 8
)spool
)lisp (bye)
```

———-

— IntegerLinearDependence.help —

```
=====================================================================
IntegerLinearDependence examples
=====================================================================

The elements v1,...,vN of a module M over a ring R are said to be
linearly dependent over R if there exist c1,...,cN in R, not all 0,
such that c1 v1 + ...  cN vNn = 0.  If such ci's exist, they form
what is called a linear dependence relation over R for the vi's.

The package IntegerLinearDependence provides functions for testing
whether some elements of a module over the integers are linearly
dependent over the integers, and to find the linear dependence
relations, if any.

Consider the domain of two by two square matrices with integer entries.

  M := SQMATRIX(2,INT)
    SquareMatrix(2,Integer)
```

```
                    Type: Domain
```

Now create three such matrices.

```
  m1: M := squareMatrix matrix [ [1, 2], [0, -1] ]
     +1   2 +
     |      |
     +0  - 1+
                    Type: SquareMatrix(2,Integer)

  m2: M := squareMatrix matrix [ [2, 3], [1, -2] ]
    +2   3 +
    |      |
    +1  - 2+
                    Type: SquareMatrix(2,Integer)

  m3: M := squareMatrix matrix [ [3, 4], [2, -3] ]
    +3   4 +
    |      |
    +2  - 3+
                    Type: SquareMatrix(2,Integer)
```

This tells you whether m1, m2 and m3 are linearly dependent over the integers.

```
  linearlyDependentOverZ? vector [m1, m2, m3]
    true
                    Type: Boolean
```

Since they are linearly dependent, you can ask for the dependence relation.

```
  c := linearDependenceOverZ vector [m1, m2, m3]
    [1,- 2,1]
                    Type: Union(Vector Integer,...)
```

This means that the following linear combination should be 0.

```
  c.1 * m1 + c.2 * m2 + c.3 * m3
    +0  0+
    |    |
    +0  0+
                    Type: SquareMatrix(2,Integer)
```

When a given set of elements are linearly dependent over R, this also means that at least one of them can be rewritten as a linear combination of the others with coefficients in the quotient field of R.

To express a given element in terms of other elements, use the operation solveLinearlyOverQ.

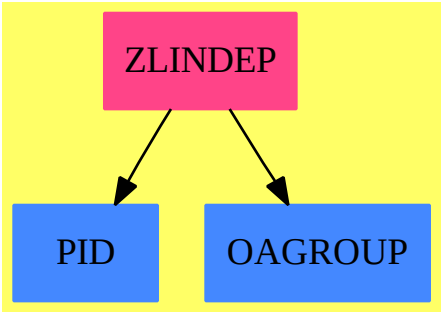
```
  solveLinearlyOverQ(vector [m1, m3], m2)
```

```
     1 1
    [-,-]
     2 2
                        Type: Union(Vector Fraction Integer,...)

See Also:
o )show IntegerLinearDependence
```

———-

10.26.1 IntegerLinearDependence (ZLINDEP)



Exports:
linearDependenceOverZ linearlyDependentOverZ? solveLinearlyOverQ

— package ZLINDEP IntegerLinearDependence —

```
)abbrev package ZLINDEP IntegerLinearDependence
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 14 May 1991
++ Description:
++ Test for linear dependence over the integers.

IntegerLinearDependence(R): Exports == Implementation where
  R: LinearlyExplicitRingOver Integer

  Z ==> Integer

  Exports ==> with
    linearlyDependentOverZ?: Vector R -> Boolean
      ++ \spad{linearlyDependentOverZ?([v1,...,vn])} returns true if the
      ++ vi's are linearly dependent over the integers, false otherwise.
```

```
    linearDependenceOverZ  : Vector R -> Union(Vector Z, "failed")
      ++ \spad{linearlyDependenceOverZ([v1,...,vn])} returns
      ++ \spad{[c1,...,cn]} if
      ++ \spad{c1*v1 + ... + cn*vn = 0} and not all the ci's are 0, "failed"
      ++ if the vi's are linearly independent over the integers.
    solveLinearlyOverQ     : (Vector R, R) ->
                                    Union(Vector Fraction Z, "failed")
      ++ \spad{solveLinearlyOverQ([v1,...,vn], u)} returns \spad{[c1,...,cn]}
      ++ such that \spad{c1*v1 + ... + cn*vn = u},
      ++ "failed" if no such rational numbers ci's exist.

  Implementation ==> add
    import LinearDependence(Z, R)

    linearlyDependentOverZ? v == linearlyDependent? v
    linearDependenceOverZ   v == linearDependence v
    solveLinearlyOverQ(v, c)  == solveLinear(v, c)
```

———-

— **ZLINDEP.dotabb** —

```
"ZLINDEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ZLINDEP"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ZLINDEP" -> "PID"
"ZLINDEP" -> "OAGROUP"
```

———-

10.27 package INTHEORY IntegerNumberTheoryFunctions

— **IntegerNumberTheoryFunctions.input** —

```
)set break resume
)spool IntegerNumberTheoryFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 30
div144 := divisors(144)
```

```
--R
--R
--R   (1)  [1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
--R                                                    Type: List(Integer)
--E 1

--S 2 of 30
#(div144)
--R
--R
--R   (2)  15
--R                                                  Type: PositiveInteger
--E 2

--S 3 of 30
reduce(+,div144)
--R
--R
--R   (3)  403
--R                                                  Type: PositiveInteger
--E 3

--S 4 of 30
numberOfDivisors(144)
--R
--R
--R   (4)  15
--R                                                  Type: PositiveInteger
--E 4

--S 5 of 30
sumOfDivisors(144)
--R
--R
--R   (5)  403
--R                                                  Type: PositiveInteger
--E 5

--S 6 of 30
f1(n)==reduce(+,[moebiusMu(d)*numberOfDivisors(quo(n,d))_
     for d in divisors(n)])
--R
--R                                                            Type: Void
--E 6

--S 7 of 30
f1(200)
--R
--R   Compiling function f1 with type PositiveInteger -> Integer
--R
```

```
--R   (7)  1
--R                                                    Type: PositiveInteger
--E 7

--S 8 of 30
f1(846)
--R
--R
--R   (8)  1
--R                                                    Type: PositiveInteger
--E 8

--S 9 of 30
f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d))_
     for d in divisors(n)])
--R
--R                                                                Type: Void
--E 9

--S 10 of 30
f2(200)
--R
--R   Compiling function f2 with type PositiveInteger -> Integer
--R
--R   (10)  200
--R                                                    Type: PositiveInteger
--E 10

--S 11 of 30
f2(846)
--R
--R
--R   (11)  846
--R                                                    Type: PositiveInteger
--E 11

--S 12 of 30
fibonacci(25)
--R
--R
--R   (12)  75025
--R                                                    Type: PositiveInteger
--E 12

--S 13 of 30
[fibonacci(n) for n in 1..15]
--R
--R
--R   (13)  [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
--R                                                      Type: List(Integer)
```

```
--E 13

--S 14 of 30
fib(n) == reduce(+,[binomial(n-1-k,k) for k in 0..quo(n-1,2)])
--R
--R                                                                    Type: Void
--E 14

--S 15 of 30
fib(25)
--R
--R   Compiling function fib with type PositiveInteger -> Integer
--R
--R   (15)  75025
--R                                                          Type: PositiveInteger
--E 15

--S 16 of 30
[fib(n) for n in 1..15]
--R
--R
--R   (16)  [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
--R                                                            Type: List(Integer)
--E 16

--S 17 of 30
legendre(3,5)
--R
--R
--R   (17)  - 1
--R                                                                 Type: Integer
--E 17

--S 18 of 30
legendre(23,691)
--R
--R
--R   (18)  - 1
--R                                                                 Type: Integer
--E 18

--S 19 of 30
h(d) == quo(reduce(+,[jacobi(d,k) for k in 1..quo(-d, 2)]),2-jacobi(d,2))
--R
--R                                                                    Type: Void
--E 19

--S 20 of 30
h(-163)
--R
```

```
--R   Compiling function h with type Integer -> Integer
--R
--R   (20)  1
--R                                                    Type: PositiveInteger
--E 20

--S 21 of 30
h(-499)
--R
--R
--R   (21)  3
--R                                                    Type: PositiveInteger
--E 21

--S 22 of 30
h(-1832)
--R
--R
--R   (22)  26
--R                                                    Type: PositiveInteger
--E 22

--S 23 of 30
inverse:(INT,INT)->INT
--R
--R                                                          Type: Void
--E 23

--S 24 of 30
inverse(a,b) ==
  borg:INT:=b
  c1:INT := 1
  d1:INT := 0
  while b ~= 0 repeat
    q := a quo b
    r := a-q*b
    print [a, "=", q, "*(", b, ")+", r]
    (a,b):=(b,r)
    (c1,d1):=(d1,c1-q*d1)
  a ~= 1 => error("moduli are not relatively prime")
  positiveRemainder(c1,borg)
--R
--R                                                          Type: Void
--E 24

--S 25 of 30
inverse(15,26)
--R
--R   Compiling function inverse with type (Integer,Integer) -> Integer
--R   [15,"=",0,"*(",26,")+",15]
```

```
--R   [26,"=",1,"*(",15,")+",11]
--R   [15,"=",1,"*(",11,")+",4]
--R   [11,"=",2,"*(",4,")+",3]
--R   [4,"=",1,"*(",3,")+",1]
--R   [3,"=",3,"*(",1,")+",0]
--R
--R   (25)  7
--R                                                    Type: PositiveInteger
--E 25

--S 26 of 30
x1:=4
--R
--R
--R   (26)  4
--R                                                    Type: PositiveInteger
--E 26

--S 27 of 30
m1:=5
--R
--R
--R   (27)  5
--R                                                    Type: PositiveInteger
--E 27

--S 28 of 30
x2:=2
--R
--R
--R   (28)  2
--R                                                    Type: PositiveInteger
--E 28

--S 29 of 30
m2:=3
--R
--R
--R   (29)  3
--R                                                    Type: PositiveInteger
--E 29

--S 30 of 30
result:=chineseRemainder(x1,m1,x2,m2)
--R
--R
--R   (30)  14
--R                                                    Type: PositiveInteger
--E 30
```

```
)spool
)lisp (bye)
```

———-

— IntegerNumberTheoryFunctions.help —

```
====================================================================
IntegerNumberTheoryFunctions examples
====================================================================

The IntegerNumberTheoryFunctions package contains a variety of operations
of interest to number theorists.  Many of these operations deal with
divisibility properties of integers.  (Recall that an integer a divides
an integer b if there is an integer c such that b = a * c.)

The operation divisors returns a list of the divisors of an integer.

  div144 := divisors(144)
    [1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
                            Type: List Integer

You can now compute the number of divisors of 144 and the sum of the
divisors of 144 by counting and summing the elements of the list we
just created.

  #(div144)
    15
                            Type: PositiveInteger

  reduce(+,div144)
    403
                            Type: PositiveInteger

Of course, you can compute the number of divisors of an integer n,
usually denoted d(n), and the sum of the divisors of an integer n,
usually denoted sigma(n), without ever listing the divisors of n.

In Axiom, you can simply call the operations numberOfDivisors and
sumOfDivisors.

  numberOfDivisors(144)
    15
                            Type: PositiveInteger

  sumOfDivisors(144)
    403
                            Type: PositiveInteger
```

```
The key is that d(n) and sigma(n) are "multiplicative functions."
This means that when n and m are relatively prime, that is, when
n and m have no prime factor in common, then d(nm) = d(n) d(m) and
sigma(nm) = sigma(n) sigma(m).  Note that these functions are trivial to
compute when n is a prime power and are computed for general n from the
prime factorization of n.  Other examples of multiplicative functions
are sigma_k(n), the sum of the k-th powers of the divisors of n and
varphi(n), the number of integers between 1 and n which are prime to n.
The corresponding Axiom operations are called sumOfKthPowerDivisors and
eulerPhi.

An interesting function is mu(n), the Moebius mu function, defined as
follows: mu(1) = 1, mu(n) = 0, when n is divisible by a square, and
mu = (-1)^k, when n is the product of k distinct primes.  The corresponding
Axiom operation is moebiusMu. This function occurs in the following theorem:

Theorem: (Moebius Inversion Formula):
  Let f(n) be a function on the positive integers and let F(n)
  be defined by
    F(n) = \sum_{d | n} f(n)
  the sum of f(n) over d | n where the sum is taken over the
  positive divisors of n.  Then the values of f(n) can be recovered
  from the values of F(n): f(n) = sum_{d | n} \mu(n) F(n/d) where
  again the sum is taken over the positive divisors of n.

When f(n) = 1, then F(n) = d(n).  Thus, if you sum mu(d)..d(n/d) over
the positive divisors d of n, you should always get 1.

  f1(n)==reduce(+,[moebiusMu(d)*numberOfDivisors(quo(n,d))_
     for d in divisors(n)])
                            Type: Void

  f1(200)
    1
                            Type: PositiveInteger

  f1(846)
    1
                            Type: PositiveInteger

Similarly, when f(n) = n, then F(n) = sigma(n).  Thus, if you sum
mu(d)..sigma(n/d) over the positive divisors d of n, you should always get n.

  f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d))_
     for d in divisors(n)])
                            Type: Void

  f2(200)
    200
                            Type: PositiveInteger
```

```
  f2(846)
    846
                            Type: PositiveInteger

The Fibonacci numbers are defined by
   F(1) = F(2) = 1 and
   F(n) = F(n-1) + F(n-2) for n = 3,4,...

The operation fibonacci computes the n-th Fibonacci number.

  fibonacci(25)
    75025
                            Type: PositiveInteger

  [fibonacci(n) for n in 1..15]
    [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
                            Type: List Integer

Fibonacci numbers can also be expressed as sums of binomial coefficients.

  fib(n) == reduce(+,[binomial(n-1-k,k) for k in 0..quo(n-1,2)])
                             Type: Void

  fib(25)
    75025
                             Type: PositiveInteger

  [fib(n) for n in 1..15]
    [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
                             Type: List Integer

Quadratic symbols can be computed with the operations legendre and
jacobi.  The Legendre symbol a/p is defined for integers a and p with
p an odd prime number.  By definition,

  (a/p) = +1, when a is a square (mod p),
  (a/p)= -1, when a is not a square (mod p), and
  (a/p) = 0, when a is divisible by p.

You compute (a/p) via the command legendre(a,p).

  legendre(3,5)
    - 1
                              Type: Integer

  legendre(23,691)
    - 1
                              Type: Integer
```

```
The Jacobi symbol (a/n) is the usual extension of the Legendre symbol,
where n is an arbitrary integer.  The most important property of the
Jacobi symbol is the following: if K is a quadratic field with
discriminant d and quadratic character chi, then chi(n) = (d/n).
Thus, you can use the Jacobi symbol to compute, say, the class numbers
of imaginary quadratic fields from a standard class number formula.

This function computes the class number of the imaginary quadratic
field with discriminant d.

  h(d) == quo(reduce(+,[jacobi(d,k) for k in 1..quo(-d, 2)]),2-jacobi(d,2))
                              Type: Void

  h(-163)
    1
                              Type: PositiveInteger

  h(-499)
    3
                              Type: PositiveInteger

  h(-1832)
    26
                              Type: PositiveInteger

========================================================================
The inverse function
========================================================================

The inverse function is derived from the Extended Euclidean Algorithm.
If we divide one integer by another nonzero integer we get an integer
quotient plus a remainder which is, in general, a rational number.
For instance,
  13/5 = 2 + 3/5
where 2 is the quotient and 3/5 is the remainder.

If we multiply thru by the denominator of the remainder we get an answer
in integer terms which no longer involves division:
  13 = 2(5) + 3

This gives a method of dividing integers. Specifically, if a and b are
positive integers, there exist unique non-negative integers q and r so that

  a = qb + r , where 0 <= r < b

q is called the quotient and r the remainder.

The greatest common divisor of integers a and b, denoted by gcd(a,b),
is the largest integer that divides (without remainder) both a and
b. So, for example:
```

```
  gcd(15, 5)  = 5,
  gcd(7, 9)   = 1,
  gcd(12, 9)  = 3,
  gcd(81, 57) = 3.

The gcd of two integers can be found by repeated application of the
division algorithm, this is known as the Euclidean Algorithm. You
repeatedly divide the divisor by the remainder until the remainder is
0. The gcd is the last non-zero remainder in this algorithm. The
following example shows the algorithm.

Finding the gcd of 81 and 57 by the Euclidean Algorithm:
  81 = 1(57) + 24
  57 = 2(24) + 9
  24 = 2(9)  + 6
  9  = 1(6)  + 3
  6  = 2(3)  + 0

So the greatest commmon divisor, the GCD(81,51)=3.

If the gcd(a, b) = r then there exist integers s and t so that

  s(a) + t(b) = r

By back substitution in the steps in the Euclidean Algorithm, it is
possible to find these integers s and t. We shall do this with the
above example:

Starting with the next to last line, we have:

   3 = 9 -1(6)

From the line before that, we see that 6 = 24 - 2(9), so:

  3 = 9 - 1(24 - 2(9)) = 3(9) - 1(24)

From the line before that, we have 9 = 57 - 2(24), so:

  3 = 3( 57 - 2(24)) - 1(24) = 3(57) - 7(24)

And, from the line before that 24 = 81 - 1(57), giving us:

  3 = 3(57) - 7( 81 - 1(57)) = 10(57) -7(81)

So we have found s = -7 and t = 10.

The Extended Euclidean Algorithm computes the GCD(a,b) and
the values for s and t.

Suppose we were doing arithmetics modulo 26 and we needed to find the
```

```
inverse of a number mod 26. This turned out to be a difficult task (and
not always possible). We observed that a number x had an inverse mod 26
(i.e., a number y so that xy = 1 mod 26) if and only if gcd(x,26) = 1.
In the general case the inverse of x exists if and only if gcd(x, n) = 1
and if it exists then there exist integers s and t so that

  sx + tn = 1

But this says that sx = 1 + (-t)n, or in other words,

  sx == 1 mod n

So, s (reduced mod n if need be) is the inverse of x mod n.
The extended Euclidean algorithm calculates s efficiently.

========================================================================
Finding the inverse mod n
========================================================================

We will number the steps of the Euclidean algorithm starting with step 0.
The quotient obtained at step i will be denoted by qi and an auxillary
number, si. For the first two steps, the value of this number is given:
  s(0) = 0 and
  s(1) = 1.

For the remainder of the steps, we recursively calculate
  s(i) = s(i-2) - s(i-1) q(i-2) mod n

The algorithm starts by "dividing" n by x. If the last non-zero remainder
occurs at step k, then if this remainder is 1, x has an inverse and it is
s(k+2). If the remainder is not 1, then x does not have an inverse.

For example, find the inverse of 15 mod 26.

Step 0: 26 = 1(15) + 11 s(0) = 0
Step 1: 15 = 1(11) + 4  s(1) = 1
Step 2: 11 = 2(4) + 3   s(2) = 0  -  1( 1) mod 26 =  25
Step 3: 4  = 1(3) + 1   s(3) = 1  - 25( 1) mod 26 = -24 mod 26 = 2
Step 4: 3  = 3(1) + 0   s(4) = 25 -  2( 2) mod 26 =  21
                        s(5) = 2  - 21( 1) mod 26 = -19 mod 26 = 7

Notice that 15(7) = 105 = 1 + 4(26) == 1 (mod 26).

Using the half extended Euclidean algorithm we compute 1/a mod b.

  inverse:(INT,INT)->INT
                              Type: Void

  inverse(a,b) ==
    borg:INT:=b
```

```
    c1:INT := 1
    d1:INT := 0
    while b ~= 0 repeat
      q := a quo b
      r := a-q*b
      print [a, "=", q, "*(", b, ")+", r]
      (a,b):=(b,r)
      (c1,d1):=(d1,c1-q*d1)
    a ~= 1 => error("moduli are not relatively prime")
    positiveRemainder(c1,borg)
                                Type: Void

  inverse(15,26)
   [15,"=",0,"*(",26,")+",15]
   [26,"=",1,"*(",15,")+",11]
   [15,"=",1,"*(",11,")+",4]
   [11,"=",2,"*(",4,")+",3]
   [4,"=",1,"*(",3,")+",1]
   [3,"=",3,"*(",1,")+",0]

   7
                                Type: PositiveInteger


======================================================================
The Chinese Remainder Algorithm
======================================================================

The Chinese Remainder Theorem says that given n moduli m_i for
(1 <= i <= n) of pairwise coprime integers and a set of congruential
equations x \equiv c_i (mod m_i) (1 <= i <= n) for an arbitrary
integer x, there is a unique solution c for x modulo the product
M = \prod_{i=1}^n {m_i}, say x \equiv c (mod M).

If the upper bound B of the absolute value of x is known, that is, if
|x| <= B then the number of moduli can be chosen such that B < M/2. If
c is the absolutely least residue modulo M then c and x coincide so x
is uniquely determined.

Let m1,m2,...,mr be positive integers that are pairwise relatively prime.
Let x1,x2,..,xr be integers with 0 <= xi < mi. Then, there is exactly one
x in the interval 0 <= x < m1 ... m2 ... mr
that satisfies the remainder equations

  irem(x,mi) = xi, i=1,2,...,r

where irem is the positive integer remainder function.

For example, et x1 = 4, m1 = 5, x2 = 2, m2 = 3. We know that
  irem(x,m1) = x1
```

```
  irem(x,m2) = x2
where 0 <= x_ < m1 and 0 <= x2 < m2.

By the extended Euclidean Algorithm there are integers c and d such that
  c m1 + d m2 = 1

In this case we are looking for an integer such that
 irem(x,5) = 4,
 irem(x,3) = 2

The algorithm we use is to first compute the positive integer remainder of
x1 and m1 to get a new x1:

 x1 = positiveRemainder(x1,m1)
  4 = positiveRemainder(4,5)

Next compute the positive integer remainder of x2 and m2 to get a new x2:

 x2 = positiveRemainder(x2,m2)
  2 = positiveRemainder(2,3)

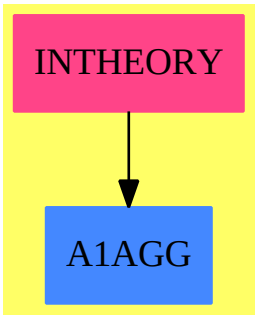
Then we compute x1 + m1 ... positiveRemainder(((x2-x1)*inverse(m1,m2)),m2)
or
  4+5*positiveRemainder(((2-4)*inverse(5,3)),3)
or
  4+5*positiveRemainder(-2*2),3)
or
  4+5*2
or
  14

This function has a restricted signature which only allows for
computing the chinese remainder of two numbers and two moduli.
  x1:=4
    4
                          Type: PositiveInteger
  m1:=5
    5
                          Type: PositiveInteger
  x2:=2
    2
                          Type: PositiveInteger
  m2:=3
    3
                          Type: PositiveInteger
  result:=chineseRemainder(x1,m1,x2,m2)
    14
                          Type: PositiveInteger

See Also:
```

```
o )show IntegerNumberTheoryFunctions
```

10.27.1 IntegerNumberTheoryFunctions (INTHEORY)



Exports:

bernoulli	chineseRemainder	divisors	euler	eulerPhi
fibonacci	harmonic	jacobi	legendre	moebiusMu
numberOfDivisors	sumOfDivisors	sumOfKthPowerDivisors		

— package INTHEORY IntegerNumberTheoryFunctions —

```
)abbrev package INTHEORY IntegerNumberTheoryFunctions
++ Author: Michael Monagan, Martin Brock, Robert Sutor, Timothy Daly
++ Date Created: June 1987
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: number theory, integer
++ Examples:
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various number theoretic functions on the integers.

IntegerNumberTheoryFunctions(): Exports == Implementation where
 I ==> Integer
 RN ==> Fraction I
 SUP ==> SparseUnivariatePolynomial
 NNI ==> NonNegativeInteger

 Exports ==> with
```

```
bernoulli : I -> RN
  ++ \spad{bernoulli(n)} returns the nth Bernoulli number.
  ++ this is \spad{B(n,0)}, where \spad{B(n,x)} is the \spad{n}th Bernoulli
  ++ polynomial.
chineseRemainder: (I,I,I,I) -> I
  ++ \spad{chineseRemainder(x1,m1,x2,m2)} returns w, where w is such that
  ++ \spad{w = x1 mod m1} and \spad{w = x2 mod m2}. Note that \spad{m1} and
  ++ \spad{m2} must be relatively prime.
divisors : I -> List I
  ++ \spad{divisors(n)} returns a list of the divisors of n.
euler : I -> I
  ++ \spad{euler(n)} returns the \spad{n}th Euler number. This is
  ++ \spad{2^n E(n,1/2)}, where \spad{E(n,x)} is the nth Euler polynomial.
eulerPhi : I -> I
  ++ \spad{eulerPhi(n)} returns the number of integers between 1 and n
  ++ (including 1) which are relatively prime to n. This is the Euler phi
  ++ function \spad{\phi(n)} is also called the totient function.
fibonacci : I -> I
  ++ \spad{fibonacci(n)} returns the nth Fibonacci number. the Fibonacci
  ++ numbers \spad{F[n]} are defined by \spad{F[0] = F[1] = 1} and
  ++ \spad{F[n] = F[n-1] + F[n-2]}.
  ++ The algorithm has running time \spad{O(log(n)^3)}.
  ++ Reference: Knuth, The Art of Computer Programming
  ++ Vol 2, Semi-Numerical Algorithms.
harmonic : I -> RN
  ++ \spad{harmonic(n)} returns the nth harmonic number. This is
  ++ \spad{H[n] = sum(1/k,k=1..n)}.
jacobi : (I,I) -> I
  ++ \spad{jacobi(a,b)} returns the Jacobi symbol \spad{J(a/b)}.
  ++ When b is odd, \spad{J(a/b) = product(L(a/p) for p in factor b )}.
  ++ Note that by convention, 0 is returned if \spad{gcd(a,b) ^= 1}.
  ++ Iterative \spad{O(log(b)^2)} version coded by Michael Monagan June 1987.
legendre : (I,I) -> I
  ++ \spad{legendre(a,p)} returns the Legendre symbol \spad{L(a/p)}.
  ++ \spad{L(a/p) = (-1)**((p-1)/2) mod p} (p prime), which is 0 if \spad{a}
  ++ is 0, 1 if \spad{a} is a quadratic residue \spad{mod p} and -1 otherwise.
  ++ Note that because the primality test is expensive,
  ++ if it is known that p is prime then use \spad{jacobi(a,p)}.
moebiusMu : I -> I
  ++ \spad{moebiusMu(n)} returns the Moebius function \spad{mu(n)}.
  ++ \spad{mu(n)} is either -1,0 or 1 as follows:
  ++ \spad{mu(n) = 0} if n is divisible by a square > 1,
  ++ \spad{mu(n) = (-1)^k} if n is square-free and has k distinct
  ++ prime divisors.
numberOfDivisors: I -> I
  ++ \spad{numberOfDivisors(n)} returns the number of integers between 1 and n
  ++ (inclusive) which divide n. The number of divisors of n is often
  ++ denoted by \spad{tau(n)}.
sumOfDivisors : I -> I
  ++ \spad{sumOfDivisors(n)} returns the sum of the integers between 1 and n
```

```
    ++ (inclusive) which divide n. The sum of the divisors of n is often
    ++ denoted by \spad{sigma(n)}.
  sumOfKthPowerDivisors: (I,NNI) -> I
    ++ \spad{sumOfKthPowerDivisors(n,k)} returns the sum of the \spad{k}th
    ++ powers of the integers between 1 and n (inclusive) which divide n.
    ++ the sum of the \spad{k}th powers of the divisors of n is often denoted
    ++ by \spad{sigma_k(n)}.
 Implementation ==> add
  import IntegerPrimesPackage(I)

  -- we store the euler and bernoulli numbers computed so far in
  -- a Vector because they are computed from an n-term recurrence
  E: IndexedFlexibleArray(I,0)   := new(1, 1)
  B: IndexedFlexibleArray(RN,0)  := new(1, 1)
  H: Record(Hn:I,Hv:RN) := [1, 1]

  harmonic n ==
    s:I; h:RN
    n < 0 => error("harmonic not defined for negative integers")
    if n >= H.Hn then (s,h) := H else (s := 0; h := 0)
    for k in s+1..n repeat h := h + 1/k
    H.Hn := n
    H.Hv := h
    h

  fibonacci n ==
    n = 0 => 0
    n < 0 => (odd? n => 1; -1) * fibonacci(-n)
    f1, f2 : I
    (f1,f2) := (0,1)
    for k in length(n)-2 .. 0 by -1 repeat
      t := f2**2
      (f1,f2) := (t+f1**2,t+2*f1*f2)
      if bit?(n,k) then (f1,f2) := (f2,f1+f2)
    f2

  euler n ==
    n < 0 => error "euler not defined for negative integers"
    odd? n => 0
    l := (#E) :: I
    n < l => E(n)
    concat_!(E, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(I,0))
    for i in 1 .. l by 2 repeat E(i) := 0
    -- compute E(i) i = l+2,l+4,...,n given E(j) j = 0,2,...,i-2
    t,e : I
    for i in l+1 .. n by 2 repeat
      t := e := 1
      for j in 2 .. i-2 by 2 repeat
        t := (t*(i-j+1)*(i-j+2)) quo (j*(j-1))
        e := e + t*E(j)
```

```
    E(i) := -e
  E(n)

bernoulli n ==
  n < 0 => error "bernoulli not defined for negative integers"
  odd? n =>
    n = 1 => -1/2
    0
  l := (#B) :: I
  n < l => B(n)
  concat_!(B, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(RN,0))
  for i in 1 .. l by 2 repeat B(i) := 0
  -- compute B(i) i = l+2,l+4,...,n given B(j) j = 0,2,...,i-2
  for i in l+1 .. n by 2 repeat
    t:I := 1
    b := (1-i)/2
    for j in 2 .. i-2 by 2 repeat
      t := (t*(i-j+2)*(i-j+3)) quo (j*(j-1))
      b := b + (t::RN) * B(j)
    B(i) := -b/((i+1)::RN)
  B(n)

inverse : (I,I) -> I

inverse(a,b) ==
  borg:I:=b
  c1:I := 1
  d1:I := 0
  while b ^= 0 repeat
    q:I := a quo b
    r:I := a-q*b
    (a,b):=(b,r)
    (c1,d1):=(d1,c1-q*d1)
  a ^= 1 => error("moduli are not relatively prime")
  positiveRemainder(c1,borg)

chineseRemainder(x1,m1,x2,m2) ==
  m1 < 0 or m2 < 0 => error "moduli must be positive"
  x1 := positiveRemainder(x1,m1)
  x2 := positiveRemainder(x2,m2)
  x1 + m1 * positiveRemainder(((x2-x1) * inverse(m1,m2)),m2)

jacobi(a,b) ==
  -- Revised by Clifton Williamson January 1989.
  -- Previous version returned incorrect answers when b was even.
  -- The formula J(a/b) = product ( L(a/p) for p in factor b) is only
  -- valid when b is odd (the Legendre symbol L(a/p) is not defined
  -- for p = 2).  When b is even, the Jacobi symbol J(a/b) is only
  -- defined for a = 0 or 1 (mod 4).  When a = 1 (mod 8),
  -- J(a/2) = +1 and when a = 5 (mod 8), we define J(a/2) = -1.
```

```
  -- Extending by multiplicativity, we have J(a/b) for even b and
  -- appropriate a.
  -- We also define J(a/1) = 1.
  -- The point of this is the following: if d is the discriminant of
  -- a quadratic field K and chi is the quadratic character for K,
  -- then J(d/n) = chi(n) for n > 0.
  -- Reference: Hecke, Vorlesungen ueber die Theorie der Algebraischen
  -- Zahlen.
  if b < 0 then b := -b
  b = 0 => error "second argument of jacobi may not be 0"
  b = 1 => 1
  even? b and positiveRemainder(a,4) > 1 =>
    error "J(a/b) not defined for b even and a = 2 or 3 (mod 4)"
  even? b and even? a => 0
  for k in 0.. while even? b repeat b := b quo 2
  j:I := (odd? k and positiveRemainder(a,8) = 5 => -1; 1)
  b = 1 => j
  a := positiveRemainder(a,b)
  -- assertion: 0 < a < b and odd? b
  while a > 1 repeat
    if odd? a then
      -- J(a/b) = J(b/a) (-1) ** (a-1)/2 (b-1)/2
      if a rem 4 = 3 and b rem 4 = 3 then j := -j
      (a,b) := (b rem a,a)
    else
      -- J(2*a/b) = J(a/b) (-1) (b**2-1)/8
      for k in 0.. until odd? a repeat a := a quo 2
      if odd? k and (b+2) rem 8 > 4 then j := -j
  a = 0 => 0
  j

legendre(a,p) ==
  prime? p => jacobi(a,p)
  error "characteristic of legendre must be prime"

eulerPhi n ==
  n = 0 => 0
  r : RN := 1
  for entry in factors factor n repeat
    r := ((entry.factor - 1) /$RN entry.factor) * r
  numer(n * r)

divisors n ==
  oldList : List Integer := [1]
  for f in factors factor n repeat
    newList : List Integer := oldList
    for k in 1..f.exponent repeat
      pow := f.factor ** k
      for m in oldList repeat
        newList := concat(pow * m,newList)
```

```
      oldList := newList
    sort((i1:Integer,i2:Integer):Boolean +-> i1 < i2,oldList)

  numberOfDivisors n ==
    n = 0 => 0
    */[1+entry.exponent for entry in factors factor n]

  sumOfDivisors n ==
    n = 0 => 0
    r : RN := */[(entry.factor**(entry.exponent::NNI + 1)-1)/
      (entry.factor-1) for entry in factors factor n]
    numer r

  sumOfKthPowerDivisors(n,k) ==
    n = 0 => 0
    r : RN := */[(entry.factor**(k*entry.exponent::NNI+k)-1)/
      (entry.factor**k-1) for entry in factors factor n]
    numer r

  moebiusMu n ==
    n = 1 => 1
    t := factor n
    for k in factors t repeat
      k.exponent > 1 => return 0
    odd? numberOfFactors t => -1
    1
```

———-

— INTHEORY.dotabb —

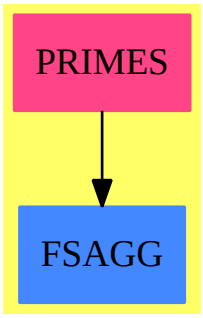
```
"INTHEORY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTHEORY"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"INTHEORY" -> "A1AGG"
```

———-

10.28 package PRIMES IntegerPrimesPackage

We've expanded the list of small primes to include those between 1 and 10000.

10.28.1 IntegerPrimesPackage (PRIMES)



Exports:
nextPrime prevPrime prime? primes

— package PRIMES IntegerPrimesPackage —

```
)abbrev package PRIMES IntegerPrimesPackage
++ Author: Michael Monagan
++ Date Created: August 1987
++ Date Last Updated: 31 May 1993
++ Updated by: James Davenport
++ Updated Because: of problems with strong pseudo-primes
++   and for some efficiency reasons.
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: integer, prime
++ Examples:
++ References: Davenport's paper in ISSAC 1992
++             AXIOM Technical Report ATR/6
++ Description:
++ The \spadtype{IntegerPrimesPackage} implements a modification of
++ Rabin's probabilistic
++ primality test and the utility functions \spadfun{nextPrime},
++ \spadfun{prevPrime} and \spadfun{primes}.

IntegerPrimesPackage(I:IntegerNumberSystem): with
   prime?: I -> Boolean
     ++ \spad{prime?(n)} returns true if n is prime and false if not.
     ++ The algorithm used is Rabin's probabilistic primality test
     ++ (reference: Knuth Volume 2 Semi Numerical Algorithms).
     ++ If \spad{prime? n} returns false, n is proven composite.
     ++ If \spad{prime? n} returns true, prime? may be in error
     ++ however, the probability of error is very low.
     ++ and is zero below 25*10**9 (due to a result of Pomerance et al),
```

```
    ++ below 10**12 and 10**13 due to results of Pinch,
    ++ and below 341550071728321 due to a result of Jaeschke.
    ++ Specifically, this implementation does at least 10 pseudo prime
    ++ tests and so the probability of error is \spad{< 4**(-10)}.
    ++ The running time of this method is cubic in the length
    ++ of the input n, that is \spad{O( (log n)**3 )}, for n<10**20.
    ++ beyond that, the algorithm is quartic, \spad{O( (log n)**4 )}.
    ++ Two improvements due to Davenport have been incorporated
    ++ which catches some trivial strong pseudo-primes, such as
    ++ [Jaeschke, 1991] 1377161253229053 * 413148375987157, which
    ++ the original algorithm regards as prime
  nextPrime: I -> I
    ++ \spad{nextPrime(n)} returns the smallest prime strictly larger than n
  prevPrime: I -> I
    ++ \spad{prevPrime(n)} returns the largest prime strictly smaller than n
  primes: (I,I) -> List I
    ++ \spad{primes(a,b)} returns a list of all primes p with
    ++ \spad{a <= p <= b}
== add
```

———-

10.28.2 smallPrimes

This is a table of all of the primes in [2..10000]. It is used by the prime? function to check for primality. It is used by the primes function to generate arrays of primes in a given range. Changing the range included in this table implies changing the value of the nextSmallPrime variable. There is a constant in the function squareFree from IntegerFactorizationPackage that is the square of the upper bound of the table range, in this case 10000000.

— package PRIMES IntegerPrimesPackage —

```
  smallPrimes: List I :=
    [2::I, 3::I, 5::I, 7::I, 11::I, 13::I, 17::I, 19::I,_
     23::I, 29::I, 31::I, 37::I, 41::I, 43::I, 47::I, 53::I,_
     59::I, 61::I, 67::I, 71::I, 73::I, 79::I, 83::I, 89::I,_
     97::I, 101::I, 103::I, 107::I, 109::I, 113::I, 127::I,_
     131::I, 137::I, 139::I, 149::I, 151::I, 157::I, 163::I,_
     167::I, 173::I, 179::I, 181::I, 191::I, 193::I, 197::I,_
     199::I, 211::I, 223::I, 227::I, 229::I, 233::I, 239::I,_
     241::I, 251::I, 257::I, 263::I, 269::I, 271::I, 277::I,_
     281::I, 283::I, 293::I, 307::I, 311::I, 313::I, 317::I,_
     331::I, 337::I, 347::I, 349::I, 353::I, 359::I, 367::I,_
     373::I, 379::I, 383::I, 389::I, 397::I, 401::I, 409::I,_
     419::I, 421::I, 431::I, 433::I, 439::I, 443::I, 449::I,_
     457::I, 461::I, 463::I, 467::I, 479::I, 487::I, 491::I,_
     499::I, 503::I, 509::I, 521::I, 523::I, 541::I, 547::I,_
     557::I, 563::I, 569::I, 571::I, 577::I, 587::I, 593::I,_
     599::I, 601::I, 607::I, 613::I, 617::I, 619::I, 631::I,_
```

```
641::I, 643::I, 647::I, 653::I, 659::I, 661::I, 673::I,_
677::I, 683::I, 691::I, 701::I, 709::I, 719::I, 727::I,_
733::I, 739::I, 743::I, 751::I, 757::I, 761::I, 769::I,_
773::I, 787::I, 797::I, 809::I, 811::I, 821::I, 823::I,_
827::I, 829::I, 839::I, 853::I, 857::I, 859::I, 863::I,_
877::I, 881::I, 883::I, 887::I, 907::I, 911::I, 919::I,_
929::I, 937::I, 941::I, 947::I, 953::I, 967::I, 971::I,_
977::I, 983::I, 991::I, 997::I, 1009::I, 1013::I,_
1019::I, 1021::I, 1031::I, 1033::I, 1039::I, 1049::I,_
1051::I, 1061::I, 1063::I, 1069::I, 1087::I, 1091::I,_
1093::I, 1097::I, 1103::I, 1109::I, 1117::I, 1123::I,_
1129::I, 1151::I, 1153::I, 1163::I, 1171::I, 1181::I,_
1187::I, 1193::I, 1201::I, 1213::I, 1217::I, 1223::I,_
1229::I, 1231::I, 1237::I, 1249::I, 1259::I, 1277::I,_
1279::I, 1283::I, 1289::I, 1291::I, 1297::I, 1301::I,_
1303::I, 1307::I, 1319::I, 1321::I, 1327::I, 1361::I,_
1367::I, 1373::I, 1381::I, 1399::I, 1409::I, 1423::I,_
1427::I, 1429::I, 1433::I, 1439::I, 1447::I, 1451::I,_
1453::I, 1459::I, 1471::I, 1481::I, 1483::I, 1487::I,_
1489::I, 1493::I, 1499::I, 1511::I, 1523::I, 1531::I,_
1543::I, 1549::I, 1553::I, 1559::I, 1567::I, 1571::I,_
1579::I, 1583::I, 1597::I, 1601::I, 1607::I, 1609::I,_
1613::I, 1619::I, 1621::I, 1627::I, 1637::I, 1657::I,_
1663::I, 1667::I, 1669::I, 1693::I, 1697::I, 1699::I,_
1709::I, 1721::I, 1723::I, 1733::I, 1741::I, 1747::I,_
1753::I, 1759::I, 1777::I, 1783::I, 1787::I, 1789::I,_
1801::I, 1811::I, 1823::I, 1831::I, 1847::I, 1861::I,_
1867::I, 1871::I, 1873::I, 1877::I, 1879::I, 1889::I,_
1901::I, 1907::I, 1913::I, 1931::I, 1933::I, 1949::I,_
1951::I, 1973::I, 1979::I, 1987::I, 1993::I, 1997::I,_
1999::I, 2003::I, 2011::I, 2017::I, 2027::I, 2029::I,_
2039::I, 2053::I, 2063::I, 2069::I, 2081::I, 2083::I,_
2087::I, 2089::I, 2099::I, 2111::I, 2113::I, 2129::I,_
2131::I, 2137::I, 2141::I, 2143::I, 2153::I, 2161::I,_
2179::I, 2203::I, 2207::I, 2213::I, 2221::I, 2237::I,_
2239::I, 2243::I, 2251::I, 2267::I, 2269::I, 2273::I,_
2281::I, 2287::I, 2293::I, 2297::I, 2309::I, 2311::I,_
2333::I, 2339::I, 2341::I, 2347::I, 2351::I, 2357::I,_
2371::I, 2377::I, 2381::I, 2383::I, 2389::I, 2393::I,_
2399::I, 2411::I, 2417::I, 2423::I, 2437::I, 2441::I,_
2447::I, 2459::I, 2467::I, 2473::I, 2477::I, 2503::I,_
2521::I, 2531::I, 2539::I, 2543::I, 2549::I, 2551::I,_
2557::I, 2579::I, 2591::I, 2593::I, 2609::I, 2617::I,_
2621::I, 2633::I, 2647::I, 2657::I, 2659::I, 2663::I,_
2671::I, 2677::I, 2683::I, 2687::I, 2689::I, 2693::I,_
2699::I, 2707::I, 2711::I, 2713::I, 2719::I, 2729::I,_
2731::I, 2741::I, 2749::I, 2753::I, 2767::I, 2777::I,_
2789::I, 2791::I, 2797::I, 2801::I, 2803::I, 2819::I,_
2833::I, 2837::I, 2843::I, 2851::I, 2857::I, 2861::I,_
2879::I, 2887::I, 2897::I, 2903::I, 2909::I, 2917::I,_
```

```
2927::I, 2939::I, 2953::I, 2957::I, 2963::I, 2969::I,_
2971::I, 2999::I, 3001::I, 3011::I, 3019::I, 3023::I,_
3037::I, 3041::I, 3049::I, 3061::I, 3067::I, 3079::I,_
3083::I, 3089::I, 3109::I, 3119::I, 3121::I, 3137::I,_
3163::I, 3167::I, 3169::I, 3181::I, 3187::I, 3191::I,_
3203::I, 3209::I, 3217::I, 3221::I, 3229::I, 3251::I,_
3253::I, 3257::I, 3259::I, 3271::I, 3299::I, 3301::I,_
3307::I, 3313::I, 3319::I, 3323::I, 3329::I, 3331::I,_
3343::I, 3347::I, 3359::I, 3361::I, 3371::I, 3373::I,_
3389::I, 3391::I, 3407::I, 3413::I, 3433::I, 3449::I,_
3457::I, 3461::I, 3463::I, 3467::I, 3469::I, 3491::I,_
3499::I, 3511::I, 3517::I, 3527::I, 3529::I, 3533::I,_
3539::I, 3541::I, 3547::I, 3557::I, 3559::I, 3571::I,_
3581::I, 3583::I, 3593::I, 3607::I, 3613::I, 3617::I,_
3623::I, 3631::I, 3637::I, 3643::I, 3659::I, 3671::I,_
3673::I, 3677::I, 3691::I, 3697::I, 3701::I, 3709::I,_
3719::I, 3727::I, 3733::I, 3739::I, 3761::I, 3767::I,_
3769::I, 3779::I, 3793::I, 3797::I, 3803::I, 3821::I,_
3823::I, 3833::I, 3847::I, 3851::I, 3853::I, 3863::I,_
3877::I, 3881::I, 3889::I, 3907::I, 3911::I, 3917::I,_
3919::I, 3923::I, 3929::I, 3931::I, 3943::I, 3947::I,_
3967::I, 3989::I, 4001::I, 4003::I, 4007::I, 4013::I,_
4019::I, 4021::I, 4027::I, 4049::I, 4051::I, 4057::I,_
4073::I, 4079::I, 4091::I, 4093::I, 4099::I, 4111::I,_
4127::I, 4129::I, 4133::I, 4139::I, 4153::I, 4157::I,_
4159::I, 4177::I, 4201::I, 4211::I, 4217::I, 4219::I,_
4229::I, 4231::I, 4241::I, 4243::I, 4253::I, 4259::I,_
4261::I, 4271::I, 4273::I, 4283::I, 4289::I, 4297::I,_
4327::I, 4337::I, 4339::I, 4349::I, 4357::I, 4363::I,_
4373::I, 4391::I, 4397::I, 4409::I, 4421::I, 4423::I,_
4441::I, 4447::I, 4451::I, 4457::I, 4463::I, 4481::I,_
4483::I, 4493::I, 4507::I, 4513::I, 4517::I, 4519::I,_
4523::I, 4547::I, 4549::I, 4561::I, 4567::I, 4583::I,_
4591::I, 4597::I, 4603::I, 4621::I, 4637::I, 4639::I,_
4643::I, 4649::I, 4651::I, 4657::I, 4663::I, 4673::I,_
4679::I, 4691::I, 4703::I, 4721::I, 4723::I, 4729::I,_
4733::I, 4751::I, 4759::I, 4783::I, 4787::I, 4789::I,_
4793::I, 4799::I, 4801::I, 4813::I, 4817::I, 4831::I,_
4861::I, 4871::I, 4877::I, 4889::I, 4903::I, 4909::I,_
4919::I, 4931::I, 4933::I, 4937::I, 4943::I, 4951::I,_
4957::I, 4967::I, 4969::I, 4973::I, 4987::I, 4993::I,_
4999::I, 5003::I, 5009::I, 5011::I, 5021::I, 5023::I,_
5039::I, 5051::I, 5059::I, 5077::I, 5081::I, 5087::I,_
5099::I, 5101::I, 5107::I, 5113::I, 5119::I, 5147::I,_
5153::I, 5167::I, 5171::I, 5179::I, 5189::I, 5197::I,_
5209::I, 5227::I, 5231::I, 5233::I, 5237::I, 5261::I,_
5273::I, 5279::I, 5281::I, 5297::I, 5303::I, 5309::I,_
5323::I, 5333::I, 5347::I, 5351::I, 5381::I, 5387::I,_
5393::I, 5399::I, 5407::I, 5413::I, 5417::I, 5419::I,_
5431::I, 5437::I, 5441::I, 5443::I, 5449::I, 5471::I,_
```

```
5477::I, 5479::I, 5483::I, 5501::I, 5503::I, 5507::I,_
5519::I, 5521::I, 5527::I, 5531::I, 5557::I, 5563::I,_
5569::I, 5573::I, 5581::I, 5591::I, 5623::I, 5639::I,_
5641::I, 5647::I, 5651::I, 5653::I, 5657::I, 5659::I,_
5669::I, 5683::I, 5689::I, 5693::I, 5701::I, 5711::I,_
5717::I, 5737::I, 5741::I, 5743::I, 5749::I, 5779::I,_
5783::I, 5791::I, 5801::I, 5807::I, 5813::I, 5821::I,_
5827::I, 5839::I, 5843::I, 5849::I, 5851::I, 5857::I,_
5861::I, 5867::I, 5869::I, 5879::I, 5881::I, 5897::I,_
5903::I, 5923::I, 5927::I, 5939::I, 5953::I, 5981::I,_
5987::I, 6007::I, 6011::I, 6029::I, 6037::I, 6043::I,_
6047::I, 6053::I, 6067::I, 6073::I, 6079::I, 6089::I,_
6091::I, 6101::I, 6113::I, 6121::I, 6131::I, 6133::I,_
6143::I, 6151::I, 6163::I, 6173::I, 6197::I, 6199::I,_
6203::I, 6211::I, 6217::I, 6221::I, 6229::I, 6247::I,_
6257::I, 6263::I, 6269::I, 6271::I, 6277::I, 6287::I,_
6299::I, 6301::I, 6311::I, 6317::I, 6323::I, 6329::I,_
6337::I, 6343::I, 6353::I, 6359::I, 6361::I, 6367::I,_
6373::I, 6379::I, 6389::I, 6397::I, 6421::I, 6427::I,_
6449::I, 6451::I, 6469::I, 6473::I, 6481::I, 6491::I,_
6521::I, 6529::I, 6547::I, 6551::I, 6553::I, 6563::I,_
6569::I, 6571::I, 6577::I, 6581::I, 6599::I, 6607::I,_
6619::I, 6637::I, 6653::I, 6659::I, 6661::I, 6673::I,_
6679::I, 6689::I, 6691::I, 6701::I, 6703::I, 6709::I,_
6719::I, 6733::I, 6737::I, 6761::I, 6763::I, 6779::I,_
6781::I, 6791::I, 6793::I, 6803::I, 6823::I, 6827::I,_
6829::I, 6833::I, 6841::I, 6857::I, 6863::I, 6869::I,_
6871::I, 6883::I, 6899::I, 6907::I, 6911::I, 6917::I,_
6947::I, 6949::I, 6959::I, 6961::I, 6967::I, 6971::I,_
6977::I, 6983::I, 6991::I, 6997::I, 7001::I, 7013::I,_
7019::I, 7027::I, 7039::I, 7043::I, 7057::I, 7069::I,_
7079::I, 7103::I, 7109::I, 7121::I, 7127::I, 7129::I,_
7151::I, 7159::I, 7177::I, 7187::I, 7193::I, 7207::I,_
7211::I, 7213::I, 7219::I, 7229::I, 7237::I, 7243::I,_
7247::I, 7253::I, 7283::I, 7297::I, 7307::I, 7309::I,_
7321::I, 7331::I, 7333::I, 7349::I, 7351::I, 7369::I,_
7393::I, 7411::I, 7417::I, 7433::I, 7451::I, 7457::I,_
7459::I, 7477::I, 7481::I, 7487::I, 7489::I, 7499::I,_
7507::I, 7517::I, 7523::I, 7529::I, 7537::I, 7541::I,_
7547::I, 7549::I, 7559::I, 7561::I, 7573::I, 7577::I,_
7583::I, 7589::I, 7591::I, 7603::I, 7607::I, 7621::I,_
7639::I, 7643::I, 7649::I, 7669::I, 7673::I, 7681::I,_
7687::I, 7691::I, 7699::I, 7703::I, 7717::I, 7723::I,_
7727::I, 7741::I, 7753::I, 7757::I, 7759::I, 7789::I,_
7793::I, 7817::I, 7823::I, 7829::I, 7841::I, 7853::I,_
7867::I, 7873::I, 7877::I, 7879::I, 7883::I, 7901::I,_
7907::I, 7919::I, 7927::I, 7933::I, 7937::I, 7949::I,_
7951::I, 7963::I, 7993::I, 8009::I, 8011::I, 8017::I,_
8039::I, 8053::I, 8059::I, 8069::I, 8081::I, 8087::I,_
8089::I, 8093::I, 8101::I, 8111::I, 8117::I, 8123::I,_
```

```
   8147::I, 8161::I, 8167::I, 8171::I, 8179::I, 8191::I,_
   8209::I, 8219::I, 8221::I, 8231::I, 8233::I, 8237::I,_
   8243::I, 8263::I, 8269::I, 8273::I, 8287::I, 8291::I,_
   8293::I, 8297::I, 8311::I, 8317::I, 8329::I, 8353::I,_
   8363::I, 8369::I, 8377::I, 8387::I, 8389::I, 8419::I,_
   8423::I, 8429::I, 8431::I, 8443::I, 8447::I, 8461::I,_
   8467::I, 8501::I, 8513::I, 8521::I, 8527::I, 8537::I,_
   8539::I, 8543::I, 8563::I, 8573::I, 8581::I, 8597::I,_
   8599::I, 8609::I, 8623::I, 8627::I, 8629::I, 8641::I,_
   8647::I, 8663::I, 8669::I, 8677::I, 8681::I, 8689::I,_
   8693::I, 8699::I, 8707::I, 8713::I, 8719::I, 8731::I,_
   8737::I, 8741::I, 8747::I, 8753::I, 8761::I, 8779::I,_
   8783::I, 8803::I, 8807::I, 8819::I, 8821::I, 8831::I,_
   8837::I, 8839::I, 8849::I, 8861::I, 8863::I, 8867::I,_
   8887::I, 8893::I, 8923::I, 8929::I, 8933::I, 8941::I,_
   8951::I, 8963::I, 8969::I, 8971::I, 8999::I, 9001::I,_
   9007::I, 9011::I, 9013::I, 9029::I, 9041::I, 9043::I,_
   9049::I, 9059::I, 9067::I, 9091::I, 9103::I, 9109::I,_
   9127::I, 9133::I, 9137::I, 9151::I, 9157::I, 9161::I,_
   9173::I, 9181::I, 9187::I, 9199::I, 9203::I, 9209::I,_
   9221::I, 9227::I, 9239::I, 9241::I, 9257::I, 9277::I,_
   9281::I, 9283::I, 9293::I, 9311::I, 9319::I, 9323::I,_
   9337::I, 9341::I, 9343::I, 9349::I, 9371::I, 9377::I,_
   9391::I, 9397::I, 9403::I, 9413::I, 9419::I, 9421::I,_
   9431::I, 9433::I, 9437::I, 9439::I, 9461::I, 9463::I,_
   9467::I, 9473::I, 9479::I, 9491::I, 9497::I, 9511::I,_
   9521::I, 9533::I, 9539::I, 9547::I, 9551::I, 9587::I,_
   9601::I, 9613::I, 9619::I, 9623::I, 9629::I, 9631::I,_
   9643::I, 9649::I, 9661::I, 9677::I, 9679::I, 9689::I,_
   9697::I, 9719::I, 9721::I, 9733::I, 9739::I, 9743::I,_
   9749::I, 9767::I, 9769::I, 9781::I, 9787::I, 9791::I,_
   9803::I, 9811::I, 9817::I, 9829::I, 9833::I, 9839::I,_
   9851::I, 9857::I, 9859::I, 9871::I, 9883::I, 9887::I,_
   9901::I, 9907::I, 9923::I, 9929::I, 9931::I, 9941::I,_
   9949::I, 9967::I, 9973::I]

productSmallPrimes    := */smallPrimes
nextSmallPrime        := 10007::I
nextSmallPrimeSquared := nextSmallPrime**2
two                   := 2::I
tenPowerTwenty:=(10::I)**20
PomeranceList:= [25326001::I, 161304001::I, 960946321::I, 1157839381::I,
                 -- 3215031751::I, -- has a factor of 151
                 3697278427::I, 5764643587::I, 6770862367::I,
                  14386156093::I, 15579919981::I, 18459366157::I,
                   19887974881::I, 21276028621::I ]::(List I)
PomeranceLimit:=27716349961::I  -- replaces (25*10**9) due to Pinch
PinchList:= _
  [3215031751::I, 118670087467::I, 128282461501::I, 354864744877::I,
   546348519181::I, 602248359169::I, 669094855201::I ]
```

```
PinchLimit:= (10**12)::I
PinchList2:= [2152302898747::I, 3474749660383::I]
PinchLimit2:= (10**13)::I
JaeschkeLimit:=341550071728321::I
rootsMinus1:Set I := empty()
-- used to check whether we detect too many roots of -1
count2Order:Vector NonNegativeInteger := new(1,0)
-- used to check whether we observe an element of maximal two-order
```

———-

10.28.3 primes

— package PRIMES IntegerPrimesPackage —

```
primes(m, n) ==
   -- computes primes from m to n inclusive using prime?
   l:List(I) :=
     m <= two => [two]
     empty()
   n < two or n < m => empty()
   if even? m then m := m + 1
   ll:List(I) := [k::I for k in
          convert(m)@Integer..convert(n)@Integer by 2 | prime?(k::I)]
   reverse_! concat_!(ll, l)

rabinProvesComposite : (I,I,I,I,NonNegativeInteger) -> Boolean
rabinProvesCompositeSmall : (I,I,I,I,NonNegativeInteger) -> Boolean
```

———-

10.28.4 rabinProvesCompositeSmall

— package PRIMES IntegerPrimesPackage —

```
rabinProvesCompositeSmall(p,n,nm1,q,k) ==
      -- probability n prime is > 3/4 for each iteration
      -- for most n this probability is much greater than 3/4
      t := powmod(p, q, n)
      -- neither of these cases tells us anything
      if not ((t = 1) or t = nm1) then
         for j in 1..k-1 repeat
```

```
            oldt := t
            t := mulmod(t, t, n)
            (t = 1) => return true
            -- we have squared someting not -1 and got 1
            t = nm1 =>
                leave
         not (t = nm1) => return true
      false
```

———-

10.28.5 rabinProvesComposite

— package PRIMES IntegerPrimesPackage —

```
rabinProvesComposite(p,n,nm1,q,k) ==
      -- probability n prime is > 3/4 for each iteration
      -- for most n this probability is much greater than 3/4
      t := powmod(p, q, n)
      -- neither of these cases tells us anything
      if t=nm1 then count2Order(1):=count2Order(1)+1
      if not ((t = 1) or t = nm1) then
         for j in 1..k-1 repeat
            oldt := t
            t := mulmod(t, t, n)
            (t = 1) => return true
            -- we have squared someting not -1 and got 1
            t = nm1 =>
                rootsMinus1:=union(rootsMinus1,oldt)
                count2Order(j+1):=count2Order(j+1)+1
                leave
         not (t = nm1) => return true
      # rootsMinus1 > 2 => true  -- Z/nZ can't be a field
      false
```

———-

10.28.6 prime?

— package PRIMES IntegerPrimesPackage —

```
prime? n ==
   n < two => false
```

```
n < nextSmallPrime => member?(n, smallPrimes)
not (gcd(n, productSmallPrimes) = 1) => false
n < nextSmallPrimeSquared => true

nm1 := n-1
q := (nm1) quo two
for k in 1.. while not odd? q repeat q := q quo two
-- q = (n-1) quo 2**k for largest possible k

n < JaeschkeLimit =>
    rabinProvesCompositeSmall(2::I,n,nm1,q,k) => return false
    rabinProvesCompositeSmall(3::I,n,nm1,q,k) => return false

    n < PomeranceLimit =>
        rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
        member?(n,PomeranceList) => return false
        true

    rabinProvesCompositeSmall(7::I,n,nm1,q,k) => return false
    n < PinchLimit =>
        rabinProvesCompositeSmall(10::I,n,nm1,q,k) => return false
        member?(n,PinchList) => return false
        true

    rabinProvesCompositeSmall(5::I,n,nm1,q,k) => return false
    rabinProvesCompositeSmall(11::I,n,nm1,q,k) => return false
    n < PinchLimit2 =>
        member?(n,PinchList2) => return false
        true

    rabinProvesCompositeSmall(13::I,n,nm1,q,k) => return false
    rabinProvesCompositeSmall(17::I,n,nm1,q,k) => return false
    true

rootsMinus1:= empty()
count2Order := new(k,0) -- vector of k zeroes

mn := minIndex smallPrimes
for i in mn+1..mn+10 repeat
    rabinProvesComposite(smallPrimes i,n,nm1,q,k) => return false
import IntegerRoots(I)
q > 1 and perfectSquare?(3*n+1) => false
((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare?(8*n+1) => false
-- Both previous tests from Damgard & Landrock
currPrime:=smallPrimes(mn+10)
probablySafe:=tenPowerTwenty
while count2Order(k) = 0 or n > probablySafe repeat
    currPrime := nextPrime currPrime
    probablySafe:=probablySafe*(100::I)
    rabinProvesComposite(currPrime,n,nm1,q,k) => return false
```

```
    true
```

———-

10.28.7 nextPrime

— package PRIMES IntegerPrimesPackage —

```
nextPrime n ==
   -- computes the first prime after n
   n < two => two
   if odd? n then n := n + two else n := n + 1
   while not prime? n repeat n := n + two
   n
```

———-

10.28.8 prevPrime

— package PRIMES IntegerPrimesPackage —

```
prevPrime n ==
   -- computes the first prime before n
   n < 3::I => error "no primes less than 2"
   n = 3::I => two
   if odd? n then n := n - two else n := n - 1
   while not prime? n repeat n := n - two
   n
```

———-

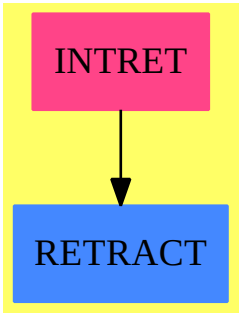
— PRIMES.dotabb —

```
"PRIMES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMES"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"PRIMES" -> "FSAGG"
```

———-

10.29 package INTRET IntegerRetractions

10.29.1 IntegerRetractions (INTRET)



Exports:
integer integer? integerIfCan

— package INTRET IntegerRetractions —

```
)abbrev package INTRET IntegerRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Provides integer testing and retraction functions.

IntegerRetractions(S:RetractableTo(Integer)): with
    integer     : S -> Integer
      ++ integer(x) returns x as an integer;
      ++ error if x is not an integer;
    integer?    : S -> Boolean
      ++ integer?(x) is true if x is an integer, false otherwise;
    integerIfCan: S -> Union(Integer, "failed")
      ++ integerIfCan(x) returns x as an integer,
      ++ "failed" if x is not an integer;
  == add
    integer s      == retract s
    integer? s     == retractIfCan(s) case Integer
    integerIfCan s == retractIfCan s
```

———-

— INTRET.dotabb —

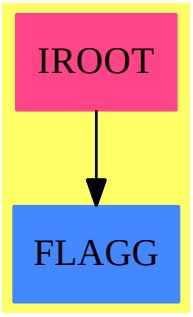
```
"INTRET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRET"]
```

```
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"INTRET" -> "RETRACT"
```

———-

10.30 package IROOT IntegerRoots

10.30.1 IntegerRoots (IROOT)



Exports:
approxSqrt approxNthRoot perfectNthPower? perfectNthRoot perfectSqrt perfectSquare?

— **package IROOT IntegerRoots** —

```
)abbrev package IROOT IntegerRoots
++ Author: Michael Monagan
++ Date Created: November 1987
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: integer roots
++ Examples:
++ References:
++ Description:
++ The \spadtype{IntegerRoots} package computes square roots and
++ nth roots of integers efficiently.

IntegerRoots(I:IntegerNumberSystem): Exports == Implementation where
  NNI ==> NonNegativeInteger

  Exports ==> with
    perfectNthPower?: (I, NNI) -> Boolean
```

```
    ++ \spad{perfectNthPower?(n,r)} returns true if n is an \spad{r}th
    ++ power and false otherwise
  perfectNthRoot: (I,NNI) -> Union(I,"failed")
    ++ \spad{perfectNthRoot(n,r)} returns the \spad{r}th root of n if n
    ++ is an \spad{r}th power and returns "failed" otherwise
  perfectNthRoot: I -> Record(base:I, exponent:NNI)
    ++ \spad{perfectNthRoot(n)} returns \spad{[x,r]}, where \spad{n = x\^r}
    ++ and r is the largest integer such that n is a perfect \spad{r}th power
  approxNthRoot: (I,NNI) -> I
    ++ \spad{approxRoot(n,r)} returns an approximation x
    ++ to \spad{n**(1/r)} such that \spad{-1 < x - n**(1/r) < 1}
  perfectSquare?: I -> Boolean
    ++ \spad{perfectSquare?(n)} returns true if n is a perfect square
    ++ and false otherwise
  perfectSqrt: I -> Union(I,"failed")
    ++ \spad{perfectSqrt(n)} returns the square root of n if n is a
    ++ perfect square and returns "failed" otherwise
  approxSqrt: I -> I
    ++ \spad{approxSqrt(n)} returns an approximation x
    ++ to \spad{sqrt(n)} such that \spad{-1 < x - sqrt(n) < 1}.
    ++ Compute an approximation s to \spad{sqrt(n)} such that
    ++           \spad{-1 < s - sqrt(n) < 1}
    ++ A variable precision Newton iteration is used.
    ++ The running time is \spad{O( log(n)**2 )}.

Implementation ==> add
  import IntegerPrimesPackage(I)

  resMod144: List I := [0::I,1::I,4::I,9::I,16::I,25::I,36::I,49::I,_
                  52::I,64::I,73::I,81::I,97::I,100::I,112::I,121::I]
  two := 2::I
```

———-

10.30.2 perfectSquare?

— package IROOT IntegerRoots —

```
  perfectSquare? a       == (perfectSqrt a) case I
```

———-

10.30.3 perfectNthPower?

— package IROOT IntegerRoots —

```
perfectNthPower?(b, n) == perfectNthRoot(b, n) case I
```

———-

10.30.4 perfectNthRoot

— package IROOT IntegerRoots —

```
perfectNthRoot n ==  -- complexity (log log n)**2 (log n)**2
  m:NNI
  (n = 1) or zero? n or n = -1 => [n, 1]
  e:NNI := 1
  p:NNI := 2
  while p::I <= length(n) + 1 repeat
     for m in 0.. while (r := perfectNthRoot(n, p)) case I repeat
        n := r::I
     e := e * p ** m
     p := convert(nextPrime(p::I))@Integer :: NNI
  [n, e]
```

———-

10.30.5 approxNthRoot

— package IROOT IntegerRoots —

```
approxNthRoot(a, n) ==   -- complexity (log log n) (log n)**2
  zero? n => error "invalid arguments"
  (n = 1) => a
  n=2 => approxSqrt a
  negative? a =>
    odd? n => - approxNthRoot(-a, n)
    0
  zero? a => 0
  (a = 1) => 1
  -- quick check for case of large n
  ((3*n) quo 2)::I >= (l := length a) => two
  -- the initial approximation must be >= the root
```

```
    y := max(two, shift(1, (n::I+l-1) quo (n::I)))
    z:I := 1
    n1:= (n-1)::NNI
    while z > 0 repeat
      x := y
      xn:= x**n1
      y := (n1*x*xn+a) quo (n*xn)
      z := x-y
    x
```

———-

10.30.6 perfectNthRoot

— package IROOT IntegerRoots —

```
perfectNthRoot(b, n) ==
  (r := approxNthRoot(b, n)) ** n = b => r
  "failed"
```

———-

10.30.7 perfectSqrt

— package IROOT IntegerRoots —

```
perfectSqrt a ==
  a < 0 or not member?(a rem (144::I), resMod144) => "failed"
  (s := approxSqrt a) * s = a => s
  "failed"
```

———-

10.30.8 approxSqrt

— package IROOT IntegerRoots —

```
approxSqrt a ==
  a < 1 => 0
  if (n := length a) > (100::I) then
```

```
      -- variable precision newton iteration
      n := n quo (4::I)
      s := approxSqrt shift(a, -2 * n)
      s := shift(s, n)
      return ((1 + s + a quo s) quo two)
   -- initial approximation for the root is within a factor of 2
   (new, old) := (shift(1, n quo two), 1)
   while new ^= old repeat
      (new, old) := ((1 + new + a quo new) quo two, new)
   new
```

———-

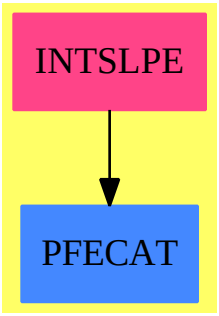
— **IROOT.dotabb** —

```
"IROOT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IROOT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"IROOT" -> "FLAGG"
```

———-

10.31 package INTSLPE IntegerSolveLinearPolynomialEquation

10.31.1 IntegerSolveLinearPolynomialEquation (INTSLPE)



Exports:
solveLinearPolynomialEquation

— **package INTSLPE IntegerSolveLinearPolynomialEquation** —

```
)abbrev package INTSLPE IntegerSolveLinearPolynomialEquation
```

```
++ Author: Davenport
++ Date Created: 1991
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides the implementation for the
++ \spadfun{solveLinearPolynomialEquation}
++ operation over the integers. It uses a lifting technique
++ from the package GenExEuclid

IntegerSolveLinearPolynomialEquation(): C ==T
 where
  ZP ==> SparseUnivariatePolynomial Integer
  C == with
      solveLinearPolynomialEquation: (List ZP,ZP) -> Union(List ZP,"failed")
           ++ solveLinearPolynomialEquation([f1, ..., fn], g)
           ++ (where the fi are relatively prime to each other)
           ++ returns a list of ai such that
           ++ \spad{g/prod fi = sum ai/fi}
           ++ or returns "failed" if no such list of ai's exists.
  T == add
      oldlp:List ZP := []
      slpePrime:Integer:=(2::Integer)
      oldtable:Vector List ZP := empty()
      solveLinearPolynomialEquation(lp,p) ==
         if (oldlp ^= lp) then
            -- we have to generate a new table
            deg:= _+/[degree u for u in lp]
            ans:Union(Vector List ZP,"failed"):="failed"
            slpePrime:=2147483647::Integer   -- 2**31 -1 : a prime
                 -- a good test case for this package is
                 --  ([x**31-1,x-2],2)
            while (ans case "failed") repeat
              ans:=tablePow(deg,slpePrime,lp)$GenExEuclid(Integer,ZP)
              if (ans case "failed") then
                 slpePrime:= prevPrime(slpePrime)$IntegerPrimesPackage(Integer)
            oldtable:=(ans:: Vector List ZP)
         answer:=solveid(p,slpePrime,oldtable)
         answer
```

———-

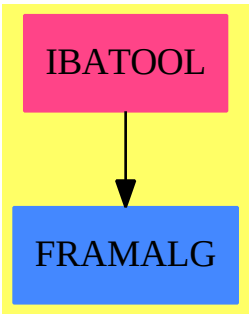
— INTSLPE.dotabb —

```
"INTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTSLPE" -> "PFECAT"
```

———-

10.32 package IBATOOL IntegralBasisTools

10.32.1 IntegralBasisTools (IBATOOL)



Exports:
diagonalProduct divideIfCan! idealiser idealiserMatrix leastPower matrixGcd moduleSum

— package IBATOOL IntegralBasisTools —

```
)abbrev package IBATOOL IntegralBasisTools
++ Author: Victor Miller, Barry Trager, Clifton Williamson
++ Date Created: 11 April 1990
++ Date Last Updated: 20 September 1994
++ Keywords: integral basis, function field, number field
++ Examples:
++ References:
++ Description:
++ This package contains functions used in the packages
++ FunctionFieldIntegralBasis and NumberFieldIntegralBasis.

IntegralBasisTools(R,UP,F): Exports == Implementation where
  R  : EuclideanDomain with
        squareFree: $ -> Factored $
          ++ squareFree(x) returns a square-free factorisation of x
  UP : UnivariatePolynomialCategory R
  F  : FramedAlgebra(R,UP)
  Mat ==> Matrix R
```

```
NNI ==> NonNegativeInteger
Ans ==> Record(basis: Mat, basisDen: R, basisInv:Mat)

Exports ==> with

  diagonalProduct: Mat -> R
    ++ diagonalProduct(m) returns the product of the elements on the
    ++ diagonal of the matrix m
  matrixGcd: (Mat,R,NNI) -> R
    ++ matrixGcd(mat,sing,n) is \spad{gcd(sing,g)} where \spad{g} is the
    ++ gcd of the entries of the \spad{n}-by-\spad{n} upper-triangular
    ++ matrix \spad{mat}.
  divideIfCan_!: (Matrix R,Matrix R,R,Integer) -> R
    ++ divideIfCan!(matrix,matrixOut,prime,n) attempts to divide the
    ++ entries of \spad{matrix} by \spad{prime} and store the result in
    ++ \spad{matrixOut}.  If it is successful, 1 is returned and if not,
    ++ \spad{prime} is returned.  Here both \spad{matrix} and
    ++ \spad{matrixOut} are \spad{n}-by-\spad{n} upper triangular matrices.
  leastPower: (NNI,NNI) -> NNI
    ++ leastPower(p,n) returns e, where e is the smallest integer
    ++ such that \spad{p **e >= n}
  idealiser: (Mat,Mat) -> Mat
    ++ idealiser(m1,m2) computes the order of an ideal defined by m1 and m2
  idealiser: (Mat,Mat,R) -> Mat
    ++ idealiser(m1,m2,d) computes the order of an ideal defined by m1 and m2
    ++ where d is the known part of the denominator
  idealiserMatrix: (Mat, Mat) -> Mat
    ++ idealiserMatrix(m1, m2) returns the matrix representing the linear
    ++ conditions on the Ring associatied with an ideal defined by m1 and m2.
  moduleSum: (Ans,Ans) -> Ans
    ++ moduleSum(m1,m2) returns the sum of two modules in the framed
    ++ algebra \spad{F}.  Each module \spad{mi} is represented as follows:
    ++ F is a framed algebra with R-module basis \spad{w1,w2,...,wn} and
    ++ \spad{mi} is a record \spad{[basis,basisDen,basisInv]}.  If
    ++ \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ a basis \spad{v1,...,vn} for \spad{mi} is given by
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of 'basis' contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
    ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
    ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.

Implementation ==> add
  import ModularHermitianRowReduction(R)
  import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

  diagonalProduct m ==
    ans : R := 1
```

```
      for i in minRowIndex m .. maxRowIndex m
        for j in minColIndex m .. maxColIndex m repeat
          ans := ans * qelt(m, i, j)
      ans

    matrixGcd(mat,sing,n) ==
      -- note that 'matrix' is upper triangular;
      -- no need to do anything below the diagonal
      d := sing
      for i in 1..n repeat
        for j in i..n repeat
          if not zero?(mij := qelt(mat,i,j)) then d := gcd(d,mij)
--          one? d => return d
          (d = 1) => return d
      d

    divideIfCan_!(matrix,matrixOut,prime,n) ==
    -- note that both 'matrix' and 'matrixOut' will be upper triangular;
    -- no need to do anything below the diagonal
      for i in 1..n repeat
        for j in i..n repeat
          (a := (qelt(matrix,i,j) exquo prime)) case "failed" => return prime
          qsetelt_!(matrixOut,i,j,a :: R)
      1

    leastPower(p,n) ==
      -- efficiency is not an issue here
      e : NNI := 1; q := p
      while q < n repeat (e := e + 1; q := q * p)
      e

    idealiserMatrix(ideal,idealinv) ==
      -- computes the Order of the ideal
      n  := rank()$F
      bigm := zero(n * n,n)$Mat
      mr := minRowIndex bigm; mc := minColIndex bigm
      v := basis()$F
      for i in 0..n-1 repeat
        r := regularRepresentation qelt(v,i + minIndex v)
        m := ideal * r * idealinv
        for j in 0..n-1 repeat
          for k in 0..n-1 repeat
            bigm(j * n + k + mr,i + mc) := qelt(m,j + mr,k + mc)
      bigm

    idealiser(ideal,idealinv) ==
      bigm := idealiserMatrix(ideal, idealinv)
      transpose squareTop rowEch bigm

    idealiser(ideal,idealinv,denom) ==
```

```
      bigm := (idealiserMatrix(ideal, idealinv) exquo denom)::Mat
      transpose squareTop rowEchelon(bigm,denom)

    moduleSum(mod1,mod2) ==
      rb1 := mod1.basis; rbden1 := mod1.basisDen; rbinv1 := mod1.basisInv
      rb2 := mod2.basis; rbden2 := mod2.basisDen; rbinv2 := mod2.basisInv
      -- compatibility check: doesn't take much computation time
      (not square? rb1) or (not square? rbinv1) or (not square? rb2) _
        or (not square? rbinv2) =>
        error "moduleSum: matrices must be square"
      ((n := nrows rb1) ^= (nrows rbinv1)) or (n ^= (nrows rb2)) _
        or (n ^= (nrows rbinv2)) =>
        error "moduleSum: matrices of imcompatible dimensions"
      (zero? rbden1) or (zero? rbden2) =>
        error "moduleSum: denominator must be non-zero"
      den := lcm(rbden1,rbden2); c1 := den quo rbden1; c2 := den quo rbden2
      rb := squareTop rowEchelon(vertConcat(c1 * rb1,c2 * rb2),den)
      rbinv := UpTriBddDenomInv(rb,den)
      [rb,den,rbinv]
```

———-

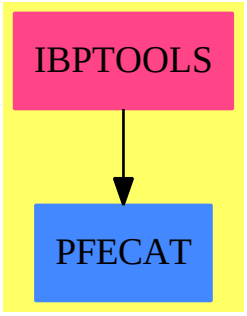
— IBATOOL.dotabb —

```
"IBATOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBATOOL"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"IBATOOL" -> "FRAMALG"
```

———-

10.33 package IBPTOOLS IntegralBasisPolynomialTools

10.33.1 IntegralBasisPolynomialTools (IBPTOOLS)



Exports:
mapBivariate mapMatrixIfCan mapUnivariate mapUnivariateIfCan

— package IBPTOOLS IntegralBasisPolynomialTools —

```
)abbrev package IBPTOOLS IntegralBasisPolynomialTools
++ Author: Clifton Williamson
++ Date Created: 13 August 1993
++ Date Last Updated: 17 August 1993
++ Basic Operations: mapUnivariate, mapBivariate
++ Related Domains: PAdicWildFunctionFieldIntegralBasis(K,R,UP,F)
++ Also See: WildFunctionFieldIntegralBasis, FunctionFieldIntegralBasis
++ AMS Classifications:
++ Keywords: function field, finite field, integral basis
++ Examples:
++ References:
++ Description:
++ IntegralBasisPolynomialTools provides functions for mapping functions
++ on the coefficients of univariate and bivariate polynomials.

IntegralBasisPolynomialTools(K,R,UP,L): Exports == Implementation where
  K  : Ring
  R  : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R
  L  : Ring

  MAT ==> Matrix
  SUP ==> SparseUnivariatePolynomial

  Exports ==> with
    mapUnivariate: (L -> K,SUP L) -> R
      ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
      ++ coefficients of \spad{p(x)}.

    mapUnivariate: (K -> L,R) -> SUP L
      ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
      ++ coefficients of \spad{p(x)}.

    mapUnivariateIfCan: (L -> Union(K,"failed"),SUP L) -> Union(R,"failed")
      ++ mapUnivariateIfCan(f,p(x)) applies the function \spad{f} to the
      ++ coefficients of \spad{p(x)}, if possible, and returns
      ++ \spad{"failed"} otherwise.

    mapMatrixIfCan: (L -> Union(K,"failed"),MAT SUP L) -> Union(MAT R,"failed")
      ++ mapMatrixIfCan(f,mat) applies the function \spad{f} to the
      ++ coefficients of the entries of \spad{mat} if possible, and returns
      ++ \spad{"failed"} otherwise.

    mapBivariate: (K -> L,UP) -> SUP SUP L
```

```
    ++ mapBivariate(f,p(x,y)) applies the function \spad{f} to the
    ++ coefficients of \spad{p(x,y)}.

  Implementation ==> add

    mapUnivariate(f:L -> K,poly:SUP L) ==
      ans : R := 0
      while not zero? poly repeat
        ans := ans + monomial(f leadingCoefficient poly,degree poly)
        poly := reductum poly
      ans

    mapUnivariate(f:K -> L,poly:R) ==
      ans : SUP L := 0
      while not zero? poly repeat
        ans := ans + monomial(f leadingCoefficient poly,degree poly)
        poly := reductum poly
      ans

    mapUnivariateIfCan(f,poly) ==
      ans : R := 0
      while not zero? poly repeat
        (lc := f leadingCoefficient poly) case "failed" => return "failed"
        ans := ans + monomial(lc :: K,degree poly)
        poly := reductum poly
      ans

    mapMatrixIfCan(f,mat) ==
      m := nrows mat; n := ncols mat
      matOut : MAT R := new(m,n,0)
      for i in 1..m repeat for j in 1..n repeat
        (poly := mapUnivariateIfCan(f,qelt(mat,i,j))) case "failed" =>
          return "failed"
        qsetelt_!(matOut,i,j,poly :: R)
      matOut

    mapBivariate(f,poly) ==
      ans : SUP SUP L := 0
      while not zero? poly repeat
        ans :=
          ans + monomial(mapUnivariate(f,leadingCoefficient poly),degree poly)
        poly := reductum poly
      ans
```

———-

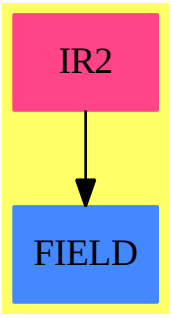
— IBPTOOLS.dotabb —

```
"IBPTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBPTOOLS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"IBPTOOLS" -> "PFECAT"
```

———-

10.34 package IR2 IntegrationResultFunctions2

10.34.1 IntegrationResultFunctions2 (IR2)



Exports:
map

— package IR2 IntegrationResultFunctions2 —

```
)abbrev package IR2 IntegrationResultFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 August 1992
++ Keywords: integration.
++ Description:
++ Internally used by the integration packages

IntegrationResultFunctions2(E, F): Exports == Implementation where
  E : Field
  F : Field

  SE  ==> Symbol
  Q   ==> Fraction Integer
  IRE ==> IntegrationResult E
  IRF ==> IntegrationResult F
  UPE ==> SparseUnivariatePolynomial E
  UPF ==> SparseUnivariatePolynomial F
  NEE ==> Record(integrand:E, intvar:E)
```

```
NEF ==> Record(integrand:F, intvar:F)
LGE ==> Record(scalar:Q, coeff:UPE, logand:UPE)
LGF ==> Record(scalar:Q, coeff:UPF, logand:UPF)
NLE ==> Record(coeff:E, logand:E)
NLF ==> Record(coeff:F, logand:F)
UFE ==> Union(Record(mainpart:E, limitedlogs:List NLE), "failed")
URE ==> Union(Record(ratpart:E, coeff:E), "failed")
UE  ==> Union(E, "failed")

Exports ==> with
  map: (E -> F, IRE) -> IRF
    ++ map(f,ire) \undocumented
  map: (E -> F, URE) -> Union(Record(ratpart:F, coeff:F), "failed")
    ++ map(f,ure) \undocumented
  map: (E -> F,  UE) -> Union(F, "failed")
    ++ map(f,ue) \undocumented
  map: (E -> F, UFE) ->
            Union(Record(mainpart:F, limitedlogs:List NLF), "failed")
    ++ map(f,ufe) \undocumented

Implementation ==> add
  import SparseUnivariatePolynomialFunctions2(E, F)

  NEE2F: (E -> F, NEE) -> NEF
  LGE2F: (E -> F, LGE) -> LGF
  NLE2F: (E -> F, NLE) -> NLF

  NLE2F(func, r)         == [func(r.coeff), func(r.logand)]
  NEE2F(func, n)         == [func(n.integrand), func(n.intvar)]
  map(func:E -> F, u:UE) == (u case "failed" => "failed"; func(u::E))

  map(func:E -> F, ir:IRE) ==
    mkAnswer(func ratpart ir, [LGE2F(func, f) for f in logpart ir],
                               [NEE2F(func, g) for g in notelem ir])

  map(func:E -> F, u:URE) ==
    u case "failed" => "failed"
    [func(u.ratpart), func(u.coeff)]

  map(func:E -> F, u:UFE) ==
    u case "failed" => "failed"
    [func(u.mainpart), [NLE2F(func, f) for f in u.limitedlogs]]

  LGE2F(func, lg) ==
    [lg.scalar, map(func, lg.coeff), map(func, lg.logand)]
```

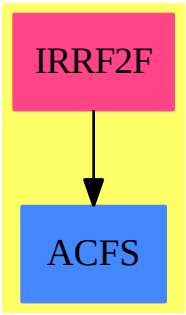
— **IR2.dotabb** —

```
"IR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IR2"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"IR2" -> "FIELD"
```

———-

10.35 package IRRF2F IntegrationResultRFToFunction

10.35.1 IntegrationResultRFToFunction (IRRF2F)



Exports:
complexExpand complexIntegrate expand integrate split

— **package IRRF2F IntegrationResultRFToFunction** —

```
)abbrev package IRRF2F IntegrationResultRFToFunction
++ Author: Manuel Bronstein
++ Date Created: 21 August 1988
++ Date Last Updated: 4 October 1993
++ Description:
++ Conversion of integration results to top-level expressions.
++ This package allows a sum of logs over the roots of a polynomial
++ to be expressed as explicit logarithms and arc tangents, provided
++ that the indexing polynomial can be factored into quadratics.

IntegrationResultRFToFunction(R): Exports == Implementation where
  R: Join(GcdDomain, RetractableTo Integer, OrderedSet,
          LinearlyExplicitRingOver Integer)

  RF  ==> Fraction Polynomial R
```

```
  F   ==> Expression R
  IR  ==> IntegrationResult RF

  Exports ==> with
    split           : IR -> IR
       ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
       ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + ... + sum_{Pn(a)=0} Q(a,x)}
       ++ where P1,...,Pn are the factors of P.
    expand          : IR -> List F
       ++ expand(i) returns the list of possible real functions
       ++ corresponding to i.
    complexExpand   : IR -> F
       ++ complexExpand(i) returns the expanded complex function
       ++ corresponding to i.
    if R has CharacteristicZero then
      integrate       : (RF, Symbol) -> Union(F, List F)
        ++ integrate(f, x) returns the integral of \spad{f(x)dx}
        ++ where x is viewed as a real variable..
      complexIntegrate: (RF, Symbol) -> F
        ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
        ++ where x is viewed as a complex variable.

  Implementation ==> add
    import IntegrationTools(R, F)
    import TrigonometricManipulations(R, F)
    import IntegrationResultToFunction(R, F)

    toEF: IR -> IntegrationResult F

    toEF i          == map(z1+->z1::F, i)$IntegrationResultFunctions2(RF, F)
    expand i        == expand toEF i
    complexExpand i == complexExpand toEF i

    split i ==
      map(retract, split toEF i)$IntegrationResultFunctions2(F, RF)

    if R has CharacteristicZero then
      import RationalFunctionIntegration(R)

      complexIntegrate(f, x) == complexExpand internalIntegrate(f, x)

-- do not use real integration if R is complex
      if R has imaginary: () -> R then integrate(f, x) == complexIntegrate(f, x)
      else
        integrate(f, x) ==
          l := [mkPrim(real g, x) for g in expand internalIntegrate(f, x)]
          empty? rest l => first l
          l
```

———-

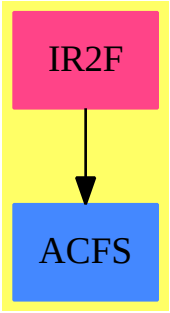
— **IRRF2F.dotabb** —

```
"IRRF2F" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRRF2F"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"IRRF2F" -> "ACFS"
```

———-

10.36 package IR2F IntegrationResultToFunction

10.36.1 IntegrationResultToFunction (IR2F)



Exports:
complexExpand expand split

— **package IR2F IntegrationResultToFunction** —

```
)abbrev package IR2F IntegrationResultToFunction
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 9 October 1991
++ Keywords: integration, expansion, function.
++ Description:
++ Conversion of integration results to top-level expressions
++ This package allows a sum of logs over the roots of a polynomial
++ to be expressed as explicit logarithms and arc tangents, provided
++ that the indexing polynomial can be factored into quadratics.

IntegrationResultToFunction(R, F): Exports == Implementation where
  R: Join(GcdDomain, RetractableTo Integer, OrderedSet,
          LinearlyExplicitRingOver Integer)
```

```
F: Join(AlgebraicallyClosedFunctionSpace R,
        TranscendentalFunctionCategory)

N   ==> NonNegativeInteger
Z   ==> Integer
Q   ==> Fraction Z
K   ==> Kernel F
P   ==> SparseMultivariatePolynomial(R, K)
UP  ==> SparseUnivariatePolynomial F
IR  ==> IntegrationResult F
REC ==> Record(ans1:F, ans2:F)
LOG ==> Record(scalar:Q, coeff:UP, logand:UP)

Exports ==> with
  split        : IR -> IR
     ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
     ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + ... + sum_{Pn(a)=0} Q(a,x)}
     ++ where P1,...,Pn are the factors of P.
  expand       : IR -> List F
     ++ expand(i) returns the list of possible real functions
     ++ corresponding to i.
  complexExpand: IR -> F
     ++ complexExpand(i) returns the expanded complex function
     ++ corresponding to i.

Implementation ==> add
  import AlgebraicManipulations(R, F)
  import ElementaryFunctionSign(R, F)

  IR2F       : IR -> F
  insqrt     : F  -> Record(sqrt:REC, sgn:Z)
  pairsum    : (List F, List F) -> List F
  pairprod   : (F, List F) -> List F
  quadeval   : (UP, F, F, F) -> REC
  linear     : (UP, UP) -> F
  tantrick   : (F, F) -> F
  ilog       : (F, F, List K) -> F
  ilog0      : (F, F, UP, UP, F) -> F
  nlogs      : LOG -> List LOG
  lg2func    : LOG -> List F
  quadratic  : (UP, UP) -> List F
  mkRealFunc : List LOG -> List F
  lg2cfunc   : LOG -> F
  loglist    : (Q, UP, UP) -> List LOG
  cmplex     : (F, UP) -> F
  evenRoots  : F -> List F
  compatible?: (List F, List F) -> Boolean

  cmplex(alpha, p) == alpha * log p alpha
  IR2F i           == retract mkAnswer(ratpart i, empty(), notelem i)
```

```
    pairprod(x, l)   == [x * y for y in l]

    evenRoots x ==
      [first argument k for k in tower x |
       is?(k,"nthRoot"::Symbol) and even?(retract(second argument k)@Z)
         and (not empty? variables first argument k)]

    expand i ==
      j := split i
      pairsum([IR2F j], mkRealFunc logpart j)

    split i ==
      mkAnswer(ratpart i,concat [nlogs l for l in logpart i],notelem i)

    complexExpand i ==
      j := split i
      IR2F j + +/[lg.scalar::F * lg2cfunc lg for lg in logpart j]

-- p = a t^2 + b t + c
-- Expands sum_{p(t) = 0} t log(lg(t))
    quadratic(p, lg) ==
      zero?(delta := (b := coefficient(p, 1))**2 - 4 *
       (a := coefficient(p,2)) * (p0 := coefficient(p, 0))) =>
         [linear(monomial(1, 1) + (b / a)::UP, lg)]
      e := (q := quadeval(lg, c := - b * (d := inv(2*a)),d, delta)).ans1
      lgp  := c * log(nrm := (e**2 - delta * (f := q.ans2)**2))
      s    := (sqr := insqrt delta).sqrt
      pp := nn := 0$F
      if sqr.sgn >= 0 then
        sqrp := s.ans1 * rootSimp sqrt(s.ans2)
        pp := lgp + d * sqrp * log(((2 * e * f) / nrm) * sqrp
                                      + (e**2 + delta * f**2) / nrm)
      if sqr.sgn <= 0 then
        sqrn := s.ans1 * rootSimp sqrt(-s.ans2)
        nn := lgp + d * sqrn * ilog(e, f * sqrn,
                    setUnion(setUnion(kernels a, kernels b), kernels p0))
      sqr.sgn > 0 => [pp]
      sqr.sgn < 0 => [nn]
      [pp, nn]

-- returns 2 atan(a/b) or 2 atan(-b/a) whichever looks better
-- they differ by a constant so it's ok to do it from an IR
    tantrick(a, b) ==
      retractIfCan(a)@Union(Q, "failed") case Q => 2 * atan(-b/a)
      2 * atan(a/b)

-- transforms i log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo's algorithm
-- lk is a list of kernels which are parameters for the integral
    ilog(a, b, lk) ==
```

```
      l := setDifference(setUnion(variables numer a, variables numer b),
           setUnion(lk, setUnion(variables denom a, variables denom b)))
      empty? l => tantrick(a, b)
      k := "max"/l
      ilog0(a, b, numer univariate(a, k), numer univariate(b, k), k::F)

-- transforms i log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo's algorithm
-- the arc-tangents will not have k in the denominator
-- we always keep upa(k) = a  and upb(k) = b
    ilog0(a, b, upa, upb, k) ==
      if degree(upa) < degree(upb) then
        (upa, upb) := (-upb, upa)
        (a, b) := (-b, a)
      zero? degree upb => tantrick(a, b)
      r := extendedEuclidean(upa, upb)
      (g:= retractIfCan(r.generator)@Union(F,"failed")) case "failed" =>
        tantrick(a, b)
      if degree(r.coef1) >= degree upb then
        qr := divide(r.coef1, upb)
        r.coef1 := qr.remainder
        r.coef2 := r.coef2 + qr.quotient * upa
      aa := (r.coef2) k
      bb := -(r.coef1) k
      tantrick(aa * a + bb * b, g::F) + ilog0(aa,bb,r.coef2,-r.coef1,k)

    lg2func lg ==
      zero?(d := degree(p := lg.coeff)) => error "poly has degree 0"
--      one? d => [linear(p, lg.logand)]
      (d = 1) => [linear(p, lg.logand)]
      d = 2  => quadratic(p, lg.logand)
      odd? d and
        ((r := retractIfCan(reductum p)@Union(F, "failed")) case F) =>
            pairsum([cmplex(alpha := rootSimp zeroOf p, lg.logand)],
                    lg2func [lg.scalar,
                     (p exquo (monomial(1, 1)$UP - alpha::UP))::UP,
                      lg.logand])
      [lg2cfunc lg]

    lg2cfunc lg ==
      +/[cmplex(alpha, lg.logand) for alpha in zerosOf(lg.coeff)]

    mkRealFunc l ==
      ans := empty()$List(F)
      for lg in l repeat
        ans := pairsum(ans, pairprod(lg.scalar::F, lg2func lg))
      ans

-- returns a log(b)
    linear(p, lg) ==
```

```
      alpha := - coefficient(p, 0) / coefficient(p, 1)
      alpha * log lg alpha

-- returns (c, d) s.t. p(a + b t) = c + d t, where t^2 = delta
    quadeval(p, a, b, delta) ==
      zero? p => [0, 0]
      bi := c := d := 0$F
      ai := 1$F
      v  := vectorise(p, 1 + degree p)
      for i in minIndex v .. maxIndex v repeat
        c    := c + qelt(v, i) * ai
        d    := d + qelt(v, i) * bi
        temp := a * ai + b * bi * delta
        bi   := a * bi + b * ai
        ai   := temp
      [c, d]

    compatible?(lx, ly) ==
      empty? ly => true
      for x in lx repeat
        for y in ly repeat
          ((s := sign(x*y)) case Z) and (s::Z < 0) => return false
      true

    pairsum(lx, ly) ==
      empty? lx => ly
      empty? ly => lx
      l := empty()$List(F)
      for x in lx repeat
        ls := evenRoots x
        if not empty?(ln :=
          [x + y for y in ly | compatible?(ls, evenRoots y)]) then
            l := removeDuplicates concat(l, ln)
      l

-- returns [[a, b], s] where sqrt(y) = a sqrt(b) and
-- s = 1 if b > 0, -1 if b < 0, 0 if the sign of b cannot be determined
    insqrt y ==
      rec := froot(y, 2)$PolynomialRoots(IndexedExponents K, K, R, P, F)
--      one?(rec.exponent) => [[rec.coef * rec.radicand, 1], 1]
      ((rec.exponent) = 1) => [[rec.coef * rec.radicand, 1], 1]
      rec.exponent ^=2 => error "Should not happen"
      [[rec.coef, rec.radicand],
          ((s := sign(rec.radicand)) case "failed" => 0; s::Z)]

    nlogs lg ==
      [[f.exponent * lg.scalar, f.factor, lg.logand] for f in factors
         ffactor(primitivePart(lg.coeff)
                    )$FunctionSpaceUnivariatePolynomialFactor(R, F, UP)]
```

———-

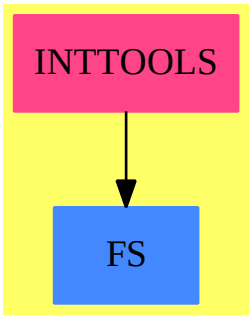
— **IR2F.dotabb** —

```
"IR2F" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IR2F"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"IR2F" -> "ACFS"
```

———-

10.37 package INTTOOLS IntegrationTools

10.37.1 IntegrationTools (INTTOOLS)



Exports:

kmax	intPatternMatch	ksec	mkPrim	removeConstantTerm
union	vark	varselect		

— **package INTTOOLS IntegrationTools** —

```
)abbrev package INTTOOLS IntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 25 April 1990
++ Date Last Updated: 9 June 1993
++ Keywords: elementary, function, integration.
++ Description:
++ Tools for the integrator

IntegrationTools(R:OrderedSet, F:FunctionSpace R): Exp == Impl where
  K   ==> Kernel F
  SE  ==> Symbol
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
```

```
IR  ==> IntegrationResult F
ANS ==> Record(special:F, integrand:F)
U   ==> Union(ANS, "failed")
ALGOP ==> "%alg"

Exp ==> with
  varselect: (List K, SE) -> List K
    ++ varselect([k1,...,kn], x) returns the ki which involve x.
  kmax     : List K -> K
    ++ kmax([k1,...,kn]) returns the top-level ki for integration.
  ksec     : (K, List K, SE) -> K
    ++ ksec(k, [k1,...,kn], x) returns the second top-level ki
    ++ after k involving x.
  union    : (List K, List K) -> List K
    ++ union(l1, l2) returns set-theoretic union of l1 and l2.
  vark     : (List F, SE) -> List K
    ++ vark([f1,...,fn],x) returns the set-theoretic union of
    ++ \spad{(varselect(f1,x),...,varselect(fn,x))}.
  if R has IntegralDomain then
    removeConstantTerm: (F, SE) -> F
      ++ removeConstantTerm(f, x) returns f minus any additive constant
      ++ with respect to x.
  if R has GcdDomain and F has ElementaryFunctionCategory then
    mkPrim: (F, SE) -> F
      ++ mkPrim(f, x) makes the logs in f which are linear in x
      ++ primitive with respect to x.
    if R has ConvertibleTo Pattern Integer and R has PatternMatchable Integer
      and F has LiouvillianFunctionCategory and F has RetractableTo SE then
        intPatternMatch: (F, SE, (F, SE) -> IR, (F, SE) -> U) -> IR
          ++ intPatternMatch(f, x, int, pmint) tries to integrate \spad{f}
          ++ first by using the integration function \spad{int}, and then
          ++ by using the pattern match intetgration function \spad{pmint}
          ++ on any remaining unintegrable part.

Impl ==> add
  better?: (K, K) -> Boolean

  union(l1, l2)   == setUnion(l1, l2)
  varselect(l, x) == [k for k in l | member?(x, variables(k::F))]
  ksec(k, l, x)   == kmax setUnion(remove(k, l), vark(argument k, x))

  vark(l, x) ==
    varselect(reduce("setUnion",[kernels f for f in l],empty()$List(K)), x)

  kmax l ==
    ans := first l
    for k in rest l repeat
      if better?(k, ans) then ans := k
    ans
```

```
-- true if x should be considered before y in the tower
    better?(x, y) ==
      height(y) ^= height(x) => height(y) < height(x)
      has?(operator y, ALGOP) or
              (is?(y, "exp"::SE) and not is?(x, "exp"::SE)
                                 and not has?(operator x, ALGOP))

    if R has IntegralDomain then
      removeConstantTerm(f, x) ==
        not freeOf?((den := denom f)::F, x) => f
        (u := isPlus(num := numer f)) case "failed" =>
          freeOf?(num::F, x) => 0
          f
        ans:P := 0
        for term in u::List(P) repeat
          if not freeOf?(term::F, x) then ans := ans + term
        ans / den

    if R has GcdDomain and F has ElementaryFunctionCategory then
      psimp     : (P, SE) -> Record(coef:Integer, logand:F)
      cont      : (P, List K) -> P
      logsimp   : (F, SE) -> F
      linearLog?: (K, F, SE)  -> Boolean

      logsimp(f, x) ==
        r1 := psimp(numer f, x)
        r2 := psimp(denom f, x)
        g := gcd(r1.coef, r2.coef)
        g * log(r1.logand ** (r1.coef quo g) / r2.logand ** (r2.coef quo g))

      cont(p, l) ==
        empty? l => p
        q := univariate(p, first l)
        cont(unitNormal(leadingCoefficient q).unit * content q, rest l)

      linearLog?(k, f, x) ==
        is?(k, "log"::SE) and
         ((u := retractIfCan(univariate(f,k))@Union(UP,"failed")) case UP)
--             and one?(degree(u::UP))
               and (degree(u::UP) = 1)
                  and not member?(x, variables leadingCoefficient(u::UP))

      mkPrim(f, x) ==
        lg := [k for k in kernels f | linearLog?(k, f, x)]
        eval(f, lg, [logsimp(first argument k, x) for k in lg])

      psimp(p, x) ==
        (u := isExpt(p := ((p exquo cont(p, varselect(variables p, x)))::P)))
          case "failed" => [1, p::F]
        [u.exponent, u.var::F]
```

```
      if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
        and F has Join(LiouvillianFunctionCategory, RetractableTo SE) then
          intPatternMatch(f, x, int, pmint) ==
            ir := int(f, x)
            empty?(l := notelem ir) => ir
            ans := ratpart ir
            nl:List(Record(integrand:F, intvar:F)) := empty()
            lg := logpart ir
            for rec in l repeat
              u := pmint(rec.integrand, retract(rec.intvar))
              if u case ANS then
                 rc := u::ANS
                 ans := ans + rc.special
                 if rc.integrand ^= 0 then
                   ir0 := intPatternMatch(rc.integrand, x, int, pmint)
                   ans := ans + ratpart ir0
                   lg  := concat(logpart ir0, lg)
                   nl  := concat(notelem ir0, nl)
              else nl := concat(rec, nl)
            mkAnswer(ans, lg, nl)
```

———-

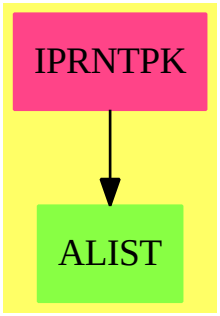
— INTTOOLS.dotabb —

```
"INTTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTTOOLS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTTOOLS" -> "FS"
```

———-

10.38 package IPRNTPK InternalPrintPackage

10.38.1 InternalPrintPackage (IPRNTPK)



Exports:
iprint

— package IPRNTPK InternalPrintPackage —

```
)abbrev package IPRNTPK InternalPrintPackage
++ Author: Themos Tsikas
++ Date Created: 09/09/1998
++ Date Last Updated: 09/09/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ A package to print strings without line-feed nor carriage-return.

InternalPrintPackage(): Exports == Implementation where

  Exports ==  with
     iprint: String -> Void
       ++ \axiom{iprint(s)} prints \axiom{s} at the current position
       ++ of the cursor.

  Implementation == add
     iprint(s:String) ==
          PRINC(coerce(s)@Symbol)$Lisp
          FORCE_-OUTPUT()$Lisp
```

———-

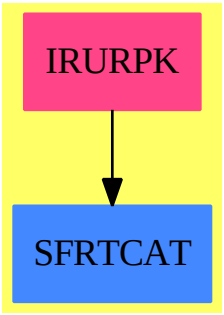
— IPRNTPK.dotabb —

```
"IPRNTPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IPRNTPK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"IPRNTPK" -> "ALIST"
```

———-

10.39 package IRURPK InternalRationalUnivariateRepresentationPackage

10.39.1 InternalRationalUnivariateRepresentationPackage (IRURPK)



Exports:
checkRur rur

— package IRURPK InternalRationalUnivariateRepresentationPackage —

```
)abbrev package IRURPK InternalRationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++  [1] D. LAZARD "Solving Zero-dimensional Algebraic Systems"
++      Journal of Symbolic Computation, 1992, 13, 117-131
++ Description:
```

```
++ An internal package for computing the rational univariate representation
++ of a zero-dimensional algebraic variety given by a square-free
++ triangular set. The main operation is rur

InternalRationalUnivariateRepresentationPackage(R,E,V,P,TS): Exports == Implementation where
  R : Join(EuclideanDomain,CharacteristicZero)
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LV ==> List V
  LP ==> List P
  PWT ==> Record(val: P, tower: TS)
  LPWT ==> Record(val: LP, tower: TS)
  WIP ==> Record(pol: P, gap: Z, tower: TS)
  BWT ==> Record(val:Boolean, tower: TS)
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  normpack ==> NormalizationPackage(R,E,V,P,TS)

  Exports ==  with

     rur: (TS,B) -> List TS
       ++ \spad{rur(ts,univ?)} returns a rational univariate representation
       ++ of \spad{ts}. This assumes that the lowest polynomial in \spad{ts}
       ++ is a variable \spad{v} which does not occur in the other polynomials
       ++ of \spad{ts}. This variable will be used to define the simple
       ++ algebraic extension over which these other polynomials will be
       ++ rewritten as univariate polynomials with degree one.
       ++ If \spad{univ?} is \spad{true} then these polynomials will have
       ++ a constant initial.
     checkRur: (TS, List TS) -> Boolean
       ++ \spad{checkRur(ts,lus)} returns \spad{true} if \spad{lus}
       ++ is a rational univariate representation of \spad{ts}.

  Implementation == add

     checkRur(ts: TS, lts: List TS): Boolean ==
       f0 := last(ts)::P
       z := mvar(f0)
       ts := collectUpper(ts,z)
       dts: N := degree(ts)
       lp := parts(ts)
       dlts: N := 0
       for us in lts repeat
         dlts := dlts + degree(us)
         rems := [removeZero(p,us) for p in lp]
         not every?(zero?,rems) =>
```

```
            output(us::OutputForm)$OutputPackage
            return false
        (dts =$N dlts)@Boolean

    convert(p:P,sqfr?:B):TS ==
      -- if sqfr? ASSUME p is square-free
      newts: TS := empty()
      sqfr? => internalAugment(p,newts)
      p := squareFreePart(p)
      internalAugment(p,newts)

    prepareRur(ts: TS): List LPWT ==
      not purelyAlgebraic?(ts)$TS =>
        error "rur$IRURPK: #1 is not zero-dimensional"
      lp: LP := parts(ts)$TS
      lp := sort(infRittWu?,lp)
      empty? lp =>
        error "rur$IRURPK: #1 is empty"
      f0 := first lp; lp := rest lp
--      not (one?(init(f0)) and one?(mdeg(f0)) and zero?(tail(f0))) =>
      not ((init(f0) = 1) and (mdeg(f0) = 1) and zero?(tail(f0))) =>
        error "rur$IRURPK: #1 has no generating root."
      empty? lp =>
        error "rur$IRURPK: #1 has a generating root but no indeterminates"
      z: V :=  mvar(f0)
      f1 := first lp; lp := rest lp
      x1: V := mvar(f1)
      newf1 := x1::P - z::P
      toSave: List LPWT := []
      for ff1 in irreducibleFactors([f1])$polsetpack repeat
        newf0 := eval(ff1,mvar(f1),f0)
        ts := internalAugment(newf1,convert(newf0,true)@TS)
        toSave := cons([lp,ts],toSave)
      toSave

    makeMonic(z:V,c:P,r:P,ts:TS,s:P,univ?:B): TS ==
      --ASSUME r is a irreducible univariate polynomial in z
      --ASSUME c and s only depends on z and mvar(s)
      --ASSUME c and a have main degree 1
      --ASSUME c and s have a constant initial
      --ASSUME mvar(ts) < mvar(s)
      lp: LP := parts(ts)
      lp := sort(infRittWu?,lp)
      newts: TS := convert(r,true)@TS
      s := remainder(s,newts).polnum
      if univ?
        then
          s := normalizedAssociate(s,newts)$normpack
      for p in lp repeat
        p := lazyPrem(eval(p,z,c),s)
```

```
        p := remainder(p,newts).polnum
        newts := internalAugment(p,newts)
      internalAugment(s,newts)

    next(lambda:Z):Z ==
      if lambda < 0 then lambda := - lambda + 1 else lambda := - lambda

    makeLinearAndMonic(p: P, xi: V, ts: TS, univ?:B, check?: B, info?: B): List TS ==
      -- if check? THEN some VERIFICATIONS are performed
      -- if info? THEN some INFORMATION is displayed
      f0 := last(ts)::P
      z: V := mvar(f0)
      lambda: Z := 1
      ts := collectUpper(ts,z)
      toSee: List WIP := [[f0,lambda,ts]$WIP]
      toSave: List TS := []
      while not empty? toSee repeat
        wip := first toSee; toSee := rest toSee
        (f0, lambda, ts) := (wip.pol, wip.gap, wip.tower)
        if check? and ((not univariate?(f0)$polsetpack) or (mvar(f0) ~= z))
          then
              output("Bad f0: ")$OutputPackage
              output(f0::OutputForm)$OutputPackage
        c: P := lambda * xi::P + z::P
        f := eval(f0,z,c); q := eval(p,z,c)
        prs := subResultantChain(q,f)
        r := first prs; prs := rest prs
        check? and ((not zero? degree(r,xi)) or (empty? prs)) =>
          error "rur$IRURPK: should never happen !"
        s := first prs; prs := rest prs
        check? and (zero? degree(s,xi)) and (empty? prs) =>
          error "rur$IRURPK: should never happen !!"
        if zero? degree(s,xi) then s := first prs
--        not one? degree(s,xi) =>
        not (degree(s,xi) = 1) =>
          toSee := cons([f0,next(lambda),ts]$WIP,toSee)
        h := init(s)
        r := squareFreePart(r)
        ground?(h) or ground?(gcd(h,r)) =>
          for fr in irreducibleFactors([r])$polsetpack repeat
            ground? fr => "leave"
            toSave := cons(makeMonic(z,c,fr,ts,s,univ?),toSave)
        if info?
          then
            output("Unlucky lambda")$OutputPackage
            output(h::OutputForm)$OutputPackage
            output(r::OutputForm)$OutputPackage
        toSee := cons([f0,next(lambda),ts]$WIP,toSee)
      toSave
```

```
    rur (ts: TS,univ?:Boolean): List TS ==
      toSee: List LPWT := prepareRur(ts)
      toSave: List TS := []
      while not empty? toSee repeat
        wip := first toSee; toSee := rest toSee
        ts: TS := wip.tower
        lp: LP := wip.val
        empty? lp => toSave := cons(ts,toSave)
        p := first lp; lp := rest lp
        xi: V := mvar(p)
        p := remainder(p,ts).polnum
        if not univ?
          then
            p := primitivePart stronglyReduce(p,ts)
        ground?(p) or (mvar(p) < xi) =>
          error "rur$IRUROK: should never happen"
--        (one? mdeg(p)) and (ground? init(p)) =>
        (mdeg(p) = 1) and (ground? init(p)) =>
          ts := internalAugment(p,ts)
          wip := [lp,ts]
          toSee := cons(wip,toSee)
        lts := makeLinearAndMonic(p,xi,ts,univ?,false,false)
        for ts in lts repeat
          wip := [lp,ts]
          toSee := cons(wip,toSee)
      toSave
```

———-

— IRURPK.dotabb —

```
"IRURPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRURPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"IRURPK" -> "SFRTCAT"
```

———-

10.40 package INTFRSP InterpolateFormsPackage

— InterpolateFormsPackage.input —

```
)set break resume
)sys rm -f InterpolateFormsPackage.output
)spool InterpolateFormsPackage.output
```

```
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterpolateFormsPackage
--R
--R InterpolateFormsPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,Ordered
--R Abbreviation for InterpolateFormsPackage is INTFRSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTFRSP
--R
--R------------------------------ Operations ------------------------------
--R basisOfInterpolateForms : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R basisOfInterpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R interpolateForms : (DIVISOR,NonNegativeInteger,PolyRing,List(PolyRing)) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R
--E 1

)spool
)lisp (bye)
```

———-

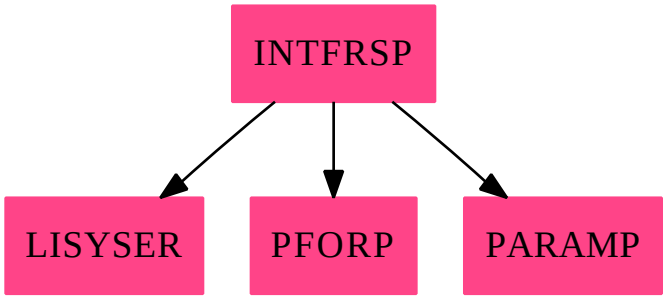
— InterpolateFormsPackage.help —

```
=====================================================================
InterpolateFormsPackage examples
=====================================================================

See Also:
o )show InterpolateFormsPackage
```

———-

10.40.1 InterpolateFormsPackage (INTFRSP)



Exports:

basisOfInterpolateForms	basisOfInterpolateFormsForFact
interpolateForms	interpolateFormsForFact

— package INTFRSP InterpolateFormsPackage —

```
)abbrev package INTFRSP InterpolateFormsPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
InterpolateFormsPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR):_
 Exports == Implementation where
  K:Field
  symb: List(Symbol)

  OV ==> OrderedVariableList(symb)
  E :  DirectProductCategory(#symb,NonNegativeInteger)

  PolyRing    : PolynomialCategory(K,E,OV)
  ProjPt      : ProjectiveSpaceCategory(K)
  PCS         : LocalPowerSeriesCategory(K)
  Plc         : PlacesCategory(K,PCS)
  DIVISOR     : DivisorCategory(Plc)

  INT       ==> Integer
  NNI       ==> NonNegativeInteger
  ParamPack ==>  ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
  PackPoly  ==>    PackageForPoly(K,PolyRing,E,#symb)
  LINPACK   ==> LinearSystemFromPowerSeriesPackage(K,PCS)

  Exports ==>  with

    basisOfInterpolateForms: (DIVISOR,List PolyRing)  -> List(Vector(K))
```

```
    basisOfInterpolateFormsForFact: (DIVISOR,List PolyRing)  -> List(Vector(K))

    interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

    interpolateForms: (DIVISOR,NNI,PolyRing,List PolyRing) -> List(PolyRing)
      ++ interpolateForms(D,n,pol,base) compute the basis of the sub-vector
      ++ space W of V = <base>, such that for all G in W, the
      ++ divisor (G) >= D. All the elements in base must be homogeneous
      ++ polynomial of degree n. Typicaly, base is the set of all monomial
      ++ of degree n: in that case, interpolateForms(D,n,pol,base)
      ++ returns the basis of the vector space of all forms of degree d that
      ++ interpolated D. The argument pol must be the same polynomial that
      ++ defined the curve form which the divisor D is defined.

  Implementation ==>  add

    import PolyRing
    import PCS

    sbSpcOfCurve: (NNI,PolyRing) -> List(List(K))

    exponant2monomial: List(NNI) -> PolyRing

    crtV: (List(K),List(INT),NNI) -> List(K)

    createLinSys: (List Plc, List INT,List PolyRing) -> Matrix(K)

    createLinSysWOVectorise: (List Plc, List INT,List PolyRing) -> Matrix(K)

    basisOfInterpolateFormsForFact(divis,lm)==
      -- permet d'intepoler un diviseur qui n'est pas rationnel.
      -- La partie non rationel
      -- est dans sptdiv (note: une place de sptdiv est une place qui identidie
      -- l'ensemble des places qui lui sont conjuguees.
      -- Note: On utilise ici la fonction createLinSysWOVectorise
      -- qui ne vectorise pas les elements du corps de base.
      lstOfPlc:= supp divis
      lstOfv:= [coefficient(pl,divis)  for pl in lstOfPlc]
      -- ppsol contiendra la base des formes interpolant ke diviseur divis
      linSys:Matrix(K)
      linSysT:Matrix(K)
      ll:List Matrix K
      ^empty?(lstOfPlc) =>
        linSys:=createLinSysWOVectorise(lstOfPlc,lstOfv,lm)
        nullSpace linSys
      zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
      nullSpace zeroMat

    interpolateForms(divis,d,laCrb,lm)==
```

```
      -- ppsol contiendra la base des formes interpolant le diviseur divis
      -- mieux vaut prendre  divOfZero de divis ?
      ppsol:= basisOfInterpolateForms(divis,lm)

      psol:List(List(K)):=[entries(vec) for vec in ppsol]
      mpsol:=psol
      sbspc:List(List(K))
      if ^(totalDegree(laCrb)$PackPoly > d) then
        -- retourne une base des formes de degres d
-- qui sont un multiple de la courbe
        sbspc:=sbSpcOfCurve(d,laCrb)
        mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)

      empty?(mpsol) => [0]

      rowEchmpsol:=rowEchelon(matrix(mpsol)$Matrix(K))
      npsol:=listOfLists(rowEchmpsol)
      [reduce("+",[a*f  for a in ll for f in lm]) for ll in npsol]

    interpolateFormsForFact(divis,lm)==
      -- ppsol contiendra la base des formes interpolant le diviseur divis
      ppsol:= basisOfInterpolateFormsForFact(divis,lm)
      psol:List(List(K)):=[entries(vec) for vec in ppsol]
      mpsol:=psol
      empty?(mpsol) => [0]
      rowEchmpsol:=rowEchelon matrix(mpsol)$Matrix(K)
      npsol:=listOfLists(rowEchmpsol)
      [reduce("+",[a*f  for a in ll for f in lm]) for ll in npsol]

    createLinSys(lstOfPlc,lstOfv,lm)==
        lplsT:=[ [parametrize(f,pl)$ParamPack  for f in lm]_
           for pl in lstOfPlc]
        lpls:=[[filterUpTo(s,v) for s in souslplsT] _
          for souslplsT in lplsT_
  for v in lstOfv]
        linSys:=reduce("vertConcat",_
           [finiteSeries2LinSys(souslplsT,v)$LINPACK_
          for souslplsT in lpls_
  for v in lstOfv])
        linSys

    createLinSysWOVectorise(lstOfPlc,lstOfv,lm)==
        lplsT:=[ [parametrize(f,pl)$ParamPack  for f in lm]_
           for pl in lstOfPlc]
        lpls:=[[filterUpTo(s,v) for s in souslplsT] _
          for souslplsT in lplsT_
  for v in lstOfv]
        linSys:=reduce("vertConcat",_
           [finiteSeries2LinSysWOVectorise(souslplsT,v)$LINPACK_
          for souslplsT in lpls_
```

```
  for v in lstOfv])
        linSys

    basisOfInterpolateForms(divis,lm)==
      lstOfPlc:= supp divis
      lstOfv:= [coefficient(pl,divis)  for pl in lstOfPlc]
      -- ppsol contiendra la base des formes interpolant ke diviseur divis
      linSys:Matrix(K)
      ^empty?(lstOfPlc) =>
        linSys:=createLinSys(lstOfPlc,lstOfv,lm)
        -- ppsol contient la base des formes passant par le diviseur divv
        nullSpace(linSys)
      zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
      nullSpace zeroMat


--    interpolateForms(divis,d,laCrb,lm)==
--      lstOfPlc:= supp divis
--      lstOfv:= [coefficient(pl,divis)  for pl in lstOfPlc]
--
--      lpls : List(List(PCS))
--      lplsT: List(List(PCS))
--
--      -- ppsol contiendra la base des formes interpolant ke diviseur divis
--      ppsol:List(Vector(K))
--      linSys:Matrix(K)
--      if ^empty?(lstOfPlc) then
--        linSys:=createLinSys(lstOfPlc,lstOfv,lm)
--
--        -- ppsol contient la base des formes passant par le diviseur divv
--        ppsol:=nullSpace(linSys)
--      else
--        zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
--        ppsol:=nullSpace zeroMat
--      mpsol:=psol:List(List(K)):=[entries(vec) for vec in ppsol]
--
--      if ^(totalDegree(laCrb) > d) then
--        -- retourne une base des formes de degres d
-- -- qui sont un multiple de la courbe
--        sbspc:=sbSpcOfCurve(d,laCrb)
--        mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
--
--      empty?(mpsol) => [0]
--
--      rowEchmpsol:=rowEchelon(matrix(mpsol))
--      npsol:=listOfLists(rowEchmpsol)
--      [reduce("+",[a*f  for a in ll for f in lm]) for ll in npsol]
--

--    interpolateForms(divis,d,laCrb,lm)==
```

```
--      lstOfPlc:= supp divis
--      lstOfv:= [coefficient(pl,divis)  for pl in lstOfPlc]
--
--      lpls : List(List(PCS))
--      lplsT: List(List(PCS))
--
--      -- ppsol contiendra la base des formes interpolant ke diviseur divis
--      ppsol:List(Vector(K))
--      linSys:Matrix(K)
--      if ^empty?(lstOfPlc) then
--
--        lplsT:=[ [parametrize(f,pl)$ParamPack  for f in lm]_
--            for pl in lstOfPlc]
--
--        lpls:=[[filterUpTo(s,v) for s in souslplsT] _
--           for souslplsT in lplsT_
--   for v in lstOfv]
--
--        linSys:=reduce("vertConcat",_
--            [finiteSeries2LinSys(souslplsT,v)$LINPACK_
--           for souslplsT in lpls_
--   for v in lstOfv])
--
--        -- ppsol contient la base des formes passant par le diviseur divv
--        ppsol:=nullSpace(linSys)
--      else
--        zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
--        ppsol:=nullSpace zeroMat
--      mpsol:=psol:List(List(K)):=[entries(vec) for vec in ppsol]
--
--      if ^(totalDegree(laCrb) > d) then
--        -- retourne une base des formes de degres d
-- -- qui sont un multiple de la courbe
--        sbspc:=sbSpcOfCurve(d,laCrb)
--        mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
--
--      empty?(mpsol) => [0]
--
--      rowEchmpsol:=rowEchelon(matrix(mpsol))
--      npsol:=listOfLists(rowEchmpsol)
--      [reduce("+",[a*f  for a in ll for f in lm]) for ll in npsol]

    listVar:List(OV):= [index(i::PositiveInteger)$OV for i in 1..#symb]

    listMonoPols:List(PolyRing):=[monomial(1,vv,1) for vv in listVar]

    crtV(lcoef,lpos,l)==
      vvv:List(K):=[0 for i in 1..l]
      for c in lcoef for p in lpos repeat
        setelt(vvv,p,c)
```

```
      vvv

    sbSpcOfCurve(m,laCrb)==
      d:=totalDegree(laCrb)$PackPoly
      lm:List(PolyRing):=listAllMono(m)$PackPoly
      m<d => [[0$K for i in 1..#lm]]
      sd:NNI:=((m pretend INT)-(d pretend INT)) pretend NNI
      slm:List(PolyRing):=listAllMono(sd)$PackPoly
      allPol:=[laCrb*f for f in slm]
      lpos:=[[position(m,lm) for m in primitiveMonomials(f)] for f in allPol]
      lcoef:=[coefficients(f) for f in allPol]
      clm:=#lm
      [crtV(lc,lp,clm) for lc in lcoef for lp in lpos]

    inVecSpace?: (List(K),List(List(K))) -> Boolean
    inVecSpace?(line,basis)==
      mat:Matrix(K):=matrix(basis)
      rmat:=rank(mat)
      augmat:Matrix(K):=matrix(concat(line,basis))
      raugmat:=rank(augmat)
      rmat=raugmat


    exponant2monomial(lexp)==
      reduce("*",[m**e for m in listMonoPols for e in lexp])

--    interpolateFunctions(lstOfPlc,lstOfv,lmnumer)==
----      lstOfPlc:= supp divis
----      lstOfv:= [coef(divis,pl) for pl in lstOfPlc]
--
--      lpls:List(List(PCS))
--      lplsT:List(List(PCS))
--      llll:List(List(Integer))
--      lOrd:List(Integer)
--      ordMin:Integer
--      ppsol:List(Vector(K))
--      linSys:Matrix(K)
--      if ^empty?(lstOfPlc) then
--        lplsT:=[[parametrize(f,pl)$ParamPack  for f in lmnumer ] _
--          for pl in lstOfPlc]
-- lplsT:=[[removeFirstZeroes(s) for s in l] for l in lplsT]
--
-- -- series must be shift if somme of them has negetive order
-- llll:= [[order(s)$PCS for s in l] for l in lplsT]
-- lOrd:=  concat llll
-- lOrd:=cons(0,lOrd)
-- ordMin:Integer:= "min"/lOrd
-- lplsT:=[[shift(s,-ordMin) for s in l] for l in lplsT]
--
--        lpls:=[[filterUpTo(s,v-ordMin) for s in souslplsT] _
```

```
--          for souslplsT in lplsT  for v in lstOfv]
--        linSys:=reduce("vertConcat",_
--          [finiteSeries2LinSys(souslplsT,v-ordMin)$LINPACK _
--            for souslplsT in lpls for v in lstOfv])
--        -- ppsol contient la base des formes passant par le diviseur divv
--        ppsol:=nullSpace(linSys)
--      else
--        zeroMat:Matrix(K):=zero(1,#lmnumer)$Matrix(K)
--        ppsol:=nullSpace zeroMat
--      mpsol:=psol:List(List(K)):=[entries(vec) for vec in ppsol]
--      -- inserer ici le code pour calculer la base modulo l'ideal ...
--      empty?(mpsol) => [0]
--      rowEchmpsol:=rowEchelon(matrix(mpsol))
--      npsol:=listOfLists(rowEchmpsol)
--      [reduce("+",[a*f  for a in ll for f in lmnumer]) for ll in npsol]
```

———-

— **INTFRSP.dotabb** —

```
"INTFRSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTFRSP"]
"LISYSER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LISYSER"]
"PFORP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFORP"]
"PARAMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARAMP"]
"INTFRSP" -> "LISYSER"
"INTFRSP" -> "PFORP"
"INTFRSP" -> "PARAMP"
```

———-

10.41 package INTDIVP IntersectionDivisorPackage

— **IntersectionDivisorPackage.input** —

```
)set break resume
)sys rm -f IntersectionDivisorPackage.output
)spool IntersectionDivisorPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntersectionDivisorPackage
--R
```

```
--R IntersectionDivisorPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,Orde
--R Abbreviation for IntersectionDivisorPackage is INTDIVP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTDIVP
--R
--R------------------------------ Operations ------------------------------
--R intersectionDivisor : (PolyRing,PolyRing,List(DesTree),List(ProjPt)) -> DIVISOR
--R placesOfDegree : (PositiveInteger,PolyRing,List(ProjPt)) -> Void
--R
--E 1

)spool
)lisp (bye)
```

———-

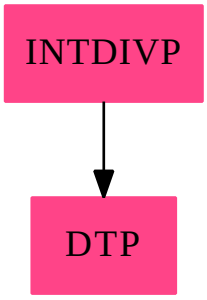
— **IntersectionDivisorPackage.help** —

```
=====================================================================
IntersectionDivisorPackage examples
=====================================================================

See Also:
o )show IntersectionDivisorPackage
```

———-

10.41.1 IntersectionDivisorPackage (INTDIVP)



Exports:
intersectionDivisor placesOfDegree

— **package INTDIVP IntersectionDivisorPackage** —

```
)abbreviation package INTDIVP IntersectionDivisorPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt, PCS,Plc,DIVISOR,_
                           InfClsPoint,DesTree,BLMET):_
 Exports == Implementation where

  K:Field
  symb: List(Symbol)

  OV ==> OrderedVariableList(symb)
  E  : DirectProductCategory(#symb,NonNegativeInteger)

  PolyRing    : PolynomialCategory(K,E,OV)
  ProjPt      : ProjectiveSpaceCategory(K)
  PCS         : LocalPowerSeriesCategory(K)
  Plc         : PlacesCategory(K,PCS)
  DIVISOR     : DivisorCategory(Plc)
  InfClsPoint : InfinitlyClosePointCategory(K,symb,PolyRing,E,ProjPt,_
                                            PCS,Plc,DIVISOR,BLMET)
  DesTree     : DesingTreeCategory(InfClsPoint)
  BLMET       : BlowUpMethodCategory

  OF          ==> OutputForm
  PackPoly    ==> PackageForPoly(K,PolyRing,E,#symb)
  PPFC1       ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
  ParamPackFC ==> LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_
                                                           E,ProjPt,PCS,Plc)
  ParamPack   ==> ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
  RatSingPack ==> ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
  DesingPack  ==> DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,_
                                    DIVISOR,InfClsPoint,DesTree,BLMET)

  Exports ==>  with

    intersectionDivisor:(PolyRing,PolyRing,List DesTree,List ProjPt) -> DIVISOR
      ++ intersectionDivisor(f,pol,listOfTree) returns the intersection
      ++ divisor of f with a curve defined by pol. listOfTree must contain
      ++ all the desingularisation trees of all singular points on the curve
      ++  defined by pol.

    placesOfDegree: (PositiveInteger, PolyRing, List ProjPt) -> Void()
      ++ placesOfDegree(d, f, pts) compute the places of degree
      ++ dividing d of the curve f. pts should be the singular points
      ++ of the curve f. For d > 1 this only works if K has
      ++ \axiomType{PseudoAlgebraicClosureOfFiniteFieldCategory}.
```

```
 Implementation ==>  add

   intersectionDivisor(pol,curve,ltr,listOfSingPt)==
      intDeg:Integer:=  (totalDegree(pol)$PackPoly * _
                         totalDegree(curve)$PackPoly) pretend Integer
      -- compute at places over singular Points
      lDivAtSingPt:DIVISOR:=_
        reduce("+",[divisorAtDesingTree(pol,tr)$DesingPack for tr in ltr],0)
      -- By Bezout Thorem, if all intersection points with mult.
      -- have been found then return the divisor
      degD:Integer:=degree lDivAtSingPt
      degD = intDeg  => lDivAtSingPt
      setOfFdPlc:List Plc:=foundPlaces()$Plc
      plcFrSplPts:List Plc:=[pl for pl in setOfFdPlc | ^leaf?(pl)]
      ordAtPlcFrSplPts:List Integer:=_
        [order(parametrize(pol,pl)$ParamPack)$PCS for pl in plcFrSplPts]
      divAtSplPts:DIVISOR:=_
        reduce("+",[o * (pl :: DIVISOR) _
          for o in ordAtPlcFrSplPts _
            for pl in plcFrSplPts],0)
      tDiv:=lDivAtSingPt+divAtSplPts
      -- By Bezout Thorem, if all intersection points with mult.
      -- have been found then return the divisor
      degD:Integer:=degree tDiv
      degD = intDeg  => tDiv
      intPts:List ProjPt:=algebraicSet([pol,curve])$RatSingPack
      intPtsNotSing:=setDifference(intPts,listOfSingPt)
      intPls:List(Plc):=_
        [pointToPlace(pt,curve)$ParamPackFC for pt in intPtsNotSing]
      remPlc:=setDifference(intPls , plcFrSplPts)
      ordAtPlcRem:List Integer:=_
        [order(parametrize(pol,pl)$ParamPack)$PCS for pl in remPlc]
      divAtRem:DIVISOR:=_
       reduce("+",[o*(pl :: DIVISOR) for o in ordAtPlcRem for pl in remPlc],0)
      theDivisor:= lDivAtSingPt +  divAtSplPts + divAtRem
      degD:Integer:=degree theDivisor
      if ^(degD = intDeg) then
        print("error while computing the intersection divisor" :: OF )
print("Otherwise the Bezout Theoreme is not true !!!! " :: OF)
print("Of course its the machine that make the mistake !!!!!" :: OF)
      theDivisor

   placesOfDegree(d, curve, singPts) ==
     --Return the number of places of degree i of the functionfield, no
     --constant field extension
     allPoints: List ProjPt:= rationalPoints(curve, d)$RatSingPack
     remindingSimplePts: List ProjPt :=setDifference(allPoints,singPts)
     for tpt in remindingSimplePts repeat
        pointToPlace(tpt,curve)$ParamPackFC
     Void()
```

———-

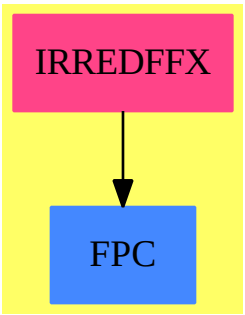
— **INTDIVP.dotabb** —

```
"INTDIVP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTDIVP"]
"DTP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DTP"]
"INTDIVP" -> "DTP"
```

———-

10.42 package IRREDFFX IrredPolyOverFiniteField

10.42.1 IrredPolyOverFiniteField (IRREDFFX)



Exports:
generateIrredPoly

— **package IRREDFFX IrredPolyOverFiniteField** —

```
)abbrev package IRREDFFX IrredPolyOverFiniteField
++ Author: Robert S. Sutor (original)
++ Date Created: ???
++ Date Last Updated: 29 May 1990
++ Description:
++ This package exports the function generateIrredPoly that computes
++ a monic irreducible polynomial of degree n over a finite field.

IrredPolyOverFiniteField(GF:FiniteFieldCategory): Exports == Impl where
  N    ==> PositiveInteger
  Z    ==> Integer
  SUP  ==> SparseUnivariatePolynomial GF
```

```
  QR   ==> Record(quotient: Z, remainder: Z)

  Exports ==> with
    generateIrredPoly: N -> SUP
      ++ generateIrredPoly(n) generates an irreducible univariate
      ++ polynomial of the given degree n over the finite field.

  Impl ==> add
    import DistinctDegreeFactorize(GF, SUP)

    getIrredPoly  : (Z, N) -> SUP
    qAdicExpansion: Z -> SUP

    p := characteristic()$GF :: N
    q := size()$GF :: N

    qAdicExpansion(z : Z): SUP ==
      -- expands z as a sum of powers of q, with coefficients in GF
      -- z = HornerEval(qAdicExpansion z,q)
      qr := divide(z, q)
      zero?(qr.remainder) => monomial(1, 1) * qAdicExpansion(qr.quotient)
      r := index(qr.remainder pretend N)$GF :: SUP
      zero?(qr.quotient) => r
      r + monomial(1, 1) * qAdicExpansion(qr.quotient)

    getIrredPoly(start : Z, n : N) : SUP ==
      -- idea is to iterate over possibly irreducible monic polynomials
      -- until we find an irreducible one. The obviously reducible ones
      -- are avoided.
      mon := monomial(1, n)$SUP
      pol: SUP := 0
      found: Boolean := false
      end: Z := q**n - 1
      while not ((end < start) or found) repeat
        if gcd(start, p) = 1 then
          if irreducible?(pol := mon + qAdicExpansion(start)) then
            found := true
        start := start + 1
      zero? pol => error "no irreducible poly found"
      pol

    generateIrredPoly(n : N) : SUP ==
      -- want same poly every time
--      one?(n) => monomial(1, 1)$SUP
      (n = 1) => monomial(1, 1)$SUP
--      one?(gcd(p, n)) or (n < q) =>
      (gcd(p, n) = 1) or (n < q) =>
        odd?(n) => getIrredPoly(2, n)
        getIrredPoly(1, n)
      getIrredPoly(q + 1, n)
```

———-

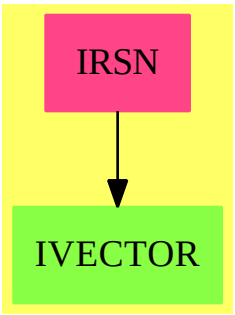
— **IRREDFFX.dotabb** —

```
"IRREDFFX" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRREDFFX"]
"FPC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FPC"]
"IRREDFFX" -> "FPC"
```

———-

10.43 package IRSN IrrRepSymNatPackage

10.43.1 IrrRepSymNatPackage (IRSN)



Exports:
dimensionOfIrreducibleRepresentation irreducibleRepresentation

— **package IRSN IrrRepSymNatPackage** —

```
)abbrev package IRSN IrrRepSymNatPackage
++ Authors: Johannes Grabmeier, Thorsten Werther
++ Date Created: 04 August 1988
++ Date Last Updated: 24 May 1991
++ Basic Operations: dimensionOfIrreducibleRepresentation
++   irreducibleRepresentation
++ Related Constructors: RepresentationTheoryPackage1
++   RepresentationTheoryPackage2
++ Also See: SymmetricGroupCombinatoricFunctions
++ AMS Classifications:
++ Keywords:
++ References:
```

```
++   G. James, A. Kerber: The Representation Theory of the Symmetric
++    Group. Encycl. of Math. and its Appl. Vol 16., Cambr. Univ Press 1981;
++   J. Grabmeier, A. Kerber: The Evaluation of Irreducible
++    Polynomial Representations of the General Linear Groups
++    and of the Unitary Groups over Fields of Characteristic 0,
++    Acta Appl. Math. 8 (1987), 271-291;
++   H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++    their Realization in the Computer Algebra System Scratchpad,
++    Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
++ Description:
++ IrrRepSymNatPackage contains functions for computing
++ the ordinary irreducible representations of symmetric groups on
++ n letters {1,2,...,n} in Young's natural form and their dimensions.
++ These representations can be labelled by number partitions of n,
++ i.e. a weakly decreasing sequence of integers summing up to n, e.g.
++ [3,3,3,1] labels an irreducible representation for n equals 10.
++ Note that whenever a \spadtype{List Integer} appears in a signature,
++ a partition required.
-- NOT TRUE in current system, but should:
-- also could be an element of \spadtype(Partition)

IrrRepSymNatPackage(): public == private where
  NNI   ==> NonNegativeInteger
  I     ==> Integer
  L     ==> List
  M     ==> Matrix
  V     ==> Vector
  B     ==> Boolean
  SGCF  ==> SymmetricGroupCombinatoricFunctions
  ICF   ==> IntegerCombinatoricFunctions Integer
  PP    ==> PartitionsAndPermutations
  PERM  ==> Permutation

  public ==> with

    dimensionOfIrreducibleRepresentation  : L I -> NNI
      ++ dimensionOfIrreducibleRepresentation(lambda) is the dimension
      ++ of the ordinary irreducible representation of the symmetric group
      ++ corresponding to lambda.
      ++ Note that the Robinson-Thrall hook formula is implemented.
    irreducibleRepresentation : (L I, PERM I) -> M I
      ++ irreducibleRepresentation(lambda,pi) is the irreducible representation
      ++ corresponding to partition lambda in Young's natural form of the
      ++ permutation pi in the symmetric group, whose elements permute
      ++ {1,2,...,n}.
    irreducibleRepresentation : L I -> L M I
      ++ irreducibleRepresentation(lambda) is the list of the two
      ++ irreducible representations corresponding to the partition lambda
      ++ in Young's natural form for the following two generators
      ++ of the symmetric group, whose elements permute
```

```
    ++ {1,2,...,n}, namely (1 2) (2-cycle) and
    ++ (1 2 ... n) (n-cycle).
  irreducibleRepresentation : (L I, L PERM I)  -> L M I
    ++ irreducibleRepresentation(lambda,listOfPerm) is the list of the
    ++ irreducible representations corresponding to lambda
    ++ in Young's natural form for the list of permutations
    ++ given by listOfPerm.

private ==> add

  -- local variables
  oldlambda : L I  := nil$(L I)
  flambda   : NNI := 0           -- dimension of the irreducible repr.
  younglist : L M I := nil$(L M I)     -- list of all standard tableaus
  lprime    : L I  := nil$(L I)      -- conjugated partition of lambda
  n         : NNI := 0           -- concerning symmetric group S_n
  rows      : NNI := 0           -- # of rows of standard tableau
  columns   : NNI := 0           -- # of columns of standard tableau
  aId       : M I  := new(1,1,0)

  -- declaration of local functions

  aIdInverse : () -> Void
    -- computes aId, the inverse of the matrix
    -- (signum(k,l,id))_1 <= k,l <= flambda, where id
    -- denotes the identity permutation

  alreadyComputed? : L I -> Void
    -- test if the last calling of an exported function concerns
    -- the same partition lambda as the previous call

  listPermutation : PERM I -> L I   -- should be in Permutation
    -- converts a permutation pi into the list
    -- [pi(1),pi(2),..,pi(n)]

  signum : (NNI, NNI, L I) -> I
    -- if there exists a vertical permutation v of the tableau
    -- tl := pi o younglist(l) (l-th standard tableau)
    -- and a horizontal permutation h of the tableau
    -- tk := younglist(k) (k-th standard tableau)  such that
    --                v o tl = h o tk,
    -- then
    --            signum(k,l,pi) = sign(v),
    -- otherwise
    --            signum(k,l,pi) = 0.

  sumPartition : L I -> NNI
    -- checks if lambda is a proper partition and results in
    -- the sum of the entries
```

```
testPermutation : L I -> NNI
  -- testPermutation(pi) checks if pi is an element of S_n,
  -- the set of permutations of the set {1,2,...,n}.
  -- If not, an error message will occur, if yes it replies n.


-- definition of local functions


aIdInverse() ==

  aId := new(flambda,flambda,0)
  for k in 1..flambda repeat
    aId(k,k) := 1
  if n < 5 then return aId

  idperm      : L I  := nil$(L I)
  for k in n..1 by -1 repeat
    idperm := cons(k,idperm)
  for k in 1..(flambda-1) repeat
    for l in (k+1)..flambda repeat
      aId(k::NNI,l::NNI) := signum(k::NNI,l::NNI,idperm)

  -- invert the upper triangular matrix aId
  for j in flambda..2 by -1 repeat
    for i in (j-1)..1 by -1 repeat
      aId(i::NNI,j:NNI) := -aId(i::NNI,j::NNI)
    for k in (j+1)..flambda repeat
      for i in (j-1)..1 by -1 repeat
        aId(i::NNI,k:NNI) := aId(i::NNI,k::NNI) +
            aId(i::NNI,j:NNI) * aId(j::NNI,k::NNI)


alreadyComputed?(lambda) ==
  if not(lambda = oldlambda) then
    oldlambda := lambda
    lprime    := conjugate(lambda)$PP
    rows      := (first(lprime)$(L I))::NNI
    columns   := (first(lambda)$(L I))::NNI
    n         := (+/lambda)::NNI
    younglist := listYoungTableaus(lambda)$SGCF
    flambda   := #younglist
    aIdInverse()        -- side effect: creates actual aId

listPermutation(pi) ==
  li : L I := nil$(L I)
  for k in n..1 by -1 repeat
    li := cons(eval(pi,k)$(PERM I),li)
  li
```

```
signum(numberOfRowTableau, numberOfColumnTableau,pi) ==

  rowtab : M I  := copy younglist numberOfRowTableau
  columntab : M I  := copy younglist numberOfColumnTableau
  swap : I
  sign : I   := 1
  end  : B   := false
  endk : B
  ctrl : B

  -- k-loop for all rows of tableau rowtab
  k    : NNI := 1
  while (k <= rows) and (not end) repeat
    -- l-loop along the k-th row of rowtab
    l : NNI := 1
    while (l <= oldlambda(k)) and (not end) repeat
      z : NNI := l
      endk := false
      -- z-loop for k-th row of rowtab beginning at column l.
      -- test wether the entry rowtab(k,z) occurs in the l-th column
      -- beginning at row k of pi o columntab
      while (z <= oldlambda(k)) and (not endk) repeat
        s : NNI := k
        ctrl := true
        while ctrl repeat
          if (s <= lprime(l))
            then
              if (1+rowtab(k,z) = pi(1+columntab(s,l)))
              -- if entries in the tableaus were from 1,..,n, then
              -- it should be ..columntab(s,l)... .
                then ctrl := false
                else s := s + 1
            else ctrl := false
        -- end of ctrl-loop
        endk := (s <= lprime(l)) -- same entry found ?
        if not endk
          then       -- try next entry
            z := z + 1
          else
            if k < s
              then     -- verticalpermutation
                sign := -sign
                swap := columntab(s,l)
                columntab(s,l) := columntab(k,l)
                columntab(k,l) := swap
            if l < z
              then     -- horizontalpermutation
                swap := rowtab(k,z)
                rowtab(k,z) := rowtab(k,l)
                rowtab(k,l) := swap
```

```
          -- end of else
      -- end of z-loop
      if (z > oldlambda(k))  -- no coresponding entry found
        then
          sign := 0
          end := true
      l := l + 1
    -- end of l-loop
    k := k + 1
  -- end of k-loop

  sign


sumPartition(lambda) ==
  ok   : B := true
  prev : I := first lambda
  sum  : I := 0
  for x in lambda repeat
    sum := sum + x
    ok := ok and (prev >= x)
    prev := x
  if not ok then
    error("No proper partition ")
  sum::NNI


testPermutation(pi : L I) : NNI ==
  ok : B := true
  n  : I := 0
  for i in pi repeat
    if i > n then n  := i     -- find the largest entry n in pi
    if i < 1 then ok := false -- check whether there are entries < 1
  -- now n should be the number of permuted objects
  if (not (n=#pi)) or (not ok) then
    error("No permutation of 1,2,..,n")
  -- now we know that pi has n Elements ranging from 1 to n
  test : Vector(B) := new((n)::NNI,false)
  for i in pi repeat
    test(i) := true   -- this means that i occurs in pi
  if member?(false,test) then error("No permutation")  -- pi not surjective
  n::NNI


-- definitions of exported functions


dimensionOfIrreducibleRepresentation(lambda) ==
  nn : I :=  sumPartition(lambda)::I --also checks whether lambda
  dd : I := 1                        --is a partition
```

```
  lambdaprime : L I := conjugate(lambda)$PP
  -- run through all rows of the Youngtableau corr. to lambda
  for i in 1..lambdaprime.1 repeat
    -- run through all nodes in row i of the Youngtableau
    for j in 1..lambda.i repeat
        -- the hooklength of node (i,j) of the Youngtableau
        -- is the new factor, remember counting starts with 1
        dd := dd * (lambda.i + lambdaprime.j - i - j + 1)
  (factorial(nn)$ICF quo dd)::NNI


irreducibleRepresentation(lambda:(L I),pi:(PERM I)) ==
  nn : NNI := sumPartition(lambda)
  alreadyComputed?(lambda)
  piList : L I := listPermutation pi
  if not (nn = testPermutation(piList)) then
    error("Partition and permutation are not consistent")
  aPi : M I := new(flambda,flambda,0)
  for k in 1..flambda repeat
    for l in 1..flambda repeat
      aPi(k,l) := signum(k,l,piList)
  aId * aPi


irreducibleRepresentation(lambda) ==
  listperm : L PERM I := nil$(L PERM I)
  li       : L I  := nil$(L I)
  sumPartition(lambda)
  alreadyComputed?(lambda)
  listperm :=
    n = 1 =>  cons(1$(PERM I),listperm)
    n = 2 =>  cons(cycle([1,2])$(PERM I),listperm)
    -- the n-cycle (1,2,..,n) and the 2-cycle (1,2) generate S_n
    for k in n..1 by -1 repeat
      li := cons(k,li)  -- becomes n-cycle (1,2,..,n)
    listperm := cons(cycle(li)$(PERM I),listperm)
    -- 2-cycle (1,2)
    cons(cycle([1,2])$(PERM I),listperm)
  irreducibleRepresentation(lambda,listperm)


irreducibleRepresentation(lambda:(L I),listperm:(L PERM I)) ==
  sumPartition(lambda)
  alreadyComputed?(lambda)
  [irreducibleRepresentation(lambda, pi) for pi in listperm]
```

———-

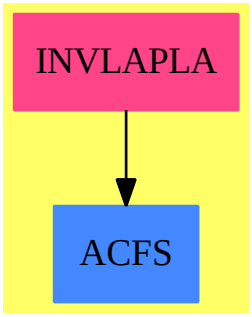
— **IRSN.dotabb** —

```
"IRSN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRSN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"IRSN" -> "IVECTOR"
```

———-

10.44 package INVLAPLA InverseLaplaceTransform

10.44.1 InverseLaplaceTransform (INVLAPLA)



Exports:
inverseLaplace

— **package INVLAPLA InverseLaplaceTransform** —

```
)abbrev package INVLAPLA InverseLaplaceTransform
++ Author: Barry Trager
++ Date Created: 3 Sept 1991
++ Date Last Updated: 3 Sept 1991
++ Description:
++ This package computes the inverse Laplace Transform.

InverseLaplaceTransform(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
           SpecialFunctionCategory, AlgebraicallyClosedFunctionSpace R)

  SE  ==> Symbol
  PI  ==> PositiveInteger
```

```
N   ==> NonNegativeInteger
K   ==> Kernel F
UP  ==> SparseUnivariatePolynomial F
RF  ==> Fraction UP

Exports ==> with
  inverseLaplace: (F, SE, SE) -> Union(F,"failed")
    ++ inverseLaplace(f, s, t) returns the Inverse
    ++ Laplace transform of \spad{f(s)}
    ++ using t as the new variable or "failed" if unable to find
    ++ a closed form.
    ++ Handles only rational \spad{f(s)}.

Implementation ==> add

  -- local ops --
  ilt : (F,Symbol,Symbol) -> Union(F,"failed")
  ilt1 : (RF,F) -> F
  iltsqfr : (RF,F) -> F
  iltirred: (UP,UP,F) -> F
  freeOf?: (UP,Symbol) -> Boolean

  inverseLaplace(expr,ivar,ovar) == ilt(expr,ivar,ovar)

  freeOf?(p:UP,v:Symbol) ==
    "and"/[freeOf?(c,v) for c in coefficients p]

  ilt(expr,var,t) ==
    expr = 0 => 0
    r := univariate(expr,kernel(var))

    -- Check that r is a rational function such that degree of
    -- the numarator is lower that degree of denominator
    not(numer(r) quo denom(r) = 0) => "failed"
    not( freeOf?(numer r,var) and freeOf?(denom r,var)) => "failed"

    ilt1(r,t::F)

  hintpac := TranscendentalHermiteIntegration(F, UP)

  ilt1(r,t) ==
    r = 0 => 0
    rsplit := HermiteIntegrate(r, differentiate)$hintpac
    -t*ilt1(rsplit.answer,t) + iltsqfr(rsplit.logpart,t)

  iltsqfr(r,t) ==
     r = 0 => 0
     p:=numer r
     q:=denom r
   --  ql := [qq.factor for qq in factors factor q]
```

```
      ql := [qq.factor for qq in factors squareFree q]
      # ql = 1 => iltirred(p,q,t)
      nl := multiEuclidean(ql,p)::List(UP)
      +/[iltirred(a,b,t) for a in nl for b in ql]

   -- q is irreducible, monic, degree p < degree q
   iltirred(p,q,t) ==
     degree q = 1 =>
       cp := coefficient(p,0)
       (c:=coefficient(q,0))=0 => cp
       cp*exp(-c*t)
     degree q = 2 =>
       a := coefficient(p,1)
       b := coefficient(p,0)
       c:=(-1/2)*coefficient(q,1)
       d:= coefficient(q,0)
       e := exp(c*t)
       b := b+a*c
       d := d-c**2
       d > 0 =>
           alpha:F := sqrt d
           e*(a*cos(t*alpha) + b*sin(t*alpha)/alpha)
       alpha :F := sqrt(-d)
       e*(a*cosh(t*alpha) + b*sinh(t*alpha)/alpha)
     roots:List F := zerosOf q
     q1 := differentiate q
     +/[p(root)/q1(root)*exp(root*t) for root in roots]
```

———-

— INVLAPLA.dotabb —

```
"INVLAPLA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INVLAPLA"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"INVLAPLA" -> "ACFS"
```

———-

Chapter 11

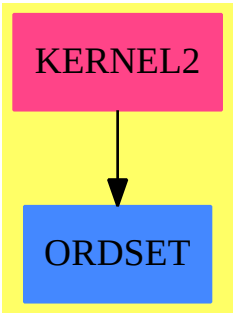
Chapter J

Chapter 12

Chapter K

12.1 package KERNEL2 KernelFunctions2

12.1.1 KernelFunctions2 (KERNEL2)



Exports:
constantKernel constantIfCan

— package KERNEL2 KernelFunctions2 —

```
)abbrev package KERNEL2 KernelFunctions2
++ Description:
++ This package exports some auxiliary functions on kernels

KernelFunctions2(R:OrderedSet, S:OrderedSet): with
  constantKernel: R -> Kernel S
    ++ constantKernel(r) \undocumented
  constantIfCan : Kernel S -> Union(R, "failed")
    ++ constantIfCan(k) \undocumented
```

```
== add
 import BasicOperatorFunctions1(R)

 constantKernel r == kernel(constantOperator r, nil(), 1)
 constantIfCan k  == constantOpIfCan operator k
```

———-

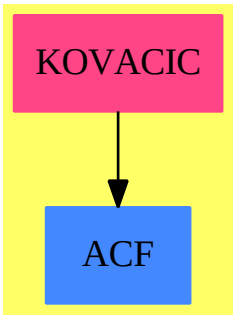
— **KERNEL2.dotabb** —

```
"KERNEL2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=KERNEL2"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"KERNEL2" -> "ORDSET"
```

———-

12.2 package KOVACIC Kovacic

12.2.1 Kovacic (KOVACIC)



Exports:
kovacic

— **package KOVACIC Kovacic** —

```
)abbrev package KOVACIC Kovacic
++ Author: Manuel Bronstein
++ Date Created: 14 January 1992
++ Date Last Updated: 3 February 1994
++ Keywords: differential equation, ODE
++ Description:
```

```
++ \spadtype{Kovacic} provides a modified Kovacic's algorithm for
++ solving explicitely irreducible 2nd order linear ordinary
++ differential equations.

Kovacic(F, UP): Exports == Impl where
  F  : Join(CharacteristicZero, AlgebraicallyClosedField,
            RetractableTo Integer, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

  RF  ==> Fraction UP
  SUP ==> SparseUnivariatePolynomial RF
  LF  ==> List Record(factor:UP, exponent:Integer)
  LODO==> LinearOrdinaryDifferentialOperator1 RF

  Exports ==> with
    kovacic: (RF, RF, RF) -> Union(SUP, "failed")
      ++ kovacic(a_0,a_1,a_2) returns either "failed" or P(u) such that
      ++ \spad{$e^{\int(-a_1/2a_2)} e^{\int u}$} is a solution of
      ++      \spad{a_2 y'' + a_1 y' + a0 y = 0}
      ++ whenever \spad{u} is a solution of \spad{P u = 0}.
      ++ The equation must be already irreducible over the rational functions.
    kovacic: (RF, RF, RF, UP -> Factored UP) -> Union(SUP, "failed")
      ++ kovacic(a_0,a_1,a_2,ezfactor) returns either "failed" or P(u) such
      ++ that \spad{$e^{\int(-a_1/2a_2)} e^{\int u}$} is a solution of
      ++      \spad{$a_2 y'' + a_1 y' + a0 y = 0$}
      ++ whenever \spad{u} is a solution of \spad{P u = 0}.
      ++ The equation must be already irreducible over the rational functions.
      ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
      ++ not necessarily into irreducibles.

  Impl ==> add
    import RationalRicDE(F, UP)

    case2       : (RF, LF, UP -> Factored UP) -> Union(SUP, "failed")
    cannotCase2?: LF -> Boolean

    kovacic(a0, a1, a2) == kovacic(a0, a1, a2, squareFree)

    -- it is assumed here that a2 y'' + a1 y' + a0 y is already irreducible
    -- over the rational functions, i.e. that the associated Riccati equation
    -- does NOT have rational solutions (so we don't check case 1 of Kovacic's
    -- algorithm)
    -- currently only check case 2, not 3
    kovacic(a0, a1, a2, ezfactor) ==
      -- transform first the equation to the form y'' = r y
      -- which makes the Galois group unimodular
      -- this does not change irreducibility over the rational functions
    -- the following is split into 5 lines in order to save a couple of
    -- hours of compile time.
      r:RF := a1**2
```

```
      r := r + 2 * a2 * differentiate a1
      r := r - 2 * a1 * differentiate a2
      r := r - 4 * a0 * a2
      r := r  / (4 * a2**2)
      lf := factors squareFree denom r
      case2(r, lf, ezfactor)

    -- this is case 2 of Kovacic's algorithm, i.e. look for a solution
    -- of the associated Riccati equation in a quadratic extension
    -- lf is the squarefree factorisation of denom(r) and is used to
    -- check the necessary condition
    case2(r, lf, ezfactor) ==
      cannotCase2? lf => "failed"
      -- build the symmetric square of the operator L = y'' - r y
      -- which is L2 = y''' - 4 r y' - 2 r' y
      l2:LODO := monomial(1, 3) - monomial(4*r, 1) - 2 * differentiate(r)::LODO
      -- no solution in this case if L2 has no rational solution
      empty?(sol := ricDsolve(l2, ezfactor)) => "failed"
      -- otherwise the defining polynomial for an algebraic solution
      -- of the Ricatti equation associated with L is
      -- u^2 - b u + (1/2 b' + 1/2 b^2 - r) = 0
      -- where b is a rational solution of the Ricatti of L2
      b := first sol
      monomial(1, 2)$SUP - monomial(b, 1)$SUP
                         + ((differentiate(b) + b**2 - 2 * r) / (2::RF))::SUP

    -- checks the necessary condition for case 2
    -- returns true if case 2 cannot have solutions
    -- the necessary condition is that there is either a factor with
    -- exponent 2 or odd exponent > 2
    cannotCase2? lf ==
      for rec in lf repeat
        rec.exponent = 2 or (odd?(rec.exponent) and rec.exponent > 2) =>
          return false
      true
```

———-

— KOVACIC.dotabb —

```
"KOVACIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=KOVACIC"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"KOVACIC" -> "ACF"
```

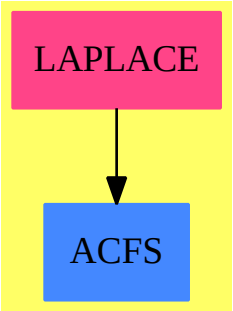
———-

Chapter 13

Chapter L

13.1 package LAPLACE LaplaceTransform

13.1.1 LaplaceTransform (LAPLACE)



Exports:
laplace

— package LAPLACE LaplaceTransform —

```
)abbrev package LAPLACE LaplaceTransform
++ Author: Manuel Bronstein
++ Date Created: 30 May 1990
++ Date Last Updated: 13 December 1995
++ Description:
++ This package computes the forward Laplace Transform.

LaplaceTransform(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
```

```
          RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
          AlgebraicallyClosedFunctionSpace R)

  SE  ==> Symbol
  PI  ==> PositiveInteger
  N   ==> NonNegativeInteger
  K   ==> Kernel F
  OFE ==> OrderedCompletion F
  EQ  ==> Equation OFE

  ALGOP       ==> "%alg"
  SPECIALDIFF ==> "%specialDiff"

  Exports ==> with
    laplace: (F, SE, SE) -> F
      ++ laplace(f, t, s) returns the Laplace transform of \spad{f(t)}
      ++ using \spad{s} as the new variable.
      ++ This is \spad{integral(exp(-s*t)*f(t), t = 0..%plusInfinity)}.
      ++ Returns the formal object \spad{laplace(f, t, s)} if it cannot
      ++ compute the transform.

  Implementation ==> add
    import IntegrationTools(R, F)
    import ElementaryIntegration(R, F)
    import PatternMatchIntegration(R, F)
    import PowerSeriesLimitPackage(R, F)
    import FunctionSpaceIntegration(R, F)
    import TrigonometricManipulations(R, F)

    locallaplace : (F, SE, F, SE, F) -> F
    lapkernel    : (F, SE, F, F) -> Union(F, "failed")
    intlaplace   : (F, F, F, SE, F) -> Union(F, "failed")
    isLinear     : (F, SE) -> Union(Record(const:F, nconst:F), "failed")
    mkPlus       : F -> Union(List F, "failed")
    dvlap        : (List F, SE) -> F
    tdenom       : (F, F) -> Union(F, "failed")
    atn          : (F, SE) -> Union(Record(coef:F, deg:PI), "failed")
    aexp         : (F, SE) -> Union(Record(coef:F, coef1:F, coef0:F), "failed")
    algebraic?   : (F, SE) -> Boolean

    oplap := operator("laplace"::Symbol, 3)$BasicOperator

    laplace(f,t,s) == locallaplace(complexElementary(f,t),t,t::F,s,s::F)

-- returns true if the highest kernel of f is algebraic over something
    algebraic?(f, t) ==
      l := varselect(kernels f, t)
      m:N := reduce(max, [height k for k in l], 0)$List(N)
      for k in l repeat
```

```
         height k = m and has?(operator k, ALGOP) => return true
      false

-- differentiate a kernel of the form  laplace(l.1,l.2,l.3) w.r.t x.
-- note that x is not necessarily l.3
-- if x = l.3, then there is no use recomputing the laplace transform,
-- it will remain formal anyways
    dvlap(l, x) ==
      l1 := first l
      l2 := second l
      x = (v := retract(l3 := third l)@SE) => - oplap(l2 * l1, l2, l3)
      e := exp(- l3 * l2)
      locallaplace(differentiate(e * l1, x) / e, retract(l2)@SE, l2, v, l3)

-- returns [b, c] iff f = c * t + b
-- and b and c do not involve t
    isLinear(f, t) ==
      ff := univariate(f, kernel(t)@K)
      ((d := retractIfCan(denom ff)@Union(F, "failed")) case "failed")
        or (degree(numer ff) > 1) => "failed"
      freeOf?(b := coefficient(numer ff, 0) / d, t) and
        freeOf?(c := coefficient(numer ff, 1) / d, t) => [b, c]
      "failed"

-- returns [a, n] iff f = a * t**n
    atn(f, t) ==
      if ((v := isExpt f) case Record(var:K, exponent:Integer)) then
        w := v::Record(var:K, exponent:Integer)
        (w.exponent > 0) and
          ((vv := symbolIfCan(w.var)) case SE) and (vv::SE = t) =>
            return [1, w.exponent::PI]
      (u := isTimes f) case List(F) =>
        c:F  := 1
        d:N  := 0
        for g in u::List(F) repeat
          if (rec := atn(g, t)) case Record(coef:F, deg:PI) then
            r := rec::Record(coef:F, deg:PI)
            c := c * r.coef
            d := d + r.deg
          else c := c * g
        zero? d => "failed"
        [c, d::PI]
      "failed"

-- returns [a, c, b] iff f = a * exp(c * t + b)
-- and b and c do not involve t
    aexp(f, t) ==
      is?(f, "exp"::SE) =>
        (v := isLinear(first argument(retract(f)@K),t)) case "failed" =>
           "failed"
```

```
        [1, v.nconst, v.const]
      (u := isTimes f) case List(F) =>
        c:F := 1
        c1 := c0 := 0$F
        for g in u::List(F) repeat
          if (r := aexp(g,t)) case Record(coef:F,coef1:F,coef0:F) then
            rec := r::Record(coef:F, coef1:F, coef0:F)
            c   := c * rec.coef
            c0  := c0 + rec.coef0
            c1  := c1 + rec.coef1
          else c := c * g
        zero? c0 and zero? c1 => "failed"
        [c, c1, c0]
      if (v := isPower f) case Record(val:F, exponent:Integer) then
        w := v::Record(val:F, exponent:Integer)
        (w.exponent ^= 1) and
          ((r := aexp(w.val, t)) case Record(coef:F,coef1:F,coef0:F)) =>
            rec := r::Record(coef:F, coef1:F, coef0:F)
            return [rec.coef ** w.exponent, w.exponent * rec.coef1,
                                            w.exponent * rec.coef0]
      "failed"

    mkPlus f ==
      (u := isPlus numer f) case "failed" => "failed"
      d := denom f
      [p / d for p in u::List(SparseMultivariatePolynomial(R, K))]

-- returns g if f = g/t
    tdenom(f, t) ==
      (denom f exquo numer t) case "failed" => "failed"
      t * f

    intlaplace(f, ss, g, v, vv) ==
      is?(g, oplap) or ((i := integrate(g, v)) case List(F)) => "failed"
      (u:=limit(i::F,equation(vv::OFE,plusInfinity()$OFE)$EQ)) case OFE =>
        (l := limit(i::F, equation(vv::OFE, ss::OFE)$EQ)) case OFE =>
          retractIfCan(u::OFE - l::OFE)@Union(F, "failed")
        "failed"
      "failed"

    lapkernel(f, t, tt, ss) ==
      (k := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
      empty?(arg := argument(k::K)) => "failed"
      is?(op := operator k, "%diff"::SE) =>
        not( #arg = 3) => "failed"
        not(is?(arg.3, t)) => "failed"
        fint := eval(arg.1, arg.2, tt)
        s := name operator (kernels(ss).1)
        ss * locallaplace(fint, t, tt, s, ss) - eval(fint, tt = 0)
      not (empty?(rest arg)) => "failed"
```

```
      member?(t, variables(a := first(arg) / tt)) => "failed"
      is?(op := operator k, "Si"::SE) => atan(a / ss) / ss
      is?(op, "Ci"::SE) => log((ss**2 + a**2) / a**2) / (2 * ss)
      is?(op, "Ei"::SE) => log((ss + a) / a) / ss
      -- digamma (or Gamma) needs SpecialFunctionCategory
      -- which we do not have here
      -- is?(op, "log"::SE) => (digamma(1) - log(a) - log(ss)) / ss
      "failed"

    -- Below we try to apply one of the texbook rules for computing
    -- Laplace transforms, either reducing problem to simpler cases
    -- or using one of known base cases
    locallaplace(f, t, tt, s, ss) ==
      zero? f => 0
--      one? f  => inv ss
      (f = 1)  => inv ss

      -- laplace(f(t)/t,t,s)
      --              = integrate(laplace(f(t),t,v), v = s..%plusInfinity)
      (x := tdenom(f, tt)) case F =>
        g := locallaplace(x::F, t, tt, vv := new()$SE, vvv := vv::F)
        (x := intlaplace(f, ss, g, vv, vvv)) case F => x::F
        oplap(f, tt, ss)

      -- Use linearity
      (u := mkPlus f) case List(F) =>
        +/[locallaplace(g, t, tt, s, ss) for g in u::List(F)]
      (rec := splitConstant(f, t)).const ^= 1 =>
        rec.const * locallaplace(rec.nconst, t, tt, s, ss)

      -- laplace(t^n*f(t),t,s) = (-1)^n*D(laplace(f(t),t,s), s, n))
      (v := atn(f, t)) case Record(coef:F, deg:PI) =>
        vv := v::Record(coef:F, deg:PI)
        is?(la := locallaplace(vv.coef, t, tt, s, ss), oplap) => oplap(f,tt,ss)
        (-1$Integer)**(vv.deg) * differentiate(la, s, vv.deg)

      -- Complex shift rule
      (w := aexp(f, t)) case Record(coef:F, coef1:F, coef0:F) =>
        ww := w::Record(coef:F, coef1:F, coef0:F)
        exp(ww.coef0) * locallaplace(ww.coef,t,tt,s,ss - ww.coef1)

      -- Try base cases
      (x := lapkernel(f, t, tt, ss)) case F => x::F

--    -- The following does not seem to help computing transforms, but
--    -- quite frequently leads to loops, so I (wh) disabled it for now
--    -- last chance option: try to use the fact that
--    --    laplace(f(t),t,s) = s laplace(g(t),t,s) - g(0)  where dg/dt = f(t)
--    elem?(int := lfintegrate(f, t)) and (rint := retractIfCan int) case F =>
--        fint := rint :: F
```

```
--        -- to avoid infinite loops, we don't call laplace recursively
--        -- if the integral has no new logs and f is an algebraic function
--        empty?(logpart int) and algebraic?(f, t) => oplap(fint, tt, ss)
--        ss * locallaplace(fint, t, tt, s, ss) - eval(fint, tt = 0)
      oplap(f, tt, ss)

    setProperty(oplap,SPECIALDIFF,dvlap@((List F,SE)->F) pretend None)
```

———-

— LAPLACE.dotabb —

```
"LAPLACE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LAPLACE"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"LAPLACE" -> "ACFS"
```

———-

13.2 package LAZM3PK LazardSetSolvingPackage

— LazardSetSolvingPackage.input —

```
)set break resume
)sys rm -f LazardSetSolvingPackage.output
)spool LazardSetSolvingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 36
R := Integer
--R
--R
--R   (1)  Integer
--R                                                          Type: Domain
--E 1

--S 2 of 36
ls : List Symbol := [b1,x,y,z,t,v,u,w]
--R
--R
--R   (2)  [b1,x,y,z,t,v,u,w]
--R                                                    Type: List(Symbol)
--E 2
```

```
--S 3 of 36
V := OVAR(ls)
--R
--R
--R   (3)  OrderedVariableList([b1,x,y,z,t,v,u,w])
--R                                                              Type: Domain
--E 3

--S 4 of 36
E := IndexedExponents V
--R
--R
--R   (4)  IndexedExponents(OrderedVariableList([b1,x,y,z,t,v,u,w]))
--R                                                              Type: Domain
--E 4

--S 5 of 36
P := NSMP(R, V)
--R
--R
--R   (5)
--R  NewSparseMultivariatePolynomial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w
--R  ]))
--R                                                              Type: Domain
--E 5

--S 6 of 36
b1: P := 'b1
--R
--R
--R   (6)  b1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 6

--S 7 of 36
x: P := 'x
--R
--R
--R   (7)  x
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 7

--S 8 of 36
y: P := 'y
--R
--R
--R   (8)  y
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 8
```

```
--S 9 of 36
z: P := 'z
--R
--R
--R   (9)  z
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 9

--S 10 of 36
t: P := 't
--R
--R
--R   (10)  t
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 10

--S 11 of 36
u: P := 'u
--R
--R
--R   (11)  u
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 11

--S 12 of 36
v: P := 'v
--R
--R
--R   (12)  v
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 12

--S 13 of 36
w: P := 'w
--R
--R
--R   (13)  w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 13

--S 14 of 36
T := REGSET(R,E,V,P)
--R
--R
--R   (14)
--R  RegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([b1,x,y,z,t
--R  ,v,u,w])),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePolyno
--R  mial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])))
--R                                                                  Type: Domain
```

```
--E 14

--S 15 of 36
p0 := b1 + y + z - t - w
--R
--R
--R   (15)  b1 + y + z - t - w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 15

--S 16 of 36
p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1
--R
--R
--R                                 2
--R   (16)  2v y + 2u z + 2w t - 2w  - w - 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 16

--S 17 of 36
p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w
--R
--R
--R           2      2         2           3     2
--R   (17)  3v y + 3u z + (- 3w  - 1)t + 3w  + 3w  + 4w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 17

--S 18 of 36
p3 := 6*x*z*v - 6*t*w**2 + 6*w**3 - 3*t*w + 6*w**2 - t + 4*w
--R
--R
--R                       2                3     2
--R   (18)  6v z x + (- 6w  - 3w - 1)t + 6w  + 6w  + 4w
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 18

--S 19 of 36
p4 := 4*z*u**3+ 4*y*v**3+ 4*t*w**3- 4*w**4 - 6*w**3+ 4*t*w- 10*w**2- w- 1
--R
--R
--R           3      3       3            4     3      2
--R   (19)  4v y + 4u z + (4w  + 4w)t - 4w  - 6w  - 10w  - w - 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 19

--S 20 of 36
p5 := 8*x*z*u*v +8*t*w**3 -8*w**4 +4*t*w**2 -12*w**3 +4*t*w -14*w**2 -3*w -1
--R
--R
```

```
--R                        3     2            4      3      2
--R   (20)  8u v z x + (8w  + 4w  + 4w)t - 8w  - 12w  - 14w  - 3w - 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 20

--S 21 of 36
p6 := 12*x*z*v**2+12*t*w**3 -12*w**4 +12*t*w**2 -18*w**3 +8*t*w -14*w**2 -w -1
--R
--R
--R            2          3      2             4      3      2
--R   (21)  12v z x + (12w  + 12w  + 8w)t - 12w  - 18w  - 14w  - w - 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 21

--S 22 of 36
p7 := -24*t*w**3 + 24*w**4 - 24*t*w**2 + 36*w**3 - 8*t*w + 26*w**2 + 7*w + 1
--R
--R
--R               3      2             4      3      2
--R   (22)  (- 24w  - 24w  - 8w)t + 24w  + 36w  + 26w  + 7w + 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 22

--S 23 of 36
lp := [p0, p1, p2, p3, p4, p5, p6, p7]
--R
--R
--R   (23)
--R                                                  2
--R   [b1 + y + z - t - w, 2v y + 2u z + 2w t - 2w  - w - 1,
--R      2      2         2           3     2
--R    3v y + 3u z + (- 3w  - 1)t + 3w  + 3w  + 4w,
--R                  2               3     2
--R    6v z x + (- 6w  - 3w - 1)t + 6w  + 6w  + 4w,
--R      3      3       3            4     3      2
--R    4v y + 4u z + (4w  + 4w)t - 4w  - 6w  - 10w  - w - 1,
--R                  3     2           4      3      2
--R    8u v z x + (8w  + 4w  + 4w)t - 8w  - 12w  - 14w  - 3w - 1,
--R       2          3      2            4      3      2
--R    12v z x + (12w  + 12w  + 8w)t - 12w  - 18w  - 14w  - w - 1,
--R          3      2             4      3      2
--R    (- 24w  - 24w  - 8w)t + 24w  + 36w  + 26w  + 7w + 1]
--IType: List(NewSparseMultivariatePolynomial(Integer,...
--E 23

--S 24 of 36
lts := zeroSetSplit(lp,false)$T
--R
--R
--R   (24)
```

```
--R   [{w + 1,u,v,t + 1,b1 + y + z + 2}, {w + 1,v,t + 1,z,b1 + y + 2},
--R    {w + 1,t + 1,z,y,b1 + 2}, {w + 1,v - u,t + 1,y + z,x,b1 + 2},
--R    {w + 1,u,t + 1,y,x,b1 + z + 2},
--R
--R          5       4      3     2
--R     {144w  + 216w  + 96w  + 6w  - 11w - 1,
--R          2                 5       4      3     2
--R      (12w  + 9w + 1)u - 72w  - 108w  - 42w  - 9w  - 3w,
--R          2                 4      3      2
--R      (12w  + 9w + 1)v + 36w  + 54w  + 18w ,
--R          3      2             4      3      2
--R      (24w  + 24w  + 8w)t - 24w  - 36w  - 26w  - 7w - 1,
--R
--R                      2                 2                          4      3     2
--R         (12u v - 12u )z + (12w v + 12w  + 4)t + (3w - 5)v + 36w  + 42w  + 6w
--R       +
--R         - 16w
--R       ,
--R                             2
--R      2v y + 2u z + 2w t - 2w  - w - 1,
--R                     2                3     2
--R      6v z x + (- 6w  - 3w - 1)t + 6w  + 6w  + 4w, b1 + y + z - t - w}
--R     ]
--IType: List(RegularTriangularSet(Integer,...
--E 24

--S 25 of 36
[coHeight(ts) for ts in lts]
--R
--R
--R   (25)  [3,3,3,2,2,0]
--R                                                    Type: List(NonNegativeInteger)
--E 25

--S 26 of 36
ST := SREGSET(R,E,V,P)
--R
--R
--R   (26)
--R  SquareFreeRegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([
--R  b1,x,y,z,t,v,u,w])),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivar
--R  iatePolynomial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])))
--R                                                                   Type: Domain
--E 26

--S 27 of 36
pack := LAZM3PK(R,E,V,P,T,ST)
--R
--R
--R   (27)
```

```
--R  LazardSetSolvingPackage(Integer,IndexedExponents(OrderedVariableList([b1,x,y,
--R  z,t,v,u,w])),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePol
--R  ynomial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])),RegularTriangularSet
--R  (Integer,IndexedExponents(OrderedVariableList([b1,x,y,z,t,v,u,w])),OrderedVar
--R  iableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePolynomial(Integer,Ordered
--R  VariableList([b1,x,y,z,t,v,u,w]))),SquareFreeRegularTriangularSet(Integer,Ind
--R  exedExponents(OrderedVariableList([b1,x,y,z,t,v,u,w])),OrderedVariableList([b
--R  1,x,y,z,t,v,u,w]),NewSparseMultivariatePolynomial(Integer,OrderedVariableList
--R  ([b1,x,y,z,t,v,u,w])))
--R                                                                  Type: Domain
--E 27

--S 28 of 36
zeroSetSplit(lp,false)$pack
--R
--R
--R   (28)
--R   [{w + 1,t + 1,z,y,b1 + 2}, {w + 1,v,t + 1,z,b1 + y + 2},
--R    {w + 1,u,v,t + 1,b1 + y + z + 2}, {w + 1,v - u,t + 1,y + z,x,b1 + 2},
--R    {w + 1,u,t + 1,y,x,b1 + z + 2},
--R
--R          5       4      3     2                   4      3      2
--R     {144w  + 216w  + 96w  + 6w  - 11w - 1, u - 24w  - 36w  - 14w  + w + 1,
--R              4      3      2                  4      3      2
--R      3v - 48w  - 60w  - 10w  + 8w + 2, t - 24w  - 36w  - 14w  - w + 1,
--R                 4        3        2
--R      486z - 2772w  - 4662w  - 2055w  + 30w + 127,
--R                   4         3        2
--R      2916y - 22752w  - 30312w  - 8220w  + 2064w + 1561,
--R                  4        3       2
--R      356x - 3696w  - 4536w  - 968w  + 822w + 371,
--R                    4         3         2
--R      2916b1 - 30600w  - 46692w  - 20274w  - 8076w + 593}
--R     ]
--IType: List(SquareFreeRegularTriangularSet(Integer,...
--E 28

--S 29 of 36
f0 := (w - v) ** 2 + (u - t) ** 2 - 1
--R
--R
--R          2           2           2    2
--R   (29)  t  - 2u t + v  - 2w v + u  + w  - 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 29

--S 30 of 36
f1 := t ** 2 - v ** 3
--R
--R
```

```
--R          2    3
--R   (30)  t  - v
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 30

--S 31 of 36
f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)
--R
--R
--R              2                   2
--R   (31)  (- 3v  - 2v + 2w)t + 3u v
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 31

--S 32 of 36
f3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1)
--R
--R
--R           2   2             2
--R   (32)  6v t z  + (- 2t - 3v )z + 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 32

--S 33 of 36
lf := [f0, f1, f2, f3]
--R
--R
--R   (33)
--R     2           2           2    2       2    3       2                   2
--R   [t  - 2u t + v  - 2w v + u  + w  - 1, t  - v , (- 3v  - 2v + 2w)t + 3u v ,
--R      2   2             2
--R    6v t z  + (- 2t - 3v )z + 1]
--IType: List(NewSparseMultivariatePolynomial(Integer,...
--E 33

--S 34 of 36
zeroSetSplit(lf,true)$T
--R
--R
--R   (34)
--R   [
--R     {
--R             6           3       2                 4
--R         729u  + (- 1458w  + 729w  - 4158w - 1685)u
--R       +
--R              6        5        4        3       2                2       8
--R         (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
--R       +
--R             7        6        5        4        3        2
--R         216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
```

```
--R       ,
--R
--R                  4           3       2                  2        6        5
--R             2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
--R           +
--R                     4        3        2
--R             - 10125w  - 4800w  + 2501w  + 4968w - 1587
--R        *
--R           v
--R       +
--R               3       2  2       6        5        4       3        2
--R         (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
--R       ,
--R         2                   2                    2  2           2
--R      (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
--R     ]
--IType: List(RegularTriangularSet(Integer,IndexedExponents(...
--E 34

--S 35 of 36
zeroSetSplit(lf,false)$T
--R
--R
--R   (35)
--R   [
--R     {
--R             6           3       2                 4
--R         729u  + (- 1458w  + 729w  - 4158w - 1685)u
--R       +
--R              6        5        4        3       2                2       8
--R         (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
--R       +
--R             7        6        5        4        3        2
--R         216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
--R       ,
--R
--R                  4           3       2                  2        6        5
--R             2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
--R           +
--R                     4        3        2
--R             - 10125w  - 4800w  + 2501w  + 4968w - 1587
--R        *
--R           v
--R       +
--R               3       2  2       6        5        4       3        2
--R         (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
--R       ,
--R         2                   2                    2  2           2
--R      (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
--R     ,
```

```
--R
--R         4     3      2                                2
--R     {27w  + 4w  - 54w  - 36w + 23, u, (12w + 2)v - 9w  - 2w + 9,
--R        2          2
--R      6t  - 2v - 3w  + 2w + 3, 2t z - 1}
--R     ,
--R
--R            6         5         4          3         2
--R     {59049w  + 91854w  - 45198w  + 145152w  + 63549w  + 60922w + 21420,
--R
--R                             5                  4                  3
--R              31484448266904w  - 18316865522574w  + 23676995746098w
--R           +
--R                            2
--R              6657857188965w  + 8904703998546w + 3890631403260
--R        *
--R            2
--R           u
--R       +
--R                          5                  4                  3
--R         94262810316408w  - 82887296576616w  + 89801831438784w
--R       +
--R                         2
--R         28141734167208w  + 38070359425432w + 16003865949120
--R       ,
--R            2             2         2       3      2                    3     2
--R      (243w  + 36w + 85)v  + (- 81u  - 162w  + 36w  + 154w + 72)v - 72w  + 4w ,
--R         2                   2                      2  2           2
--R      (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
--R     ,
--R
--R         4     3      2                                2
--R     {27w  + 4w  - 54w  - 36w + 23, u, (12w + 2)v - 9w  - 2w + 9,
--R        2          2             2
--R      6t  - 2v - 3w  + 2w + 3, 3v z - 1}
--R     ]
--IType: List(RegularTriangularSet(Integer,IndexedExponents(...
--E 35

--S 36 of 36
zeroSetSplit(lf,false)$pack
--R
--R
--R   (36)
--R   [
--R     {
--R             6           3       2                 4
--R         729u  + (- 1458w  + 729w  - 4158w - 1685)u
--R       +
--R              6        5        4        3       2                2       8
```

```
--R         (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
--R       +
--R             7        6        5        4        3        2
--R         216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
--R       ,
--R
--R                 4           3       2                  2        6        5
--R             2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
--R           +
--R                     4        3        2
--R             - 10125w  - 4800w  + 2501w  + 4968w - 1587
--R        *
--R           v
--R       +
--R               3       2  2       6        5        4       3        2
--R         (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
--R       ,
--R         2                   2                    2  2           2
--R      (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
--R     ,
--R
--R         2                 2                   2
--R     {81w  + 18w + 28, 729u  - 1890w - 533, 81v  + (- 162w + 27)v - 72w - 112,
--R      11881t + (972w + 2997)u v + (- 11448w - 11536)u,
--R
--R                         2
--R         641237934604288z
--R       +
--R                 (78614584763904w + 26785578742272)u + 236143618655616w
--R               +
--R                 70221988585728
--R            *
--R               v
--R           +
--R             (358520253138432w + 101922133759488)u + 142598803536000w
--R           +
--R             54166419595008
--R        *
--R           z
--R       +
--R         (32655103844499w - 44224572465882)u v
--R       +
--R         (43213900115457w - 32432039102070)u
--R       }
--R     ,
--R
--R         4     3      2                            3     2
--R     {27w  + 4w  - 54w  - 36w + 23, u, 218v - 162w  + 3w  + 160w + 153,
--R          2      3      2                             3      2
--R      109t  - 27w  - 54w  + 63w + 80, 1744z + (- 1458w  + 27w  + 1440w + 505)t}
```

```
--R     ,
--R
--R         4     3      2                            3     2
--R     {27w  + 4w  - 54w  - 36w + 23, u, 218v - 162w  + 3w  + 160w + 153,
--R          2      3      2                           3     2
--R      109t  - 27w  - 54w  + 63w + 80, 1308z + 162w  - 3w  - 814w - 153}
--R     ,
--R
--R          4       3        2                    2      2
--R     {729w  + 972w  - 1026w  + 1684w + 765, 81u  + 72w  + 16w - 72,
--R                 3       2
--R      702v - 162w  - 225w  + 40w - 99,
--R                    3       2
--R      11336t + (324w  - 603w  - 1718w - 1557)u,
--R
--R                   2
--R         595003968z
--R       +
--R                           3             2
--R             (- 963325386w  - 898607682w  + 1516286466w - 3239166186)u
--R           +
--R                           3              2
--R             - 1579048992w  - 1796454288w  + 2428328160w - 4368495024
--R        *
--R           z
--R       +
--R                      3              2
--R         (9713133306w  + 9678670317w  - 16726834476w + 28144233593)u
--R       }
--R     ]
--IType: List(SquareFreeRegularTriangularSet(Integer,...
--E 36

)spool
)lisp (bye)
```

———-

— LazardSetSolvingPackage.help —

```
=====================================================================
LazardSetSolvingPackage examples
=====================================================================

The LazardSetSolvingPackage package constructor solves polynomial
systems by means of Lazard triangular sets.  However one condition is
relaxed: Regular triangular sets whose saturated ideals have positive
dimension are not necessarily normalized.
```

```
The decompositions are computed in two steps.  First the algorithm of
Moreno Maza (implemented in the RegularTriangularSet domain
constructor) is called.  Then the resulting decompositions are
converted into lists of square-free regular triangular sets and the
redundant components are removed.  Moreover, zero-dimensional regular
triangular sets are normalized.

Note that the way of understanding triangular decompositions is
detailed in the example of the RegularTriangularSet constructor.

The LazardSetSolvingPackage constructor takes six arguments.  The
first one, R, is the coefficient ring of the polynomials; it must
belong to the category GcdDomain.  The second one, E, is the exponent
monoid of the polynomials; it must belong to the category
OrderedAbelianMonoidSup.  the third one, V, is the ordered set of
variables; it must belong to the category OrderedSet.  The fourth one
is the polynomial ring; it must belong to the category
RecursivePolynomialCategory(R,E,V).  The fifth one is a domain of the
category RegularTriangularSetCategory(R,E,V,P) and the last one is a
domain of the category SquareFreeRegularTriangularSetCategory(R,E,V,P).
The abbreviation for LazardSetSolvingPackage is LAZM3PK.

For the purpose of solving zero-dimensional algebraic systems, see
also LexTriangularPackage and ZeroDimensionalSolvePackage.  These
packages are easier to call than LAZM3PK.  Moreover, the
ZeroDimensionalSolvePackage package provides operations to compute
either the complex roots or the real roots.

We illustrate now the use of the LazardSetSolvingPackage package
constructor with two examples (Butcher and Vermeer).

Define the coefficient ring.

R := Integer
    Integer

Define the list of variables,

ls : List Symbol := [b1,x,y,z,t,v,u,w]
    [b1,x,y,z,t,v,u,w]

and make it an ordered set:

V := OVAR(ls)
    OrderedVariableList [b1,x,y,z,t,v,u,w]

then define the exponent monoid.

E := IndexedExponents V
```

```
Define the polynomial ring.

P := NSMP(R, V)

Let the variables be polynomial.

b1: P := 'b1

x: P := 'x

y: P := 'y

z: P := 'z

t: P := 't

u: P := 'u

v: P := 'v

w: P := 'w

Now call the {\tt RegularTriangularSet} domain constructor.

T := REGSET(R,E,V,P)

Define a polynomial system (the Butcher example).

p0 := b1 + y + z - t - w

p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1

p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w

p3 := 6*x*z*v - 6*t*w**2 + 6*w**3 - 3*t*w + 6*w**2 - t + 4*w

p4 := 4*z*u**3+ 4*y*v**3+ 4*t*w**3- 4*w**4 - 6*w**3+ 4*t*w- 10*w**2- w- 1

p5 := 8*x*z*u*v +8*t*w**3 -8*w**4 +4*t*w**2 -12*w**3 +4*t*w -14*w**2 -3*w -1

p6 := 12*x*z*v**2+12*t*w**3 -12*w**4 +12*t*w**2 -18*w**3 +8*t*w -14*w**2 -w -1

p7 := -24*t*w**3 + 24*w**4 - 24*t*w**2 + 36*w**3 - 8*t*w + 26*w**2 + 7*w + 1

lp := [p0, p1, p2, p3, p4, p5, p6, p7]
   [b1 + y + z - t - w, 2v y + 2u z + 2w t - 2w  - w - 1,
      2      2         2           3     2
    3v y + 3u z + (- 3w  - 1)t + 3w  + 3w  + 4w,
                  2                3     2
    6v z x + (- 6w  - 3w - 1)t + 6w  + 6w  + 4w,
```

```
      3      3       3            4     3      2
    4v y + 4u z + (4w  + 4w)t - 4w  - 6w  - 10w  - w - 1,
                  3     2           4      3      2
    8u v z x + (8w  + 4w  + 4w)t - 8w  - 12w  - 14w  - 3w - 1,
       2          3      2             4      3      2
    12v z x + (12w  + 12w  + 8w)t - 12w  - 18w  - 14w  - w - 1,
          3      2             4      3      2
    (- 24w  - 24w  - 8w)t + 24w  + 36w  + 26w  + 7w + 1]
```

First of all, let us solve this system in the sense of Lazard by means of the REGSET constructor:

```
lts := zeroSetSplit(lp,false)$T
   [{w + 1,u,v,t + 1,b1 + y + z + 2}, {w + 1,v,t + 1,z,b1 + y + 2},
    {w + 1,t + 1,z,y,b1 + 2}, {w + 1,v - u,t + 1,y + z,x,b1 + 2},
    {w + 1,u,t + 1,y,x,b1 + z + 2},

          5       4      3     2
     {144w  + 216w  + 96w  + 6w  - 11w - 1,
          2                 5       4      3     2
      (12w  + 9w + 1)u - 72w  - 108w  - 42w  - 9w  - 3w,
          2                 4      3      2
      (12w  + 9w + 1)v + 36w  + 54w  + 18w ,
          3      2             4      3      2
      (24w  + 24w  + 8w)t - 24w  - 36w  - 26w  - 7w - 1,

                     2                 2                          4      3     2
        (12u v - 12u )z + (12w v + 12w  + 4)t + (3w - 5)v + 36w  + 42w  + 6w
       +
         - 16w
       ,
                           2
      2v y + 2u z + 2w t - 2w  - w - 1,
                    2                3     2
      6v z x + (- 6w  - 3w - 1)t + 6w  + 6w  + 4w, b1 + y + z - t - w}
     ]
```

We can get the dimensions of each component of a decomposition as follows.

```
[coHeight(ts) for ts in lts]
    [3,3,3,2,2,0]
```

The first five sets have a simple shape. However, the last one, which has dimension zero, can be simplified by using Lazard triangular sets.

Thus we call the SquareFreeRegularTriangularSet domain constructor,

```
ST := SREGSET(R,E,V,P)
```

```
and set the LAZM3PK package constructor to our situation.

pack := LAZM3PK(R,E,V,P,T,ST)

We are ready to solve the system by means of Lazard triangular sets:

zeroSetSplit(lp,false)$pack
   [{w + 1,t + 1,z,y,b1 + 2}, {w + 1,v,t + 1,z,b1 + y + 2},
    {w + 1,u,v,t + 1,b1 + y + z + 2}, {w + 1,v - u,t + 1,y + z,x,b1 + 2},
    {w + 1,u,t + 1,y,x,b1 + z + 2},

         5       4      3     2                    4      3      2
     {144w  + 216w  + 96w  + 6w  - 11w - 1, u - 24w  - 36w  - 14w  + w + 1,
              4      3      2                  4      3      2
      3v - 48w  - 60w  - 10w  + 8w + 2, t - 24w  - 36w  - 14w  - w + 1,
                 4        3        2
      486z - 2772w  - 4662w  - 2055w  + 30w + 127,
                   4         3        2
      2916y - 22752w  - 30312w  - 8220w  + 2064w + 1561,
                 4        3       2
      356x - 3696w  - 4536w  - 968w  + 822w + 371,
                    4         3         2
      2916b1 - 30600w  - 46692w  - 20274w  - 8076w + 593}
     ]

We see the sixth triangular set is nicer now: each one of its
polynomials has a constant initial.

We follow with the Vermeer example. The ordering is the usual one
for this system.

Define the polynomial system.

f0 := (w - v) ** 2 + (u - t) ** 2 - 1


      2           2           2    2
     t  - 2u t + v  - 2w v + u  + w  - 1


f1 := t ** 2 - v ** 3

      2    3
     t  - v

f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)

          2                   2
     (- 3v  - 2v + 2w)t + 3u v
```

```
f3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1)
```

```
       2   2            2
     6v t z  + (- 2t - 3v )z + 1
```

```
lf := [f0, f1, f2, f3]
```

```
     2          2          2    2       2    3       2                   2
   [t  - 2u t + v  - 2w v + u  + w  - 1, t  - v , (- 3v  - 2v + 2w)t + 3u v ,
      2   2            2
    6v t z  + (- 2t - 3v )z + 1]
```

First of all, let us solve this system in the sense of Kalkbrener by means of the REGSET constructor:

```
zeroSetSplit(lf,true)$T
   [
    {
           6          3       2                4
       729u  + (- 1458w  + 729w  - 4158w - 1685)u
     +
            6        5        4        3       2                2       8
       (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
     +
           7        6        5        4        3        2
       216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
     ,

                 4          3       2                  2        6        5
          2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
        +
                  4        3        2
          - 10125w  - 4800w  + 2501w  + 4968w - 1587
      *
        v
     +
           3       2  2       6        5        4       3        2
       (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
     ,
       2                  2                    2  2          2
     (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
    ]
```

We have obtained one regular chain (i.e. regular triangular set) with dimension 1. This set is in fact a characterist set of the (radical of) of the ideal generated by the input system lf. Thus we have only the generic points of the variety associated with lf (for the elimination ordering given by ls).

So let us get now a full description of this variety.

Hence, we solve this system in the sense of Lazard by means of the REGSET constructor:

```
zeroSetSplit(lf,false)$T
   [
    {
            6           3       2                 4
        729u  + (- 1458w  + 729w  - 4158w - 1685)u
      +
             6        5        4        3       2                2       8
        (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
      +
            7        6        5        4        3        2
        216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
      ,

                 4           3       2                  2        6        5
          2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
        +
                   4        3        2
          - 10125w  - 4800w  + 2501w  + 4968w - 1587
       *
          v
      +
            3       2  2       6        5        4       3        2
        (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
      ,
        2                   2                     2  2           2
     (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
    ,

        4     3      2                               2
    {27w  + 4w  - 54w  - 36w + 23, u, (12w + 2)v - 9w  - 2w + 9,
       2          2
     6t  - 2v - 3w  + 2w + 3, 2t z - 1}
    ,

           6         5         4          3         2
    {59049w  + 91854w  - 45198w  + 145152w  + 63549w  + 60922w + 21420,

                            5                  4                  3
          31484448266904w  - 18316865522574w  + 23676995746098w
        +
                         2
          6657857188965w  + 8904703998546w + 3890631403260
       *
           2
          u
      +
```

```
                    5                  4                 3
      94262810316408w  - 82887296576616w  + 89801831438784w
    +
                    2
      28141734167208w  + 38070359425432w + 16003865949120
    ,
        2            2         2       3      2                    3     2
    (243w  + 36w + 85)v  + (- 81u  - 162w  + 36w  + 154w + 72)v - 72w  + 4w ,
      2                  2                    2  2           2
    (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
   ,

       4     3      2                               2
   {27w  + 4w  - 54w  - 36w + 23, u, (12w + 2)v - 9w  - 2w + 9,
      2          2            2
    6t  - 2v - 3w  + 2w + 3, 3v z - 1}
   ]
```

We retrieve our regular chain of dimension 1 and we get three regular chains of dimension 0 corresponding to the degenerated cases.
We want now to simplify these zero-dimensional regular chains by using Lazard triangular sets. Moreover, this will allow us to prove that the above decomposition has no redundant component.

Generally, decompositions computed by the REGSET constructor do not have redundant components. However, to be sure that no redundant component occurs one needs to use the SREGSET or LAZM3PK constructors.

So let us solve the input system in the sense of Lazard by means of the LAZM3PK constructor:

```
zeroSetSplit(lf,false)$pack
   [
     {
            6           3       2                 4
       729u  + (- 1458w  + 729w  - 4158w - 1685)u
     +
           6        5        4        3       2                2       8
       (729w  - 1458w  - 2619w  - 4892w  - 297w  + 5814w + 427)u  + 729w
     +
           7        6        5        4        3        2
       216w  - 2900w  - 2376w  + 3870w  + 4072w  - 1188w  - 1656w + 529
     ,

                4           3       2                  2        6        5
          2187u  + (- 4374w  - 972w  - 12474w - 2868)u  + 2187w  - 1944w
        +
                  4        3        2
          - 10125w  - 4800w  + 2501w  + 4968w - 1587
```

```
   *
      v
  +
          3       2  2       6        5        4       3        2
    (1944w  - 108w )u  + 972w  + 3024w  - 1080w  + 496w  + 1116w
  ,
    2                   2                    2  2           2
 (3v  + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z  + (2t + 3v )z - 1}
,

    2                2                   2
{81w  + 18w + 28, 729u  - 1890w - 533, 81v  + (- 162w + 27)v - 72w - 112,
 11881t + (972w + 2997)u v + (- 11448w - 11536)u,

                    2
    641237934604288z
  +
            (78614584763904w + 26785578742272)u + 236143618655616w
          +
            70221988585728
       *
          v
      +
        (358520253138432w + 101922133759488)u + 142598803536000w
      +
        54166419595008
   *
      z
  +
    (32655103844499w - 44224572465882)u v
  +
    (43213900115457w - 32432039102070)u
  }
,

    4     3      2                           3     2
{27w  + 4w  - 54w  - 36w + 23, u, 218v - 162w  + 3w  + 160w + 153,
     2      3      2                            3      2
 109t  - 27w  - 54w  + 63w + 80, 1744z + (- 1458w  + 27w  + 1440w + 505)t}
,

    4     3      2                           3     2
{27w  + 4w  - 54w  - 36w + 23, u, 218v - 162w  + 3w  + 160w + 153,
     2      3      2                           3     2
 109t  - 27w  - 54w  + 63w + 80, 1308z + 162w  - 3w  - 814w - 153}
,

     4       3        2                   2      2
{729w  + 972w  - 1026w  + 1684w + 765, 81u  + 72w  + 16w - 72,
            3       2
```

```
      702v - 162w  - 225w  + 40w - 99,
                    3       2
      11336t + (324w  - 603w  - 1718w - 1557)u,

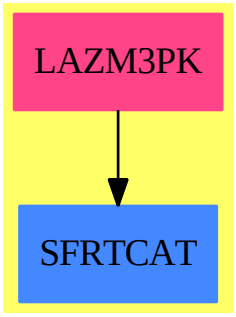
                  2
        595003968z
       +
                          3             2
             (- 963325386w  - 898607682w  + 1516286466w - 3239166186)u
          +
                          3             2
            - 1579048992w  - 1796454288w  + 2428328160w - 4368495024
       *
          z
       +
                     3             2
        (9713133306w  + 9678670317w  - 16726834476w + 28144233593)u
       }
     ]
```

Due to square-free factorization, we obtained now four zero-dimensional regular chains. Moreover, each of them is normalized (the initials are constant). Note that these zero-dimensional components may be investigated further with the ZeroDimensionalSolvePackage package constructor.

See also:
o)show LazardSetSolvingPackage
o)show ZeroDimensionalSolvePackage

———-

13.2.1 LazardSetSolvingPackage (LAZM3PK)



Exports:
normalizeIfCan zeroSetSplit

— package LAZM3PK LazardSetSolvingPackage —

```
)abbrev package LAZM3PK LazardSetSolvingPackage
++ Author: Marc Moreno Maza
++ Date Created: 10/02/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] D. LAZARD "A new method for solving algebraic systems of
++      positive dimension" Discr. App. Math. 33:147-160,1991
++  [2] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A package for solving polynomial systems by means of Lazard triangular
++ sets. This package provides two operations. One for solving in the sense
++ of the regular zeros, and the other for solving in the sense of
++ the Zariski closure. Both produce square-free regular sets.
++ Moreover, the decompositions do not contain any redundant component.
++ However, only zero-dimensional regular sets are normalized, since
++ normalization may be time consumming in positive dimension.
++ The decomposition process is that of [2].

LazardSetSolvingPackage(R,E,V,P,TS,ST): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS: RegularTriangularSetCategory(R,E,V,P)
  ST : SquareFreeRegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  K ==> Fraction R
  LP ==> List P
  PWT ==> Record(val : P, tower : TS)
  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Split ==> List TS
  --KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
  --EntryGcd ==> List PWT
```

```
--HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
--KeyInvSet ==> Record(arg1: P, arg3: TS)
--EntryInvSet ==> List TS
--HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,ST)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,ST)
normalizpack ==> NormalizationPackage(R,E,V,P,ST)

Exports ==  with

   normalizeIfCan: ST -> ST
     ++ \axiom{normalizeIfCan(ts)} returns \axiom{ts} in an normalized shape
     ++ if \axiom{ts} is zero-dimensional.
   zeroSetSplit: (LP, B) -> List ST
     ++ \axiom{zeroSetSplit(lp,clos?)} has the same specifications as
     ++ zeroSetSplit(lp,clos?) from RegularTriangularSetCategory.

Implementation == add

   convert(st: ST): TS ==
     ts: TS := empty()
     lp: LP := members(st)$ST
     lp := sort(infRittWu?,lp)
     for p in lp repeat
       ts := internalAugment(p,ts)$TS
     ts

   squareFree(ts: TS): List ST ==
     empty? ts => [empty()$ST]
     lp: LP := members(ts)$TS
     lp := sort(infRittWu?,lp)
     newts: ST := empty()$ST
     toSee: List ST := [newts]
     toSave: List ST
     for p in lp repeat
       toSave := []
       while (not empty? toSee) repeat
         us := first toSee; toSee := rest toSee
         lpwt := stoseSquareFreePart(p,us)$regsetgcdpack
         for pwt in lpwt repeat
           newus := internalAugment(pwt.val,pwt.tower)$ST
           toSave := cons(newus,toSave)
       toSee := toSave
     toSave

   normalizeIfCan(ts: ST): ST ==
     empty? ts => ts
     lp: LP := members(ts)$ST
     lp := sort(infRittWu?,lp)
```

```
      p: P := first lp
      not univariate?(p)$polsetpack => ts
      lp := rest lp
      newts: ST := empty()$ST
      newts := internalAugment(p,newts)$ST
      while (not empty? lp) repeat
        p := first lp
        lv := variables(p)
        for v in lv repeat
          v = mvar(p) => "leave"
          not algebraic?(v,newts) => return internalAugment(lp,newts)$ST
        lp := rest lp
        p := normalizedAssociate(p,newts)$normalizpack
        newts := internalAugment(p,newts)$ST
      newts

    zeroSetSplit(lp:List(P), clos?:B): List ST ==
      -- if clos? then SOLVE in the closure sense
      toSee: Split := zeroSetSplit(lp, clos?)$TS
      toSave: List ST := []
      for ts in toSee repeat
        toSave := concat(squareFree(ts),toSave)
      toSave := removeSuperfluousQuasiComponents(toSave)$quasicomppack
      [normalizeIfCan(ts) for ts in toSave]
```

———-

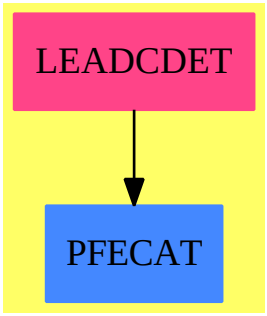
— LAZM3PK.dotabb —

```
"LAZM3PK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LAZM3PK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"LAZM3PK" -> "SFRTCAT"
```

———-

13.3 package LEADCDET LeadingCoefDetermination

13.3.1 LeadingCoefDetermination (LEADCDET)



Exports:
distFact polCase

— package LEADCDET LeadingCoefDetermination —

```
)abbrev package LEADCDET LeadingCoefDetermination
++ Author : P.Gianni, May 1990
++ Description:
++ Package for leading coefficient determination in the lifting step.
++ Package working for every R euclidean with property "F".

LeadingCoefDetermination(OV,E,Z,P) : C == T
 where
  OV    :   OrderedSet
  E     :   OrderedAbelianMonoidSup
  Z     :   EuclideanDomain
  BP        ==> SparseUnivariatePolynomial Z
  P         :   PolynomialCategory(Z,E,OV)
  NNI       ==> NonNegativeInteger
  LeadFact  ==> Record(polfac:List(P),correct:Z,corrfact:List(BP))
  ParFact   ==> Record(irr:P,pow:Integer)
  FinalFact ==> Record(contp:Z,factors:List(ParFact))

  C == with
   polCase  : (Z,NNI,List(Z)) -> Boolean
     ++ polCase(contprod, numFacts, evallcs), where contprod is the
     ++ product of the content of the leading coefficient of
     ++ the polynomial to be factored with the content of the
     ++ evaluated polynomial, numFacts is the number of factors
     ++ of the leadingCoefficient, and evallcs is the list of
     ++ the evaluated factors of the leadingCoefficient, returns
     ++ true if the factors of the leading Coefficient can be
```

```
    ++ distributed with this valuation.
  distFact : (Z,List(BP),FinalFact,List(Z),List(OV),List(Z)) ->
                                          Union(LeadFact,"failed")
    ++ distFact(contm,unilist,plead,vl,lvar,lval), where contm is
    ++ the content of the evaluated polynomial, unilist is the list
    ++ of factors of the evaluated polynomial, plead is the complete
    ++ factorization of the leading coefficient, vl is the list
    ++ of factors of the leading coefficient evaluated, lvar is the
    ++ list of variables, lval is the list of values, returns a record
    ++ giving the list of leading coefficients to impose on the univariate
    ++ factors,

 T == add
   distribute: (Z,List(BP),List(P),List(Z),List(OV),List(Z)) -> LeadFact
   checkpow  : (Z,Z) -> NNI

   polCase(d:Z,nk:NNI,lval:List(Z)):Boolean ==
     -- d is the product of the content lc m (case polynomial)
     -- and the cont of the polynomial evaluated
     q:Z
     distlist:List(Z) := [d]
     for i in 1..nk repeat
       q := unitNormal(lval.i).canonical
       for j in 0..(i-1)::NNI repeat
         y := distlist.((i-j)::NNI)
         while y^=1  repeat
           y := gcd(y,q)
           q := q quo y
         if q=1 then return false
       distlist := append(distlist,[q])
     true

   checkpow(a:Z,b:Z) : NonNegativeInteger ==
     qt: Union(Z,"failed")
     for i in 0.. repeat
       qt:= b exquo a
       if qt case "failed" then return i
       b:=qt::Z

   distribute(contm:Z,unilist:List(BP),pl:List(P),vl:List(Z),
                            lvar:List(OV),lval:List(Z)): LeadFact ==
     d,lcp : Z
     nf:NNI:=#unilist
     for i in 1..nf repeat
         lcp := leadingCoefficient (unilist.i)
         d:= gcd(lcp,vl.i)
         pl.i := (lcp quo d)*pl.i
         d := vl.i quo d
         unilist.i := d*unilist.i
         contm := contm quo d
```

```
      if contm ^=1 then for i in 1..nf repeat pl.i := contm*pl.i
      [pl,contm,unilist]$LeadFact

    distFact(contm:Z,unilist:List(BP),plead:FinalFact,
             vl:List(Z),lvar:List(OV),lval:List(Z)):Union(LeadFact,"failed") ==
      h:NonNegativeInteger
      c,d : Z
      lpol:List(P):=[]
      lexp:List(Integer):=[]
      nf:NNI := #unilist
      vl := reverse vl --lpol and vl reversed so test from right to left
      for fpl in plead.factors repeat
       lpol:=[fpl.irr,:lpol]
       lexp:=[fpl.pow,:lexp]
      vlp:List(Z):= [1$Z for i in 1..nf]
      aux : List(P) := [1$P for i in 1..nf]
      for i in 1..nf repeat
        c := contm*leadingCoefficient unilist.i
        c=1 or c=-1  => "next i"
        for k in 1..(# lpol) repeat
          lexp.k=0 => "next factor"
          h:= checkpow(vl.k,c)
          if h ^=0 then
           if h>lexp.k then return "failed"
           lexp.k:=lexp.k-h
           aux.i := aux.i*(lpol.k ** h)
           d:= vl.k**h
           vlp.i:= vlp.i*d
           c:= c quo d
        if contm=1 then vlp.i:=c
      for k in 1..(# lpol) repeat if lexp.k ^= 0 then return "failed"
      contm =1 => [[vlp.i*aux.i for i in 1..nf],1,unilist]$LeadFact
      distribute(contm,unilist,aux,vlp,lvar,lval)
```

———-

— LEADCDET.dotabb —

```
"LEADCDET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LEADCDET"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LEADCDET" -> "PFECAT"
```

———-

13.4 package LEXTRIPK LexTriangularPackage

— LexTriangularPackage.input —

```
)set break resume
)spool LexTriangularPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 22
R := Integer
--R
--R
--R   (1)  Integer
--R                                                              Type: Domain
--E 1

--S 2 of 22
ls : List Symbol := [a,b,c,d,e,f]
--R
--R
--R   (2)  [a,b,c,d,e,f]
--R                                                    Type: List(Symbol)
--E 2

--S 3 of 22
V := OVAR(ls)
--R
--R
--R   (3)  OrderedVariableList([a,b,c,d,e,f])
--R                                                              Type: Domain
--E 3

--S 4 of 22
P := NSMP(R, V)
--R
--R
--R   (4)
--R   NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--R                                                              Type: Domain
--E 4

--S 5 of 22
p1: P :=  a*b*c*d*e*f - 1
--R
--R
--R   (5)  f e d c b a - 1
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
```

```
--E 5

--S 6 of 22
p2: P := a*b*c*d*e +a*b*c*d*f +a*b*c*e*f +a*b*d*e*f +a*c*d*e*f +b*c*d*e*f
--R
--R
--R   (6)  ((((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--E 6

--S 7 of 22
p3: P :=  a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f
--R
--R
--R   (7)  (((d + f)c + f e)b + f e d)a + e d c b + f e d c
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--E 7

--S 8 of 22
p4: P := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f
--R
--R
--R   (8)  ((c + f)b + f e)a + d c b + e d c + f e d
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--E 8

--S 9 of 22
p5: P := a*b + a*f + b*c + c*d + d*e + e*f
--R
--R
--R   (9)  (b + f)a + c b + d c + e d + f e
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--E 9

--S 10 of 22
p6: P := a + b + c + d + e + f
--R
--R
--R   (10)  a + b + c + d + e + f
--RType: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
--E 10

--S 11 of 22
lp := [p1, p2, p3, p4, p5, p6]
--R
--R
--R   (11)
--R   [f e d c b a - 1, ((((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b,
--R    (((d + f)c + f e)b + f e d)a + e d c b + f e d c,
--R    ((c + f)b + f e)a + d c b + e d c + f e d,
```

```
--R    (b + f)a + c b + d c + e d + f e, a + b + c + d + e + f]
--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))
--E 11

--S 12 of 22
lextripack :=  LEXTRIPK(R,ls)
--R
--R
--R   (12)  LexTriangularPackage(Integer,[a,b,c,d,e,f])
--R                                                                  Type: Domain
--E 12

--S 13 of 22
lg := groebner(lp)$lextripack
--R
--R
--R   (13)
--R   [a + b + c + d + e + f,
--R
--R                         2                                          2
--R       3968379498283200b  + 15873517993132800f b + 3968379498283200d
--R     +
--R                                                3 5                    4 4
--R       15873517993132800f d + 3968379498283200f e  - 15873517993132800f e
--R     +
--R                          5 3                      6                       2
--R       23810276989699200f e  + (206355733910726400f  + 230166010900425600)e
--R     +
--R                              43                        37
--R           - 729705987316687f   + 1863667496867205421f
--R         +
--R                                 31                          25
--R           291674853771731104461f   + 365285994691106921745f
--R         +
--R                               19                           13
--R           549961185828911895f   - 365048404038768439269f
--R         +
--R                                   7
--R           - 292382820431504027669f  - 2271898467631865497f
--R      *
--R         e
--R     +
--R                            44                         38
--R       - 3988812642545399f   + 10187423878429609997f
--R     +
--R                               32                           26
--R       1594377523424314053637f   + 19947393084399916238065f
--R     +
--R                             20                            14
--R       1596840088052642815f   - 19934941183011621454l3f
```

```
--R     +
--R                                8                          2
--R       - 1596049742289689815053f  - 11488171330159667449f
--R     ,
--R
--R                                                                        2
--R       (23810276989699200c - 23810276989699200f)b + 23810276989699200c
--R     +
--R                                          2
--R       71430830969097600f c - 23810276989699200d  - 95241107958796800f d
--R     +
--R                            3 5                       4 4                       5 3
--R       - 55557312975964800f e  + 174608697924460800f e  - 174608697924460800f e
--R     +
--R                              6                          2
--R       (- 2428648252949318400f  - 2611193709870345600)e
--R     +
--R                             43                         37
--R           8305444561289527f   - 212120871519454596 41f
--R         +
--R                                    31                           25
--R           - 3319815883093451385381f   - 4157691646261657136445f
--R         +
--R                                  19                            13
--R           - 6072721607510764095f   + 4154986709036460221649f
--R         +
--R                                   7
--R           3327761311138587096749f  + 25885340608290841637f
--R       *
--R           e
--R     +
--R                           44                          38
--R       45815897629010329f   - 117013765582151891207f
--R     +
--R                                  32                             26
--R       - 18313166848970865074187f   - 229099712396492974389 15f
--R     +
--R                                20                              14
--R       - 16133250761305157265f   + 228973058576361782566 23f
--R     +
--R                                 8                            2
--R       183299447818672424979 23f  + 1302585310020204206 99f
--R     ,
--R
--R       (7936758996566400d - 7936758996566400f)b - 7936758996566400f d
--R     +
--R                            3 5                       4 4                       5 3
--R       - 7936758996566400f e  + 23810276989699200f e  - 23810276989699200f e
--R     +
--R                              6                          2
```

```
--R       (- 337312257354072000f  - 369059293340337600)e
--R     +
--R                            43                        37
--R           1176345388640471f   - 30043835828914730 73f
--R         +
--R                                  31                          25
--R           - 470203502707246105653f   - 588858183402644348085f
--R         +
--R                                19                        13
--R           - 856939308623513535f   + 588472674242340526377f
--R         +
--R                                 7
--R           4713132419583711035 17f  + 3659742549078552381f
--R      *
--R         e
--R     +
--R                        44                         38                             32
--R       6423170513956901f   - 164047721370364808 03f   - 25674191652275287744 63f
--R     +
--R                                26                         20
--R       - 3211938090825682172335f   - 2330490332697587485f
--R     +
--R                              14                           8
--R       32101001094447548645 87f   + 25698583153951626178 47f
--R     +
--R                            2
--R       1832608948742773575 1f
--R     ,
--R
--R                                                                        3 5
--R       (11905138494849600e - 11905138494849600f)b - 3968379498283200f e
--R     +
--R                           4 4                      5 3
--R       15873517993132800f e  - 27778656487982400f e
--R     +
--R                              6                         2
--R       (- 208339923659868000f  - 240086959646133600)e
--R     +
--R                            43                         37
--R           786029984751110f   - 20075190081822452 50f
--R         +
--R                                  31                          25
--R           - 3141880629080738070 90f   - 3934236675379295752 50f
--R         +
--R                                19                          13
--R           - 550329120654394950f   + 3931964087288896127 70f
--R         +
--R                                 7
--R           3148923727991764957 30f  + 2409386515146668530f
--R      *
```

```
--R         e
--R     +
--R                        44                          38                            32
--R       4177638546747827f   - 10669685294602576381f   - 1669852980419949524601f
--R     +
--R                                26                         20
--R       - 2089077057287904170745f   - 1569899763580278795f
--R     +
--R                               14                           8
--R       2087864026859015573349f   + 16714960859451995779 69f
--R     +
--R                             2
--R       11940257226216280177f
--R     ,
--R
--R                            6                                              2 5
--R       (11905138494849600f  - 11905138494849600)b - 15873517993132800f e
--R     +
--R                          3 4                        4 3
--R       39683794982832000f e  - 39683794982832000f e
--R     +
--R                             11                        5  2
--R       (- 686529653202993600f   - 6071620632373296 00f )e
--R     +
--R                           42                         36                          30
--R           65144531306704f   - 166381280901088652f   - 260334345024702834 72f
--R         +
--R                                  24                          18
--R           - 316962595838606501 40f   + 971492093167581360f
--R         +
--R                               12                           6
--R         32220085033691389548f   + 25526177666070529808f  + 138603268355749244
--R       *
--R         e
--R     +
--R                         43                          37                            31
--R       167620036074811f   - 428102417974791473f   - 669972438012316793 13f
--R     +
--R                                25                          19
--R       - 834267167221487504 85f   + 203673895369980765f
--R     +
--R                              13                            7
--R       835230563260104324 57f   + 669957896402380669 37f  + 4785928555495879 01f
--R     ,
--R
--R                     3                    2                  2                 45
--R       801692827936c  + 2405078483808f c  - 2405078483808f c - 13752945467f
--R     +
--R                       39                      33                     27
--R       35125117815561f   + 5496946957826433f   + 6834659447749117f
```

```
--R     +
--R                        21                    15                   9
--R       - 44484880462461f   - 6873406230093057f   - 5450844938762633f
--R     +
--R                     3
--R       1216586044571f
--R     ,
--R
--R                                                                     2
--R       (23810276989699200d - 2381027698969920f)c + 23810276989699200d
--R     +
--R                                              3 5                     4 4
--R       71430830969097600f d + 7936758996566400f e  - 31747035986265600f e
--R     +
--R                           5 3                        6                        2
--R       31747035986265600f e  + (404774708824886400f  + 396837949828320000)e
--R     +
--R                               43                         37
--R           - 1247372229446701f   + 3185785654596621203f
--R         +
--R                                 31                          25
--R           498594866849974751463f   + 624542545845791047935f
--R         +
--R                               19                           13
--R           931085755769682885f   - 6241506635824170633387f
--R         +
--R                                  7
--R           - 4998818593883604756 47f  - 3926885313819527351f
--R      *
--R         e
--R     +
--R                             44                          38
--R       - 7026011547118141f   + 17944427051950691243f
--R     +
--R                               32                            26
--R       28083835225939866035 43f   + 3513624142354807530135f
--R     +
--R                            20                             14
--R       2860757006705537685f   - 3511356735642190737267f
--R     +
--R                                 8                           2
--R       - 2811332494697103819887f  - 20315011631522847311f
--R     ,
--R
--R       (7936758996566400e - 7936758996566400f)c
--R     +
--R                            43                       37                          31
--R           - 4418748183673f   + 11285568707456559f   + 1765998617294451019f
--R         +
--R                                25                       19
```

```
--R           2173749283622606155f   - 55788292195402895f
--R         +
--R                                13                      7
--R         - 2215291421788292951f   - 17181426653474308 51f  + 30256569458230237f
--R      *
--R         e
--R     +
--R                     44                      38                      32
--R       4418748183673f   - 112855687074565 59f   - 1765998617294451019f
--R     +
--R                              26                      20                        14
--R       - 2173749283622606155f   + 55788292195402895f   + 2215291421788292951f
--R     +
--R                            8                     2
--R       17181426653474308 51f  - 30256569458230237f
--R     ,
--R
--R                       6                                 43
--R       (72152354514240f  - 72152354514240)c + 40950859449f
--R     +
--R                         37                      31                      25
--R       - 104588980990367f   - 1636722739557530 7f   - 202685234165273 55f
--R     +
--R                       19                     13                     7
--R       442205002259535f   + 20576059935789063f   + 1599713379697056 3f
--R     +
--R       - 275099892785581f
--R     ,
--R
--R                        3                      2                     2
--R       19841897491416 00d  + 5952569247424800f d  - 5952569247424800f d
--R     +
--R                           4 5                     5 4                     3
--R       - 3968379498283200f e  + 15873517993132800f e  + 17857707742274400e
--R     +
--R                              7                        2
--R       (- 148814231185620000f  - 162703559429611200f)e
--R     +
--R                              44                      38
--R         - 3900009146788 78f   + 9960627045937564 34f
--R       +
--R                                 32                        26
--R         1558863239720348 23914f   + 194745956143985421330f
--R       +
--R                             20                        14
--R         6205077595574430f   - 1945965126532990 68786f
--R       +
--R                                  8                      2
--R         - 1557968979407569 22666f  - 1036375759077320978f
--R      *
```

```
--R         e
--R     +
--R                          45                       39                          33
--R       - 3749986300359 91f   + 957747106595453993f   + 1498891555667648916 93f
--R     +
--R                              27                       21
--R       18715417144349464 1685f   - 127129015426348065f
--R     +
--R                              15                           9                       3
--R     - 1872415332431150404 17f   - 1497199835679765340 37f  - 836654081239648061f
--R     ,
--R
--R                                                                          3 5
--R       (5952569247424800e - 5952569247424800f)d - 3968379498283200f e
--R     +
--R                         4 4                      5 3
--R       9920948745708000f e  - 3968379498283200f e
--R     +
--R                              6                        2
--R       (- 148814231185620000f  - 150798420934761600)e
--R     +
--R                            43                        37
--R           492558110242553f   - 12579923596080745 99f
--R         +
--R                                    31                            25
--R           - 1968830945393685139 59f   - 2465621157457354280 55f
--R         +
--R                                  19                           13
--R           - 325698701993885505f   + 2464177698836518081 11f
--R         +
--R                                   7
--R           1973273520682006529 11f  + 1523373796389332143f
--R      *
--R         e
--R     +
--R                         44                        38                            32
--R       2679481081803026f   - 6843392695421906608f   - 1071020459642646913578f
--R     +
--R                                26                       20
--R       - 13397891696920412400 60f   - 85274675091075021 0f
--R     +
--R                               14                           8
--R       13391051019718784013 12f   + 1071900289758712984762f
--R     +
--R                             2
--R       75552390720727275 6f
--R     ,
--R
--R                           6                                          2 5
--R       (11905138494849600f  - 11905138494849600)d - 7936758996566400f e
```

```
--R     +
--R                          3 4                     4 3
--R       31747035986265600f e  - 31747035986265600f e
--R     +
--R                             11                      5  2
--R       (- 420648226818019200f   - 404774708824886400f )e
--R     +
--R                            42                     36                      30
--R           15336187600889f   - 39169739565161107f   - 6127176127489690827f
--R         +
--R                                 24                      18
--R           - 7217708742310509615f   + 538628483890722735f
--R         +
--R                                12                       6
--R           7506804353843507643f   + 58861607697826072O3f  + 63576108396535879
--R      *
--R         e
--R     +
--R                       43                      37                         31
--R       71737781777066f   - 183218856207557938f   - 286728742711322760 78f
--R     +
--R                              25                      19
--R       - 35625223686939812010f   + 164831339634084390f
--R     +
--R                             13                       7
--R       357241604230730526 42f   + 286270225786649106 22f  + 187459987029680506f
--R     ,
--R
--R                           6                     5                    2 4
--R       1322793166094400e  - 3968379498283200f e  + 3968379498283200f e
--R     +
--R                            3 3
--R       - 5291172664377600f e
--R     +
--R                              10                      4  2
--R       (- 230166010900425600f   - 226197631402142400f )e
--R     +
--R                                47                         41
--R           - 152375364610443885f   + 3891666260648548904 15f
--R         +
--R                                   35                            29
--R           609060978413605589873 35f   + 761673679346087986972 75f
--R         +
--R                                23                          17
--R           27855066785995181125f   - 76144952817052723145495f
--R         +
--R                                   11                         5
--R           - 609336298924635175469 75f   - 4114150716820025477 95f
--R      *
--R         e
```

```
--R     +
--R                          42                       36                          30
--R       - 209493533143822f   + 535045979490560586f   + 83737947964973553146f
--R     +
--R                            24                       18
--R       104889507084213371570f   + 167117997269207870f
--R     +
--R                              12                         6
--R       - 104793725781390615514f   - 83842685189903180394f  - 569978796672974242
--R     ,
--R
--R                         6                   3
--R       (25438330117200f  + 25438330117200)e
--R     +
--R                         7                    2
--R       (76314990351600f  + 76314990351600f)e
--R     +
--R                            44                    38                       32
--R           - 1594966552735f   + 4073543370415745f   + 6375271592311489 25f
--R         +
--R                              26                    20                        14
--R           7975211761136065 25f   + 530440941097175f   - 797160527306433145f
--R         +
--R                               8                     2
--R           - 638132320196044965f  - 4510507167940725f
--R      *
--R         e
--R     +
--R                          45                      39                         33
--R       - 6036376800443f   + 15416903421476909f   + 24128076461923044 49f
--R     +
--R                             27                     21                          15
--R       3017679923028013705f   + 1422320037411955f   - 30165604024178439 41f
--R     +
--R                               9                      3
--R       - 2414249368183033161f  - 165618623617638 73f
--R     ,
--R
--R                         12                  2
--R       (1387545279120f   - 1387545279120)e
--R     +
--R                         43                  37                    31
--R           4321823003f   - 11037922310209f   - 1727510711947989f
--R         +
--R                             25                  19                     13
--R           - 2165150991154425f   - 5114342560755f   + 2162682824948601f
--R         +
--R                            7
--R           1732620732685741f  + 13506088516033f
--R      *
```

```
--R         e
--R     +
--R                   44                  38                    32
--R       24177661775f   - 61749727185325f   - 9664106795754225f
--R     +
--R                           26                 20                      14
--R       - 12090487758628245f   - 8787672733575f   + 12083693383005045f
--R     +
--R                         8                  2
--R       9672870290826025f  + 6854402808525f
--R     ,
--R     48        42          36          30          18          12        6
--R    f   - 2554f   - 399710f   - 499722f   + 499722f   + 399710f   + 2554f  - 1]
--RType: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))
--E 13

--S 14 of 22
lexTriangular(lg,false)$lextripack
--R
--R
--R   (14)
--R   [
--R       6       6       5     2 4     3 3     4 2     5
--R     {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
--R            2 5     3 4     4 3     5 2
--R      3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
--R             2 5     3 4     4 3      5 2
--R      3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
--R           2 5     3 4     4 3     5 2
--R      a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
--R     ,
--R      6                  2           2                    2
--R    {f  - 1,e - f,d - f,c  + 4f c + f ,(c - f)b - f c - 5f ,a + b + c + 3f},
--R      6                         2           2
--R    {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
--R      6            2           2                    2
--R    {f  - 1,e - f,d  + 4f d + f ,(d - f)c - f d - 5f ,b - f,a + c + d + 3f},
--R
--R       36        30          24          18          12        6
--R     {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
--R
--R                    12             2
--R         (161718564f   - 161718564)e
--R       +
--R                       31              25                19                13
--R             - 504205f   + 1287737951f   + 201539391380f   + 253982817368f
--R           +
--R                            7
--R             201940704665f  + 1574134601f
--R        *
```

```
--R           e
--R       +
--R                    32              26                 20                  14
--R         - 2818405f   + 7198203911f   + 1126548149060f   + 1416530563364f
--R       +
--R                     8             2
--R         1127377589345f  + 7988820725f
--R       ,
--R
--R                     6                                 2 5                 3 4
--R         (693772639560f  - 693772639560)d - 462515093040f e  + 1850060372160f e
--R       +
--R                       4 3                      11                  5  2
--R         - 1850060372160f e  + (- 24513299931120f   - 23588269745040f )e
--R       +
--R                      30                 24                   18
--R             - 890810428f   + 2275181044754f   + 355937263869776f
--R           +
--R                         12                   6
--R             413736880104344f   + 342849304487996f  + 3704966481878
--R        *
--R           e
--R       +
--R                     31                  25                    19
--R         - 4163798003f   + 10634395752169f   + 1664161760192806f
--R       +
--R                       13                   7
--R         2079424391370694f   + 1668153650635921f  + 10924274392693f
--R       ,
--R
--R                     6                           31               25
--R         (12614047992f  - 12614047992)c - 7246825f   + 18508536599f
--R       +
--R                      19                 13                 7
--R         2896249516034f   + 3581539649666f   + 2796477571739f  - 48094301893f
--R       ,
--R
--R                     6                                 2 5                 3 4
--R         (693772639560f  - 693772639560)b - 925030186080f e  + 2312575465200f e
--R       +
--R                       4 3                      11                  5  2
--R         - 2312575465200f e  + (- 40007555547960f   - 35382404617560f )e
--R       +
--R                      30                 24                   18
--R             - 3781280823f   + 9657492291789f   + 1511158913397906f
--R           +
--R                          12                   6
--R             1837290892286154f   + 1487216006594361f  + 8077238712093
--R        *
--R           e
```

```
--R       +
--R                      31                   25                    19
--R         - 9736390478f   + 24866827916734f   + 3891495681905296f
--R       +
--R                        13                    7
--R         4872556418871424f   + 3904047887269606f  + 27890075838538f
--R       ,
--R      a + b + c + d + e + f}
--R     ,
--R      6      2           2                    2
--R    {f  - 1,e  + 4f e + f ,(e - f)d - f e - 5f ,c - f,b - f,a + d + e + 3f}]
--R                              Type: List(RegularChain(Integer,[a,b,c,d,e,f]))
--E 14

--S 15 of 22
lts := lexTriangular(lg,true)$lextripack
--R
--R
--R   (15)
--R   [
--R       6       6       5     2 4     3 3     4 2     5
--R     {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
--R            2 5     3 4     4 3     5 2
--R      3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
--R             2 5     3 4     4 3      5 2
--R      3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
--R           2 5     3 4     4 3     5 2
--R      a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
--R     ,
--R      6                  2           2
--R    {f  - 1,e - f,d - f,c  + 4f c + f ,b + c + 4f,a - f},
--R      6                        2           2
--R    {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
--R      6            2           2
--R    {f  - 1,e - f,d  + 4f d + f ,c + d + 4f,b - f,a - f},
--R
--R       36        30          24          18          12        6
--R     {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
--R
--R                       2
--R         1387545279120e
--R       +
--R                        31                  25                     19
--R             4321823003f   - 11037922310209f   - 1727506390124986f
--R           +
--R                                13                    7
--R             - 2176188913464634f   - 1732620732685741f  - 13506088516033f
--R        *
--R           e
--R       +
```

```
--R                    32                  26                   20
--R         24177661775f   - 61749727185325f   - 9664082618092450f
--R       +
--R                           14                    8                 2
--R         - 12152237485813570f   - 9672870290826025f  - 68544102808525f
--R       ,
--R
--R         1387545279120d
--R       +
--R                         30                 24                   18
--R             - 1128983050f   + 2883434331830f   + 451234998755840f
--R           +
--R                           12                   6
--R             562426491685760f   + 447129055314890f  - 165557857270
--R        *
--R           e
--R       +
--R                        31                 25                   19
--R         - 1816935351f   + 4640452214013f   + 726247129626942f
--R       +
--R                       13                   7
--R         912871801716798f   + 726583262666877f  + 4909358645961f
--R       ,
--R
--R                                   31                 25                   19
--R         1387545279120c + 778171189f   - 1987468196267f   - 310993556954378f
--R       +
--R                          13                   7
--R         - 383262822316802f   - 300335488637543f  + 5289595037041f
--R       ,
--R
--R         1387545279120b
--R       +
--R                        30                 24                   18
--R             1128983050f   - 2883434331830f   - 451234998755840f
--R           +
--R                             12                   6
--R             - 562426491685760f   - 447129055314890f  + 165557857270
--R        *
--R           e
--R       +
--R                        31                 25                   19
--R         - 3283058841f   + 8384938292463f   + 1312252817452422f
--R       +
--R                         13                   7
--R         1646579934064638f   + 1306372958656407f  + 4694680112151f
--R       ,
--R
--R                                                  31                  25
--R         1387545279120a + 1387545279120e + 4321823003f   - 110379223102O9f
```

```
--R       +
--R                            19                    13                    7
--R         - 1727506390124986f   - 2176188913464634f   - 1732620732685741f
--R       +
--R         - 13506088516033f
--R       }
--R     ,
--R      6      2           2
--R    {f  - 1,e  + 4f e + f ,d + e + 4f,c - f,b - f,a - f}]
--R                                Type: List(RegularChain(Integer,[a,b,c,d,e,f]))
--E 15

--S 16 of 22
[ [init(p) for p in (ts :: List(P))] for ts in lts]
--R
--R
--R   (16)
--R   [[1,3,1,3,1,1], [1,1,1,1,1,1], [1,1,1,1,1,1], [1,1,1,1,1,1],
--R    [1387545279120,1387545279120,1387545279120,1387545279120,1387545279120,1],
--R    [1,1,1,1,1,1]]
--RType: List(List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))))
--E 16

--S 17 of 22
squareFreeLexTriangular(lg,true)$lextripack
--R
--R
--R   (17)
--R   [
--R       6       6       5     2 4     3 3     4 2     5
--R     {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
--R            2 5     3 4     4 3     5 2
--R      3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
--R             2 5     3 4     4 3      5 2
--R      3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
--R           2 5     3 4     4 3     5 2
--R      a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
--R     ,
--R      6                  2           2
--R    {f  - 1,e - f,d - f,c  + 4f c + f ,b + c + 4f,a - f},
--R      6                        2           2
--R    {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
--R      6            2           2
--R    {f  - 1,e - f,d  + 4f d + f ,c + d + 4f,b - f,a - f},
--R
--R       36        30          24          18          12        6
--R     {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
--R
--R                         2
--R         1387545279120e
```

```
--R       +
--R                        31                  25                    19
--R             4321823003f   - 11037922310209f   - 1727506390124986f
--R          +
--R                                13                   7
--R             - 2176188913464634f   - 1732620732685741f  - 13506088516033f
--R        *
--R           e
--R       +
--R                      32                 26                   20
--R         24177661775f   - 61749727185325f   - 9664082618092450f
--R       +
--R                              14                   8                  2
--R         - 12152237485813570f   - 9672870290826025f  - 68544102808525f
--R       ,
--R
--R         1387545279120d
--R       +
--R                         30                 24                   18
--R             - 1128983050f   + 2883434331830f   + 451234998755840f
--R           +
--R                              12                   6
--R             562426491685760f   + 447129055314890f  - 165557857270
--R        *
--R           e
--R       +
--R                       31                 25                   19
--R         - 1816935351f   + 4640452214013f   + 726247129626942f
--R       +
--R                          13                   7
--R         912871801716798f   + 726583262666877f  + 4909358645961f
--R       ,
--R
--R                                     31                 25                   19
--R         1387545279120c + 778171189f   - 1987468196267f   - 310993556954378f
--R       +
--R                            13                   7
--R         - 383262822316802f   - 300335488637543f  + 5289595037041f
--R       ,
--R
--R         1387545279120b
--R       +
--R                        30                 24                   18
--R             1128983050f   - 2883434331830f   - 451234998755840f
--R           +
--R                               12                   6
--R             - 562426491685760f   - 447129055314890f  + 165557857270
--R        *
--R           e
--R       +
```

```
--R                      31                 25                    19
--R         - 3283058841f   + 8384938292463f   + 1312252817452422f
--R       +
--R                        13                   7
--R         1646579934064638f   + 1306372958656407f  + 4694680112151f
--R       ,
--R                                                  31                  25
--R         1387545279120a + 1387545279120e + 4321823003f   - 11037922310209f
--R       +
--R                           19                   13                    7
--R         - 1727506390124986f   - 2176188913464634f   - 1732620732685741f
--R       +
--R         - 13506088516033f
--R       }
--R     ,
--R      6      2           2
--R    {f  - 1,e  + 4f e + f ,d + e + 4f,c - f,b - f,a - f}]
--RType: List(SquareFreeRegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([a,b,c
--E 17

--S 18 of 22
reduce(+,[degree(ts) for ts in lts])
--R
--R
--R   (18)  156
--R                                                        Type: PositiveInteger
--E 18

--S 19 of 22
ls2 : List Symbol := concat(ls,new()$Symbol)
--R
--R
--R   (19)  [a,b,c,d,e,f,%A]
--R                                                           Type: List(Symbol)
--E 19

--S 20 of 22
zdpack := ZDSOLVE(R,ls,ls2)
--R
--R
--R   (20)  ZeroDimensionalSolvePackage(Integer,[a,b,c,d,e,f],[a,b,c,d,e,f,%A])
--R                                                                 Type: Domain
--E 20

--S 21 of 22
concat [univariateSolve(ts)$zdpack for ts in lts]
--R
--R
--R   (21)
```

```
--R   [
--R                    4      2
--R     [complexRoots= ?  - 13?  + 49,
--R
--R       coordinates =
--R                 3                3                3                3
--R         [7a + %A  - 6%A, 21b + %A  + %A, 21c - 2%A  + 19%A, 7d - %A  + 6%A,
--R                  3                3
--R          21e - %A  - %A, 21f + 2%A  - 19%A]
--R       ]
--R     ,
--R
--R                    4      2
--R     [complexRoots= ?  + 11?  + 49,
--R
--R       coordinates =
--R                  3                 3                  3
--R         [35a + 3%A  + 19%A, 35b + %A  + 18%A, 35c - 2%A  - %A,
--R                  3                 3                  3
--R          35d - 3%A  - 19%A, 35e - %A  - 18%A, 35f + 2%A  + %A]
--R       ]
--R     ,
--R
--R     [
--R       complexRoots =
--R          8      7      6       5       4       3       2
--R         ?  - 12?  + 58?  - 120?  + 207?  - 360?  + 802?  - 1332? + 1369
--R       ,
--R
--R       coordinates =
--R         [
--R                                7           6            5            4
--R             43054532a + 33782%A  - 546673%A  + 3127348%A  - 6927123%A
--R           +
--R                      3             2
--R             4365212%A  - 25086957%A  + 39582814%A - 107313172
--R           ,
--R
--R                                7           6            5            4
--R             43054532b - 33782%A  + 546673%A  - 3127348%A  + 6927123%A
--R           +
--R                        3             2
--R             - 4365212%A  + 25086957%A  - 39582814%A + 107313172
--R           ,
--R
--R                                7           6            5            4
--R             21527266c - 22306%A  + 263139%A  - 1166076%A  + 1821805%A
--R           +
--R                        3             2
--R             - 2892788%A  + 10322663%A  - 9026596%A + 12950740
```

```
--R          ,
--R
--R                                7           6            5            4
--R            43054532d + 22306%A  - 263139%A  + 1166076%A  - 1821805%A
--R          +
--R                      3             2
--R            2892788%A  - 10322663%A  + 30553862%A - 12950740
--R          ,
--R
--R                                7           6            5            4
--R            43054532e - 22306%A  + 263139%A  - 1166076%A  + 1821805%A
--R          +
--R                        3             2
--R            - 2892788%A  + 10322663%A  - 30553862%A + 12950740
--R          ,
--R
--R                                7           6            5            4
--R            21527266f + 22306%A  - 263139%A  + 1166076%A  - 1821805%A
--R          +
--R                      3             2
--R            2892788%A  - 10322663%A  + 9026596%A - 12950740
--R          ]
--R       ]
--R     ,
--R
--R     [
--R       complexRoots =
--R          8      7      6       5       4       3       2
--R         ?  + 12?  + 58?  + 120?  + 207?  + 360?  + 802?  + 1332? + 1369
--R       ,
--R
--R       coordinates =
--R         [
--R                                7           6            5            4
--R            43054532a + 33782%A  + 546673%A  + 3127348%A  + 6927123%A
--R          +
--R                      3             2
--R            4365212%A  + 25086957%A  + 39582814%A + 107313172
--R          ,
--R
--R                                7           6            5            4
--R            43054532b - 33782%A  - 546673%A  - 3127348%A  - 6927123%A
--R          +
--R                        3             2
--R            - 4365212%A  - 25086957%A  - 39582814%A - 107313172
--R          ,
--R
--R                                7           6            5            4
--R            21527266c - 22306%A  - 263139%A  - 1166076%A  - 1821805%A
--R          +
```

```
--R                        3            2
--R             - 2892788%A  - 10322663%A  - 9026596%A - 12950740
--R           ,
--R
--R                                7           6            5            4
--R             43054532d + 22306%A  + 263139%A  + 1166076%A  + 1821805%A
--R           +
--R                       3            2
--R             2892788%A  + 10322663%A  + 30553862%A + 12950740
--R           ,
--R
--R                                7           6            5            4
--R             43054532e - 22306%A  - 263139%A  - 1166076%A  - 1821805%A
--R           +
--R                        3            2
--R             - 2892788%A  - 10322663%A  - 30553862%A - 12950740
--R           ,
--R
--R                                7           6            5            4
--R             21527266f + 22306%A  + 263139%A  + 1166076%A  + 1821805%A
--R           +
--R                       3            2
--R             2892788%A  + 10322663%A  + 9026596%A + 12950740
--R           ]
--R       ]
--R     ,
--R
--R                     4    2
--R     [complexRoots= ?  - ?  + 1,
--R                                  3            3             3            3
--R      coordinates= [a - %A,b + %A  - %A,c + %A ,d + %A,e - %A  + %A,f - %A ]]
--R     ,
--R
--R                     8     6      4      2
--R     [complexRoots= ?  + 4?  + 12?  + 16?  + 4,
--R
--R       coordinates =
--R                  7      5       3                  7      5       3
--R         [4a - 2%A  - 7%A  - 20%A  - 22%A, 4b + 2%A  + 7%A  + 20%A  + 22%A,
--R                 7      5       3                7      5       3
--R          4c + %A  + 3%A  + 10%A  + 10%A, 4d + %A  + 3%A  + 10%A  + 6%A,
--R                7      5       3               7      5       3
--R          4e - %A  - 3%A  - 10%A  - 6%A, 4f - %A  - 3%A  - 10%A  - 10%A]
--R       ]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  + 6?  + 30?  + 36? + 36,
--R
--R       coordinates =
```

```
--R                  3      2                    3      2
--R         [30a - %A  - 5%A  - 30%A - 6, 6b + %A  + 5%A  + 24%A + 6,
--R                  3      2              3      2
--R          30c - %A  - 5%A  - 6, 30d - %A  - 5%A  - 30%A - 6,
--R                  3      2                     3      2
--R          30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
--R       ]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  - 6?  + 30?  - 36? + 36,
--R
--R       coordinates =
--R                  3      2                    3      2
--R         [30a - %A  + 5%A  - 30%A + 6, 6b + %A  - 5%A  + 24%A - 6,
--R                  3      2              3      2
--R          30c - %A  + 5%A  + 6, 30d - %A  + 5%A  - 30%A + 6,
--R                  3      2                     3      2
--R          30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
--R       ]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  + 6? + 6,
--R      coordinates= [a + 1,b - %A - 5,c + %A + 1,d + 1,e + 1,f + 1]]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  - 6? + 6,
--R      coordinates= [a - 1,b - %A + 5,c + %A - 1,d - 1,e - 1,f - 1]]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  + 6?  + 30?  + 36? + 36,
--R
--R       coordinates =
--R                   3      2                    3      2
--R         [6a + %A  + 5%A  + 24%A + 6, 30b - %A  - 5%A  - 6,
--R                  3      2                     3      2
--R          30c - %A  - 5%A  - 30%A - 6, 30d - %A  - 5%A  - 30%A - 6,
--R                  3      2                     3      2
--R          30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
--R       ]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  - 6?  + 30?  - 36? + 36,
--R
--R       coordinates =
--R                   3      2                    3      2
```

```
--R         [6a + %A  - 5%A  + 24%A - 6, 30b - %A  + 5%A  + 6,
--R                  3      2                     3      2
--R          30c - %A  + 5%A  - 30%A + 6, 30d - %A  + 5%A  - 30%A + 6,
--R                  3      2                     3      2
--R          30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
--R       ]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  + 6? + 6,
--R      coordinates= [a - %A - 5,b + %A + 1,c + 1,d + 1,e + 1,f + 1]]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  - 6? + 6,
--R      coordinates= [a - %A + 5,b + %A - 1,c - 1,d - 1,e - 1,f - 1]]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  + 6?  + 30?  + 36? + 36,
--R
--R       coordinates =
--R                  3      2                     3      2
--R         [30a - %A  - 5%A  - 30%A - 6, 30b - %A  - 5%A  - 30%A - 6,
--R                  3      2                     3      2
--R          6c + %A  + 5%A  + 24%A + 6, 30d - %A  - 5%A  - 6,
--R                  3      2                     3      2
--R          30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
--R       ]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  - 6?  + 30?  - 36? + 36,
--R
--R       coordinates =
--R                  3      2                     3      2
--R         [30a - %A  + 5%A  - 30%A + 6, 30b - %A  + 5%A  - 30%A + 6,
--R                  3      2                     3      2
--R          6c + %A  - 5%A  + 24%A - 6, 30d - %A  + 5%A  + 6,
--R                  3      2                     3      2
--R          30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
--R       ]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  + 6? + 6,
--R      coordinates= [a + 1,b + 1,c - %A - 5,d + %A + 1,e + 1,f + 1]]
--R     ,
--R
--R                     2
```

```
--R     [complexRoots= ?  - 6? + 6,
--R      coordinates= [a - 1,b - 1,c - %A + 5,d + %A - 1,e - 1,f - 1]]
--R     ,
--R
--R                     8     7      6      5      4     2
--R     [complexRoots= ?  + 6?  + 16?  + 24?  + 18?  - 8?  + 4,
--R
--R       coordinates =
--R                  7      6       5       4      3       2
--R         [2a + 2%A  + 9%A  + 18%A  + 19%A  + 4%A  - 10%A  - 2%A + 4,
--R                  7      6       5       4      3       2
--R          2b + 2%A  + 9%A  + 18%A  + 19%A  + 4%A  - 10%A  - 4%A + 4,
--R                  7      6      5      4      3
--R          2c - %A  - 4%A  - 8%A  - 9%A  - 4%A  - 2%A - 4,
--R                  7      6      5      4      3
--R          2d + %A  + 4%A  + 8%A  + 9%A  + 4%A  + 2%A + 4,
--R                  7      6       5       4      3       2
--R          2e - 2%A  - 9%A  - 18%A  - 19%A  - 4%A  + 10%A  + 4%A - 4,
--R                  7      6       5       4      3       2
--R          2f - 2%A  - 9%A  - 18%A  - 19%A  - 4%A  + 10%A  + 2%A - 4]
--R       ]
--R     ,
--R
--R     [
--R       complexRoots =
--R          8      7      6       5       4       3        2
--R         ?  + 12?  + 64?  + 192?  + 432?  + 768?  + 1024?  + 768? + 256
--R       ,
--R
--R       coordinates =
--R         [
--R                            7        6        5         4         3         2
--R             1408a - 19%A  - 200%A  - 912%A  - 2216%A  - 4544%A  - 6784%A
--R           +
--R             - 6976%A - 1792
--R           ,
--R
--R                            7        6         5         4          3          2
--R             1408b - 37%A  - 408%A  - 1952%A  - 5024%A  - 10368%A  - 16768%A
--R           +
--R             - 17920%A - 5120
--R           ,
--R
--R                            7        6         5         4          3          2
--R             1408c + 37%A  + 408%A  + 1952%A  + 5024%A  + 10368%A  + 16768%A
--R           +
--R             17920%A + 5120
--R           ,
--R
--R                            7        6        5         4         3         2
```

```
--R             1408d + 19%A  + 200%A  + 912%A  + 2216%A  + 4544%A  + 6784%A
--R           +
--R             6976%A + 1792
--R           ,
--R          2e + %A, 2f - %A]
--R       ]
--R     ,
--R
--R                     8     6      4      2
--R     [complexRoots= ?  + 4?  + 12?  + 16?  + 4,
--R
--R       coordinates =
--R                  7      5       3               7      5       3
--R         [4a - %A  - 3%A  - 10%A  - 6%A, 4b - %A  - 3%A  - 10%A  - 10%A,
--R                 7      5       3                 7      5       3
--R          4c - 2%A  - 7%A  - 20%A  - 22%A, 4d + 2%A  + 7%A  + 20%A  + 22%A,
--R                 7      5       3               7      5       3
--R          4e + %A  + 3%A  + 10%A  + 10%A, 4f + %A  + 3%A  + 10%A  + 6%A]
--R       ]
--R     ,
--R
--R                     8      6      4       2
--R     [complexRoots= ?  + 16?  - 96?  + 256?  + 256,
--R
--R       coordinates =
--R                     7       5        3
--R         [512a - %A  - 12%A  + 176%A  - 448%A,
--R                     7       5       3
--R          128b - %A  - 16%A  + 96%A  - 256%A,
--R                     7       5       3
--R          128c + %A  + 16%A  - 96%A  + 256%A,
--R                     7       5        3
--R          512d + %A  + 12%A  - 176%A  + 448%A, 2e + %A, 2f - %A]
--R       ]
--R     ,
--R
--R     [
--R       complexRoots =
--R          8      7      6       5       4       3        2
--R         ?  - 12?  + 64?  - 192?  + 432?  - 768?  + 1024?  - 768? + 256
--R       ,
--R
--R       coordinates =
--R         [
--R                          7        6        5         4         3         2
--R             1408a - 19%A  + 200%A  - 912%A  + 2216%A  - 4544%A  + 6784%A
--R           +
--R             - 6976%A + 1792
--R           ,
--R
```

```
--R                          7        6         5         4          3          2
--R             1408b - 37%A  + 408%A  - 1952%A  + 5024%A  - 10368%A  + 16768%A
--R           +
--R             - 17920%A + 5120
--R           ,
--R
--R                          7        6         5         4          3          2
--R             1408c + 37%A  - 408%A  + 1952%A  - 5024%A  + 10368%A  - 16768%A
--R           +
--R             17920%A - 5120
--R           ,
--R
--R                          7        6        5         4         3         2
--R             1408d + 19%A  - 200%A  + 912%A  - 2216%A  + 4544%A  - 6784%A
--R           +
--R             6976%A - 1792
--R           ,
--R          2e + %A, 2f - %A]
--R       ]
--R     ,
--R
--R                     8     7      6      5      4     2
--R     [complexRoots= ?  - 6?  + 16?  - 24?  + 18?  - 8?  + 4,
--R
--R       coordinates =
--R                  7      6       5       4      3       2
--R         [2a + 2%A  - 9%A  + 18%A  - 19%A  + 4%A  + 10%A  - 2%A - 4,
--R                  7      6       5       4      3       2
--R          2b + 2%A  - 9%A  + 18%A  - 19%A  + 4%A  + 10%A  - 4%A - 4,
--R                 7      6      5      4      3
--R          2c - %A  + 4%A  - 8%A  + 9%A  - 4%A  - 2%A + 4,
--R                 7      6      5      4      3
--R          2d + %A  - 4%A  + 8%A  - 9%A  + 4%A  + 2%A - 4,
--R                  7      6       5       4      3       2
--R          2e - 2%A  + 9%A  - 18%A  + 19%A  - 4%A  - 10%A  + 4%A + 4,
--R                  7      6       5       4      3       2
--R          2f - 2%A  + 9%A  - 18%A  + 19%A  - 4%A  - 10%A  + 2%A + 4]
--R       ]
--R     ,
--R
--R                     4      2
--R     [complexRoots= ?  + 12?  + 144,
--R
--R       coordinates =
--R                  2               2               2               2
--R         [12a - %A  - 12, 12b - %A  - 12, 12c - %A  - 12, 12d - %A  - 12,
--R                  2                    2
--R          6e + %A  + 3%A + 12, 6f + %A  - 3%A + 12]
--R       ]
--R     ,
```

```
--R
--R                    4     3      2
--R     [complexRoots= ?  + 6?  + 30?  + 36? + 36,
--R
--R       coordinates =
--R                 3      2                     3      2
--R         [6a - %A  - 5%A  - 24%A - 6, 30b + %A  + 5%A  + 30%A + 6,
--R                 3      2                     3      2
--R          30c + %A  + 5%A  + 30%A + 6, 30d + %A  + 5%A  + 30%A + 6,
--R                 3      2                     3      2
--R          30e + %A  + 5%A  + 30%A + 6, 30f + %A  + 5%A  + 6]
--R       ]
--R     ,
--R
--R                    4     3      2
--R     [complexRoots= ?  - 6?  + 30?  - 36? + 36,
--R
--R       coordinates =
--R                 3      2                     3      2
--R         [6a - %A  + 5%A  - 24%A + 6, 30b + %A  - 5%A  + 30%A - 6,
--R                 3      2                     3      2
--R          30c + %A  - 5%A  + 30%A - 6, 30d + %A  - 5%A  + 30%A - 6,
--R                 3      2                     3      2
--R          30e + %A  - 5%A  + 30%A - 6, 30f + %A  - 5%A  - 6]
--R       ]
--R     ,
--R
--R                    4      2
--R     [complexRoots= ?  + 12?  + 144,
--R
--R       coordinates =
--R                  2               2               2               2
--R         [12a + %A  + 12, 12b + %A  + 12, 12c + %A  + 12, 12d + %A  + 12,
--R                  2                    2
--R          6e - %A  + 3%A - 12, 6f - %A  - 3%A - 12]
--R       ]
--R     ,
--R
--R                    2
--R     [complexRoots= ?  - 12,
--R      coordinates= [a - 1,b - 1,c - 1,d - 1,2e + %A + 4,2f - %A + 4]]
--R     ,
--R
--R                    2
--R     [complexRoots= ?  + 6? + 6,
--R      coordinates= [a + %A + 5,b - 1,c - 1,d - 1,e - 1,f - %A - 1]]
--R     ,
--R
--R                    2
--R     [complexRoots= ?  - 6? + 6,
```

```
--R      coordinates= [a + %A - 5,b + 1,c + 1,d + 1,e + 1,f - %A + 1]]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  - 12,
--R      coordinates= [a + 1,b + 1,c + 1,d + 1,2e + %A - 4,2f - %A - 4]]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  + 6?  + 30?  + 36? + 36,
--R
--R       coordinates =
--R                  3      2                      3      2
--R         [30a - %A  - 5%A  - 30%A - 6, 30b - %A  - 5%A  - 30%A - 6,
--R                  3      2                      3      2
--R          30c - %A  - 5%A  - 30%A - 6, 6d + %A  + 5%A  + 24%A + 6,
--R                  3      2               3      2
--R          30e - %A  - 5%A  - 6, 30f - %A  - 5%A  - 30%A - 6]
--R       ]
--R     ,
--R
--R                     4     3      2
--R     [complexRoots= ?  - 6?  + 30?  - 36? + 36,
--R
--R       coordinates =
--R                  3      2                      3      2
--R         [30a - %A  + 5%A  - 30%A + 6, 30b - %A  + 5%A  - 30%A + 6,
--R                  3      2                      3      2
--R          30c - %A  + 5%A  - 30%A + 6, 6d + %A  - 5%A  + 24%A - 6,
--R                  3      2               3      2
--R          30e - %A  + 5%A  + 6, 30f - %A  + 5%A  - 30%A + 6]
--R       ]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  + 6? + 6,
--R      coordinates= [a + 1,b + 1,c + 1,d - %A - 5,e + %A + 1,f + 1]]
--R     ,
--R
--R                     2
--R     [complexRoots= ?  - 6? + 6,
--R      coordinates= [a - 1,b - 1,c - 1,d - %A + 5,e + %A - 1,f - 1]]
--R     ]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynom
--E 21

--S 22 of 22
concat [realSolve(ts)$zdpack for ts in lts]
--R
--R
```

```
--R   (22)
--R   [[%B1,%B1,%B1,%B5,- %B5 - 4%B1,%B1], [%B1,%B1,%B1,%B6,- %B6 - 4%B1,%B1],
--R    [%B2,%B2,%B2,%B3,- %B3 - 4%B2,%B2], [%B2,%B2,%B2,%B4,- %B4 - 4%B2,%B2],
--R    [%B7,%B7,%B7,%B7,%B11,- %B11 - 4%B7], [%B7,%B7,%B7,%B7,%B12,- %B12 - 4%B7],
--R    [%B8,%B8,%B8,%B8,%B9,- %B9 - 4%B8], [%B8,%B8,%B8,%B8,%B10,- %B10 - 4%B8],
--R    [%B13,%B13,%B17,- %B17 - 4%B13,%B13,%B13],
--R    [%B13,%B13,%B18,- %B18 - 4%B13,%B13,%B13],
--R    [%B14,%B14,%B15,- %B15 - 4%B14,%B14,%B14],
--R    [%B14,%B14,%B16,- %B16 - 4%B14,%B14,%B14],
--R
--R     [%B19, %B29,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B19   - ---------- %B19   - ----------- %B19
--R         6006689520           2002229840            49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------ %B19   - ------------ %B19  - ----------- %B19
--R             3003344760             2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------ %B19   + ------------ %B19   + --------------- %B19
--R           1387545279120          1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B19   + --------------- %B19  - ------------ %B19
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B19   - ------------ %B19   - --------------- %B19
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B19   - --------------- %B19  - ------------ %B19
--R             77085848840            154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B29 - ------------ %B19   + ------------ %B19
--R                  1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B19   + ---------------- %B19
--R           693772639560            693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B19  + -------------- %B19
--R           1387545279120           1387545279120
```

```
--R       ]
--R     ,
--R
--R     [%B19, %B30,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B19   - ---------- %B19   - ----------- %B19
--R         6006689520           2002229840             49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B19   - ------------- %B19  - ----------- %B19
--R             3003344760             2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B19   + ------------- %B19   + --------------- %B19
--R           1387545279120          1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B19   + --------------- %B19  - ------------ %B19
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B19   - ------------- %B19   - --------------- %B19
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B19   - --------------- %B19  - ------------- %B19
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B30 - ------------- %B19   + ------------ %B19
--R                 1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B19   + ---------------- %B19
--R           693772639560            693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B19  + -------------- %B19
--R           1387545279120          1387545279120
--R       ]
--R     ,
--R
--R     [%B20, %B27,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B20   - ---------- %B20   - ----------- %B20
```

```
--R         6006689520           2002229840             49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B20   - ------------- %B20  - ------------ %B20
--R             3003344760             2002229840            6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B20   + ------------- %B20   + ---------------- %B20
--R           1387545279120          1387545279120             693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         ---------------- %B20   + ---------------- %B20  - ------------ %B20
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B20   - ------------- %B20   - ---------------- %B20
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B20   - ---------------- %B20  - ------------- %B20
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B27 - ------------- %B20   + ------------ %B20
--R                  1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         ---------------- %B20   + ----------------- %B20
--R           693772639560             693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B20  + -------------- %B20
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R
--R     [%B20, %B28,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B20   - ---------- %B20   - ----------- %B20
--R         6006689520           2002229840             49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B20   - ------------- %B20  - ----------- %B20
--R             3003344760             2002229840            6006689520
--R       ,
--R
```

```
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B20   + ------------- %B20   + --------------- %B20
--R           1387545279120           1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B20   + --------------- %B20  - ------------ %B20
--R           693772639560            1387545279120           198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B20   - ------------- %B20   - --------------- %B20
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B20   - --------------- %B20  - ------------- %B20
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                   4321823003      31   180949546069     25
--R         - %B28 - ------------- %B20   + ------------ %B20
--R                 1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B20   + ---------------- %B20
--R           693772639560            693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B20  + -------------- %B20
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R
--R     [%B21, %B25,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B21   - ---------- %B21   - ----------- %B21
--R         6006689520          2002229840            49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B21   - ------------- %B21  - ----------- %B21
--R             3003344760            2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B21   + ------------- %B21   + --------------- %B21
--R           1387545279120           1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B21   + --------------- %B21  - ------------ %B21
--R           693772639560            1387545279120           198220754160
```

```
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B21   - ------------- %B21   - --------------- %B21
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B21   - ---------------- %B21  - ------------- %B21
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B25 - ------------- %B21   + ------------ %B21
--R                  1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         ---------------- %B21   + ----------------- %B21
--R           693772639560             693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ----------------- %B21  + --------------- %B21
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R
--R     [%B21, %B26,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B21   - ---------- %B21   - ----------- %B21
--R         6006689520          2002229840            49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B21   - ------------- %B21  - ------------ %B21
--R             3003344760             2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B21   + ------------- %B21   + ---------------- %B21
--R           1387545279120          1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         ---------------- %B21   + ----------------- %B21  - ------------- %B21
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B21   - ------------- %B21   - --------------- %B21
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
```

```
--R         - -------------- %B21   - --------------- %B21  - ------------- %B21
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B26 - ------------- %B21   + ------------ %B21
--R                  1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B21   + ---------------- %B21
--R           693772639560             693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B21  + -------------- %B21
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R
--R     [%B22, %B23,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B22   - ---------- %B22   - ----------- %B22
--R         6006689520           2002229840            49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B22   - ------------- %B22  - ----------- %B22
--R             3003344760             2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B22   + ------------- %B22   + --------------- %B22
--R           1387545279120           1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B22   + --------------- %B22  - ------------ %B22
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B22   - ------------- %B22   - --------------- %B22
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B22   - --------------- %B22  - ------------- %B22
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B23 - ------------- %B22   + ------------ %B22
--R                  1387545279120           22746643920
```

```
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B22   + ---------------- %B22
--R           693772639560             693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B22  + -------------- %B22
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R
--R     [%B22, %B24,
--R
--R           7865521      31   6696179241     25   25769893181     19
--R         ---------- %B22   - ---------- %B22   - ----------- %B22
--R         6006689520          2002229840            49235160
--R       +
--R           1975912990729     13   1048460696489     7   21252634831
--R         - ------------- %B22   - ------------- %B22  - ------------ %B22
--R             3003344760             2002229840           6006689520
--R       ,
--R
--R             778171189       31   1987468196267     25   155496778477189     19
--R         - ------------- %B22   + ------------- %B22   + --------------- %B22
--R           1387545279120          1387545279120            693772639560
--R       +
--R         191631411158401     13   300335488637543     7   755656433863
--R         --------------- %B22   + --------------- %B22  - ------------ %B22
--R           693772639560            1387545279120          198220754160
--R       ,
--R
--R          1094352947      31   2794979430821     25   218708802908737     19
--R         ------------ %B22   - ------------- %B22   - --------------- %B22
--R         462515093040           462515093040            231257546520
--R       +
--R           91476663003591     13   145152550961823     7   1564893370717
--R         - -------------- %B22   - --------------- %B22  - ------------- %B22
--R             77085848840             154171697680           462515093040
--R       ,
--R
--R                    4321823003      31   180949546069     25
--R         - %B24 - ------------- %B22   + ------------ %B22
--R                  1387545279120           22746643920
--R       +
--R         863753195062493     19   1088094456732317     13
--R         --------------- %B22   + ---------------- %B22
--R           693772639560             693772639560
--R       +
--R         1732620732685741     7   13506088516033
--R         ---------------- %B22  + -------------- %B22
```

```
--R           1387545279120           1387545279120
--R       ]
--R     ,
--R    [%B31,%B35,- %B35 - 4%B31,%B31,%B31,%B31],
--R    [%B31,%B36,- %B36 - 4%B31,%B31,%B31,%B31],
--R    [%B32,%B33,- %B33 - 4%B32,%B32,%B32,%B32],
--R    [%B32,%B34,- %B34 - 4%B32,%B32,%B32,%B32]]
--R                              Type: List(List(RealClosure(Fraction(Integer))))
--E 22
)spool
)lisp (bye)
```

———-

— LexTriangularPackage.help —

```
====================================================================
LexTriangularPackage examples
====================================================================

The LexTriangularPackage package constructor provides an
implementation of the lexTriangular algorithm (D. Lazard "Solving
Zero-dimensional Algebraic Systems", J. of Symbol. Comput., 1992).
This algorithm decomposes a zero-dimensional variety into zero-sets of
regular triangular sets.  Thus the input system must have a finite
number of complex solutions.  Moreover, this system needs to be a
lexicographical Groebner basis.

This package takes two arguments: the coefficient-ring R of the
polynomials, which must be a GcdDomain and their set of variables
given by ls a List Symbol.  The type of the input polynomials must be
NewSparseMultivariatePolynomial(R,V) where V is OrderedVariableList(ls).
The abbreviation for LexTriangularPackage is LEXTRIPK.  The main operations
are lexTriangular and squareFreeLexTriangular.  The later provide
decompositions by means of square-free regular triangular sets, built
with the SquareFreeRegularTriangularSet constructor, whereas the former
uses the RegularTriangularSet constructor.  Note that these constructors
also implement another algorithm for solving algebraic systems by
means of regular triangular sets; in that case no computations of
Groebner bases are needed and the input system may have any dimension
(i.e. it may have an infinite number of solutions).

The implementation of the lexTriangular algorithm provided in the
LexTriangularPackage constructor differs from that reported in
"Computations of gcd over algebraic towers of simple extensions" by
M. Moreno Maza and R. Rioboo (in proceedings of AAECC11, Paris, 1995).
Indeed, the squareFreeLexTriangular operation removes all multiplicities
of the solutions (i.e. the computed solutions are pairwise different)
and the lexTriangular operation may keep some multiplicities; this
```

later operation runs generally faster than the former.

The interest of the lexTriangular algorithm is due to the
following experimental remark. For some examples, a triangular
decomposition of a zero-dimensional variety can be computed faster via
a lexicographical Groebner basis computation than by using a direct method
(like that of SquareFreeRegularTriangularSet and RegularTriangularSet).
This happens typically when the total degree of the system relies
essentially on its smallest variable (like in the Katsura systems).
When this is not the case, the direct method may give better timings
(like in the Rose system).

Of course, the direct method can also be applied to a lexicographical
Groebner basis. However, the lexTriangular algorithm takes advantage
of the structure of this basis and avoids many unnecessary computations
which are performed by the direct method.

For this purpose of solving algebraic systems with a finite number of
solutions, see also the ZeroDimensionalSolvePackage. It allows
to use both strategies (the lexTriangular algorithm and the direct
method) for computing either the complex or real roots of a system.

Note that the way of understanding triangular decompositions is
detailed in the example of the RegularTriangularSet constructor.

Since the LexTriangularPackage package constructor is limited to
zero-dimensional systems, it provides a zeroDimensional? operation to
check whether this requirement holds. There is also a groebner operation
to compute the lexicographical Groebner basis of a set of polynomials
with type NewSparseMultivariatePolynomial(R,V). The elimination ordering
is that given by ls (the greatest variable being the first element
of ls). This basis is computed by the FLGM algorithm (Faugere et al.
"Efficient Computation of Zero-Dimensional Groebner Bases by Change
of Ordering" , J. of Symbol. Comput., 1993) implemented in the
LinGroebnerPackage package constructor.

Once a lexicographical Groebner basis is computed, then one can call
the operations lexTriangular and squareFreeLexTriangular. Note that
these operations admit an optional argument to produce normalized
triangular sets. There is also a zeroSetSplit operation which does
all the job from the input system; an error is produced if this system
is not zero-dimensional.

Let us illustrate the facilities of the LexTriangularPackage constructor
by a famous example, the cyclic-6 root system.

Define the coefficient ring.

```
  R := Integer
   Integer
```

```
                      Type: Domain

Define the list of variables,

  ls : List Symbol := [a,b,c,d,e,f]
    [a,b,c,d,e,f]
                      Type: List Symbol

and make it an ordered set.

  V := OVAR(ls)
    OrderedVariableList [a,b,c,d,e,f]
                      Type: Domain

Define the polynomial ring.

  P := NSMP(R, V)
   NewSparseMultivariatePolynomial(Integer,OrderedVariableList [a,b,c,d,e,f])
                      Type: Domain

Define the polynomials.

  p1: P :=  a*b*c*d*e*f - 1
   f e d c b a - 1
    Type: NewSparseMultivariatePolynomial(Integer,
                                          OrderedVariableList [a,b,c,d,e,f])

  p2: P := a*b*c*d*e +a*b*c*d*f +a*b*c*e*f +a*b*d*e*f +a*c*d*e*f +b*c*d*e*f
   ((((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b
    Type: NewSparseMultivariatePolynomial(Integer,
                                          OrderedVariableList [a,b,c,d,e,f])

  p3: P :=  a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f
   (((d + f)c + f e)b + f e d)a + e d c b + f e d c
    Type: NewSparseMultivariatePolynomial(Integer,
                                          OrderedVariableList [a,b,c,d,e,f])

  p4: P := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f
   ((c + f)b + f e)a + d c b + e d c + f e d
    Type: NewSparseMultivariatePolynomial(Integer,
                                          OrderedVariableList [a,b,c,d,e,f])

  p5: P := a*b + a*f + b*c + c*d + d*e + e*f
   (b + f)a + c b + d c + e d + f e
    Type: NewSparseMultivariatePolynomial(Integer,
                                          OrderedVariableList [a,b,c,d,e,f])

  p6: P := a + b + c + d + e + f
   a + b + c + d + e + f
    Type: NewSparseMultivariatePolynomial(Integer,
```

```
                                        OrderedVariableList [a,b,c,d,e,f])

  lp := [p1, p2, p3, p4, p5, p6]
    [f e d c b a - 1, ((((e + f)d + f e)c + f e d)b + f e d c)a + f e d c b,
     (((d + f)c + f e)b + f e d)a + e d c b + f e d c,
     ((c + f)b + f e)a + d c b + e d c + f e d,
     (b + f)a + c b + d c + e d + f e, a + b + c + d + e + f]
   Type: List NewSparseMultivariatePolynomial(Integer,
                                         OrderedVariableList [a,b,c,d,e,f])
```

Now call LEXTRIPK.

```
  lextripack :=  LEXTRIPK(R,ls)
    LexTriangularPackage(Integer,[a,b,c,d,e,f])
                        Type: Domain
```

Compute the lexicographical Groebner basis of the system. This may take between 5 minutes and one hour, depending on your machine.

```
  lg := groebner(lp)$lextripack
   [a + b + c + d + e + f,
                        2                                          2
     3968379498283200b  + 15873517993132800f b + 3968379498283200d
    +
                                              3 5                    4 4
     15873517993132800f d + 3968379498283200f e  - 15873517993132800f e
    +
                        5 3                       6                      2
     23810276989699200f e  + (206355733910726400f  + 230166010900425600)e
    +
                            43                        37
          - 729705987316687f   + 18636674968672054 21f
       +
                               31                            25
         2916748537717311044 61f   + 365285994691106921745f
       +
                             19                             13
         549961185828911895f   - 365048404038768439269f
       +
                                7
         - 2923828204315040276 69f  - 2271898467631865497f
     *
        e
    +
                          44                          38
      - 3988812642545399f   + 10187423878429609997f
    +
                           32                           26
     1594377523424314053637f   + 19947393084399162380 65f
    +
```

```
                      20                            14
  1596840088052642815f   - 1993494118301162145413f
+
                        8                          2
  - 15960497422896898150 53f  - 11488171330159667449f
,
                                                                2
  (23810276989699200c - 23810276989699200f)b + 23810276989699200c
+
                                        2
  71430830969097600f c - 23810276989699200d  - 95241107958796800f d
+
                      3 5                       4 4                       5 3
  - 55557312975964800f e  + 174608697924460800f e  - 174608697924460800f e
+
                         6                         2
  (- 2428648252949318400f  - 2611193709870345600)e
+
                       43                         37
      8305444561289527f   - 21212087151945459641f
    +
                                31                           25
      - 3319815883093451385381f   - 41576916462616571364 45f
    +
                             19                            13
      - 6072721607510764095f   + 41549867090364602216 49f
    +
                              7
      33277613111385870967 49f  + 25885340608290841637f
 *
    e
+
                     44                          38
  45815897629010329f   - 1170137655821518912 07f
+
                              32                             26
  - 18313166848970865074187f   - 22909971239649297438915f
+
                           20                              14
  - 161332507613051572 65f   + 22897305857636178256623f
+
                            8                            2
  183299447818672424979 23f  + 1302585310020204206 99f
,
   (7936758996566400d - 7936758996566400f)b - 7936758996566400f d
+
                      3 5                       4 4                       5 3
  - 7936758996566400f e  + 23810276989699200f e  - 23810276989699200f e
+
                         6                         2
```

```
  (- 337312257354072000f  - 369059293340337600)e
+
                       43                       37
     1176345388640471f   - 3004383582891473073f
   +
                            31                          25
     - 4702035027072461056 53f   - 588858183402644348085f
   +
                          19                          13
     - 856939308623513535f   + 5884726742423405263 77f
   +
                           7
     4713132419583711035 17f  + 3659742549078552381f
 *
   e
+
                  44                        38                            32
  6423170513956901f   - 16404772137036480803f   - 25674191652275287744 63f
+
                           26                        20
  - 3211938090825682172335f   - 2330490332697587485f
+
                         14                           8
  32101001094447548645 87f   + 25698583153951626178 47f
+
                        2
  18326089487427735751f
,
                                                                    3 5
  (11905138494849600e - 11905138494849600f)b - 3968379498283200f e
+
                    4 4                    5 3
  15873517993132800f e  - 27778656487982400f e
+
                       6                       2
  (- 208339923659868000f  - 240086959646133600)e
+
                      43                       37
     786029984751110f   - 2007519008182245250f
   +
                           31                          25
     - 3141880629080738070 90f   - 3934236675379295752 50f
   +
                          19                          13
     - 550329120654394950f   + 3931964087288896127 70f
   +
                           7
     3148923727991764957 30f  + 2409386515146668530f
 *
   e
```

```
+
                  44                          38                            32
  4177638546747827f   - 10669685294602576381f   - 1669852980419949524601f
+
                         26                        20
  - 2089077057287904170745f   - 1569899763580278795f
+
                        14                          8
  20878640268590155733 49f   + 1671496085945199577969f
+
                       2
  11940257226216280177f
,
                      6                                          2 5
  (11905138494849600f  - 11905138494849600)b - 15873517993132800f e
+
                    3 4                     4 3
  39683794982832000f e  - 39683794982832000f e
+
                        11                      5  2
  (- 6865296532029936 00f   - 6071620632373296 00f )e
+
                     42                        36                          30
      65144531306704f   - 166381280901088652f   - 260334345024702834 72f
    +
                           24                      18
      - 31696259583860650140f   + 9714920931675813 60f
    +
                         12                        6
    322200850336913895 48f   + 255261776660705298 08f  + 138603268355749244
 *
    e
+
                  43                       37                          31
  167620036074811f   - 428102417974791473f   - 669972438012316793 13f
+
                          25                       19
  - 834267167221487504 85f   + 203673895369980765f
+
                        13                         7
  835230563260104324 57f   + 669957896402380669 37f  + 478592855549587901f
,
                3                   2                 2                45
  801692827936c  + 2405078483808f c  - 2405078483808f c - 13752945467f
+
                 39                     33                    27
  35125117815561f   + 5496946957826433f   + 6834659447749117f
+
                   21                     15                    9
  - 44484880462461f   - 6873406230093057f   - 5450844938762633f
```

```
+
                3
  1216586044571f
,
                                                                      2
  (23810276989699200d - 23810276989699200f)c + 23810276989699200d
+
                                          3 5                     4 4
  71430830969097600f d + 7936758996566400f e  - 31747035986265600f e
+
                     5 3                        6                       2
  31747035986265600f e  + (404774708824886400f  + 396837949828320000)e
+
                           43                          37
      - 1247372229446701f   + 3185785654596621203f
    +
                             31                            25
      498594866849974751463f   + 6245425458457910479 35f
    +
                           19                            13
      931085755769682885f   - 6241506635824170633 87f
    +
                               7
      - 4998818593883604756 47f  - 39268853138195273 51f
 *
    e
+
                       44                          38
  - 7026011547118141f   + 17944427051950691243f
+
                           32                            26
  28083835225939866035 43f   + 35136241423548075301 35f
+
                         20                             14
  2860757006705537685f   - 35113567356421907372 67f
+
                             8                           2
  - 2811332494697103819887f  - 203150116315228473 11f
,
   (7936758996566400e - 7936758996566400f)c
+
                         43                        37                          31
      - 4418748183673f   + 11285568707456559f   + 1765998617294451019f
    +
                           25                         19
      2173749283622606155f   - 55788292195402895f
    +
                           13                          7
    - 2215291421788292951f   - 17181426653474308 51f  + 30256569458230237f
 *
```

```
    e
+
                44                      38                      32
  4418748183673f   - 11285568707456559f   - 1765998617294451019f
+
                        26                      20                        14
  - 2173749283622606155f   + 55788292195402895f   + 22152914217882929 51f
+
                      8                     2
  17181426653474308 51f  - 302565694582302 37f
,
                   6                                  43
  (72152354514240f  - 72152354514240)c + 40950859449f
+
                    37                     31                     25
  - 104588980990367f   - 1636722739557530 7f   - 20268523416527355f
+
                  19                     13                     7
  442205002259535f   + 20576059935789063f   + 15997133796970563f
+
  - 275099892785581f
,
                    3                     2                     2
  1984189749141600d  + 5952569247424800f d  - 5952569247424800f d
+
                     4 5                    5 4                    3
  - 3968379498283200f e  + 15873517993132800f e  + 17857707742274400e
+
                       7                      2
  (- 148814231185620000f  - 162703559429611200f)e
+
                       44                     38
      - 390000914678878f   + 996062704593756434f
    +
                           32                         26
      15588632397203482 3914f   + 194745956143985421330f
    +
                      20                        14
      62050775955744 30f   - 1945965126532990 68786f
    +
                            8                       2
      - 155796897940756922666f  - 10363757590773209 78f
 *
    e
+
                    45                     39                          33
  - 374998630035991f   + 957747106595453993f   + 1498891555667648916 93f
+
                       27                     21
  1871541714434946416 85f   - 127129015426348065f
```

```
+
                        15                          9                       3
- 187241533243115040417f   - 149719983567976534037f  - 836654081239648061f
,
                                                                    3 5
  (5952569247424800e - 595256924742480 0f)d - 3968379498283200f e
+
                   4 4                    5 3
  9920948745708000f e  - 3968379498283200f e
+
                        6                       2
  (- 148814231185620000f  - 150798420934761600)e
+
                      43                        37
      492558110242553f   - 1257992359608074599f
    +
                             31                          25
      - 196883094539368513959f   - 2465621157457354280 55f
    +
                           19                           13
      - 325698701993885505f   + 2464177698836518081 11f
    +
                            7
      197327352068200652911f  + 1523373796389332143f
 *
    e
+
                   44                        38                           32
  2679481081803026f   - 6843392695421906608f   - 1071020459642646913578f
+
                           26                        20
  - 13397891696920412400 60f   - 852746750910750210f
+
                         14                            8
  1339105101971878401312f   + 1071900289758712984762f
+
                       2
  7555239072072727756f
,
                       6                                          2 5
  (11905138494849600f  - 11905138494849600)d - 7936758996566400f e
+
                     3 4                      4 3
  31747035986265600f e  - 31747035986265600f e
+
                         11                       5  2
  (- 420648226818019200f   - 404774708824886400f )e
+
                       42                        36                        30
      153361876008 89f   - 39169739565161107f   - 61271761274896908 27f
```

```
    +
                            24                       18
      - 7217708742310509615f   + 538628483890722735f
    +
                           12                       6
      7506804353843507643f   + 5886160769782607203f  + 63576108396535879
 *
    e
+
                 43                      37                         31
  71737781777066f   - 183218856207557938f   - 28672874271132276078f
+
                          25                       19
  - 356252236869398120l0f   + 164831339634084390f
+
                         13                        7
  357241604230730526 2f   + 28627022578664910622f  + 1874599870296805 6f
,
                     6                       5                    2 4
  1322793166094400e  - 3968379498283200f e  + 3968379498283200f e
+
                      3 3
  - 5291172664377600f e
+
                         10                       4  2
  (- 230166010900425600f   - 226197631402142400f )e
+
                           47                          41
      - 152375364610443885f   + 3891666260648548 0f
    +
                              35                             29
      6090609784136055898 335f   + 7616736793460879869 7275f
    +
                            23                             17
      27855066785995181125f   - 761449528170527231 45495f
    +
                                11                            5
      - 60933629892463517546975f   - 411415071682002547795f
 *
    e
+
                     42                       36                          30
  - 209493533143822f   + 535045979490560586f   + 837379479649735531 6f
+
                          24                       18
  1048895070842133715 0f   + 167117997269207870f
+
                            12                        6
  - 1047937257813906155 4f   - 838426851899031803 4f  - 569978796672974242
,
```

```
                   6                   3
  (25438330117200f  + 25438330117200)e
+
                  7                   2
  (76314990351600f  + 76314990351600f)e
+
                      44                    38                      32
    - 1594966552735f   + 4073543370415745f   + 637527159231148925f
    +
                         26                   20                       14
    7975211761136065 25f   + 530440941097175f   - 797160527306433145f
    +
                           8                   2
    - 638132320196044965f  - 4510507167940725f
 *
    e
+
                  45                     39                      33
  - 6036376800443f   + 15416903421476909f   + 2412807646192304449f
+
                      27                    21                      15
  3017679923028013705f   + 1422320037411955f   - 3016560402417843941f
+
                        9                   3
  - 24142493681830331 61f  - 16561862361763873f
,
                  12                  2
  (1387545279120f   - 1387545279120)e
+
                 43                  37                    31
    4321823003f   - 11037922310209f   - 1727510711947989f
    +
                       25                 19                    13
    - 2165150991154425f   - 5114342560755f   + 2162682824948601f
    +
                     7
    1732620732685741f  + 13506088516033f
 *
    e
+
              44                  38                    32
  24177661775f   - 61749727185325f   - 9664106795754225f
+
                      26                 20                     14
  - 12090487758628245f   - 8787672733575f   + 12083693383005045f
+
                   8                  2
  9672870290826025f  + 68544102808525f
,
48        42          36          30          18          12        6
```

```
   f   - 2554f   - 399710f   - 499722f   + 499722f   + 399710f   + 2554f  - 1]
  Type: List NewSparseMultivariatePolynomial(Integer,
    OrderedVariableList [a,b,c,d,e,f])
```

Apply lexTriangular to compute a decomposition into regular triangular sets.
This should not take more than 5 seconds.

```
  lexTriangular(lg,false)$lextripack
  [
     6       6       5     2 4     3 3     4 2     5
   {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
          2 5     3 4     4 3     5 2
    3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
          2 5     3 4     4 3      5 2
    3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
         2 5     3 4     4 3     5 2
    a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
   ,
    6                  2           2                    2
  {f  - 1,e - f,d - f,c  + 4f c + f ,(c - f)b - f c - 5f ,a + b + c + 3f},
    6                        2           2
  {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
    6            2           2                    2
  {f  - 1,e - f,d  + 4f d + f ,(d - f)c - f d - 5f ,b - f,a + c + d + 3f},
      36        30          24          18          12        6
   {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
                  12              2
       (161718564f   - 161718564)e
     +
                    31              25                19                13
           - 504205f   + 1287737951f   + 201539391380f   + 253982817368f
         +
                       7
           201940704665f  + 1574134601f
      *
         e
     +
                32              26                  20                  14
       - 2818405f   + 7198203911f   + 1126548149060f   + 1416530563364f
     +
                   8              2
       1127377589345f  + 7988820725f
     ,
                    6                                    2 5                 3 4
       (693772639560f  - 693772639560)d - 462515093040f e  + 1850060372160f e
     +
                     4 3                      11                  5  2
       - 1850060372160f e  + (- 24513299931120f   - 23588269745040f )e
     +
                     30                 24                   18
```

```
          - 890810428f   + 2275181044754f   + 355937263869776f
        +
                           12                   6
          413736880104344f   + 3428493044487996f  + 3704966481878
     *
        e
    +
                   31                  25                    19
      - 4163798003f   + 10634395752169f   + 1664161760192806f
    +
                     13                    7
      2079424391370694f   + 1668153650635921f  + 10924274392693f
    ,
                    6                            31               25
      (12614047992f  - 12614047992)c - 7246825f   + 18508536599f
    +
                   19                 13                 7
      2896249516034f   + 3581539649666f   + 2796477571739f  - 48094301893f
    ,
                    6                                  2 5                 3 4
      (693772639560f  - 693772639560)b - 925030186080f e  + 2312575465200f e
    +
                     4 3                     11                   5  2
      - 2312575465200f e  + (- 40007555547960f   - 35382404617560f )e
    +
                      30                 24                    18
          - 3781280823f   + 9657492291789f   + 1511158913397906f
        +
                           12                   6
          1837290892286154f   + 1487216006594361f  + 8077238712093
     *
        e
    +
                   31                  25                    19
      - 9736390478f   + 24866827916734f   + 3891495681905296f
    +
                     13                    7
      4872556418871424f   + 3904047887269606f  + 27890075838538f
    ,
   a + b + c + d + e + f}
  ,
   6      2           2                   2
 {f  - 1,e  + 4f e + f ,(e - f)d - f e - 5f ,c - f,b - f,a + d + e + 3f}]
                           Type: List RegularChain(Integer,[a,b,c,d,e,f])
```

Note that the first set of the decomposition is normalized (all initials are integer numbers) but not the second one (normalized triangular sets are defined in the description of the NormalizedTriangularSetCategory constructor).

So apply now lexTriangular to produce normalized triangular sets.

```
  lts := lexTriangular(lg,true)$lextripack
   [
      6       6       5     2 4     3 3     4 2     5
    {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
           2 5     3 4     4 3     5 2
     3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
            2 5     3 4     4 3      5 2
     3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
          2 5     3 4     4 3     5 2
     a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
    ,
     6                  2           2
    {f  - 1,e - f,d - f,c  + 4f c + f ,b + c + 4f,a - f},
     6                    2           2
    {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
     6            2           2
    {f  - 1,e - f,d  + 4f d + f ,c + d + 4f,b - f,a - f},
        36        30          24          18          12        6
     {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
                       2
         1387545279120e
       +
                        31                  25                    19
           4321823003f   - 11037922310209f   - 1727506390124986f
         +
                             13                    7
           - 2176188913464634f   - 1732620732685741f  - 13506088516033f
       *
          e
       +
                    32                  26                    20
         24177661775f   - 61749727185325f   - 9664082618092450f
       +
                           14                    8                  2
         - 12152237485813570f   - 9672870290826025f  - 68544102808525f
       ,
          1387545279120d
       +
                         30                  24                    18
           - 1128983050f   + 2883434331830f   + 451234998755840f
         +
                           12                    6
           562426491685760f   + 447129055314890f  - 165557857270
       *
          e
       +
                     31                  25                    19
         - 1816935351f   + 4640452214013f   + 726247129626942f
```

```
      +
                        13                    7
        912871801716798f   + 726583262666877f  + 4909358645961f
      ,
                                    31                 25                    19
        1387545279120c + 778171189f   - 1987468196267f   - 310993556954378f
      +
                          13                   7
        - 383262822316802f   - 300335488637543f  + 5289595037041f
      ,
         1387545279120b
      +
                       30                  24                    18
             1128983050f   - 2883434331830f   - 451234998755840f
           +
                             12                    6
             - 562426491685760f   - 447129055314890f  + 165557857270
       *
          e
      +
                     31                  25                    19
        - 3283058841f   + 8384938292463f   + 1312252817452422f
      +
                       13                    7
        1646579934064638f   + 1306372958656407f  + 4694680112151f
      ,
                                                      31                   25
        1387545279120a + 1387545279120e + 4321823003f   - 11037922310209f
      +
                           19                    13                    7
        - 1727506390124986f   - 2176188913464634f   - 1732620732685741f
      +
        - 13506088516033f
      }
    ,
     6      2           2
   {f  - 1,e  + 4f e + f ,d + e + 4f,c - f,b - f,a - f}]
                              Type: List RegularChain(Integer,[a,b,c,d,e,f])
```

We check that all initials are constant.

```
  [ [init(p) for p in (ts :: List(P))] for ts in lts]
   [[1,3,1,3,1,1], [1,1,1,1,1,1], [1,1,1,1,1,1], [1,1,1,1,1,1],
    [1387545279120,1387545279120,1387545279120,1387545279120,1387545279120,1],
    [1,1,1,1,1,1]]
     Type: List List NewSparseMultivariatePolynomial(Integer,
                                      OrderedVariableList [a,b,c,d,e,f])
```

Note that each triangular set in lts is a lexicographical Groebner basis. Recall that a point belongs to the variety associated with lp

```
if and only if it belongs to that associated with one triangular set
ts in lts.

By running the squareFreeLexTriangular operation, we retrieve the
above decomposition.

  squareFreeLexTriangular(lg,true)$lextripack
  [
      6       6       5     2 4     3 3     4 2     5
    {f  + 1, e  - 3f e  + 3f e  - 4f e  + 3f e  - 3f e - 1,
           2 5     3 4     4 3     5 2
     3d + f e  - 4f e  + 4f e  - 2f e  - 2e + 2f, c + f,
           2 5     3 4     4 3      5 2
     3b + 2f e  - 5f e  + 5f e  - 10f e  - 4e + 7f,
          2 5     3 4     4 3     5 2
     a - f e  + 3f e  - 3f e  + 4f e  + 3e - 3f}
    ,
     6                  2           2
   {f  - 1,e - f,d - f,c  + 4f c + f ,b + c + 4f,a - f},
     6                        2           2
   {f  - 1,e - f,d - f,c - f,b  + 4f b + f ,a + b + 4f},
     6            2           2
   {f  - 1,e - f,d  + 4f d + f ,c + d + 4f,b - f,a - f},
      36        30          24          18          12        6
    {f   - 2554f   - 399709f   - 502276f   - 399709f   - 2554f  + 1,
                       2
        1387545279120e
      +
                       31                  25                     19
           4321823003f   - 11037922310209f   - 1727506390124986f
         +
                                13                    7
           - 2176188913464634f   - 1732620732685741f  - 13506088516033f
       *
          e
      +
                  32                  26                    20
        24177661775f   - 61749727185325f   - 9664082618092450f
      +
                             14                    8                  2
        - 12152237485813570f   - 9672870290826025f  - 68544102808525f
      ,
         1387545279120d
      +
                        30                  24                    18
           - 1128983050f   + 2883434331830f   + 451234998755840f
          +
                             12                   6
           562426491685760f   + 447129055314890f  - 165557857270
       *
```

```
        e
     +
                   31                  25                    19
       - 1816935351f   + 4640452214013f   + 726247129626942f
     +
                     13                    7
       912871801716798f   + 726583262666877f  + 4909358645961f
     ,
                                  31                 25                    19
       1387545279120c + 778171189f   - 1987468196267f   - 310993556954378f
     +
                       13                    7
       - 383262822316802f   - 300335488637543f  + 5289595037041f
     ,
        1387545279120b
     +
                    30                  24                    18
          1128983050f   - 2883434331830f   - 451234998755840f
        +
                          12                    6
          - 562426491685760f   - 447129055314890f  + 165557857270
      *
        e
     +
                   31                  25                    19
       - 3283058841f   + 8384938292463f   + 1312252817452422f
     +
                     13                    7
       1646579934064638f   + 1306372958656407f  + 4694680112151f
     ,
                                                      31                  25
       1387545279120a + 1387545279120e + 4321823003f   - 11037922310209f
     +
                         19                    13                    7
       - 1727506390124986f   - 2176188913464634f   - 1732620732685741f
     +
       - 13506088516033f
     }
   ,
    6      2           2
  {f  - 1,e  + 4f e + f ,d + e + 4f,c - f,b - f,a - f}]
 Type: List SquareFreeRegularTriangularSet(Integer,
             IndexedExponents OrderedVariableList [a,b,c,d,e,f],
             OrderedVariableList [a,b,c,d,e,f],
             NewSparseMultivariatePolynomial(Integer,
                         OrderedVariableList [a,b,c,d,e,f]))
```

Thus the solutions given by lts are pairwise different.

We count them as follows.

```
  reduce(+,[degree(ts) for ts in lts])
    156
                        Type: PositiveInteger
```

We can investigate the triangular decomposition lts by using the ZeroDimensionalSolvePackage.

This requires to add an extra variable (smaller than the others) as follows.

```
  ls2 : List Symbol := concat(ls,new()$Symbol)
    [a,b,c,d,e,f,%A]
                        Type: List Symbol
```

Then we call the package.

```
  zdpack := ZDSOLVE(R,ls,ls2)
   (20)  ZeroDimensionalSolvePackage(Integer,[a,b,c,d,e,f],[a,b,c,d,e,f,%A])
                        Type: Domain
```

We compute a univariate representation of the variety associated with the input system as follows.

```
  concat [univariateSolve(ts)$zdpack for ts in lts]
  [
                    4      2
    [complexRoots= ?  - 13?  + 49,
       coordinates =
                 3                3                3                3
        [7a + %A  - 6%A, 21b + %A  + %A, 21c - 2%A  + 19%A, 7d - %A  + 6%A,
                  3                3
         21e - %A  - %A, 21f + 2%A  - 19%A]
      ]
    ,
                    4      2
    [complexRoots= ?  + 11?  + 49,
       coordinates =
                 3                3                3
        [35a + 3%A  + 19%A, 35b + %A  + 18%A, 35c - 2%A  - %A,
                 3                3                3
         35d - 3%A  - 19%A, 35e - %A  - 18%A, 35f + 2%A  + %A]
      ]
    ,
     [
      complexRoots =
         8      7      6       5       4       3       2
        ?  - 12?  + 58?  - 120?  + 207?  - 360?  + 802?  - 1332? + 1369
      ,
       coordinates =
        [
```

```
                        7           6            5            4
      43054532a + 33782%A  - 546673%A  + 3127348%A  - 6927123%A
     +
                 3             2
      4365212%A  - 25086957%A  + 39582814%A - 107313172
     ,
                        7           6            5            4
      43054532b - 33782%A  + 546673%A  - 3127348%A  + 6927123%A
     +
                 3             2
      - 4365212%A  + 25086957%A  - 39582814%A + 107313172
     ,
                        7           6            5            4
      21527266c - 22306%A  + 263139%A  - 1166076%A  + 1821805%A
     +
                 3             2
      - 2892788%A  + 10322663%A  - 9026596%A + 12950740
     ,
                        7           6            5            4
      43054532d + 22306%A  - 263139%A  + 1166076%A  - 1821805%A
     +
                 3             2
      2892788%A  - 10322663%A  + 30553862%A - 12950740
     ,
                        7           6            5            4
      43054532e - 22306%A  + 263139%A  - 1166076%A  + 1821805%A
     +
                 3             2
      - 2892788%A  + 10322663%A  - 30553862%A + 12950740
     ,
                        7           6            5            4
      21527266f + 22306%A  - 263139%A  + 1166076%A  - 1821805%A
     +
                 3             2
      2892788%A  - 10322663%A  + 9026596%A - 12950740
     ]
   ]
  ,
   [
    complexRoots =
       8      7      6       5       4       3       2
      ?  + 12?  + 58?  + 120?  + 207?  + 360?  + 802?  + 1332? + 1369
    ,
     coordinates =
      [
                        7           6            5            4
      43054532a + 33782%A  + 546673%A  + 3127348%A  + 6927123%A
     +
                 3             2
      4365212%A  + 25086957%A  + 39582814%A + 107313172
```

```
      ,
                          7           6            5            4
        43054532b - 33782%A  - 546673%A  - 3127348%A  - 6927123%A
      +
                  3             2
        - 4365212%A  - 25086957%A  - 39582814%A - 107313172
      ,
                          7           6            5            4
        21527266c - 22306%A  - 263139%A  - 1166076%A  - 1821805%A
      +
                  3             2
        - 2892788%A  - 10322663%A  - 9026596%A - 12950740
      ,
                          7           6            5            4
        43054532d + 22306%A  + 263139%A  + 1166076%A  + 1821805%A
      +
                 3             2
        2892788%A  + 10322663%A  + 30553862%A + 12950740
      ,
                          7           6            5            4
        43054532e - 22306%A  - 263139%A  - 1166076%A  - 1821805%A
      +
                  3             2
        - 2892788%A  - 10322663%A  - 30553862%A - 12950740
      ,
                          7           6            5            4
        21527266f + 22306%A  + 263139%A  + 1166076%A  + 1821805%A
      +
                 3             2
        2892788%A  + 10322663%A  + 9026596%A + 12950740
      ]
  ]
,
                 4    2
[complexRoots= ?  - ?  + 1,
                              3             3              3            3
 coordinates= [a - %A,b + %A  - %A,c + %A ,d + %A,e - %A  + %A,f - %A ]]
,
                 8     6      4      2
[complexRoots= ?  + 4?  + 12?  + 16?  + 4,
   coordinates =
              7      5       3                  7      5       3
    [4a - 2%A  - 7%A  - 20%A  - 22%A, 4b + 2%A  + 7%A  + 20%A  + 22%A,
            7      5       3                 7      5       3
     4c + %A  + 3%A  + 10%A  + 10%A, 4d + %A  + 3%A  + 10%A  + 6%A,
            7      5       3                7      5       3
     4e - %A  - 3%A  - 10%A  - 6%A, 4f - %A  - 3%A  - 10%A  - 10%A]
  ]
,
                 4     3      2
```

```
[complexRoots= ?  + 6?  + 30?  + 36? + 36,
   coordinates =
              3      2                    3      2
    [30a - %A  - 5%A  - 30%A - 6, 6b + %A  + 5%A  + 24%A + 6,
              3      2              3      2
     30c - %A  - 5%A  - 6, 30d - %A  - 5%A  - 30%A - 6,
              3      2                     3      2
     30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
  ]
,
                 4     3      2
[complexRoots= ?  - 6?  + 30?  - 36? + 36,
   coordinates =
              3      2                    3      2
    [30a - %A  + 5%A  - 30%A + 6, 6b + %A  - 5%A  + 24%A - 6,
              3      2              3      2
     30c - %A  + 5%A  + 6, 30d - %A  + 5%A  - 30%A + 6,
              3      2                     3      2
     30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
  ]
,
                 2
[complexRoots= ?  + 6? + 6,
 coordinates= [a + 1,b - %A - 5,c + %A + 1,d + 1,e + 1,f + 1]]
,
                 2
[complexRoots= ?  - 6? + 6,
 coordinates= [a - 1,b - %A + 5,c + %A - 1,d - 1,e - 1,f - 1]]
,
                 4     3      2
[complexRoots= ?  + 6?  + 30?  + 36? + 36,
   coordinates =
             3      2                     3      2
    [6a + %A  + 5%A  + 24%A + 6, 30b - %A  - 5%A  - 6,
              3      2                     3      2
     30c - %A  - 5%A  - 30%A - 6, 30d - %A  - 5%A  - 30%A - 6,
              3      2                     3      2
     30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
  ]
,
                 4     3      2
[complexRoots= ?  - 6?  + 30?  - 36? + 36,
   coordinates =
             3      2                     3      2
    [6a + %A  - 5%A  + 24%A - 6, 30b - %A  + 5%A  + 6,
              3      2                     3      2
     30c - %A  + 5%A  - 30%A + 6, 30d - %A  + 5%A  - 30%A + 6,
              3      2                     3      2
     30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
  ]
```

```
,
                2
[complexRoots= ?  + 6? + 6,
 coordinates= [a - %A - 5,b + %A + 1,c + 1,d + 1,e + 1,f + 1]]
,
                2
[complexRoots= ?  - 6? + 6,
 coordinates= [a - %A + 5,b + %A - 1,c - 1,d - 1,e - 1,f - 1]]
,
                4     3      2
[complexRoots= ?  + 6?  + 30?  + 36? + 36,
   coordinates =
             3      2                      3      2
    [30a - %A  - 5%A  - 30%A - 6, 30b - %A  - 5%A  - 30%A - 6,
             3      2                      3      2
     6c + %A  + 5%A  + 24%A + 6, 30d - %A  - 5%A  - 6,
             3      2                      3      2
     30e - %A  - 5%A  - 30%A - 6, 30f - %A  - 5%A  - 30%A - 6]
  ]
,
                4     3      2
[complexRoots= ?  - 6?  + 30?  - 36? + 36,
   coordinates =
             3      2                      3      2
    [30a - %A  + 5%A  - 30%A + 6, 30b - %A  + 5%A  - 30%A + 6,
             3      2                      3      2
     6c + %A  - 5%A  + 24%A - 6, 30d - %A  + 5%A  + 6,
             3      2                      3      2
     30e - %A  + 5%A  - 30%A + 6, 30f - %A  + 5%A  - 30%A + 6]
  ]
,
                2
[complexRoots= ?  + 6? + 6,
 coordinates= [a + 1,b + 1,c - %A - 5,d + %A + 1,e + 1,f + 1]]
,
                2
[complexRoots= ?  - 6? + 6,
 coordinates= [a - 1,b - 1,c - %A + 5,d + %A - 1,e - 1,f - 1]]
,
                8     7      6      5      4     2
[complexRoots= ?  + 6?  + 16?  + 24?  + 18?  - 8?  + 4,
   coordinates =
              7      6       5       4      3       2
    [2a + 2%A  + 9%A  + 18%A  + 19%A  + 4%A  - 10%A  - 2%A + 4,
              7      6       5       4      3       2
     2b + 2%A  + 9%A  + 18%A  + 19%A  + 4%A  - 10%A  - 4%A + 4,
             7      6      5      4      3
     2c - %A  - 4%A  - 8%A  - 9%A  - 4%A  - 2%A - 4,
             7      6      5      4      3
     2d + %A  + 4%A  + 8%A  + 9%A  + 4%A  + 2%A + 4,
```

```
             7      6       5       4      3       2
     2e - 2%A  - 9%A  - 18%A  - 19%A  - 4%A  + 10%A  + 4%A - 4,
             7      6       5       4      3       2
     2f - 2%A  - 9%A  - 18%A  - 19%A  - 4%A  + 10%A  + 2%A - 4]
  ]
,
 [
  complexRoots =
     8      7      6       5       4       3        2
    ?  + 12?  + 64?  + 192?  + 432?  + 768?  + 1024?  + 768? + 256
  ,
   coordinates =
    [
                    7        6        5         4         3         2
       1408a - 19%A  - 200%A  - 912%A  - 2216%A  - 4544%A  - 6784%A
      +
       - 6976%A - 1792
      ,
                     7        6         5         4          3          2
       1408b - 37%A  - 408%A  - 1952%A  - 5024%A  - 10368%A  - 16768%A
      +
       - 17920%A - 5120
      ,
                     7        6         5         4          3          2
       1408c + 37%A  + 408%A  + 1952%A  + 5024%A  + 10368%A  + 16768%A
      +
       17920%A + 5120
      ,
                     7        6        5         4         3         2
       1408d + 19%A  + 200%A  + 912%A  + 2216%A  + 4544%A  + 6784%A
      +
       6976%A + 1792
      ,
     2e + %A, 2f - %A]
  ]
,
                 8     6      4      2
[complexRoots= ?  + 4?  + 12?  + 16?  + 4,
   coordinates =
            7      5       3               7      5       3
    [4a - %A  - 3%A  - 10%A  - 6%A, 4b - %A  - 3%A  - 10%A  - 10%A,
             7      5       3                7      5       3
     4c - 2%A  - 7%A  - 20%A  - 22%A, 4d + 2%A  + 7%A  + 20%A  + 22%A,
            7      5       3               7      5       3
     4e + %A  + 3%A  + 10%A  + 10%A, 4f + %A  + 3%A  + 10%A  + 6%A]
  ]
,
                 8      6      4       2
[complexRoots= ?  + 16?  - 96?  + 256?  + 256,
   coordinates =
```

```
                7       5        3
     [512a - %A  - 12%A  + 176%A  - 448%A,
                7       5       3
      128b - %A  - 16%A  + 96%A  - 256%A,
                7       5       3
      128c + %A  + 16%A  - 96%A  + 256%A,
                7       5        3
      512d + %A  + 12%A  - 176%A  + 448%A, 2e + %A, 2f - %A]
   ]
,
  [
   complexRoots =
      8      7      6       5       4       3        2
     ?  - 12?  + 64?  - 192?  + 432?  - 768?  + 1024?  - 768? + 256
   ,
    coordinates =
     [
                    7        6        5         4         3         2
       1408a - 19%A  + 200%A  - 912%A  + 2216%A  - 4544%A  + 6784%A
      +
       - 6976%A + 1792
      ,
                     7        6         5         4          3          2
       1408b - 37%A  + 408%A  - 1952%A  + 5024%A  - 10368%A  + 16768%A
      +
       - 17920%A + 5120
      ,
                     7        6         5         4          3          2
       1408c + 37%A  - 408%A  + 1952%A  - 5024%A  + 10368%A  - 16768%A
      +
       17920%A - 5120
      ,
                     7        6        5         4         3         2
       1408d + 19%A  - 200%A  + 912%A  - 2216%A  + 4544%A  - 6784%A
      +
       6976%A - 1792
      ,
      2e + %A, 2f - %A]
   ]
,
                  8     7      6      5      4     2
[complexRoots= ?  - 6?  + 16?  - 24?  + 18?  - 8?  + 4,
   coordinates =
             7      6       5       4      3       2
    [2a + 2%A  - 9%A  + 18%A  - 19%A  + 4%A  + 10%A  - 2%A - 4,
             7      6       5       4      3       2
     2b + 2%A  - 9%A  + 18%A  - 19%A  + 4%A  + 10%A  - 4%A - 4,
             7      6      5      4      3
     2c - %A  + 4%A  - 8%A  + 9%A  - 4%A  - 2%A + 4,
             7      6      5      4      3
```

```
      2d + %A  - 4%A  + 8%A  - 9%A  + 4%A  + 2%A - 4,
              7      6       5       4      3       2
      2e - 2%A  + 9%A  - 18%A  + 19%A  - 4%A  - 10%A  + 4%A + 4,
              7      6       5       4      3       2
      2f - 2%A  + 9%A  - 18%A  + 19%A  - 4%A  - 10%A  + 2%A + 4]
   ]
,
                  4      2
[complexRoots= ?  + 12?  + 144,
   coordinates =
              2               2               2               2
    [12a - %A  - 12, 12b - %A  - 12, 12c - %A  - 12, 12d - %A  - 12,
              2                   2
     6e + %A  + 3%A + 12, 6f + %A  - 3%A + 12]
   ]
,
                  4     3      2
[complexRoots= ?  + 6?  + 30?  + 36? + 36,
   coordinates =
             3      2                     3      2
    [6a - %A  - 5%A  - 24%A - 6, 30b + %A  + 5%A  + 30%A + 6,
             3      2                     3      2
     30c + %A  + 5%A  + 30%A + 6, 30d + %A  + 5%A  + 30%A + 6,
             3      2                     3      2
     30e + %A  + 5%A  + 30%A + 6, 30f + %A  + 5%A  + 6]
   ]
,
                  4     3      2
[complexRoots= ?  - 6?  + 30?  - 36? + 36,
   coordinates =
             3      2                     3      2
    [6a - %A  + 5%A  - 24%A + 6, 30b + %A  - 5%A  + 30%A - 6,
             3      2                     3      2
     30c + %A  - 5%A  + 30%A - 6, 30d + %A  - 5%A  + 30%A - 6,
             3      2                     3      2
     30e + %A  - 5%A  + 30%A - 6, 30f + %A  - 5%A  - 6]
   ]
,
                  4      2
[complexRoots= ?  + 12?  + 144,
   coordinates =
              2               2               2               2
    [12a + %A  + 12, 12b + %A  + 12, 12c + %A  + 12, 12d + %A  + 12,
              2                   2
     6e - %A  + 3%A - 12, 6f - %A  - 3%A - 12]
   ]
,
                  2
[complexRoots= ?  - 12,
 coordinates= [a - 1,b - 1,c - 1,d - 1,2e + %A + 4,2f - %A + 4]]
```

```
   ,
                    2
   [complexRoots= ?  + 6? + 6,
    coordinates= [a + %A + 5,b - 1,c - 1,d - 1,e - 1,f - %A - 1]]
   ,
                    2
   [complexRoots= ?  - 6? + 6,
    coordinates= [a + %A - 5,b + 1,c + 1,d + 1,e + 1,f - %A + 1]]
   ,
                    2
   [complexRoots= ?  - 12,
    coordinates= [a + 1,b + 1,c + 1,d + 1,2e + %A - 4,2f - %A - 4]]
   ,
                    4     3      2
   [complexRoots= ?  + 6?  + 30?  + 36? + 36,
      coordinates =
                 3      2                      3      2
       [30a - %A  - 5%A  - 30%A - 6, 30b - %A  - 5%A  - 30%A - 6,
                 3      2                      3      2
        30c - %A  - 5%A  - 30%A - 6, 6d + %A  + 5%A  + 24%A + 6,
                 3      2               3      2
        30e - %A  - 5%A  - 6, 30f - %A  - 5%A  - 30%A - 6]
     ]
   ,
                    4     3      2
   [complexRoots= ?  - 6?  + 30?  - 36? + 36,
      coordinates =
                 3      2                      3      2
       [30a - %A  + 5%A  - 30%A + 6, 30b - %A  + 5%A  - 30%A + 6,
                 3      2                      3      2
        30c - %A  + 5%A  - 30%A + 6, 6d + %A  - 5%A  + 24%A - 6,
                 3      2               3      2
        30e - %A  + 5%A  + 6, 30f - %A  + 5%A  - 30%A + 6]
     ]
   ,
                    2
   [complexRoots= ?  + 6? + 6,
    coordinates= [a + 1,b + 1,c + 1,d - %A - 5,e + %A + 1,f + 1]]
   ,
                    2
   [complexRoots= ?  - 6? + 6,
    coordinates= [a - 1,b - 1,c - 1,d - %A + 5,e + %A - 1,f - 1]]
   ]
  Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
                    coordinates: List Polynomial Integer)
```

Since the univariateSolve operation may split a regular set, it returns a list. This explains the use of concat.

Look at the last item of the result. It consists of two parts. For

```
any complex root ? of the univariate polynomial in the first part, we
get a tuple of univariate polynomials (in a, ...,f respectively) by
replacing %A by ? in the second part.  Each of these tuples t
describes a point of the variety associated with lp by equaling to
zero the polynomials in t.

Note that the way of reading these univariate representations is
explained also in the example illustrating the ZeroDimensionalSolvePackage
constructor.

Now, we compute the points of the variety with real coordinates.

  concat [realSolve(ts)$zdpack for ts in lts]
  [[%B1,%B1,%B1,%B5,- %B5 - 4%B1,%B1], [%B1,%B1,%B1,%B6,- %B6 - 4%B1,%B1],
   [%B2,%B2,%B2,%B3,- %B3 - 4%B2,%B2], [%B2,%B2,%B2,%B4,- %B4 - 4%B2,%B2],
   [%B7,%B7,%B7,%B7,%B11,- %B11 - 4%B7], [%B7,%B7,%B7,%B7,%B12,- %B12 - 4%B7],
   [%B8,%B8,%B8,%B8,%B9,- %B9 - 4%B8], [%B8,%B8,%B8,%B8,%B10,- %B10 - 4%B8],
   [%B13,%B13,%B17,- %B17 - 4%B13,%B13,%B13],
   [%B13,%B13,%B18,- %B18 - 4%B13,%B13,%B13],
   [%B14,%B14,%B15,- %B15 - 4%B14,%B14,%B14],
   [%B14,%B14,%B16,- %B16 - 4%B14,%B14,%B14],
     [%B19, %B29,
           7865521      31   6696179241     25   25769893181     19
        ---------- %B19   - ---------- %B19   - ----------- %B19
        6006689520          2002229840            49235160
      +
          1975912990729     13   1048460696489     7   21252634831
        - ------------- %B19   - ------------- %B19  - ----------- %B19
            3003344760             2002229840           6006689520
      ,
           778171189       31   1987468196267     25   155496778477189     19
        - ------------- %B19   + ------------- %B19   + --------------- %B19
          1387545279120          1387545279120            693772639560
      +
        191631411158401     13   300335488637543     7   755656433863
        --------------- %B19   + --------------- %B19  - ------------ %B19
          693772639560           1387545279120          198220754160
      ,
          1094352947      31   2794979430821     25   218708802908737     19
        ------------ %B19   - ------------- %B19   - --------------- %B19
        462515093040          462515093040            231257546520
      +
          91476663003591     13   145152550961823     7   1564893370717
        - -------------- %B19   - --------------- %B19  - ------------- %B19
            77085848840            154171697680           462515093040
      ,
                    4321823003      31   180949546069     25
        - %B29 - ------------- %B19   + ------------ %B19
                 1387545279120           22746643920
      +
```

```
    863753195062493     19   1088094456732317     13
    --------------- %B19   + ---------------- %B19
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B19  + -------------- %B19
      1387545279120           1387545279120
  ]
,
  [%B19, %B30,
       7865521      31   6696179241     25   25769893181     19
    ---------- %B19   - ---------- %B19   - ----------- %B19
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B19   - ------------- %B19  - ----------- %B19
        3003344760             2002229840           6006689520
  ,
         778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B19   + ------------- %B19   + --------------- %B19
      1387545279120          1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B19   + --------------- %B19  - ------------ %B19
      693772639560            1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B19   - ------------- %B19   - --------------- %B19
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - -------------- %B19   - --------------- %B19  - ------------- %B19
        77085848840             154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B30 - ------------- %B19   + ------------ %B19
             1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B19   + ---------------- %B19
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B19  + -------------- %B19
      1387545279120           1387545279120
  ]
,
  [%B20, %B27,
       7865521      31   6696179241     25   25769893181     19
    ---------- %B20   - ---------- %B20   - ----------- %B20
```

```
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B20   - ------------- %B20  - ----------- %B20
        3003344760             2002229840           6006689520
  ,
         778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B20   + ------------- %B20   + --------------- %B20
      1387545279120          1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B20   + --------------- %B20  - ------------ %B20
      693772639560           1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B20   - ------------- %B20   - --------------- %B20
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - -------------- %B20   - --------------- %B20  - ------------- %B20
        77085848840            154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B27 - ------------- %B20   + ------------ %B20
             1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B20   + ---------------- %B20
      693772639560           693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B20  + -------------- %B20
      1387545279120          1387545279120
  ]
,
 [%B20, %B28,
       7865521      31   6696179241     25   25769893181     19
    ---------- %B20   - ---------- %B20   - ----------- %B20
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B20   - ------------- %B20  - ----------- %B20
        3003344760             2002229840           6006689520
  ,
         778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B20   + ------------- %B20   + --------------- %B20
      1387545279120          1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B20   + --------------- %B20  - ------------ %B20
```

```
      693772639560            1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B20   - ------------- %B20   - --------------- %B20
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - --------------- %B20   - ---------------- %B20  - ------------- %B20
        77085848840             154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B28 - ------------- %B20   + ------------ %B20
             1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B20   + ----------------- %B20
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B20  + -------------- %B20
      1387545279120           1387545279120
  ]
,
 [%B21, %B25,
       7865521      31   6696179241     25   25769893181     19
    ---------- %B21   - ---------- %B21   - ----------- %B21
    6006689520          2002229840             49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B21   - ------------- %B21  - ----------- %B21
        3003344760             2002229840           6006689520
  ,
        778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B21   + ------------- %B21   + --------------- %B21
      1387545279120          1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B21   + ---------------- %B21  - ------------ %B21
      693772639560           1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B21   - ------------- %B21   - --------------- %B21
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - --------------- %B21   - ---------------- %B21  - ------------- %B21
        77085848840             154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B25 - ------------- %B21   + ------------ %B21
```

```
            1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B21   + ---------------- %B21
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B21  + -------------- %B21
      1387545279120           1387545279120
  ]
,
  [%B21, %B26,
       7865521      31   6696179241     25   25769893181     19
    ---------- %B21   - ---------- %B21   - ----------- %B21
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B21   - ------------- %B21  - ----------- %B21
        3003344760             2002229840           6006689520
  ,
         778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B21   + ------------- %B21   + --------------- %B21
      1387545279120           1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B21   + --------------- %B21  - ------------ %B21
      693772639560            1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B21   - ------------- %B21   - --------------- %B21
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - -------------- %B21   - --------------- %B21  - ------------- %B21
        77085848840            154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B26 - ------------- %B21   + ------------ %B21
             1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B21   + ---------------- %B21
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B21  + -------------- %B21
      1387545279120           1387545279120
  ]
,
  [%B22, %B23,
```

```
      7865521      31   6696179241     25   25769893181     19
    ---------- %B22   - ---------- %B22   - ----------- %B22
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B22   - ------------- %B22  - ----------- %B22
        3003344760             2002229840           6006689520
  ,
        778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B22   + ------------- %B22   + --------------- %B22
      1387545279120          1387545279120            693772639560
  +
    191631411158401     13   300335488637543     7   755656433863
    --------------- %B22   + --------------- %B22  - ------------ %B22
      693772639560           1387545279120          198220754160
  ,
      1094352947      31   2794979430821     25   218708802908737     19
    ------------ %B22   - ------------- %B22   - --------------- %B22
    462515093040           462515093040            231257546520
  +
      91476663003591     13   145152550961823     7   1564893370717
    - -------------- %B22   - --------------- %B22  - ------------- %B22
        77085848840            154171697680           462515093040
  ,
                4321823003      31   180949546069     25
    - %B23 - ------------- %B22   + ------------ %B22
             1387545279120           22746643920
  +
    863753195062493     19   1088094456732317     13
    --------------- %B22   + ---------------- %B22
      693772639560             693772639560
  +
    1732620732685741     7   13506088516033
    ---------------- %B22  + -------------- %B22
      1387545279120           1387545279120
  ]
,
 [%B22, %B24,
      7865521      31   6696179241     25   25769893181     19
    ---------- %B22   - ---------- %B22   - ----------- %B22
    6006689520          2002229840            49235160
  +
      1975912990729     13   1048460696489     7   21252634831
    - ------------- %B22   - ------------- %B22  - ----------- %B22
        3003344760             2002229840           6006689520
  ,
        778171189       31   1987468196267     25   155496778477189     19
    - ------------- %B22   + ------------- %B22   + --------------- %B22
      1387545279120          1387545279120            693772639560
  +
```

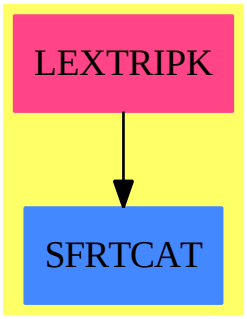
```
        191631411158401     13   300335488637543     7   755656433863
        --------------- %B22   + --------------- %B22  - ------------ %B22
          693772639560             1387545279120           198220754160
      ,
          1094352947      31   2794979430821     25   218708802908737     19
        ------------ %B22   - ------------- %B22   - --------------- %B22
        462515093040            462515093040             231257546520
      +
          91476663003591     13   145152550961823     7   1564893370717
        - -------------- %B22   - --------------- %B22  - ------------- %B22
            77085848840              154171697680           462515093040
      ,
                    4321823003      31   180949546069     25
        - %B24 - ------------- %B22   + ------------ %B22
                 1387545279120            22746643920
      +
        863753195062493     19   1088094456732317     13
        --------------- %B22   + ---------------- %B22
          693772639560             693772639560
      +
        1732620732685741     7   13506088516033
        ---------------- %B22  + -------------- %B22
          1387545279120            1387545279120
      ]
     ,
    [%B31,%B35,- %B35 - 4%B31,%B31,%B31,%B31],
    [%B31,%B36,- %B36 - 4%B31,%B31,%B31,%B31],
    [%B32,%B33,- %B33 - 4%B32,%B32,%B32,%B32],
    [%B32,%B34,- %B34 - 4%B32,%B32,%B32,%B32]]
                                Type: List List RealClosure Fraction Integer
```

We obtain 24 points given by lists of elements in the RealClosure
of Fraction of R. In each list, the first value corresponds
to the indeterminate f, the second to e and so on.

See Also:
-)help RegularChain
-)help RegularTriangularSet
-)help SquareFreeRegularTriangularSet
-)help ZeroDimensionalSolvePackage
-)help NewSparseMultivariatePolynomial
-)help LinGroebnerPackage
-)help NormalizedTriangularSetCategory
-)help RealClosure
-)help Fraction
-)show LexTriangularPackage

———-

13.4.1 LexTriangularPackage (LEXTRIPK)



Exports:

fglmIfCan	groebner	lexTriangular
squareFreeLexTriangular	zeroDimensional?	zeroSetSplit

— package LEXTRIPK LexTriangularPackage —

```
)abbrev package LEXTRIPK LexTriangularPackage
++ Author: Marc Moreno Maza
++ Date Created: 08/02/1999
++ Date Last Updated: 08/02/1999
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ [1] D. LAZARD "Solving Zero-dimensional Algebraic Systems"
++ published in the J. of Symbol. Comput. (1992) 13, 117-131.
++ [2] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++ algebraic towers of simple extensions"
++ In proceedings of AAECC11, Paris, 1995.
++ Description:
++ A package for solving polynomial systems with finitely many solutions.
++ The decompositions are given by means of regular triangular sets.
++ The computations use lexicographical Groebner bases.
++ The main operations are lexTriangular
++ and squareFreeLexTriangular. The second one provide decompositions by
++ means of square-free regular triangular sets.
++ Both are based on the lexTriangular method described in [1].
++ They differ from the algorithm described in [2] by the fact that
++ multiciplities of the roots are not kept.
++ With the squareFreeLexTriangular operation all multiciplities are removed.
```

```
++ With the other operation some multiciplities may remain. Both operations
++ admit an optional argument to produce normalized triangular sets.

LexTriangularPackage(R,ls): Exports == Implementation where

  R: GcdDomain
  ls: List Symbol
  V ==> OrderedVariableList ls
  E ==> IndexedExponents V
  P ==> NewSparseMultivariatePolynomial(R,V)
  TS  ==> RegularChain(R,ls)
  ST ==> SquareFreeRegularTriangularSet(R,E,V,P)
  Q1 ==> Polynomial R
  PS ==> GeneralPolynomialSet(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  K ==> Fraction R
  LP ==> List P
  BWTS ==> Record(val : Boolean, tower : TS)
  LpWTS ==> Record(val : (List P), tower : TS)
  BWST ==> Record(val : Boolean, tower : ST)
  LpWST ==> Record(val : (List P), tower : ST)
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppackTS ==> QuasiComponentPackage(R,E,V,P,TS)
  regsetgcdpackTS ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)
  normalizpackTS ==> NormalizationPackage(R,E,V,P,TS)
  quasicomppackST ==> QuasiComponentPackage(R,E,V,P,ST)
  regsetgcdpackST ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,ST)
  normalizpackST ==> NormalizationPackage(R,E,V,P,ST)

  Exports ==  with

     zeroDimensional?: LP -> B
         ++ \axiom{zeroDimensional?(lp)} returns true iff
         ++ \axiom{lp} generates a zero-dimensional ideal
         ++ w.r.t. the variables involved in \axiom{lp}.
     fglmIfCan:  LP -> Union(LP, "failed")
         ++ \axiom{fglmIfCan(lp)} returns the lexicographical Groebner
         ++ basis of \axiom{lp} by using the FGLM strategy,
         ++ if \axiom{zeroDimensional?(lp)} holds .
     groebner: LP -> LP
         ++ \axiom{groebner(lp)} returns the lexicographical Groebner
         ++ basis of \axiom{lp}. If \axiom{lp} generates a zero-dimensional
         ++ ideal then the FGLM strategy is used, otherwise
         ++ the Sugar strategy is used.
     lexTriangular: (LP, B) -> List TS
         ++ \axiom{lexTriangular(base, norm?)} decomposes the variety
         ++ associated with \axiom{base} into regular chains.
```

```
        ++ Thus a point belongs to this variety iff it is a regular
        ++ zero of a regular set in in the output.
        ++ Note that \axiom{base} needs to be a lexicographical Groebner basis
        ++ of a zero-dimensional ideal. If \axiom{norm?} is \axiom{true}
        ++ then the regular sets are normalized.
    squareFreeLexTriangular: (LP, B) -> List ST
        ++ \axiom{squareFreeLexTriangular(base, norm?)} decomposes the variety
        ++ associated with \axiom{base} into square-free regular chains.
        ++ Thus a point belongs to this variety iff it is a regular
        ++ zero of a regular set in in the output.
        ++ Note that \axiom{base} needs to be a lexicographical Groebner basis
        ++ of a zero-dimensional ideal. If \axiom{norm?} is \axiom{true}
        ++ then the regular sets are normalized.
    zeroSetSplit: (LP, B) -> List TS
        ++ \axiom{zeroSetSplit(lp, norm?)} decomposes the variety
        ++ associated with \axiom{lp} into regular chains.
        ++ Thus a point belongs to this variety iff it is a regular
        ++ zero of a regular set in in the output.
        ++ Note that \axiom{lp} needs to generate a zero-dimensional ideal.
        ++ If \axiom{norm?} is \axiom{true} then the regular sets are normalized.
    zeroSetSplit: (LP, B) -> List ST
        ++ \axiom{zeroSetSplit(lp, norm?)} decomposes the variety
        ++ associated with \axiom{lp} into square-free regular chains.
        ++ Thus a point belongs to this variety iff it is a regular
        ++ zero of a regular set in in the output.
        ++ Note that \axiom{lp} needs to generate a zero-dimensional ideal.
        ++ If \axiom{norm?} is \axiom{true} then the regular sets are normalized.

  Implementation == add

    trueVariables(lp: List(P)): List Symbol ==
      lv: List V := variables([lp]$PS)
      truels: List Symbol := []
      for s in ls repeat
        if member?(variable(s)::V, lv) then truels := cons(s,truels)
      reverse truels

    zeroDimensional?(lp:List(P)): Boolean ==
      truels: List Symbol := trueVariables(lp)
      fglmpack := FGLMIfCanPackage(R,truels)
      lq1: List(Q1) := [p::Q1 for p in lp]
      zeroDimensional?(lq1)$fglmpack

    fglmIfCan(lp:List(P)): Union(List(P), "failed") ==
      truels: List Symbol := trueVariables(lp)
      fglmpack := FGLMIfCanPackage(R,truels)
      lq1: List(Q1) := [p::Q1 for p in lp]
      foo := fglmIfCan(lq1)$fglmpack
      foo case "failed" => return("failed" :: Union(List(P), "failed"))
      lp := [retract(q1)$P for q1 in (foo :: List(Q1))]
```

```
    lp::Union(List(P), "failed")

  groebner(lp:List(P)): List(P) ==
    truels: List Symbol := trueVariables(lp)
    fglmpack := FGLMIfCanPackage(R,truels)
    lq1: List(Q1) := [p::Q1 for p in lp]
    lq1 := groebner(lq1)$fglmpack
    lp := [retract(q1)$P for q1 in lq1]

  lexTriangular(base: List(P), norm?: Boolean): List(TS) ==
    base := sort(infRittWu?,base)
    base := remove(zero?, base)
    any?(ground?, base) => []
    ts: TS := empty()
    toSee: List LpWTS := [[base,ts]$LpWTS]
    toSave: List TS := []
    while not empty? toSee repeat
      lpwt := first toSee; toSee := rest toSee
      lp := lpwt.val; ts := lpwt.tower
      empty? lp => toSave := cons(ts, toSave)
      p := first lp; lp := rest lp; v := mvar(p)
      algebraic?(v,ts) =>
        error "lexTriangular$LEXTRIPK: should never happen !"
      norm? and zero? remainder(init(p),ts).polnum =>
        toSee := cons([lp, ts]$LpWTS, toSee)
      (not norm?) and zero? (initiallyReduce(init(p),ts)) =>
        toSee := cons([lp, ts]$LpWTS, toSee)
      lbwt: List BWTS := invertible?(init(p),ts)$TS
      while (not empty? lbwt) repeat
        bwt := first lbwt; lbwt := rest lbwt
        b := bwt.val; us := bwt.tower
        (not b) => toSee := cons([lp, us], toSee)
        lus: List TS
        if norm?
          then
            newp := normalizedAssociate(p,us)$normalizpackTS
            lus := [internalAugment(newp,us)$TS]
          else
            newp := p
            lus := augment(newp,us)$TS
        newlp := lp
        while (not empty? newlp) and (mvar(first newlp) = v) repeat
          newlp := rest newlp
        for us in lus repeat
          toSee := cons([newlp, us]$LpWTS, toSee)
    algebraicSort(toSave)$quasicomppackTS

  zeroSetSplit(lp:List(P), norm?:B): List TS ==
    bar := fglmIfCan(lp)
    bar case "failed" =>
```

```
      error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
    lexTriangular(bar::(List P),norm?)

  squareFreeLexTriangular(base: List(P), norm?: Boolean): List(ST) ==
    base := sort(infRittWu?,base)
    base := remove(zero?, base)
    any?(ground?, base) => []
    ts: ST := empty()
    toSee: List LpWST := [[base,ts]$LpWST]
    toSave: List ST := []
    while not empty? toSee repeat
      lpwt := first toSee; toSee := rest toSee
      lp := lpwt.val; ts := lpwt.tower
      empty? lp => toSave := cons(ts, toSave)
      p := first lp; lp := rest lp; v := mvar(p)
      algebraic?(v,ts) =>
        error "lexTriangular$LEXTRIPK: should never happen !"
      norm? and zero? remainder(init(p),ts).polnum =>
        toSee := cons([lp, ts]$LpWST, toSee)
      (not norm?) and zero? (initiallyReduce(init(p),ts)) =>
        toSee := cons([lp, ts]$LpWST, toSee)
      lbwt: List BWST := invertible?(init(p),ts)$ST
      while (not empty? lbwt) repeat
        bwt := first lbwt; lbwt := rest lbwt
        b := bwt.val; us := bwt.tower
        (not b) => toSee := cons([lp, us], toSee)
        lus: List ST
        if norm?
          then
            newp := normalizedAssociate(p,us)$normalizpackST
            lus := augment(newp,us)$ST
          else
            lus := augment(p,us)$ST
        newlp := lp
        while (not empty? newlp) and (mvar(first newlp) = v) repeat
          newlp := rest newlp
        for us in lus repeat
          toSee := cons([newlp, us]$LpWST, toSee)
    algebraicSort(toSave)$quasicomppackST

  zeroSetSplit(lp:List(P), norm?:B): List ST ==
    bar := fglmIfCan(lp)
    bar case "failed" =>
      error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
    squareFreeLexTriangular(bar::(List P),norm?)
```

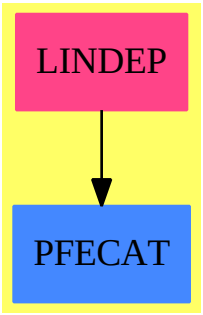
— **LEXTRIPK.dotabb** —

```
"LEXTRIPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LEXTRIPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"LEXTRIPK" -> "SFRTCAT"
```

———-

13.5 package LINDEP LinearDependence

13.5.1 LinearDependence (LINDEP)



Exports:
linearDependence linearlyDependent? solveLinear

— **package LINDEP LinearDependence** —

```
)abbrev package LINDEP LinearDependence
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 14 May 1991
++ Description:
++ Test for linear dependence.

LinearDependence(S, R): Exports == Implementation where
  S: IntegralDomain
  R: LinearlyExplicitRingOver S

  Q ==> Fraction S

  Exports ==> with
    linearlyDependent?: Vector R -> Boolean
```

```
      ++ \spad{linearlyDependent?([v1,...,vn])} returns true if
      ++ the vi's are linearly dependent over S, false otherwise.
    linearDependence  : Vector R -> Union(Vector S, "failed")
      ++ \spad{linearDependence([v1,...,vn])} returns \spad{[c1,...,cn]} if
      ++ \spad{c1*v1 + ... + cn*vn = 0} and not all the ci's are 0,
      ++ "failed" if the vi's are linearly independent over S.
    if S has Field then
      solveLinear: (Vector R, R) -> Union(Vector S, "failed")
        ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
        ++ such that \spad{c1*v1 + ... + cn*vn = u},
        ++ "failed" if no such ci's exist in S.
    else
      solveLinear: (Vector R, R) -> Union(Vector Q, "failed")
        ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
        ++ such that \spad{c1*v1 + ... + cn*vn = u},
        ++ "failed" if no such ci's exist in the quotient field of S.

  Implementation ==> add
    aNonZeroSolution: Matrix S -> Union(Vector S, "failed")

    aNonZeroSolution m ==
      every?(zero?, v := first nullSpace m) => "failed"
      v

    linearlyDependent? v ==
      zero?(n := #v) => true
--      one? n => zero?(v(minIndex v))
      (n = 1) => zero?(v(minIndex v))
      positive? nullity reducedSystem transpose v

    linearDependence v ==
      zero?(n := #v) => empty()
--      one? n =>
      (n = 1) =>
        zero?(v(minIndex v)) => new(1, 1)
        "failed"
      aNonZeroSolution reducedSystem transpose v

    if S has Field then
      solveLinear(v:Vector R, c:R):Union(Vector S, "failed") ==
        zero? c => new(#v, 0)
        empty? v => "failed"
        sys := reducedSystem(transpose v, new(1, c))
        particularSolution(sys.mat, sys.vec)$LinearSystemMatrixPackage(S,
                                          Vector S, Vector S, Matrix S)

    else
      solveLinear(v:Vector R, c:R):Union(Vector Q, "failed") ==
        zero? c => new(#v, 0)
        empty? v => "failed"
```

```
        sys := reducedSystem(transpose v, new(1, c))
        particularSolution(map((z:S):Q+->z::Q, sys.mat)_
          $MatrixCategoryFunctions2(S,
               Vector S,Vector S,Matrix S,Q,Vector Q,Vector Q,Matrix Q),
                  map((z1:S):Q+->z1::Q, sys.vec)$VectorFunctions2(S, Q)
                                  )$LinearSystemMatrixPackage(Q,
                                         Vector Q, Vector Q, Matrix Q)
```

———-

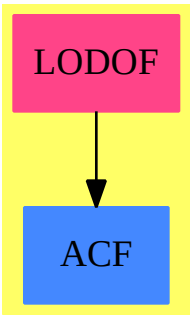
— **LINDEP.dotabb** —

```
"LINDEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LINDEP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LINDEP" -> "PFECAT"
```

———-

13.6 package LODOF LinearOrdinaryDifferentialOperatorFactorizer

13.6.1 LinearOrdinaryDifferentialOperatorFactorizer (LODOF)



Exports:
factor factor1

— **package LODOF LinearOrdinaryDifferentialOperatorFactorizer** —

```
)abbrev package LODOF LinearOrdinaryDifferentialOperatorFactorizer
++ Author: Fritz Schwarz, Manuel Bronstein
++ Date Created: 1988
```

```
++ Date Last Updated: 3 February 1994
++ Keywords: differential equation, ODE, LODO, factoring
++ Description:
++ \spadtype{LinearOrdinaryDifferentialOperatorFactorizer} provides a
++ factorizer for linear ordinary differential operators whose coefficients
++ are rational functions.

LinearOrdinaryDifferentialOperatorFactorizer(F, UP): Exports == Impl where
  F : Join(Field, CharacteristicZero,
           RetractableTo Integer, RetractableTo Fraction Integer)
  UP: UnivariatePolynomialCategory F

  RF ==> Fraction UP
  L  ==> LinearOrdinaryDifferentialOperator1 RF

  Exports ==> with
    factor: (L, UP -> List F) -> List L
      ++ factor(a, zeros) returns the factorisation of a.
      ++ \spad{zeros} is a zero finder in \spad{UP}.
    if F has AlgebraicallyClosedField then
      factor: L -> List L
        ++ factor(a) returns the factorisation of a.
      factor1: L -> List L
        ++ factor1(a) returns the factorisation of a,
        ++ assuming that a has no first-order right factor.

  Impl ==> add
    import RationalLODE(F, UP)
    import RationalRicDE(F, UP)
--  import AssociatedEquations RF

    dd := D()$L

    expsol     : (L, UP -> List F, UP -> Factored UP) -> Union(RF, "failed")
    expsols    : (L, UP -> List F, UP -> Factored UP, Boolean) -> List RF
    opeval     : (L, L) -> L
    recurfactor: (L, L, UP -> List F, UP -> Factored UP, Boolean) -> List L
    rfactor    : (L, L, UP -> List F, UP -> Factored UP, Boolean) -> List L
    rightFactor: (L, NonNegativeInteger, UP -> List F, UP -> Factored UP)
                                                        -> Union(L, "failed")
    innerFactor: (L, UP -> List F, UP -> Factored UP, Boolean) -> List L

    factor(l, zeros) == innerFactor(l, zeros, squareFree, true)

    expsol(l, zeros, ezfactor) ==
      empty?(sol := expsols(l, zeros, ezfactor, false)) => "failed"
      first sol

    expsols(l, zeros, ezfactor, all?) ==
      sol := [differentiate(f)/f for f in ratDsolve(l, 0).basis | f ^= 0]
```

```
      not(all? or empty? sol) => sol
      concat(sol, ricDsolve(l, zeros, ezfactor))

-- opeval(l1, l2) returns l1(l2)
    opeval(l1, l2) ==
      ans:L := 0
      l2n:L := 1
      for i in 0..degree l1 repeat
        ans := ans + coefficient(l1, i) * l2n
        l2n := l2 * l2n
      ans

    recurfactor(l, r, zeros, ezfactor, adj?) ==
      q := rightExactQuotient(l, r)::L
      if adj? then q := adjoint q
      innerFactor(q, zeros, ezfactor, true)

    rfactor(op, r, zeros, ezfactor, adj?) ==
--      degree r > 1 or not one? leadingCoefficient r =>
      degree r > 1 or not ((leadingCoefficient r) = 1) =>
        recurfactor(op, r, zeros, ezfactor, adj?)
      op1 := opeval(op, dd - coefficient(r, 0)::L)
      map_!((z:L):L+->opeval(z,r), recurfactor(op1, dd, zeros, ezfactor, adj?))

-- r1? is true means look for 1st-order right-factor also
    innerFactor(l, zeros, ezfactor, r1?) ==
      (n := degree l) <= 1 => [l]
      ll := adjoint l
      for i in 1..(n quo 2) repeat
        (r1? or (i > 1)) and ((u := rightFactor(l,i,zeros,ezfactor)) case L) =>
          return concat_!(rfactor(l, u::L, zeros, ezfactor, false), u::L)
        (2 * i < n) and ((u := rightFactor(ll, i, zeros, ezfactor)) case L) =>
          return concat(adjoint(u::L), rfactor(ll, u::L, zeros,ezfactor,true))
      [l]

    rightFactor(l, n, zeros, ezfactor) ==
--      one? n =>
      (n = 1) =>
        (u := expsol(l, zeros, ezfactor)) case "failed" => "failed"
        D() - u::RF::L
--    rec := associatedEquations(l, n::PositiveInteger)
--    empty?(sol := expsols(rec.eq, zeros, ezfactor, true)) => "failed"
      "failed"

    if F has AlgebraicallyClosedField then
      zro1: UP -> List F
      zro : (UP, UP -> Factored UP) -> List F

      zro(p, ezfactor) ==
        concat [zro1(r.factor) for r in factors ezfactor p]
```

```
      zro1 p ==
        [zeroOf(map((z1:F):F+->z1,p)_
          $UnivariatePolynomialCategoryFunctions2(F, UP,
                                          F, SparseUnivariatePolynomial F))]

      if F is AlgebraicNumber then
        import AlgFactor UP

        factor l  ==
          innerFactor(l,(p:UP):List(F)+->zro(p,factor),factor,true)
        factor1 l ==
          innerFactor(l,(p:UP):List(F)+->zro(p,factor),factor,false)

      else
        factor l  ==
          innerFactor(l,(p:UP):List(F)+->zro(p,squareFree),squareFree,true)
        factor1 l ==
          innerFactor(l,(p:UP):List(F)+->zro(p,squareFree),squareFree,false)
```

———-

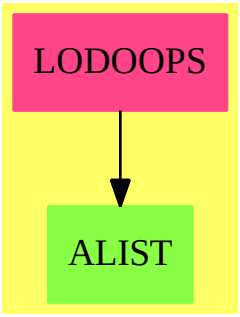
— LODOF.dotabb —

```
"LODOF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"LODOF" -> "ACF"
```

———-

13.7 package LODOOPS LinearOrdinaryDifferentialOperatorsOps

13.7.1 LinearOrdinaryDifferentialOperatorsOps (LODOOPS)



Exports:
directSum symmetricPower symmetricProduct

— **package LODOOPS LinearOrdinaryDifferentialOperatorsOps** —

```
)abbrev package LODOOPS LinearOrdinaryDifferentialOperatorsOps
++ Author: Manuel Bronstein
++ Date Created: 18 January 1994
++ Date Last Updated: 15 April 1994
++ Description:
++ \spad{LinearOrdinaryDifferentialOperatorsOps} provides symmetric
++ products and sums for linear ordinary differential operators.
-- Putting those operations here rather than defaults in LODOCAT allows
-- LODOCAT to be defined independently of the derivative used.
-- MB 1/94

LinearOrdinaryDifferentialOperatorsOps(A, L): Exports == Implementation where
    A: Field
    L: LinearOrdinaryDifferentialOperatorCategory A

    N  ==> NonNegativeInteger
    V  ==> OrderlyDifferentialVariable Symbol
    P  ==> DifferentialSparseMultivariatePolynomial(A, Symbol, V)

    Exports ==> with
          symmetricProduct: (L, L, A -> A) -> L
            ++ symmetricProduct(a,b,D) computes an operator \spad{c} of
            ++ minimal order such that the nullspace of \spad{c} is
            ++ generated by all the products of a solution of \spad{a} by
            ++ a solution of \spad{b}.
```

```
          ++ D is the derivation to use.
        symmetricPower: (L, N, A -> A) -> L
          ++ symmetricPower(a,n,D) computes an operator \spad{c} of
          ++ minimal order such that the nullspace of \spad{c} is
          ++ generated by all the products of \spad{n} solutions
          ++ of \spad{a}.
          ++ D is the derivation to use.
        directSum: (L, L, A -> A) -> L
          ++ directSum(a,b,D) computes an operator \spad{c} of
          ++ minimal order such that the nullspace of \spad{c} is
          ++ generated by all the sums of a solution of \spad{a} by
          ++ a solution of \spad{b}.
          ++ D is the derivation to use.

    Implementation ==> add
        import IntegerCombinatoricFunctions

        var1 := new()$Symbol
        var2 := new()$Symbol

        nonTrivial?: Vector A -> Boolean
        applyLODO  : (L, V) -> P
        killer     : (P, N, List V, List P, A -> A) -> L
        vec2LODO   : Vector A -> L

        nonTrivial? v == any?((x1:A):Boolean +-> x1 ^= 0, v)$Vector(A)
        vec2LODO v    == +/[monomial(v.i, (i-1)::N) for i in 1..#v]

        symmetricPower(l, m, diff) ==
          u := var1::V; n := degree l
          un := differentiate(u, n)
          a  := applyLODO(inv(- leadingCoefficient l) * reductum l, u)
          killer(u::P ** m, binomial(n + m - 1, n - 1)::N, [un], [a], diff)

-- returns an operator L such that L(u) = 0, for a given differential
-- polynomial u, given that the differential variables appearing in u
-- satisfy some linear ode's
-- m is a bound on the order of the operator searched.
-- lvar, lval describe the substitution(s) to perform when differentiating
--     the expression u (they encode the fact the the differential variables
--     satisfy some differential equations, which can be seen as the rewrite
--     rules   lvar --> lval)
-- diff is the derivation to use
        killer(u, m, lvar, lval, diff) ==
          lu:List P := [u]
          for q in 0..m repeat
            mat := reducedSystem(matrix([lu])@Matrix(P))@Matrix(A)
            (sol := find(nonTrivial?, l := nullSpace mat)) case Vector(A) =>
              return vec2LODO(sol::Vector(A))
            u := eval(differentiate(u, diff), lvar, lval)
```

```
            lu := concat_!(lu, [u])
          error "killer: no linear dependence found"

        symmetricProduct(l1, l2, diff) ==
          u  := var1::V;   v  := var2::V
          n1 := degree l1; n2 := degree l2
          un := differentiate(u, n1); vn := differentiate(v, n2)
          a  := applyLODO(inv(- leadingCoefficient l1) * reductum l1, u)
          b  := applyLODO(inv(- leadingCoefficient l2) * reductum l2, v)
          killer(u::P * v::P, n1 * n2, [un, vn], [a, b], diff)

        directSum(l1, l2, diff) ==
          u  := var1::V;   v  := var2::V
          n1 := degree l1; n2 := degree l2
          un := differentiate(u, n1); vn := differentiate(v, n2)
          a  := applyLODO(inv(- leadingCoefficient l1) * reductum l1, u)
          b  := applyLODO(inv(- leadingCoefficient l2) * reductum l2, v)
          killer(u::P + v::P, n1 + n2, [un, vn], [a, b], diff)

        applyLODO(l, v) ==
          p:P := 0
          while l ^= 0 repeat
            p := p + monomial(leadingCoefficient(l)::P,
                              differentiate(v, degree l), 1)
            l := reductum l
          p
```

———-

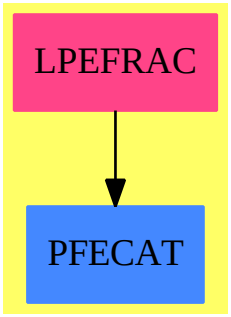
— LODOOPS.dotabb —

```
"LODOOPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOOPS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LODOOPS" -> "ALIST"
```

———-

13.8 package LPEFRAC LinearPolynomialEquationByFractions

13.8.1 LinearPolynomialEquationByFractions (LPEFRAC)



Exports:
solveLinearPolynomialEquationByFractions

— **package LPEFRAC LinearPolynomialEquationByFractions** —

```
)abbrev package LPEFRAC LinearPolynomialEquationByFractions
++ Author: James Davenport
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Given a PolynomialFactorizationExplicit ring, this package
++ provides a defaulting rule for the \spad{solveLinearPolynomialEquation}
++ operation, by moving into the field of fractions, and solving it there
++ via the \spad{multiEuclidean} operation.

LinearPolynomialEquationByFractions(R:PolynomialFactorizationExplicit): with
  solveLinearPolynomialEquationByFractions: ( _
           List SparseUnivariatePolynomial R, _
           SparseUnivariatePolynomial R) ->   _
           Union(List SparseUnivariatePolynomial R, "failed")
        ++ solveLinearPolynomialEquationByFractions([f1, ..., fn], g)
        ++ (where the fi are relatively prime to each other)
        ++ returns a list of ai such that
        ++ \spad{g/prod fi = sum ai/fi}
```

```
       ++ or returns "failed" if no such exists.
 == add
  SupR ==> SparseUnivariatePolynomial R
  F ==> Fraction R
  SupF ==> SparseUnivariatePolynomial F
  import UnivariatePolynomialCategoryFunctions2(R,SupR,F,SupF)
  lp : List SupR
  pp: SupR
  pF: SupF
  pullback : SupF -> Union(SupR,"failed")
  pullback(pF) ==
    pF = 0 => 0
    c:=retractIfCan leadingCoefficient pF
    c case "failed" => "failed"
    r:=pullback reductum pF
    r case "failed" => "failed"
    monomial(c,degree pF) + r
  solveLinearPolynomialEquationByFractions(lp,pp) ==
    lpF:List SupF:=[map((x:R):F +-> x@R::F,u) for u in lp]
    pF:SupF:=map((x:R):F +-> x::F,pp)
    ans:= solveLinearPolynomialEquation(lpF,pF)$F
    ans case "failed" => "failed"
    [(vv:= pullback v;
      vv case "failed" => return "failed";
       vv)
        for v in ans]
```

———-

— **LPEFRAC.dotabb** —

```
"LPEFRAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LPEFRAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LPEFRAC" -> "PFECAT"
```

———-

13.9 package LISYSER LinearSystemFromPowerSeries-Package

— **LinearSystemFromPowerSeriesPackage.input** —

```
)set break resume
```

```
)sys rm -f LinearSystemFromPowerSeriesPackage.output
)spool LinearSystemFromPowerSeriesPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemFromPowerSeriesPackage
--R
--R LinearSystemFromPowerSeriesPackage(K: Field,PCS: LocalPowerSeriesCategory(K))  is a package
--R Abbreviation for LinearSystemFromPowerSeriesPackage is LISYSER
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LISYSER
--R
--R------------------------------ Operations --------------------------------
--R finiteSeries2LinSys : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2LinSysWOVectorise : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2Vector : (PCS,Integer) -> List(K)
--R
--E 1

)spool
)lisp (bye)
```

———-

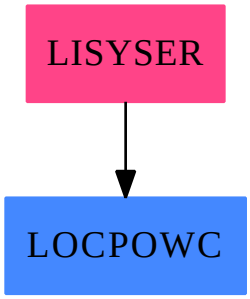
— LinearSystemFromPowerSeriesPackage.help —

```
=====================================================================
LinearSystemFromPowerSeriesPackage examples
=====================================================================

See Also:
o )show LinearSystemFromPowerSeriesPackage
```

———-

13.9.1 LinearSystemFromPowerSeriesPackage (LISYSER)



Exports:
finiteSeries2LinSys finiteSeries2LinSysWOVectorise finiteSeries2Vector

— package LISYSER LinearSystemFromPowerSeriesPackage —

```
)abbrev package LISYSER LinearSystemFromPowerSeriesPackage
++ Authors: Gaetan Hache
++ Date Created:  1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ Part of the PAFF package
LinearSystemFromPowerSeriesPackage(K,PCS):P==T where
  K : Field
  PCS: LocalPowerSeriesCategory(K)

  INT     ==> Integer
  TERM    ==> Record(k:INT,c:K)
  SER     ==> Stream(TERM)
  LOpPack ==> LinesOpPack(K)

  P==> with
    finiteSeries2LinSysWOVectorise: (List PCS, INT) -> Matrix K

    finiteSeries2LinSys: (List PCS, INT) -> Matrix K
      ++ finiteSeries2LinSys(ls,n) returns a matrix which right kernel
      ++ is the solution of the linear combinations of the series in ls
      ++ which has order greater or equal to n.
      ++ NOTE: All the series in ls must be finite and must have order
      ++ at least 0: so one must first call on each of them the
      ++ function filterUpTo(s,n) and apply an appropriate shift
      ++ (mult by a power of t).

    finiteSeries2Vector: (PCS, INT) -> List K

  T==> add
```

```
    finiteSeries2ListOfTerms: PCS -> List TERM

    finiteSeries2ListOfTermsStream: SER -> List TERM

    finiteSeries2ListOfTermsStream(s)==
      empty?(s) => empty()
      cons(frst s , finiteSeries2ListOfTermsStream(rst(s)))

    finiteSeries2LinSys(ls,n)==
      ll:List K:=  [0$K]
      lZero:=new(#ls pretend NonNegativeInteger,ll)$List(List(K))
      n <= 0  => transpose matrix lZero
      tMat:=transpose matrix [finiteSeries2Vector(s,n) for s in ls]
      rowEchWoZeroLines(tMat)$LOpPack

    finiteSeries2LinSysWOVectorise(ls,n)==
      ll:List K:=  [0$K]
      lZero:=new(#ls pretend NonNegativeInteger,ll)$List(List(K))
      n <= 0  => transpose matrix lZero
      tMat:=transpose matrix [finiteSeries2Vector(s,n) for s in ls]
      rowEchWoZeroLinesWOVectorise(tMat)$LOpPack

    finiteSeries2ListOfTerms(s)==
      ss:SER:= s :: SER
      finiteSeries2ListOfTermsStream(ss)

    finiteSeries2Vector(ins,n)==
      lZero:=new((n pretend NonNegativeInteger),0)$List(K)
      s:= removeFirstZeroes ins
      lOfTerm:=finiteSeries2ListOfTerms(s)
      for t in lOfTerm repeat lZero.((t.k)+1):= t.c
      lZero
```

———-

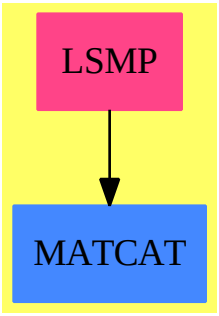
— LISYSER.dotabb —

```
"LISYSER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LISYSER"]
"LOCPOWC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LOCPOWC"]
"LISYSER" -> "LOCPOWC"
```

———-

13.10 package LSMP LinearSystemMatrixPackage

13.10.1 LinearSystemMatrixPackage (LSMP)



Exports:
hasSolution? particularSolution rank solve

— package LSMP LinearSystemMatrixPackage —

```
)abbrev package LSMP LinearSystemMatrixPackage
++ Author: P.Gianni, S.Watt
++ Date Created: Summer 1985
++ Date Last Updated:Summer 1990
++ Basic Functions: solve, particularSolution, hasSolution?, rank
++ Related Constructors: LinearSystemMatrixPackage1
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package solves linear system in the matrix form \spad{AX = B}.

LinearSystemMatrixPackage(F, Row, Col, M): Cat == Capsule where
    F: Field
    Row: FiniteLinearAggregate F with shallowlyMutable
    Col: FiniteLinearAggregate F with shallowlyMutable
    M  : MatrixCategory(F, Row, Col)

    N        ==> NonNegativeInteger
    PartialV ==> Union(Col, "failed")
    Both     ==> Record(particular: PartialV, basis: List Col)

    Cat ==> with
        solve       : (M, Col) -> Both
          ++ solve(A,B) finds a particular solution of the system \spad{AX = B}
          ++ and a basis of the associated homogeneous system \spad{AX = 0}.
```

```
    solve       : (M, List Col) -> List Both
      ++ solve(A,LB) finds a particular soln of the systems \spad{AX = B}
      ++ and a basis of the associated homogeneous systems \spad{AX = 0}
      ++ where B varies in the list of column vectors LB.

    particularSolution: (M, Col) -> PartialV
      ++ particularSolution(A,B) finds a particular solution of the linear
      ++ system \spad{AX = B}.
    hasSolution?: (M, Col) -> Boolean
      ++ hasSolution?(A,B) tests if the linear system \spad{AX = B}
      ++ has a solution.
    rank        : (M, Col) -> N
      ++ rank(A,B) computes the rank of the complete matrix \spad{(A|B)}
      ++ of the linear system \spad{AX = B}.

  Capsule ==> add
    systemMatrix      : (M, Col) -> M
    aSolution         :  M -> PartialV

    -- rank theorem
    hasSolution?(A, b) == rank A = rank systemMatrix(A, b)
    systemMatrix(m, v) == horizConcat(m, -(v::M))
    rank(A, b)         == rank systemMatrix(A, b)
    particularSolution(A, b) == aSolution rowEchelon systemMatrix(A,b)

    -- m should be in row-echelon form.
    -- last column of m is -(right-hand-side of system)
    aSolution m ==
       nvar := (ncols m - 1)::N
       rk := maxRowIndex m
       while (rk >= minRowIndex m) and every?(zero?, row(m, rk))
         repeat rk := dec rk
       rk < minRowIndex m => new(nvar, 0)
       ck := minColIndex m
       while (ck < maxColIndex m) and zero? qelt(m, rk, ck) repeat
         ck := inc ck
       ck = maxColIndex m => "failed"
       sol := new(nvar, 0)$Col
       -- find leading elements of diagonal
       v := new(nvar, minRowIndex m - 1)$PrimitiveArray(Integer)
       for i in minRowIndex m .. rk repeat
         for j in 0.. while zero? qelt(m, i, j+minColIndex m) repeat 0
         v.j := i
       for j in 0..nvar-1 repeat
         if v.j >= minRowIndex m then
           qsetelt_!(sol, j+minIndex sol, - qelt(m, v.j, maxColIndex m))
       sol

    solve(A:M, b:Col) ==
        -- Special case for homogeneous systems.
```

```
        every?(zero?, b) => [new(ncols A, 0), nullSpace A]
        -- General case.
        m   := rowEchelon systemMatrix(A, b)
        [aSolution m,
         nullSpace subMatrix(m, minRowIndex m, maxRowIndex m,
                                  minColIndex m, maxColIndex m - 1)]

    solve(A:M, l:List Col) ==
        null l => [[new(ncols A, 0), nullSpace A]]
        nl := (sol0 := solve(A, first l)).basis
        cons(sol0,
               [[aSolution rowEchelon systemMatrix(A, b), nl]
                                                    for b in rest l])
```

———-

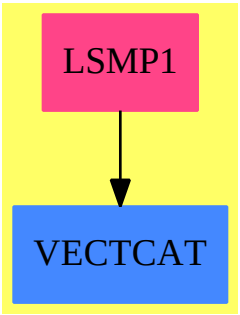
— **LSMP.dotabb** —

```
"LSMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LSMP"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"LSMP" -> "MATCAT"
```

———-

13.11 package LSMP1 LinearSystemMatrixPackage1

13.11.1 LinearSystemMatrixPackage1 (LSMP1)



Exports:
hasSolution? particularSolution rank solve

— **package LSMP1 LinearSystemMatrixPackage1** —

```
)abbrev package LSMP1 LinearSystemMatrixPackage1
++ Author: R. Sutor
++ Date Created: June, 1994
++ Date Last Updated:
++ Basic Functions: solve, particularSolution, hasSolution?, rank
++ Related Constructors: LinearSystemMatrixPackage
++ Also See:
++ AMS Classifications:
++ Keywords: solve
++ References:
++ Description:
++ This package solves linear system in the matrix form \spad{AX = B}.
++ It is essentially a particular instantiation of the package
++ \spadtype{LinearSystemMatrixPackage} for Matrix and Vector. This
++ package's existence makes it easier to use \spadfun{solve} in the
++ AXIOM interpreter.

LinearSystemMatrixPackage1(F): Cat == Capsule where
    F: Field
    Row      ==> Vector F
    Col      ==> Vector F
    M        ==> Matrix(F)
    LL       ==> List List F

    N        ==> NonNegativeInteger
    PartialV ==> Union(Col, "failed")
    Both     ==> Record(particular: PartialV, basis: List Col)
    LSMP     ==> LinearSystemMatrixPackage(F, Row, Col, M)

    Cat ==> with
        solve       : (M, Col) -> Both
          ++ solve(A,B) finds a particular solution of the system \spad{AX = B}
          ++ and a basis of the associated homogeneous system \spad{AX = 0}.
        solve       : (LL, Col) -> Both
          ++ solve(A,B) finds a particular solution of the system \spad{AX = B}
          ++ and a basis of the associated homogeneous system \spad{AX = 0}.
        solve       : (M, List Col) -> List Both
          ++ solve(A,LB) finds a particular soln of the systems \spad{AX = B}
          ++ and a basis of the associated homogeneous systems \spad{AX = 0}
          ++ where B varies in the list of column vectors LB.
        solve       : (LL, List Col) -> List Both
          ++ solve(A,LB) finds a particular soln of the systems \spad{AX = B}
          ++ and a basis of the associated homogeneous systems \spad{AX = 0}
          ++ where B varies in the list of column vectors LB.

        particularSolution: (M, Col) -> PartialV
          ++ particularSolution(A,B) finds a particular solution of the linear
          ++ system \spad{AX = B}.
        hasSolution?: (M, Col) -> Boolean
          ++ hasSolution?(A,B) tests if the linear system \spad{AX = B}
```

```
          ++ has a solution.
        rank        : (M, Col) -> N
          ++ rank(A,B) computes the rank of the complete matrix \spad{(A|B)}
          ++ of the linear system \spad{AX = B}.

    Capsule ==> add
        solve(m : M, c: Col): Both == solve(m,c)$LSMP
        solve(ll : LL, c: Col): Both == solve(matrix(ll)$M,c)$LSMP
        solve(m : M, l : List Col): List Both == solve(m, l)$LSMP
        solve(ll : LL, l : List Col): List Both == solve(matrix(ll)$M, l)$LSMP
        particularSolution (m : M, c : Col): PartialV == particularSolution(m, c)$LSMP
        hasSolution?(m :M, c : Col): Boolean == hasSolution?(m, c)$LSMP
        rank(m : M, c : Col): N == rank(m, c)$LSMP
```

———-

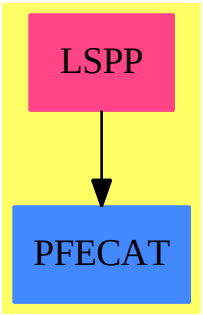
— **LSMP1.dotabb** —

```
"LSMP1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LSMP1"]
"VECTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=VECTCAT"]
"LSMP1" -> "VECTCAT"
```

———-

13.12 package LSPP LinearSystemPolynomialPackage

13.12.1 LinearSystemPolynomialPackage (LSPP)



Exports:
linSolve

— **package LSPP LinearSystemPolynomialPackage** —

```
)abbrev package LSPP LinearSystemPolynomialPackage
++ Author:  P.Gianni
++ Date Created: Summer 1985
++ Date Last Updated: Summer 1993
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References: SystemSolvePackage
++ Description:
++ This package finds the solutions of linear systems presented as a
++ list of polynomials.

LinearSystemPolynomialPackage(R, E, OV, P): Cat == Capsule where
    R          :   IntegralDomain
    OV         :   OrderedSet
    E          :   OrderedAbelianMonoidSup
    P          :   PolynomialCategory(R,E,OV)

    F        ==> Fraction P
    NNI      ==> NonNegativeInteger
    V        ==> Vector
    M        ==> Matrix
    Soln     ==> Record(particular: Union(V F, "failed"), basis: List V F)

    Cat == with
        linSolve:  (List P, List OV) -> Soln
          ++ linSolve(lp,lvar) finds the solutions of the linear system
          ++ of polynomials lp = 0 with respect to the list of symbols lvar.

    Capsule == add

                       ---- Local Functions ----

        poly2vect:    (P,     List OV)    -> Record(coefvec: V F, reductum: F)
        intoMatrix:   (List P,   List OV) -> Record(mat: M F, vec: V F)


        poly2vect(p : P, vs : List OV) : Record(coefvec: V F, reductum: F) ==
            coefs := new(#vs, 0)$(V F)
            for v in vs for i in 1.. while p ^= 0 repeat
              u := univariate(p, v)
              degree u = 0 => "next v"
              coefs.i := (c := leadingCoefficient u)::F
              p := p - monomial(c,v, 1)
            [coefs, p :: F]

        intoMatrix(ps : List P, vs : List OV ) : Record(mat: M F, vec: V F) ==
            m := zero(#ps, #vs)$M(F)
```

```
            v := new(#ps, 0)$V(F)
            for p in ps for i in 1.. repeat
                totalDegree(p,vs) > 1 => error "The system is not linear"
                r   := poly2vect(p,vs)
                m:=setRow_!(m,i,r.coefvec)
                v.i := - r.reductum
            [m, v]

        linSolve(ps, vs) ==
            r := intoMatrix(ps, vs)
            solve(r.mat, r.vec)$LinearSystemMatrixPackage(F,V F,V F,M F)
```

———-

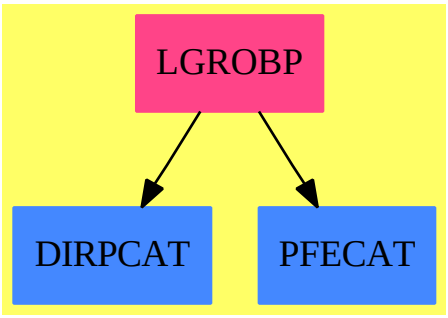
— **LSPP.dotabb** —

```
"LSPP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LSPP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LSPP" -> "PFECAT"
```

———-

13.13 package LGROBP LinGroebnerPackage

13.13.1 LinGroebnerPackage (LGROBP)



Exports:

anticoord	choosemon	computeBasis	coord	groebgen
intcompBasis	linGenPos	minPol	totolex	transform

— **package LGROBP LinGroebnerPackage** —

```
)abbrev package LGROBP LinGroebnerPackage
```

```
++ Description:
++ Given a Groebner basis B with respect to the total degree ordering for
++ a zero-dimensional ideal I, compute
++ a Groebner basis with respect to the lexicographical ordering by using
++ linear algebra.

LinGroebnerPackage(lv,F) : C == T

 where
  Z      ==>  Integer
  lv     :    List Symbol
  F      :    GcdDomain

  DP     ==>  DirectProduct(#lv,NonNegativeInteger)
  DPoly  ==>  DistributedMultivariatePolynomial(lv,F)

  HDP    ==>  HomogeneousDirectProduct(#lv,NonNegativeInteger)
  HDPoly ==>  HomogeneousDistributedMultivariatePolynomial(lv,F)

  OV     ==>  OrderedVariableList(lv)
  NNI    ==>  NonNegativeInteger
  LVals  ==>  Record(gblist : List DPoly,gvlist : List Z)
  VF     ==>  Vector F
  VV     ==>  Vector NNI
  MF     ==>  Matrix F
  cLVars ==>  Record(glbase:List DPoly,glval:List Z)

  C == with

     linGenPos    :           List HDPoly      -> LVals
        ++ linGenPos \undocumented
     groebgen     :           List DPoly       -> cLVars
        ++ groebgen \undocumented
     totolex      :           List HDPoly      -> List DPoly
        ++ totolex \undocumented
     minPol       : (List HDPoly,List HDPoly,OV) -> HDPoly
        ++ minPol \undocumented
     minPol       :        (List HDPoly,OV)     -> HDPoly
        ++ minPol \undocumented


     computeBasis :        List HDPoly           -> List HDPoly
        ++ computeBasis \undocumented
     coord        : (HDPoly,List HDPoly)         -> VF
        ++ coord \undocumented
     anticoord    : (List F,DPoly,List DPoly)    -> DPoly
        ++ anticoord \undocumented
     intcompBasis : (OV,List HDPoly,List HDPoly) -> List HDPoly
        ++ intcompBasis \undocumented
     choosemon    :     (DPoly,List  DPoly)      -> DPoly
```

```
        ++ choosemon \undocumented
     transform    :           DPoly              -> HDPoly
        ++ transform \undocumented


  T == add

    import GroebnerPackage(F,DP,OV,DPoly)
    import GroebnerPackage(F,HDP,OV,HDPoly)
    import GroebnerInternalPackage(F,HDP,OV,HDPoly)
    import GroebnerInternalPackage(F,DP,OV,DPoly)

    lvar :=[variable(yx)::OV for yx in lv]

    reduceRow(M:MF, v : VF, lastRow: Integer, pivots: Vector(Integer)) : VF ==
      a1:F := 1
      b:F := 0
      dim := #v
      for j in 1..lastRow repeat -- scan over rows
         mj := row(M,j)
         k:=pivots(j)
         b:=mj.k
         vk := v.k
         for kk in 1..(k-1) repeat
            v(kk) := ((-b*v(kk)) exquo a1) :: F
         for kk in k..dim repeat
            v(kk) := ((vk*mj(kk)-b*v(kk)) exquo a1)::F
         a1 := b
      v

    rRedPol(f:HDPoly, B:List HDPoly):Record(poly:HDPoly, mult:F) ==
      gm := redPo(f,B)
      gm.poly = 0 => gm
      gg := reductum(gm.poly)
      ggm := rRedPol(gg,B)
      [ggm.mult*(gm.poly - gg) + ggm.poly, ggm.mult*gm.mult]

----- transform the total basis B in lex basis -----
    totolex(B : List HDPoly) : List DPoly ==
      result:List DPoly :=[]
      ltresult:List DPoly :=[]
      vBasis:= computeBasis B
      nBasis:List DPoly :=[1$DPoly]
      ndim:=(#vBasis)::PositiveInteger
      ndim1:NNI:=ndim+1
      lm:VF
      linmat:MF:=zero(ndim,2*ndim+1)
      linmat(1,1):=1$F
      linmat(1,ndim1):=1
      pivots:Vector Integer := new(ndim,0)
```

```
      pivots(1) := 1
      firstmon:DPoly:=1$DPoly
      ofirstmon:DPoly:=1$DPoly
      orecfmon:Record(poly:HDPoly, mult:F) := [1,1]
      i:NNI:=2
      while (firstmon:=choosemon(firstmon,ltresult))^=1 repeat
        if (v:=firstmon exquo ofirstmon) case "failed" then
          recfmon:=rRedPol(transform firstmon,B)
        else
          recfmon:=rRedPol(transform(v::DPoly) *orecfmon.poly,B)
          recfmon.mult := recfmon.mult * orecfmon.mult
        cc := gcd(content recfmon.poly, recfmon.mult)
        recfmon.poly := (recfmon.poly exquo cc)::HDPoly
        recfmon.mult := (recfmon.mult exquo cc)::F
        veccoef:VF:=coord(recfmon.poly,vBasis)
        ofirstmon:=firstmon
        orecfmon := recfmon
        lm:=zero(2*ndim+1)
        for j in 1..ndim repeat lm(j):=veccoef(j)
        lm(ndim+i):=recfmon.mult
        lm := reduceRow(linmat, lm, i-1, pivots)
        if i=ndim1 then j:=ndim1
        else
          j:=1
          while lm(j) = 0 and j< ndim1 repeat j:=j+1
        if j=ndim1 then
          cordlist:List F:=[lm(k) for k in ndim1..ndim1+(#nBasis)]
          antc:=+/[c*b for c in reverse cordlist
                       for b in concat(firstmon,nBasis)]
          antc:=primitivePart antc
          result:=concat(antc,result)
          ltresult:=concat(antc-reductum antc,ltresult)
        else
          pivots(i) := j
          setRow_!(linmat,i,lm)
          i:=i+1
          nBasis:=cons(firstmon,nBasis)
      result

---- Compute the univariate polynomial for x
----oldBasis is a total degree Groebner basis
    minPol(oldBasis:List HDPoly,x:OV) :HDPoly ==
      algBasis:= computeBasis oldBasis
      minPol(oldBasis,algBasis,x)

---- Compute the univariate polynomial for x
---- oldBasis is total Groebner, algBasis is the basis as algebra
    minPol(oldBasis:List HDPoly,algBasis:List HDPoly,x:OV) :HDPoly ==
      nvp:HDPoly:=x::HDPoly
      f:=1$HDPoly
```

```
      omult:F :=1
      ndim:=(#algBasis)::PositiveInteger
      ndim1:NNI:=ndim+1
      lm:VF
      linmat:MF:=zero(ndim,2*ndim+1)
      linmat(1,1):=1$F
      linmat(1,ndim1):=1
      pivots:Vector Integer := new(ndim,0)
      pivots(1) := 1
      for i in 2..ndim1 repeat
        recf:=rRedPol(f*nvp,oldBasis)
        omult := recf.mult * omult
        f := recf.poly
        cc := gcd(content f, omult)
        f := (f exquo cc)::HDPoly
        omult := (omult exquo cc)::F
        veccoef:VF:=coord(f,algBasis)
        lm:=zero(2*ndim+1)
        for j in 1..ndim repeat lm(j) := veccoef(j)
        lm(ndim+i):=omult
        lm := reduceRow(linmat, lm, i-1, pivots)
        j:=1
        while lm(j)=0 and j<ndim1 repeat j:=j+1
        if j=ndim1 then return
          g:HDPoly:=0
          for k in ndim1..2*ndim+1 repeat
            g:=g+lm(k) * nvp**((k-ndim1):NNI)
          primitivePart g
        pivots(i) := j
        setRow_!(linmat,i,lm)

----- transform a DPoly in a HDPoly -----
    transform(dpol:DPoly) : HDPoly ==
      dpol=0 => 0$HDPoly
      monomial(leadingCoefficient dpol,
               directProduct(degree(dpol)::VV)$HDP)$HDPoly +
                                    transform(reductum dpol)

----- compute the basis for the vector space determined by B -----
    computeBasis(B:List HDPoly) : List HDPoly ==
      mB:List HDPoly:=[monomial(1$F,degree f)$HDPoly for f in B]
      result:List HDPoly := [1$HDPoly]
      for var in lvar repeat
        part:=intcompBasis(var,result,mB)
        result:=concat(result,part)
      result

----- internal function for computeBasis -----
    intcompBasis(x:OV,lr:List HDPoly,mB : List HDPoly):List HDPoly ==
      lr=[] => lr
```

```
      part:List HDPoly :=[]
      for f in lr repeat
        g:=x::HDPoly * f
        if redPo(g,mB).poly^=0 then part:=concat(g,part)
      concat(part,intcompBasis(x,part,mB))

----- coordinate of f with respect to the basis B -----
----- f is a reduced polynomial -----
    coord(f:HDPoly,B:List HDPoly) : VF ==
      ndim := #B
      vv:VF:=new(ndim,0$F)$VF
      while f^=0 repeat
        rf := reductum f
        lf := f-rf
        lcf := leadingCoefficient f
        i:Z:=position(monomial(1$F,degree lf),B)
        vv.i:=lcf
        f := rf
      vv

----- reconstruct the polynomial from its coordinate -----
    anticoord(vv:List F,mf:DPoly,B:List DPoly) : DPoly ==
      for f in B for c in vv repeat (mf:=mf-c*f)
      mf

----- choose the next monom -----
    choosemon(mf:DPoly,nB:List DPoly) : DPoly ==
      nB = [] => ((lvar.last)::DPoly)*mf
      for x in reverse lvar repeat
        xx:=x ::DPoly
        mf:=xx*mf
        if redPo(mf,nB).poly ^= 0 then return mf
        dx := degree(mf,x)
        mf := (mf exquo (xx ** dx))::DPoly
      mf

----- put B in general position, B is Groebner -----
    linGenPos(B : List HDPoly) : LVals ==
      result:List DPoly :=[]
      ltresult:List DPoly :=[]
      vBasis:= computeBasis B
      nBasis:List DPoly :=[1$DPoly]
      ndim:=#vBasis : PositiveInteger
      ndim1:NNI:=ndim+1
      lm:VF
      linmat:MF:=zero(ndim,2*ndim+1)
      linmat(1,1):=1$F
      linmat(1,ndim1):=1
      pivots:Vector Integer := new(ndim,0)
      pivots(1) := 1
```

```
      i:NNI:=2
      rval:List Z :=[]
      for ii in 1..(#lvar-1) repeat
        c:Z:=0
        while c=0 repeat c:=random()$Z rem 11
        rval:=concat(c,rval)
      nval:DPoly := (last.lvar)::DPoly -
                (+/[r*(vv)::DPoly for r in rval for vv in lvar])
      firstmon:DPoly:=1$DPoly
      ofirstmon:DPoly:=1$DPoly
      orecfmon:Record(poly:HDPoly, mult:F) := [1,1]
      lx:= lvar.last
      while (firstmon:=choosemon(firstmon,ltresult))^=1 repeat
        if (v:=firstmon exquo ofirstmon) case "failed" then
          recfmon:=rRedPol(transform(eval(firstmon,lx,nval)),B)
        else
          recfmon:=rRedPol(transform(eval(v,lx,nval))*orecfmon.poly,B)
          recfmon.mult := recfmon.mult * orecfmon.mult
        cc := gcd(content recfmon.poly, recfmon.mult)
        recfmon.poly := (recfmon.poly exquo cc)::HDPoly
        recfmon.mult := (recfmon.mult exquo cc)::F
        veccoef:VF:=coord(recfmon.poly,vBasis)
        ofirstmon:=firstmon
        orecfmon := recfmon
        lm:=zero(2*ndim+1)
        for j in 1..ndim repeat lm(j):=veccoef(j)
        lm(ndim+i):=recfmon.mult
        lm := reduceRow(linmat, lm, i-1, pivots)
        j:=1
        while lm(j) = 0 and j<ndim1 repeat j:=j+1
        if j=ndim1 then
          cordlist:List F:=[lm(j) for j in ndim1..ndim1+(#nBasis)]
          antc:=+/[c*b for c in reverse cordlist
                       for b in concat(firstmon,nBasis)]
          result:=concat(primitivePart antc,result)
          ltresult:=concat(antc-reductum antc,ltresult)
        else
          pivots(i) := j
          setRow_!(linmat,i,lm)
          i:=i+1
          nBasis:=concat(firstmon,nBasis)
      [result,rval]$LVals

----- given a basis of a zero-dimensional ideal,
----- performs a random change of coordinates
----- computes a Groebner basis for the lex ordering
    groebgen(L:List DPoly) : cLVars ==
      xn:=lvar.last
      val := xn::DPoly
      nvar1:NNI:=(#lvar-1):NNI
```

```
      ll: List Z :=[random()$Z rem 11 for i in 1..nvar1]
      val:=val+ +/[ll.i*(lvar.i)::DPoly for i in 1..nvar1]
      LL:=[elt(univariate(f,xn),val) for f in L]
      LL:=  groebner(LL)
      [LL,ll]$cLVars
```

———-

— LGROBP.dotabb —

```
"LGROBP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LGROBP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LGROBP" -> "DIRPCAT"
"LGROBP" -> "PFECAT"
```

———-

13.14 package LOP LinesOpPack

— LinesOpPack.input —

```
)set break resume
)sys rm -f LinesOpPack.output
)spool LinesOpPack.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinesOpPack
--R
--R LinesOpPack(K: Field)  is a package constructor
--R Abbreviation for LinesOpPack is LOP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LOP
--R
--R------------------------------ Operations --------------------------------
--R quotVecSpaceBasis : (List(List(K)),List(List(K))) -> List(List(K))
--R reduceLineOverLine : (List(K),List(K),K) -> List(K)
--R reduceRow : List(List(K)) -> List(List(K))
--R reduceRowOnList : (List(K),List(List(K))) -> List(List(K))
--R rowEchWoZeroLines : Matrix(K) -> Matrix(K)
--R rowEchWoZeroLinesWOVectorise : Matrix(K) -> Matrix(K)
```

```
--R
--E 1

)spool
)lisp (bye)
```

———-

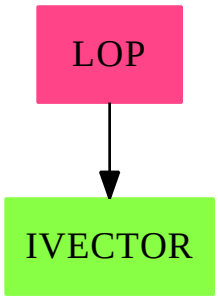
— **LinesOpPack.help** —

```
=====================================================================
LinesOpPack examples
=====================================================================

See Also:
o )show LinesOpPack
```

———-

13.14.1 LinesOpPack (LOP)



Exports:

quotVecSpaceBasis	reduceLineOverLine	reduceRow
reduceRowOnList	rowEchWoZeroLines	rowEchWoZeroLinesWOVectorise

— **package LOP LinesOpPack** —

```
)abbrev package LOP LinesOpPack
++ Authors: G. Hache
++ Date Created: 21 sept 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that exports several linear algebra operations over lines
```

```
++ of matrices. Part of the PAFF package.
LinesOpPack(K):P==T where

  K:Field

  P==> with
    rowEchWoZeroLinesWOVectorise: Matrix(K) -> Matrix(K)
    rowEchWoZeroLines: Matrix(K) -> Matrix(K)
    reduceRow: List(List(K)) -> List(List(K))
      ++ reduceRow: if the input is considered as a matrix, the output would
      ++ be the row reduction matrix. It's almost the rowEchelon form
      ++ except that no permution of lines is performed.
    quotVecSpaceBasis: (List(List(K)),List(List(K))) -> List(List(K))
      ++ quotVecSpaceBasis(b1,b2) returns a basis of V1/V2 where
      ++ V1 and V2 are vector space with basis b1  and b2 resp. and
      ++ V2 is suppose to be include in V1; Note that if
      ++ it is not the case then it returs the basis of V1/W
      ++ where W = intersection of V1 and V2
    reduceLineOverLine: (List(K),List(K),K) -> List(K)
      ++reduceLineOverLine(v1,v2,a) returns v1-a*v1 where
      ++ v1 and v2 are considered as vector space.
    reduceRowOnList: (List(K),List(List(K))) -> List(List(K))
      ++ reduceRowOnList(v,lvec) applies a row reduction on each of the
      ++ element of lv using v according to a pivot in v which is set to
      ++ be the first non nul element in v.

  T==> add
    localRowEchelon: Matrix(K) -> Matrix(K)
    localRowEchelon(m)==
      ^(K has PseudoAlgebraicClosureOfPerfectFieldCategory )  => rowEchelon m
      llm:List(List(K)):= listOfLists m
      l:= first llm
      maxT:= maxTower l
      lv := [vectorise(a,maxT)$K for a in l]
      subMatl := transpose  matrix [entries(v) for v in lv]
      matl:= subMatl
      for l in rest llm repeat
        maxT:= maxTower l
        lv := [vectorise(a,maxT)$K for a in l]
        subMatl := transpose  matrix [entries(v) for v in lv]
        matl:=vertConcat(matl,subMatl)
      rowEchelon  matl

    rowEchWoZeroLines(m)==
      mm:=localRowEchelon m
      ll:=listOfLists mm
      n:= # first ll
      lZero:=new(n pretend NonNegativeInteger,0)$List(K)
      llll:= [ l for l in ll | ^(lZero = l) ]
      empty?(llll) => matrix [lZero]
```

```
      matrix llll

    rowEchWoZeroLinesWOVectorise(m)==
      mm:=rowEchelon  m
      ll:=listOfLists mm
      n:= # first ll
      lZero:=new(n pretend NonNegativeInteger,0)$List(K)
      llll:= [ l for l in ll | ^(lZero = l) ]
      empty?(llll) => matrix [lZero]
      matrix llll

    quotVecSpaceBasis(l2,l1)==
      redBasis:=reduceRow(concat(l1,l2))
      tempRes:=rest(redBasis,#l1)
      allZero:=new(#l1.1,0$K)
      [l for l in tempRes | ^(l=allZero)]

    reduceRowOnList(line,listOfLine)==
      frsNonNul:Integer:=position(^zero?(#1),line)
      ^(frsNonNul > 0) => listOfLine
      a:= line.frsNonNul
      inva:= inv a
      newLine:=[inva*c for c in line]
      [reduceLineOverLine(newLine,l,l.frsNonNul) for l in listOfLine]

    reduceLineOverLine(l1,l2,b)==
      [c2 - b*c1 for c2 in l2 for c1 in l1]

    reduceRow(m:List(List(K)))==
      n:=#m
      mcopy:List(List(K)):=copy m
      newBottom:List(List(K))
      for i in 1..(n-1) repeat
        newBottom:=reduceRowOnList(mcopy.i,[mcopy.j for j in (i+1)..n])
        mcopy:=concat([mcopy.k for k in 1..i] :: List(List(K)),newBottom)
      mcopy
```

———-

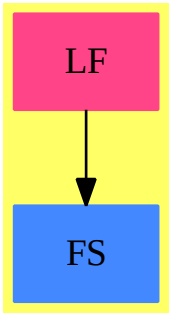
— LOP.dotabb —

```
"LOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LOP"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"LOP" -> "IVECTOR"
```

———-

13.15 package LF LiouvillianFunction

13.15.1 LiouvillianFunction (LF)



Exports:

belong?	Ci	dilog	Ei	erf
integral	li	integral	operator	Si

— package LF LiouvillianFunction —

```
)abbrev package LF LiouvillianFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 August 1994
++ Keywords: liouvillian, function, primitive, exponential.
++ Examples:  )r LF INPUT
++ Description:
++ This package provides liouvillian functions over an integral domain.

LiouvillianFunction(R, F): Exports == Implementation where
  R:Join(OrderedSet, IntegralDomain)
  F:Join(FunctionSpace R,RadicalCategory,TranscendentalFunctionCategory)

  OP  ==> BasicOperator
  PR  ==> Polynomial R
  K   ==> Kernel F
  SE  ==> Symbol
  O   ==> OutputForm
  INP ==> InputForm
  INV ==> error "Invalid argument"

  SPECIALDIFF ==> "%specialDiff"
  SPECIALDISP ==> "%specialDisp"
  SPECIALINPUT ==> "%specialInput"
  SPECIALEQUAL ==> "%specialEqual"

  Exports ==> with
```

```
    belong? : OP -> Boolean
      ++ belong?(op) checks if op is Liouvillian
    operator: OP -> OP
      ++ operator(op) returns the Liouvillian operator based on op
    Ei      : F  -> F
      ++ Ei(f) denotes the exponential integral
    Si      : F  -> F
      ++ Si(f) denotes the sine integral
    Ci      : F  -> F
      ++ Ci(f) denotes the cosine integral
    li      : F  -> F
      ++ li(f) denotes the logarithmic integral
    erf     : F  -> F
      ++ erf(f) denotes the error function
    dilog   : F  -> F
      ++ dilog(f) denotes the dilogarithm
    fresnelS : F -> F
      ++ fresnelS(f) denotes the Fresnel integral S
    fresnelC : F -> F
      ++ fresnelC(f) denotes the Fresnel integral C
    integral: (F, SE) -> F
      ++ integral(f,x) indefinite integral of f with respect to x.
    integral: (F, SegmentBinding F) -> F
      ++ integral(f,x = a..b) denotes the definite integral of f with
      ++ respect to x from \spad{a} to b.

  Implementation ==> add
    iei        : F  -> F
    isi        : F  -> F
    ici        : F  -> F
    ierf       : F  -> F
    ili        : F  -> F
    ili2       : F  -> F
    iint       : List F -> F
    eqint      : (K,K) -> Boolean
    dvint      : (List F, SE) -> F
    dvdint     : (List F, SE) -> F
    ddint      : List F -> O
    integrand  : List F -> F

    dummy := new()$SE :: F

    opint  := operator("integral"::Symbol)$CommonOperators
    opdint := operator("%defint"::Symbol)$CommonOperators
    opei   := operator("Ei"::Symbol)$CommonOperators
    opli   := operator("li"::Symbol)$CommonOperators
    opsi   := operator("Si"::Symbol)$CommonOperators
    opci   := operator("Ci"::Symbol)$CommonOperators
    opli2  := operator("dilog"::Symbol)$CommonOperators
    operf  := operator("erf"::Symbol)$CommonOperators
```

```
    opfis  := operator("fresnelS"::Symbol)$CommonOperators
    opfic  := operator("fresnelC"::Symbol)$CommonOperators

    Si x               == opsi x
    Ci x               == opci x
    Ei x               == opei x
    erf x              == operf x
    li  x              == opli x
    dilog x            == opli2 x
    fresnelS x         == opfis x
    fresnelC x         == opfic x

    belong? op     == has?(op, "prim")
    isi x          == kernel(opsi, x)
    ici x          == kernel(opci, x)
    ierf x         == (zero? x => 0; kernel(operf, x))
--    ili2 x         == (one? x => INV; kernel(opli2, x))
    ili2 x         == ((x = 1) => INV; kernel(opli2, x))
    ifis(x:F):F    == (zero? x => 0; kernel(opfis,x))
    ific(x:F):F    == (zero? x => 0; kernel(opfic,x))
    integrand l    == eval(first l, retract(second l)@K, third l)
    integral(f:F, x:SE) == opint [eval(f, k:=kernel(x)$K, dummy), dummy, k::F]

    iint l ==
      zero? first l => 0
      kernel(opint, l)

    ddint l ==
      int(integrand(l)::O * hconcat("d"::SE::O, third(l)::O),
                                  third(rest l)::O, third(rest rest l)::O)

    eqint(k1,k2) ==
      a1:=argument k1
      a2:=argument k2
      res:=operator k1 = operator k2
      if not res then return res
      res:= a1 = a2
      if res then return res
      res:= (a1.3 = a2.3) and (subst(a1.1,[retract(a1.2)@K],[a2.2]) = a2.1)

    dvint(l, x) ==
      k  := retract(second l)@K
      differentiate(third l, x) * integrand l
          + opint [differentiate(first l, x), second l, third l]


    dvdint(l, x) ==
      x = retract(y := third l)@SE => 0
      k := retract(d := second l)@K
      differentiate(h := third rest rest l,x) * eval(f := first l, k, h)
```

```
      - differentiate(g := third rest l, x) * eval(f, k, g)
           + opdint [differentiate(f, x), d, y, g, h]

  integral(f:F, s: SegmentBinding F) ==
    x := kernel(variable s)$K
    opdint [eval(f,x,dummy), dummy, x::F, lo segment s, hi segment s]

  ili x ==
    x = 1 => INV
    is?(x, "exp"::Symbol) => Ei first argument(retract(x)@K)
    kernel(opli, x)

  iei x ==
    x = 0 => INV
    is?(x, "log"::Symbol) => li first argument(retract(x)@K)
    kernel(opei, x)

  operator op ==
    is?(op, "integral"::Symbol)   => opint
    is?(op, "%defint"::Symbol)    => opdint
    is?(op, "Ei"::Symbol)         => opei
    is?(op, "Si"::Symbol)         => opsi
    is?(op, "Ci"::Symbol)         => opci
    is?(op, "li"::Symbol)         => opli
    is?(op, "erf"::Symbol)        => operf
    is?(op, "dilog"::Symbol)      => opli2
    is?(op, "fresnelC"::Symbol)   => opfis
    is?(op, "fresnelS"::Symbol)   => opfic
    error "Not a Liouvillian operator"

  evaluate(opei,    iei)$BasicOperatorFunctions1(F)
  evaluate(opli,    ili)
  evaluate(opsi,    isi)
  evaluate(opci,    ici)
  evaluate(operf,   ierf)
  evaluate(opli2,   ili2)
  evaluate(opfis,   ifis)
  evaluate(opfic,   ific)
  evaluate(opint,   iint)
  derivative(opsi,  (z1:F):F +-> sin(z1) / z1)
  derivative(opci,  (z1:F):F +-> cos(z1) / z1)
  derivative(opei,  (z1:F):F +-> exp(z1) / z1)
  derivative(opli,  (z1:F):F +-> inv log(z1))
  derivative(operf, (z1:F):F +-> 2 * exp(-(z1**2)) / sqrt(pi()))
  derivative(opli2, (z1:F):F +-> log(z1) / (1 - z1))
  derivative(opfis, (z1:F):F +-> sin(z1**2))
  derivative(opfic, (z1:F):F +-> cos(z1**2))
  setProperty(opint,SPECIALEQUAL,eqint@((K,K) -> Boolean) pretend None)
  setProperty(opint,SPECIALDIFF,dvint@((List F,SE) -> F) pretend None)
  setProperty(opdint,SPECIALDIFF,dvdint@((List F,SE)->F) pretend None)
```

```
    setProperty(opdint, SPECIALDISP, ddint@(List F -> O) pretend None)

    if R has ConvertibleTo INP then
      inint : List F -> INP
      indint: List F -> INP
      pint  : List INP -> INP


      pint l  == convert concat(convert("integral"::SE)@INP, l)
      inint l ==
        r2:= convert(
              [convert("::"::SE)@INP,
               convert(third l)@INP,
               convert("Symbol"::SE)@INP]@List INP)@INP
        pint [convert(integrand l)@INP, r2]

      indint l ==
        pint [convert(integrand l)@INP,
              convert concat(convert("="::SE)@INP,
                             [convert(third l)@INP,
                              convert concat(convert("SEGMENT"::SE)@INP,
                                             [convert(third rest l)@INP,
                                              convert(third rest rest l)@INP])])]

      setProperty(opint, SPECIALINPUT, inint@(List F -> INP) pretend None)
      setProperty(opdint, SPECIALINPUT, indint@(List F -> INP) pretend None)
```

———-

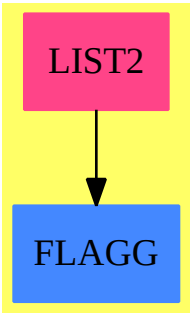
— LF.dotabb —

```
"LF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"LF" -> "FS"
```

———-

13.16 package LIST2 ListFunctions2

13.16.1 ListFunctions2 (LIST2)



Exports:
map reduce scan

— package LIST2 ListFunctions2 —

```
)abbrev package LIST2 ListFunctions2
++ Author:
++ Date Created:
++ Change History:
++ Basic Operations: map, reduce, scan
++ Related Constructors: List
++ Also See: ListFunctions3
++ AMS Classification:
++ Keywords: list, aggregate, map, reduce
++ Description:
++ \spadtype{ListFunctions2} implements utility functions that
++ operate on two kinds of lists, each with a possibly different
++ type of element.

ListFunctions2(A:Type, B:Type): public == private where
  LA     ==> List A
  LB     ==> List B
  O2     ==> FiniteLinearAggregateFunctions2(A, LA, B, LB)

  public ==> with
    scan:    ((A, B) -> B, LA, B) -> LB
      ++ scan(fn,u,ident) successively uses the binary function
      ++ \spad{fn} to reduce more and more of list \spad{u}.
      ++ \spad{ident} is returned if the \spad{u} is empty.
      ++ The result is a list of the reductions at each step. See
      ++ \spadfun{reduce} for more information. Examples:
      ++ \spad{scan(fn,[1,2],0) = [fn(2,fn(1,0)),fn(1,0)]} and
```

```
      ++ \spad{scan(*,[2,3],1) = [2 * 1, 3 * (2 * 1)]}.
    reduce:  ((A, B) -> B, LA, B) -> B
      ++ reduce(fn,u,ident) successively uses the binary function
      ++ \spad{fn} on the elements of list \spad{u} and the result
      ++ of previous applications. \spad{ident} is returned if the
      ++ \spad{u} is empty. Note the order of application in
      ++ the following examples:
      ++ \spad{reduce(fn,[1,2,3],0) = fn(3,fn(2,fn(1,0)))} and
      ++ \spad{reduce(*,[2,3],1) = 3 * (2 * 1)}.
    map:      (A -> B, LA) -> LB
      ++ map(fn,u) applies \spad{fn} to each element of
      ++ list \spad{u} and returns a new list with the results.
      ++ For example \spad{map(square,[1,2,3]) = [1,4,9]}.

  private ==> add
    map(f, l)       == map(f, l)$O2
    scan(f, l, b)   == scan(f, l, b)$O2
    reduce(f, l, b) == reduce(f, l, b)$O2
```

———-

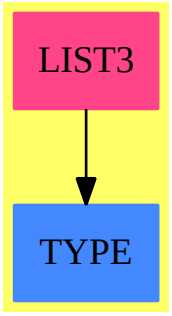
— **LIST2.dotabb** —

```
"LIST2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIST2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"LIST2" -> "FLAGG"
```

———-

13.17 package LIST3 ListFunctions3

13.17.1 ListFunctions3 (LIST3)



Exports:
map

— package LIST3 ListFunctions3 —

```
)abbrev package LIST3 ListFunctions3
++ Author:
++ Date Created:
++ Change History:
++ Basic Operations: map
++ Related Constructors: List
++ Also See: ListFunctions2
++ AMS Classification:
++ Keywords: list, aggregate, map
++ Description:
++ \spadtype{ListFunctions3} implements utility functions that
++ operate on three kinds of lists, each with a possibly different
++ type of element.

ListFunctions3(A:Type, B:Type, C:Type): public == private where
  LA     ==> List A
  LB     ==> List B
  LC     ==> List C

  public ==> with
    map: ( (A,B)->C, LA, LB) -> LC
      ++ map(fn,list1, u2) applies the binary function \spad{fn}
      ++ to corresponding elements of lists \spad{u1} and \spad{u2}
      ++ and returns a list of the results (in the same order). Thus
      ++ \spad{map(/,[1,2,3],[4,5,6]) = [1/4,2/4,1/2]}. The computation
      ++ terminates when the end of either list is reached. That is,
      ++ the length of the result list is equal to the minimum of the
      ++ lengths of \spad{u1} and \spad{u2}.

  private ==> add
    map(fn : (A,B) -> C, la : LA, lb : LB): LC ==
      empty?(la) or empty?(lb) => empty()$LC
      concat(fn(first la, first lb), map(fn, rest la, rest lb))
```

———-

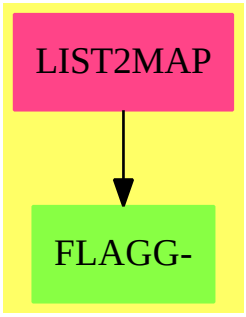
— LIST3.dotabb —

```
"LIST3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIST3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"LIST3" -> "TYPE"
```

———-

13.18 package LIST2MAP ListToMap

13.18.1 ListToMap (LIST2MAP)



Exports:
match

— package LIST2MAP ListToMap —

```
)abbrev package LIST2MAP ListToMap
++ Author: Manuel Bronstein
++ Date Created: 22 Mar 1988
++ Change History:
++   11 Oct 1989   MB   ?
++ Basic Operations: match
++ Related Constructors: List
++ Also See:
++ AMS Classification:
++ Keywords: mapping, list
++ Description:
++ \spadtype{ListToMap} allows mappings to be described by a pair of
++ lists of equal lengths.  The image of an element \spad{x},
++ which appears in position \spad{n} in the first list, is then
++ the \spad{n}th element of the second list.  A default value or
++ default function can be specified to be used when \spad{x}
++ does not appear in the first list.  In the absence of defaults,
++ an error will occur in that case.

ListToMap(A:SetCategory, B:Type): Exports == Implementation where
  LA  ==> List A
  LB  ==> List B
  AB  ==> (A -> B)
```

```
Exports ==> with
  match: (LA, LB   ) -> AB
    ++ match(la, lb) creates a map with no default source or target values
    ++ defined by lists la and lb of equal length.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
    ++ Note that when this map is applied, an error occurs when
    ++ applied to a value missing from la.
  match: (LA, LB, A) -> B
    ++ match(la, lb, a) creates a map
    ++ defined by lists la and lb of equal length, where \spad{a} is used
    ++ as the default source value if the given one is not in \spad{la}.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, B)    -> AB
    ++ match(la, lb, b) creates a map
    ++ defined by lists la and lb of equal length, where \spad{b} is used
    ++ as the default target value if the given function argument is
    ++ not in \spad{la}.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, A, B) -> B
    ++ match(la, lb, a, b) creates a map
    ++ defined by lists la and lb of equal length.
    ++ and applies this map to a.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Argument b is the default target value if a is not in la.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, AB)    -> AB
    ++ match(la, lb, f) creates a map
    ++ defined by lists la and lb of equal length.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Argument \spad{f} is used as the
    ++ function to call when the given function argument is not in
    ++ \spad{la}.
    ++ The value returned is f applied to that argument.
  match: (LA, LB, A, AB) -> B
    ++ match(la, lb, a, f) creates a map
    ++ defined by lists la and lb of equal length.
    ++ and applies this map to a.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Argument \spad{f} is a default function to call if a is not in la.
    ++ The value returned is then obtained by applying f to argument a.
```

```
  Implementation ==> add
    match(la, lb)             == (z1:A):B +-> match(la, lb, z1)
    match(la:LA, lb:LB, a:A)  == lb.position(a, la)
    match(la:LA, lb:LB, b:B)  == (z1:A):B +-> match(la, lb, z1, b)
    match(la:LA, lb:LB, f:AB) == (z1:A):B +-> match(la, lb, z1, f)

    match(la:LA, lb:LB, a:A, b:B) ==
      (p := position(a, la)) < minIndex(la) => b
      lb.p

    match(la:LA, lb:LB, a:A, f:AB) ==
      (p := position(a, la)) < minIndex(la) => f a
      lb.p
```

———-

— **LIST2MAP.dotabb** —

```
"LIST2MAP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIST2MAP"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"LIST2MAP" -> "FLAGG-"
```

———-

13.19 package LPARSPT LocalParametrizationOfSimplePointPackage

— **LocalParametrizationOfSimplePointPackage.input** —

```
)set break resume
)sys rm -f LocalParametrizationOfSimplePointPackage.output
)spool LocalParametrizationOfSimplePointPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LocalParametrizationOfSimplePointPackage
--R
--R LocalParametrizationOfSimplePointPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCat
--R Abbreviation for LocalParametrizationOfSimplePointPackage is LPARSPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LPARSPT
```

```
--R
--R------------------------------ Operations --------------------------------
--R pointDominateBy : Plc -> ProjPt       printInfo : Boolean -> Boolean
--R printInfo : () -> Boolean
--R localParamOfSimplePt : (ProjPt,PolyRing,Integer) -> List(PCS)
--R localize : (PolyRing,ProjPt,PolyRing,Integer) -> Record(fnc: PolyRing,crv: PolyRing,chart: List(Integer)
--R pointToPlace : (ProjPt,PolyRing) -> Plc
--R
--E 1

)spool
)lisp (bye)
```

———-

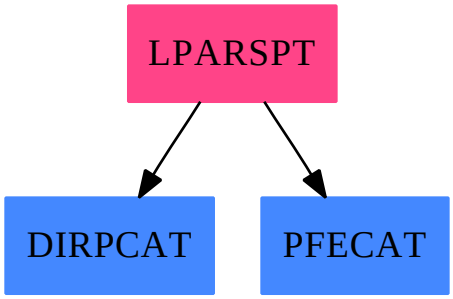
— **LocalParametrizationOfSimplePointPackage.help** —

```
=====================================================================
LocalParametrizationOfSimplePointPackage examples
=====================================================================

See Also:
o )show LocalParametrizationOfSimplePointPackage
```

———-

13.19.1 LocalParametrizationOfSimplePointPackage (LPARSPT)



Exports:
pointDominateBy printInfo localParamOfSimplePt localize pointToPlace

— **package LPARSPT LocalParametrizationOfSimplePointPackage** —

```
)abbreviation package LPARSPT LocalParametrizationOfSimplePointPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc):_
 Exports == Implementation where
  K:Field
  symb: List(Symbol)

  E : DirectProductCategory(#symb,NonNegativeInteger)
  OV  ==> OrderedVariableList(symb)

  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt   : ProjectiveSpaceCategory(K)
  PCS      : LocalPowerSeriesCategory(K)
  Plc      : PlacesCategory(K,PCS)

  PI       ==> PositiveInteger
  NNI      ==> NonNegativeInteger
  PPFC1    ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
  PackPoly ==> PackageForPoly(K,PolyRing,E,#symb)
  UP       ==> SparseUnivariatePolynomial(K)
  UPUP     ==> SparseUnivariatePolynomial(UP)

  Exports ==>  with

    printInfo: Boolean -> Boolean
      ++ printInfo(b) set a flag such that when true (b <- true) prints
      ++ some information during some critical computation.

    printInfo: () -> Boolean
      ++ returns the value of the \spad{printInfo} flag.

    pointToPlace: (ProjPt,PolyRing) -> Plc
      ++ pointToPlace(pt,pol) takes for input a simple point pt on the curve
      ++ defined by pol and set the local parametrization of the point.

    localParamOfSimplePt: (ProjPt,PolyRing,Integer) -> List PCS
      ++ localParamOfSimplePt(pt,pol,n) computes the local parametrization
      ++ of the simple point pt on the curve defined by pol. This local
      ++ parametrization is done according to the standard open affine
      ++ plane set by n

    pointDominateBy : Plc -> ProjPt
      ++ pointDominateBy(pl) returns the projective point dominated
      ++ by the place pl.

    localize:  (PolyRing,ProjPt,PolyRing,Integer) -> _
```

```
              Record(fnc:PolyRing,crv:PolyRing,chart:List(Integer))
    ++ localize(f,pt,crv,n) returns a record containing the polynomials f
    ++ and crv translate to the origin with respect to pt. The last
    ++ element of the records, consisting of three integers contains
    ++ information about the local parameter that will be used
    ++ (either x or y): the first integer correspond to the variable
    ++ that will be used as a local parameter.

Implementation ==>  add

  import PCS
  import PolyRing
  import PPFC1
  import PackPoly

  valuationAndMore: (UPUP,UPUP) -> _
                     Record(ord:Integer,value:K,fnc:UPUP,crv:UPUP)

  localize2: (PolyRing,ProjPt,PolyRing,Integer) -> _
             Record(fnc2:UPUP,crv2:UPUP)

  coerceToUPUP: (PolyRing,List Integer) -> UPUP

  paramAtOrigin: (UPUP,UPUP,Integer) -> PCS

  strictTransform: (UPUP,NNI) -> UPUP

  translate: (UPUP,K) -> UPUP

  constant: UPUP -> K

  intCoord: UPUP  -> K

  localMultiplicity: UPUP -> NNI

  mapDegree: (NNI,NNI,NNI) -> NNI

  listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]

  listMonoPols:List(PolyRing):=listVariable()

  pointDominateBy(pl)==
    lpl:List PCS:=localParam(pl)
    empty? lpl => _
     error "LPARSPT:pointDominateBy::parametrization of point not done yet"
    lK:List K:=[ findCoef(s,0) for s in lpl]
    projectivePoint(lK)

  localParamOfSimplePt(pt,curve,nV)==
    mult:NNI:=multiplicity(curve,pt,nV)
```

```
    ^one?(mult) => _
      error "The point is not simple or is not on the curve !"
    lcl:=[localize2(var,pt,curve,nV) for var in listMonoPols]
    [paramAtOrigin(l.fnc2,l.crv2,0) for l in lcl]

  pointToPlace(pt,curve)==
    -- define the chart for strictTransform (of simple point)
    nV:Integer:=lastNonNull pt
    pth:=homogenize(pt,nV)
    chart:List Integer:=[0,0,nV]
    mult:NNI:=multiplicity(curve,pth,nV)
    ^one?(mult) =>
      error "The point is not simple or is not on the curve"
    -- create a place from the simple point. This is done by giving
    -- a name to the place: in this case it is the coordinate of
    -- the projective point.
    lpth:List K:= pth :: List(K)
    plc:Plc:=create(lpth)$Plc
    ^empty?(localParam(plc)) => plc
    lcl:=[localize2(var,pth,curve,nV) for var in listMonoPols]
    lPar:=[paramAtOrigin(l.fnc2,l.crv2,0) for l in lcl]
    setParam!(plc,lPar)
    dd:=degree pth
    setDegree!(plc,dd)
    plc

  localVarForPrintInfo:Boolean:=false()$Boolean

  printInfo()==localVarForPrintInfo

  printInfo(flag)==localVarForPrintInfo:=flag

  mapDegree(n,mx,m)==
    dd:=(n+mx-m)
    dd < 0 => _
      error "LPARSPT:mapDegree called by PARAMP:strictTransform failed"
    dd  pretend NNI

  strictTransform(pol,m)==
    zero?(pol) => 0
    tc:=leadingCoefficient pol
    tk:= degree pol
    newTc:= mapExponents(mapDegree(#1,tk,m),tc)
    monomial(newTc,tk)$UPUP + strictTransform(reductum pol,m)

  Y == monomial(1,1)$UPUP

  trY: (K,NonNegativeInteger) -> UPUP
  trY(a,n)== (monomial(monomial(a,0)$UP,0)$UPUP + Y)**n
```

```
translate(pol,a)==
  zero?(pol) => 0
  tc:=leadingCoefficient pol
  tk:= degree pol
  trY(a,tk) * tc + translate(reductum pol, a)

constant(pol)==coefficient(coefficient(pol,0)$UPUP,0)$UP

intCoord(pol)==
  coefY:=coefficient(coefficient(pol,1)$UPUP,0)$UP
  cnst:=constant(pol)
  -cnst * inv coefY

localMultiplicity(pol)==
  zero?(pol) => error "Cannot compute the multiplicity for 0"
  redPol:= reductum pol
  tc:=leadingCoefficient pol
  tk:= degree pol
  m:=tk + minimumDegree(tc)$UP
  zero?(redPol) => m
  min( m, localMultiplicity(redPol))

coerceToUPUP(pol,chart)==
  zero?(pol) => 0
  lExp:=parts degree pol
  lCoef:=leadingCoefficient pol
  expX:=lExp(chart.1)
  expY:=lExp(chart.2)
  monomial(monomial(lCoef,expX)$UP,expY)$UPUP + _
    coerceToUPUP(reductum(pol),chart)

-- testing this function. See paramPack for original version.
valuationAndMore(f:UPUP,curve:UPUP)==
  -- this function evaluate the function f at the origin
  -- which must be a simple point on the curve define by "curve"
  val:= constant(f)
  ^zero?(val) => [0,val,f,curve]
  sTrCurve:=strictTransform(curve,1)
  slp:=intCoord sTrCurve
  multPtf:Integer:= localMultiplicity(f)  pretend Integer
  sTrFnc:=strictTransform(f,multPtf pretend NNI)
  newCurve:=translate(sTrCurve,slp)
  f2:=translate(sTrFnc,slp)
  val:= constant(f2)
  [multPtf, val, f2, newCurve]

paramAtOrigin(f:UPUP,curve:UPUP,ex:Integer)== delay
  -- this function must be
  -- called for parametrization a the origin
  u:=f
```

```
      zero?(u) => 0
      tt:=u exquo curve
      ^(tt case "failed") => 0
      firstTerm:=valuationAndMore(u,curve)
      od:=firstTerm.ord
      coef:=firstTerm.value
      newU:=firstTerm.fnc - monomial(monomial(coef,0)$UP,0)$UPUP
      newCurve:=firstTerm.crv
      series(od+ex,coef,paramAtOrigin(newU,newCurve,ex+od))

    localize(f:PolyRing,pt:ProjPt,curve:PolyRing,nV:Integer)==
      curveT:=translateToOrigin(curve,pt,nV)
      ft:=translateToOrigin(f,pt,nV)
      fm:=minimalForm(curveT)
      zero?(d:=totalDegree(fm)$PackPoly) => _
        error "the point is not on the curve"
      ^one?(d) => error "the point is singular"
      subChart:=[i for i in 1..#symb | ^(i= (nV pretend PI))]
      cf1:=degOneCoef(fm,(subChart.1) pretend PI)
      cf2:=degOneCoef(fm,(subChart.2) pretend PI)
      crt:List(Integer)
      sc:List(Integer):=[(i pretend Integer) for i in subChart]
      zero?(cf1) =>
        crt:=concat(sc,nV)
        [ft,curveT,crt]
      zero?(cf2) =>
        crt:=concat(reverse(sc),nV)
        [ft,curveT,crt]
      deg1:=degree(curveT,listVar(subChart.1))
      deg2:=degree(curveT,listVar(subChart.2))
      deg1 > deg2 =>
        crt:=concat(sc,nV)
        [ft,curveT,crt]
      crt:=concat(reverse(sc),nV)
      [ft,curveT,crt]

    localize2(f:PolyRing,pt:ProjPt,curve:PolyRing,nV:Integer)==
      recBlowUp:=localize(f,pt,curve,nV)
      f2:=coerceToUPUP(recBlowUp.fnc,recBlowUp.chart)
      curve2:=coerceToUPUP(recBlowUp.crv,recBlowUp.chart)
      [f2,curve2]
```

———-

— LPARSPT.dotabb —

```
"LPARSPT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LPARSPT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
```

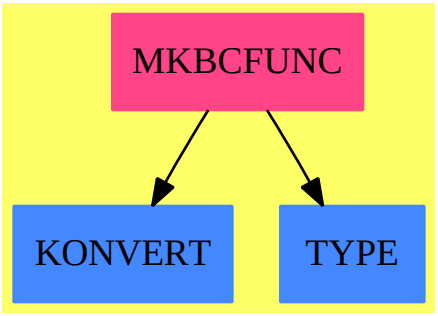
```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LPARSPT" -> "DIRPCAT"
"LPARSPT" -> "PFECAT"
```

Chapter 14

Chapter M

14.1 package MKBCFUNC MakeBinaryCompiledFunction

14.1.1 MakeBinaryCompiledFunction (MKBCFUNC)



Exports:
binaryFunction compiledFunction

— package MKBCFUNC MakeBinaryCompiledFunction —

```
)abbrev package MKBCFUNC MakeBinaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools and transforms for making compiled functions from
++ top-level expressions
```

```
MakeBinaryCompiledFunction(S, D1, D2, I):Exports == Implementation where
  S: ConvertibleTo InputForm
  D1, D2, I: Type

  SY  ==> Symbol
  DI  ==> devaluate((D1, D2) -> I)$Lisp

  Exports ==> with
    binaryFunction  : SY -> ((D1, D2) -> I)
      ++ binaryFunction(s) is a local function
    compiledFunction: (S, SY, SY) -> ((D1, D2) -> I)
      ++ compiledFunction(expr,x,y) returns a function \spad{f: (D1, D2) -> I}
      ++ defined by \spad{f(x, y) == expr}.
      ++ Function f is compiled and directly
      ++ applicable to objects of type \spad{(D1, D2)}

  Implementation ==> add
    import MakeFunction(S)

    func: (SY, D1, D2) -> I

    func(name, x, y)   == FUNCALL(name, x, y, NIL$Lisp)$Lisp
    binaryFunction name == (d1:D1,d2:D2):I +-> func(name, d1, d2)

    compiledFunction(e, x, y) ==
      t := [devaluate(D1)$Lisp, devaluate(D2)$Lisp]$List(InputForm)
      binaryFunction compile(function(e, declare DI, x, y), t)
```

———-

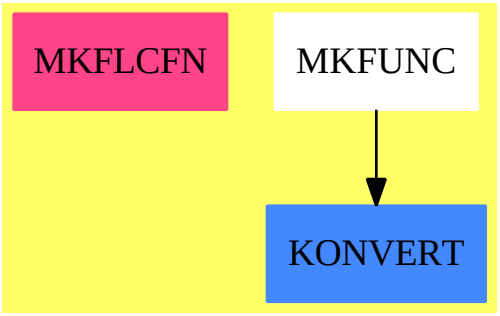
— MKBCFUNC.dotabb —

```
"MKBCFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKBCFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKBCFUNC" -> "KONVERT"
"MKBCFUNC" -> "TYPE"
```

———-

14.2 package MKFLCFN MakeFloatCompiledFunction

14.2.1 MakeFloatCompiledFunction (MKFLCFN)



Exports:

— package MKFLCFN MakeFloatCompiledFunction —

```
)abbrev package MKFLCFN MakeFloatCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 2 Mar 1990
++ Date Last Updated: 2 Dec 1996 (MCD)
++ Description:
++ Tools for making compiled functions from top-level expressions
++ MakeFloatCompiledFunction transforms top-level objects into
++ compiled Lisp functions whose arguments are Lisp floats.
++ This by-passes the \Language{} compiler and interpreter,
++ thereby gaining several orders of magnitude.

MakeFloatCompiledFunction(S): Exports == Implementation where
  S: ConvertibleTo InputForm

  INF ==> InputForm
  SF  ==> DoubleFloat
  DI1 ==> devaluate(SF -> SF)$Lisp
  DI2 ==> devaluate((SF, SF) -> SF)$Lisp

  Exports ==> with
    makeFloatFunction: (S, Symbol)         -> (SF -> SF)
      ++ makeFloatFunction(expr, x) returns a Lisp function
      ++ \spad{f: \axiomType{DoubleFloat} -> \axiomType{DoubleFloat}}
      ++ defined by \spad{f(x) == expr}.
      ++ Function f is compiled and directly
      ++ applicable to objects of type \axiomType{DoubleFloat}.
    makeFloatFunction: (S, Symbol, Symbol) -> ((SF, SF) -> SF)
```

```
    ++ makeFloatFunction(expr, x, y) returns a Lisp function
    ++ \spad{f: (\axiomType{DoubleFloat},
    ++ \axiomType{DoubleFloat}) -> \axiomType{DoubleFloat}}
    ++ defined by \spad{f(x, y) == expr}.
    ++ Function f is compiled and directly
    ++ applicable to objects of type \spad{(\axiomType{DoubleFloat},
    ++ \axiomType{DoubleFloat})}.

  Implementation ==> add
    import MakeUnaryCompiledFunction(S, SF, SF)
    import MakeBinaryCompiledFunction(S, SF, SF, SF)

    streq?    : (INF, String) -> Boolean
    streqlist?: (INF, List String) -> Boolean
    gencode   : (String, List INF) -> INF
    mkLisp    : INF -> Union(INF, "failed")
    mkLispList: List INF -> Union(List INF, "failed")
    mkDefun   : (INF, List INF) -> INF
    mkLispCall: INF -> INF
    mkPretend : INF -> INF
    mkCTOR : INF -> INF

    lsf := convert([convert("DoubleFloat"::Symbol)@INF]$List(INF))@INF

    streq?(s, st)    == s = convert(st::Symbol)@INF
    gencode(s, l)    == convert(concat(convert(s::Symbol)@INF, l))@INF
    streqlist?(s, l) == member?(string symbol s, l)

    mkPretend form ==
      convert([convert("pretend"::Symbol), form, lsf]$List(INF))@INF

    mkCTOR form ==
      convert([convert("C-TO-R"::Symbol), form]$List(INF))@INF


    mkLispCall name ==
      convert([convert("$elt"::Symbol),
                          convert("Lisp"::Symbol), name]$List(INF))@INF

    mkDefun(s, lv) ==
      name := convert(new()$Symbol)@INF
      fun  := convert([convert("DEFUN"::Symbol), name, convert lv,
              gencode("DECLARE",[gencode("FLOAT",lv)]),mkCTOR s]$List(INF))@INF
      EVAL(fun)$Lisp
      if _$compileDontDefineFunctions$Lisp then COMPILE(name)$Lisp
      name

    makeFloatFunction(f, x, y) ==
      (u := mkLisp(convert(f)@INF)) case "failed" =>
        compiledFunction(f, x, y)
```

```
  name := mkDefun(u::INF, [ix := convert x, iy := convert y])
  t    := [lsf, lsf]$List(INF)
  spadname := declare DI2
  spadform:=mkPretend convert([mkLispCall name,ix,iy]$List(INF))@INF
  interpret function(spadform, [x, y], spadname)
  binaryFunction compile(spadname, t)

makeFloatFunction(f, var) ==
  (u := mkLisp(convert(f)@INF)) case "failed" =>
    compiledFunction(f, var)
  name := mkDefun(u::INF, [ivar := convert var])
  t    := [lsf]$List(INF)
  spadname := declare DI1
  spadform:= mkPretend convert([mkLispCall name,ivar]$List(INF))@INF
  interpret function(spadform, [var], spadname)
  unaryFunction compile(spadname, t)

mkLispList l ==
  ans := nil()$List(INF)
  for s in l repeat
    (u := mkLisp s) case "failed" => return "failed"
    ans := concat(u::INF, ans)
  reverse_! ans


mkLisp s ==
  atom? s => s
  op := first(l := destruct s)
  (u := mkLispList rest l) case "failed" => "failed"
  ll := u::List(INF)
  streqlist?(op, ["+","*","/","-"]) => convert(concat(op, ll))@INF
  streq?(op, "**") => gencode("EXPT", ll)
  streqlist?(op, ["exp","sin","cos","tan","atan",
     "log", "sinh","cosh","tanh","asinh","acosh","atanh","sqrt"]) =>
        gencode(upperCase string symbol op, ll)
  streq?(op, "nthRoot") =>
    second ll = convert(2::Integer)@INF =>gencode("SQRT",[first ll])
    gencode("EXPT", concat(first ll, [1$INF / second ll]))
  streq?(op, "float") =>
    a := ll.1
    e := ll.2
    b := ll.3
    _*(a, EXPT(b, e)$Lisp)$Lisp pretend INF
  "failed"
```

———-

— **MKFLCFN.dotabb** —

```
"MKFLCFN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKFLCFN"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"MKFUNC" -> "KONVERT"
```

———-

14.3 package MKFUNC MakeFunction

— **MakeFunction.input** —

```
)set break resume
)spool MakeFunction.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
expr := (x - exp x + 1)^2 * (sin(x^2) * x + 1)^3
--R
--R
--R   (1)
--R       3   x 2        4     3   x    5     4    3      2 3
--R     (x (%e )  + (- 2x  - 2x )%e  + x  + 2x  + x )sin(x )
--R   +
--R        2   x 2        3     2   x     4     3     2      2 2
--R     (3x (%e )  + (- 6x  - 6x )%e  + 3x  + 6x  + 3x )sin(x )
--R   +
--R            x 2        2        x     3     2           2       x 2
--R     (3x (%e )  + (- 6x  - 6x)%e  + 3x  + 6x  + 3x)sin(x ) + (%e )
--R   +
--R                  x    2
--R     (- 2x - 2)%e  + x  + 2x + 1
--R                                                    Type: Expression(Integer)
--E 1

--S 2 of 9
function(expr, f, x)
--R
--R
--R   (2)  f
--R                                                                 Type: Symbol
--E 2

--S 3 of 9
tbl := [f(0.1 * i - 1) for i in 0..20]
--R
--R   Compiling function f with type Float -> Float
```

```
--R
--R   (3)
--R   [0.0005391844 0362701574, 0.0039657551 1844206653,
--R    0.0088545187 4833983689 2, 0.0116524883 0907069695,
--R    0.0108618220 9245751364 5, 0.0076366823 2120869965 06,
--R    0.0040584985 7597822062 55, 0.0015349542 8910500836 48,
--R    0.0003424903 1549879905 716, 0.0000233304 8276098819 6001, 0.0,
--R    0.0000268186 8782862599 4229, 0.0004691571 3720051642 621,
--R    0.0026924576 5968519586 08, 0.0101486881 7369135148 8,
--R    0.0313833725 8543810564 3, 0.0876991144 5154615297 9,
--R    0.2313019789 3439968362, 0.5843743955 958098772, 1.4114930171 992819197,
--R    3.2216948276 75164252]
--R                                                        Type: List(Float)
--E 3

--S 4 of 9
e := (x - y + 1)^2 * (x^2 * y + 1)^2
--R
--R
--R   (4)
--R      4 4        5     4     2  3     6     5    4     3     2      2
--R     x y  + (- 2x  - 2x  + 2x )y  + (x  + 2x  + x  - 4x  - 4x  + 1)y
--R   +
--R        4     3     2               2
--R     (2x  + 4x  + 2x  - 2x - 2)y + x  + 2x + 1
--R                                                 Type: Polynomial(Integer)
--E 4

--S 5 of 9
function(e, g, [x, y])
--R
--R
--R   (5)  g
--R                                                              Type: Symbol
--E 5

--S 6 of 9
function(e, h, x, y)
--R
--R
--R   (6)  h
--R                                                              Type: Symbol
--E 6

--S 7 of 9
m1 := squareMatrix [ [1, 2], [3, 4] ]
--R
--R
--R        +1  2+
--R   (7)  |    |
```

```
--R        +3  4+
--R                                                Type: SquareMatrix(2,Integer)
--E 7

--S 8 of 9
m2 := squareMatrix [ [1, 0], [-1, 1] ]
--R
--R
--R        + 1   0+
--R   (8)  |      |
--R        +- 1  1+
--R                                                Type: SquareMatrix(2,Integer)
--E 8

--S 9 of 9
h(m1, m2)
--R
--R   Compiling function h with type (SquareMatrix(2,Integer),SquareMatrix
--R      (2,Integer)) -> SquareMatrix(2,Integer)
--R
--R        +- 7836   8960 +
--R   (9)  |              |
--R        +- 17132  19588+
--R                                                Type: SquareMatrix(2,Integer)
--E 9
)spool
)lisp (bye)
```

———-

— MakeFunction.help —

```
=====================================================================
MakeFunction examples
=====================================================================

It is sometimes useful to be able to define a function given by
the result of a calculation.

Suppose that you have obtained the following expression after several
computations and that you now want to tabulate the numerical values of
f for x between -1 and +1 with increment 0.1.

  expr := (x - exp x + 1)^2 * (sin(x^2) * x + 1)^3
       3   x 2        4     3   x    5     4    3      2 3
     (x (%e )  + (- 2x  - 2x )%e  + x  + 2x  + x )sin(x )
   +
        2   x 2        3     2   x     4     3     2      2 2
     (3x (%e )  + (- 6x  - 6x )%e  + 3x  + 6x  + 3x )sin(x )
```

```
  +
          x 2        2        x     3     2           2       x 2
    (3x (%e )  + (- 6x  - 6x)%e  + 3x  + 6x  + 3x)sin(x ) + (%e )
  +
                x    2
    (- 2x - 2)%e  + x  + 2x + 1
                              Type: Expression Integer
```

You could, of course, use the function eval within a loop and evaluate expr twenty-one times, but this would be quite slow. A better way is to create a numerical function f such that f(x) is defined by the expression expr above, but without retyping expr! The package MakeFunction provides the operation function which does exactly this.

Issue this to create the function f(x) given by expr.

```
  function(expr, f, x)
    f
                              Type: Symbol
```

To tabulate expr, we can now quickly evaluate f 21 times.

```
  tbl := [f(0.1 * i - 1) for i in 0..20];
                              Type: List Float
```

Use the list [x1,...,xn] as the third argument to function to create a multivariate function f(x1,...,xn).

```
  e := (x - y + 1)^2 * (x^2 * y + 1)^2
      4 4        5     4     2  3     6     5    4     3     2      2
     x y  + (- 2x  - 2x  + 2x )y  + (x  + 2x  + x  - 4x  - 4x  + 1)y
   +
        4     3     2               2
     (2x  + 4x  + 2x  - 2x - 2)y + x  + 2x + 1
                               Type: Polynomial Integer

  function(e, g, [x, y])
    g
                              Type: Symbol
```

In the case of just two variables, they can be given as arguments without making them into a list.

```
  function(e, h, x, y)
    h
                              Type: Symbol
```

Note that the functions created by function are not limited to floating point numbers, but can be applied to any type for which they are defined.

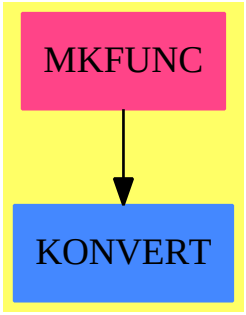
```
  m1 := squareMatrix [ [1, 2], [3, 4] ]
    +1  2+
    |    |
    +3  4+
                                Type: SquareMatrix(2,Integer)

  m2 := squareMatrix [ [1, 0], [-1, 1] ]
    + 1   0+
    |      |
    +- 1  1+
                                Type: SquareMatrix(2,Integer)

  h(m1, m2)
    +- 7836   8960 +
    |              |
    +- 17132  19588+
                                Type: SquareMatrix(2,Integer)

See Also:
o )show MakeFunction
```

14.3.1 MakeFunction (MKFUNC)



Exports:
function

— package MKFUNC MakeFunction —

```
)abbrev package MKFUNC MakeFunction
++ Author: Manuel Bronstein
```

```
++ Date Created: 22 Nov 1988
++ Date Last Updated: 8 Jan 1990
++ Description:
++ Tools for making interpreter functions from top-level expressions
++ Transforms top-level objects into interpreter functions.

MakeFunction(S:ConvertibleTo InputForm): Exports == Implementation where
  SY ==> Symbol

  Exports ==> with
    function: (S, SY         ) -> SY
      ++ function(e, foo) creates a function \spad{foo() == e}.
    function: (S, SY,      SY) -> SY
      ++ function(e, foo, x) creates a function \spad{foo(x) == e}.
    function: (S, SY, SY,  SY) -> SY
      ++ function(e, foo, x, y) creates a function \spad{foo(x, y) = e}.
    function: (S, SY, List SY) -> SY
      ++ \spad{function(e, foo, [x1,...,xn])} creates a function
      ++ \spad{foo(x1,...,xn) == e}.

  Implementation ==> add
    function(s, name)            == function(s, name, nil())
    function(s:S, name:SY, x:SY) == function(s, name, [x])
    function(s, name, x, y)      == function(s, name, [x, y])

    function(s:S, name:SY, args:List SY) ==
      interpret function(convert s, args, name)$InputForm
      name
```

———-

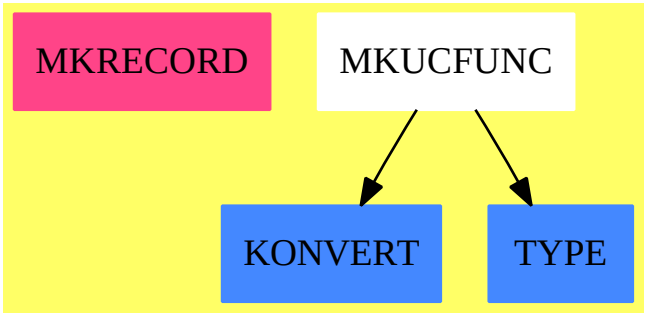
— MKFUNC.dotabb —

```
"MKFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"MKFUNC" -> "KONVERT"
```

———-

14.4 package MKRECORD MakeRecord

14.4.1 MakeRecord (MKRECORD)



Exports:
makeRecord

— **package MKRECORD MakeRecord** —

```
)abbrev package MKRECORD MakeRecord
++ Description:
++ MakeRecord is used internally by the interpreter to create record
++ types which are used for doing parallel iterations on streams.

MakeRecord(S: Type, T: Type): public == private where
  public == with
    makeRecord: (S,T) -> Record(part1: S, part2: T)
      ++ makeRecord(a,b) creates a record object with type
      ++ Record(part1:S, part2:R),
      ++ where part1 is \spad{a} and part2 is \spad{b}.
  private == add
    makeRecord(s: S, t: T)  ==
      [s,t]$Record(part1: S, part2: T)
```

———-

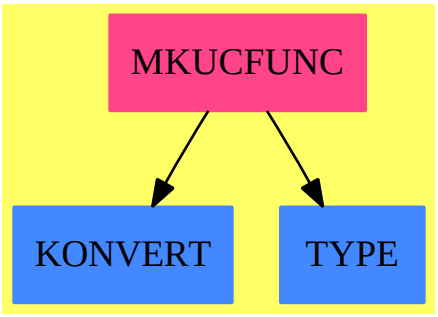
— **MKRECORD.dotabb** —

```
"MKRECORD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKRECORD"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKUCFUNC" -> "KONVERT"
"MKUCFUNC" -> "TYPE"
```

———-

14.5 package MKUCFUNC MakeUnaryCompiledFunction

14.5.1 MakeUnaryCompiledFunction (MKUCFUNC)



Exports:
unaryFunction compiledFunction

— **package MKUCFUNC MakeUnaryCompiledFunction** —

```
)abbrev package MKUCFUNC MakeUnaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools for making compiled functions from top-level expressions
++ Transforms top-level objects into compiled functions.

MakeUnaryCompiledFunction(S, D, I): Exports == Implementation where
  S: ConvertibleTo InputForm
  D, I: Type

  SY  ==> Symbol
  DI  ==> devaluate(D -> I)$Lisp

  Exports ==> with
    unaryFunction   : SY -> (D -> I)
      ++ unaryFunction(a) is a local function
    compiledFunction: (S, SY) -> (D -> I)
      ++ compiledFunction(expr, x) returns a function \spad{f: D -> I}
      ++ defined by \spad{f(x) == expr}.
      ++ Function f is compiled and directly
```

```
      ++ applicable to objects of type D.

  Implementation ==> add
    import MakeFunction(S)

    func: (SY, D) -> I

    func(name, x)       == FUNCALL(name, x, NIL$Lisp)$Lisp
    unaryFunction name  == (d1:D):I +-> func(name, d1)

    compiledFunction(e:S, x:SY) ==
      t := [convert([devaluate(D)$Lisp]$List(InputForm))
           ]$List(InputForm)
      unaryFunction compile(function(e, declare DI, x), t)
```

———-

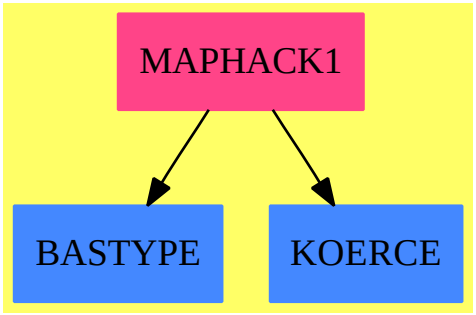
— **MKUCFUNC.dotabb** —

```
"MKUCFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKUCFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKUCFUNC" -> "KONVERT"
"MKUCFUNC" -> "TYPE"
```

———-

14.6 package MAPHACK1 MappingPackageInternalHacks1

14.6.1 MappingPackageInternalHacks1 (MAPHACK1)



Exports:
iter recur

— package MAPHACK1 MappingPackageInternalHacks1 —

```
)abbrev package MAPHACK1 MappingPackageInternalHacks1
++ Author: S.M.Watt and W.H.Burge
++ Date Created:Jan 87
++ Date Last Updated:Feb 92
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks1(A: SetCategory): MPcat == MPdef where
    NNI ==> NonNegativeInteger

    MPcat == with
        iter:  ((A -> A), NNI, A) -> A
          ++\spad{iter(f,n,x)} applies \spad{f n} times to \spad{x}.
        recur: ((NNI, A)->A, NNI, A) -> A
          ++\spad{recur(n,g,x)} is \spad{g(n,g(n-1,..g(1,x)..))}.

    MPdef == add
        iter(g,n,x)  ==
            for i in 1..n repeat x := g x     -- g(g(..(x)..))
            x
        recur(g,n,x) ==
            for i in 1..n repeat x := g(i,x)  -- g(n,g(n-1,..g(1,x)..))
            x
```

———-

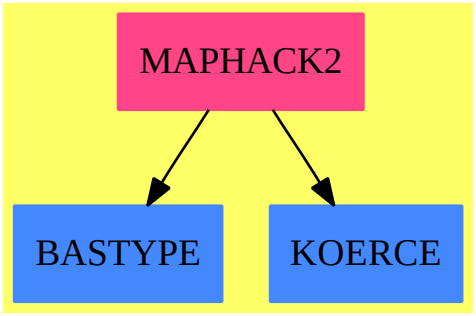
— MAPHACK1.dotabb —

```
"MAPHACK1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPHACK1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPHACK1" -> "BASTYPE"
"MAPHACK1" -> "KOERCE"
```

———-

14.7 package MAPHACK2 MappingPackageInternalHacks2

14.7.1 MappingPackageInternalHacks2 (MAPHACK2)



Exports:
arg1 arg2

— package MAPHACK2 MappingPackageInternalHacks2 —

```
)abbrev package MAPHACK2 MappingPackageInternalHacks2
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks2(A: SetCategory, C: SetCategory):_
  MPcat == MPdef where
    NNI ==> NonNegativeInteger

    MPcat == with
        arg1:  (A, C) -> A
          ++\spad{arg1(a,c)} selects its first argument.
        arg2:  (A, C) -> C
          ++\spad{arg2(a,c)} selects its second argument.

    MPdef == add
        arg1(a, c)   == a
        arg2(a, c)   == c
```

———-

— MAPHACK2.dotabb —

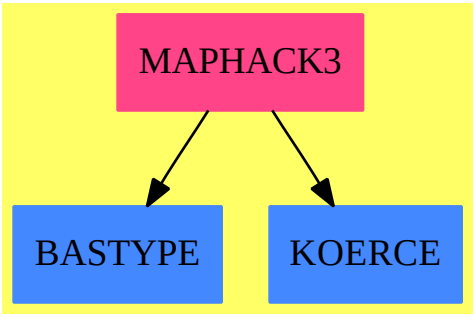
```
"MAPHACK2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPHACK2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
```

```
"MAPHACK2" -> "BASTYPE"
"MAPHACK2" -> "KOERCE"
```

———-

14.8 package MAPHACK3 MappingPackageInternalHacks3

14.8.1 MappingPackageInternalHacks3 (MAPHACK3)



Exports:
comp

— package MAPHACK3 MappingPackageInternalHacks3 —

```
)abbrev package MAPHACK3 MappingPackageInternalHacks3
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks3(A: SetCategory, B: SetCategory, C: SetCategory):_
  MPcat == MPdef where
    NNI ==> NonNegativeInteger

    MPcat == with
        comp:  (B->C, A->B, A) -> C
          ++\spad{comp(f,g,x)} is \spad{f(g x)}.

    MPdef == add
        comp(g,h,x)  == g h x
```

———-

— MAPHACK3.dotabb —

```
"MAPHACK3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPHACK3"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPHACK3" -> "BASTYPE"
"MAPHACK3" -> "KOERCE"
```

———-

14.9 package MAPPKG1 MappingPackage1

— MappingPackage1.input —

```
)set break resume
)spool MappingPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R   Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R      (Integer) has been added to workspace.
--R                                                                   Type: Void
--E 1

--S 2 of 26
power(2,3)
--R
--R   Compiling function power with type (Fraction(Integer),Integer) ->
--R      Fraction(Integer)
--R
--R   (2)  8
--R                                                  Type: Fraction(Integer)
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R   (3)  theMap(MAPPKG3;twist;MM;5!0)
--R                    Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
```

```
--R
--R
--R   (4)  8
--R                                                    Type: Fraction(Integer)
--E 4

--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R                                                                 Type: Void
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R   (6)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                              Type: (Fraction(Integer) -> Fraction(Integer))
--E 6

--S 7 of 26
square 4
--R
--R
--R   (7)  16
--R                                                    Type: Fraction(Integer)
--E 7

--S 8 of 26
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R   (8)  theMap(MAPPKG3;constantRight;MM;3!0)
--R           Type: ((Fraction(Integer),Fraction(Integer)) -> Fraction(Integer))
--E 8

--S 9 of 26
squirrel(1/2, 1/3)
--R
--R
--R        1
--R   (9)  -
--R        4
--R                                                    Type: Fraction(Integer)
--E 9

--S 10 of 26
sixteen := curry(square, 4/1)
--R
--R
```

```
--R   (10)  theMap(MAPPKG2;curry;MAM;2!0,0)
--R                                                Type: (() -> Fraction(Integer))
--E 10

--S 11 of 26
sixteen()
--R
--R
--R   (11)  16
--R                                                          Type: Fraction(Integer)
--E 11

--S 12 of 26
square2:=square*square
--R
--R
--R   (12)  theMap(MAPPKG3;*;MMM;6!0,0)
--R                                Type: (Fraction(Integer) -> Fraction(Integer))
--E 12

--S 13 of 26
square2 3
--R
--R
--R   (13)  81
--R                                                          Type: Fraction(Integer)
--E 13

--S 14 of 26
sc(x: FRAC INT): FRAC INT == x + 1
--R
--R   Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
--R      been added to workspace.
--R                                                                       Type: Void
--E 14

--S 15 of 26
incfns := [sc**i for i in 0..10]
--R
--R   Compiling function sc with type Fraction(Integer) -> Fraction(
--R      Integer)
--R
--R   (15)
--R   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0)]
--R                          Type: List((Fraction(Integer) -> Fraction(Integer)))
```

```
--E 15

--S 16 of 26
[f 4 for f in incfns]
--R
--R
--R   (16)  [4,5,6,7,8,9,10,11,12,13,14]
--R                                                Type: List(Fraction(Integer))
--E 16

--S 17 of 26
times(n:NNI, i:INT):INT == n*i
--R
--R   Function declaration times : (NonNegativeInteger,Integer) -> Integer
--R      has been added to workspace.
--R                                                                       Type: Void
--E 17

--S 18 of 26
r := recur(times)
--R
--R   Compiling function times with type (NonNegativeInteger,Integer) ->
--R      Integer
--R
--R   (18)  theMap(MAPPKG1;recur;2M;7!0,0)
--R                              Type: ((NonNegativeInteger,Integer) -> Integer)
--E 18

--S 19 of 26
fact := curryRight(r, 1)
--R
--R
--R   (19)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                                       Type: (NonNegativeInteger -> Integer)
--E 19

--S 20 of 26
fact 4
--R
--R
--R   (20)  24
--R                                                    Type: PositiveInteger
--E 20

--S 21 of 26
mto2ton(m, n) ==
  raiser := square^n
  raiser m
--R
--R                                                                       Type: Void
```

```
--E 21

--S 22 of 26
mto2ton(3, 3)
--R
--R   Compiling function mto2ton with type (PositiveInteger,
--R      PositiveInteger) -> Fraction(Integer)
--R
--R   (22)  6561
--R                                                    Type: Fraction(Integer)
--E 22

--S 23 of 26
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R
--R   Function declaration shiftfib : List(Integer) -> Integer has been
--R      added to workspace.
--R                                                                 Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R
--R   (24)  [0,1]
--R                                                      Type: List(Integer)
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
--R   Compiling function shiftfib with type List(Integer) -> Integer
--R
--R   (25)  theMap(MAPPKG2;curry;MAM;2!0,0)
--R                                                    Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R
--R   (26)
--R   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R    317811, 514229, 832040]
--R                                                      Type: List(Integer)
```

```
--E 26

)spool
)lisp (bye)
```

———-

— MappingPackage1.help —

```
========================================================================
MappingPackage examples
========================================================================

Function are objects of type Mapping.  In this section we demonstrate
some library operations from the packages MappingPackage1, MappingPackage2,
and MappingPackage3 that manipulate and create functions. Some terminology:
a nullary function takes no arguments, a unary function takes one argument,
and a binary function takes two arguments.

We begin by creating an example function that raises a rational number
to an integer exponent.

  power(q: FRAC INT, n: INT): FRAC INT == q**n
                      Type: Void

  power(2,3)
    8
                       Type: Fraction Integer

The twist operation transposes the arguments of a binary function.
Here rewop(a, b) is power(b, a).

  rewop := twist power
    theMap(MAPPKG3;twist;MM;5!0)
                       Type: ((Integer,Fraction Integer) -> Fraction Integer)

This is 2^3.

  rewop(3, 2)
    8
                       Type: Fraction Integer

Now we define square in terms of power.

  square: FRAC INT -> FRAC INT
                       Type: Void
The curryRight operation creates a unary function from a binary one by
providing a constant argument on the right.
```

```
  square:= curryRight(power, 2)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
                        Type: (Fraction Integer -> Fraction Integer)
```

Likewise, the curryLeft operation provides a constant argument on the left.

```
  square 4
    16
                        Type: Fraction Integer
```

The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that g(a,b)= f(a).

```
  squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
    theMap(MAPPKG3;constantRight;MM;3!0)
              Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)
```

Likewise, constantLeft(f) is the function g such that g(a,b)= f(b).

```
  squirrel(1/2, 1/3)
    1
    -
    4
                        Type: Fraction Integer
```

The curry operation makes a unary function nullary.

```
  sixteen := curry(square, 4/1)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                        Type: (() -> Fraction Integer)

  sixteen()
    16
                        Type: Fraction Integer
```

The * operation constructs composed functions.

```
  square2:=square*square
    theMap(MAPPKG3;*;MMM;6!0,0)
                        Type: (Fraction Integer -> Fraction Integer)

  square2 3
    81
                        Type: Fraction Integer
```

Use the ** operation to create functions that are n-fold iterations of other functions.

```
  sc(x: FRAC INT): FRAC INT == x + 1
                        Type: Void

This is a list of Mapping objects.

  incfns := [sc**i for i in 0..10]
   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0)]
                        Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

  [f 4 for f in incfns]
    [4,5,6,7,8,9,10,11,12,13,14]
                        Type: List Fraction Integer

Use the recur operation for recursion:

  g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).

  times(n:NNI, i:INT):INT == n*i
                        Type: Void

  r := recur(times)
    theMap(MAPPKG1;recur;2M;7!0,0)
                        Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

  fact := curryRight(r, 1)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
                        Type: (NonNegativeInteger -> Integer)

  fact 4
    24
                        Type: PositiveInteger

Constructed functions can be used within other functions.

  mto2ton(m, n) ==
    raiser := square^n
    raiser m
                        Type: Void

This is 3^(2^3).
```

```
  mto2ton(3, 3)
    6561
                        Type: Fraction Integer
```

Here shiftfib is a unary function that modifies its argument.

```
  shiftfib(r: List INT) : INT ==
    t := r.1
    r.1 := r.2
    r.2 := r.2 + t
    t
                        Type: Void
```

By currying over the argument we get a function with private state.

```
  fibinit: List INT := [0, 1]
    [0,1]
                        Type: List Integer

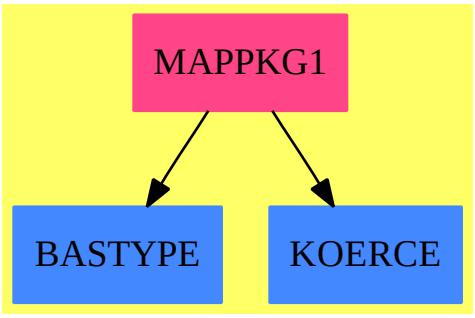
  fibs := curry(shiftfib, fibinit)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                        Type: (() -> Integer)

  [fibs() for i in 0..30]
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
    317811, 514229, 832040]
                        Type: List Integer
```

See Also:
o)show MappingPackage1
o)help MappingPackage2
o)help MappingPackage3
o)help MappingPackage4

———————

14.9.1 MappingPackage1 (MAPPKG1)



Exports:
coerce fixedPoint id nullary recur ?**?

— **package MAPPKG1 MappingPackage1** —

```
)abbrev package MAPPKG1 MappingPackage1
++ Author: S.M.Watt and W.H.Burge
++ Date Created:Jan 87
++ Date Last Updated:Feb 92
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Various Currying operations.

MappingPackage1(A:SetCategory): MPcat == MPdef where
    NNI   ==>  NonNegativeInteger

    MPcat ==  with
        nullary: A           -> (()->A)
          ++\spad{nullary A} changes its argument into a
          ++ nullary function.
        coerce:  A           -> (()->A)
          ++\spad{coerce A} changes its argument into a
          ++ nullary function.

        fixedPoint: (A->A) -> A
          ++\spad{fixedPoint f} is the fixed point of function \spad{f}.
          ++ i.e. such that \spad{fixedPoint f = f(fixedPoint f)}.
        fixedPoint: (List A->List A, Integer) -> List A
          ++\spad{fixedPoint(f,n)} is the fixed point of function
```

```
      ++ \spad{f} which is assumed to transform a list of length
      ++ \spad{n}.


    id:    A -> A
      ++\spad{id x} is \spad{x}.
    "**":  (A->A, NNI)  -> (A->A)
      ++\spad{f**n} is the  function which is the n-fold application
      ++ of \spad{f}.

    recur: ((NNI, A)->A) -> ((NNI, A)->A)
      ++\spad{recur(g)} is the function \spad{h} such that
      ++ \spad{h(n,x)= g(n,g(n-1,..g(1,x)..))}.


MPdef == add

    MappingPackageInternalHacks1(A)

    a: A
    faa:  A -> A
    f0a:  ()-> A

    nullary a   == a
    coerce  a   == nullary a
    fixedPoint faa ==
        g0 := GENSYM()$Lisp
        g1 := faa g0
        EQ(g0, g1)$Lisp => error "All points are fixed points"
        GEQNSUBSTLIST([g0]$Lisp, [g1]$Lisp, g1)$Lisp

    fixedPoint(fll, n) ==
        g0 := [(GENSYM()$Lisp):A for i in 1..n]
        g1 := fll g0
        or/[EQ(e0,e1)$Lisp for e0 in g0 for e1 in g1] =>
            error "All points are fixed points"
        GEQNSUBSTLIST(g0, g1, g1)$Lisp

    -- Composition and recursion.
    id a        == a
    g**n        == (a1:A):A +-> iter(g, n, a1)

    recur fnaa  == (n1:NNI,a2:A):A +-> recur(fnaa, n1, a2)
```

———-

— MAPPKG1.dotabb —

```
"MAPPKG1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPPKG1" -> "BASTYPE"
"MAPPKG1" -> "KOERCE"
```

———-

14.10 package MAPPKG2 MappingPackage2

— **MappingPackage2.input** —

```
)set break resume
)spool MappingPackage2.output
)set message test on
)set message auto off
)clear all
--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R   Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R      (Integer) has been added to workspace.
--R                                                                    Type: Void
--E 1

--S 2 of 26
power(2,3)
--R
--R   Compiling function power with type (Fraction(Integer),Integer) ->
--R      Fraction(Integer)
--R
--R   (2)  8
--R                                                  Type: Fraction(Integer)
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R   (3)  theMap(MAPPKG3;twist;MM;5!0)
--R                    Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
--R
```

```
--R
--R   (4)  8
--R                                                    Type: Fraction(Integer)
--E 4

--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R                                                                 Type: Void
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R   (6)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                              Type: (Fraction(Integer) -> Fraction(Integer))
--E 6

--S 7 of 26
square 4
--R
--R
--R   (7)  16
--R                                                    Type: Fraction(Integer)
--E 7

--S 8 of 26
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R   (8)  theMap(MAPPKG3;constantRight;MM;3!0)
--R           Type: ((Fraction(Integer),Fraction(Integer)) -> Fraction(Integer))
--E 8

--S 9 of 26
squirrel(1/2, 1/3)
--R
--R
--R        1
--R   (9)  -
--R        4
--R                                                    Type: Fraction(Integer)
--E 9

--S 10 of 26
sixteen := curry(square, 4/1)
--R
--R
--R   (10)  theMap(MAPPKG2;curry;MAM;2!0,0)
```

```
--R                                                Type: (() -> Fraction(Integer))
--E 10

--S 11 of 26
sixteen()
--R
--R
--R   (11)  16
--R                                                          Type: Fraction(Integer)
--E 11

--S 12 of 26
square2:=square*square
--R
--R
--R   (12)  theMap(MAPPKG3;*;MMM;6!0,0)
--R                                Type: (Fraction(Integer) -> Fraction(Integer))
--E 12

--S 13 of 26
square2 3
--R
--R
--R   (13)  81
--R                                                          Type: Fraction(Integer)
--E 13

--S 14 of 26
sc(x: FRAC INT): FRAC INT == x + 1
--R
--R   Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
--R      been added to workspace.
--R                                                                       Type: Void
--E 14

--S 15 of 26
incfns := [sc**i for i in 0..10]
--R
--R   Compiling function sc with type Fraction(Integer) -> Fraction(
--R      Integer)
--R
--R   (15)
--R   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0)]
--R                           Type: List((Fraction(Integer) -> Fraction(Integer)))
--E 15
```

```
--S 16 of 26
[f 4 for f in incfns]
--R
--R
--R   (16)  [4,5,6,7,8,9,10,11,12,13,14]
--R                                                    Type: List(Fraction(Integer))
--E 16

--S 17 of 26
times(n:NNI, i:INT):INT == n*i
--R
--R   Function declaration times : (NonNegativeInteger,Integer) -> Integer
--R      has been added to workspace.
--R                                                                        Type: Void
--E 17

--S 18 of 26
r := recur(times)
--R
--R   Compiling function times with type (NonNegativeInteger,Integer) ->
--R      Integer
--R
--R   (18)  theMap(MAPPKG1;recur;2M;7!0,0)
--R                                Type: ((NonNegativeInteger,Integer) -> Integer)
--E 18

--S 19 of 26
fact := curryRight(r, 1)
--R
--R
--R   (19)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                                          Type: (NonNegativeInteger -> Integer)
--E 19

--S 20 of 26
fact 4
--R
--R
--R   (20)  24
--R                                                        Type: PositiveInteger
--E 20

--S 21 of 26
mto2ton(m, n) ==
  raiser := square^n
  raiser m
--R
--R                                                                        Type: Void
--E 21
```

```
--S 22 of 26
mto2ton(3, 3)
--R
--R   Compiling function mto2ton with type (PositiveInteger,
--R      PositiveInteger) -> Fraction(Integer)
--R
--R   (22)  6561
--R                                                    Type: Fraction(Integer)
--E 22

--S 23 of 26
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R
--R   Function declaration shiftfib : List(Integer) -> Integer has been
--R      added to workspace.
--R                                                                 Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R
--R   (24)  [0,1]
--R                                                       Type: List(Integer)
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
--R   Compiling function shiftfib with type List(Integer) -> Integer
--R
--R   (25)  theMap(MAPPKG2;curry;MAM;2!0,0)
--R                                                    Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R
--R   (26)
--R   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R    317811, 514229, 832040]
--R                                                       Type: List(Integer)
--E 26
```

```
)spool
)lisp (bye)
```

———-

— MappingPackage2.help —

```
=====================================================================
MappingPackage examples
=====================================================================

Function are objects of type Mapping.  In this section we demonstrate
some library operations from the packages MappingPackage1, MappingPackage2,
and MappingPackage3 that manipulate and create functions. Some terminology:
a nullary function takes no arguments, a unary function takes one argument,
and a binary function takes two arguments.

We begin by creating an example function that raises a rational number
to an integer exponent.

  power(q: FRAC INT, n: INT): FRAC INT == q**n
                      Type: Void

  power(2,3)
    8
                       Type: Fraction Integer

The twist operation transposes the arguments of a binary function.
Here rewop(a, b) is power(b, a).

  rewop := twist power
    theMap(MAPPKG3;twist;MM;5!0)
                       Type: ((Integer,Fraction Integer) -> Fraction Integer)

This is 2^3.

  rewop(3, 2)
    8
                       Type: Fraction Integer

Now we define square in terms of power.

  square: FRAC INT -> FRAC INT
                       Type: Void
The curryRight operation creates a unary function from a binary one by
providing a constant argument on the right.

  square:= curryRight(power, 2)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
```

```
                    Type: (Fraction Integer -> Fraction Integer)

Likewise, the curryLeft operation provides a constant argument on the
left.

  square 4
    16
                    Type: Fraction Integer

The constantRight operation creates (in a trivial way) a binary
function from a unary one: constantRight(f) is the function g such
that g(a,b)= f(a).

  squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
    theMap(MAPPKG3;constantRight;MM;3!0)
              Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)

Likewise, constantLeft(f) is the function g such that g(a,b)= f(b).

  squirrel(1/2, 1/3)
    1
    -
    4
                    Type: Fraction Integer

The curry operation makes a unary function nullary.

  sixteen := curry(square, 4/1)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                    Type: (() -> Fraction Integer)

  sixteen()
    16
                    Type: Fraction Integer

The * operation constructs composed functions.

  square2:=square*square
    theMap(MAPPKG3;*;MMM;6!0,0)
                    Type: (Fraction Integer -> Fraction Integer)

  square2 3
    81
                    Type: Fraction Integer

Use the ** operation to create functions that are n-fold iterations of
other functions.

  sc(x: FRAC INT): FRAC INT == x + 1
                    Type: Void
```

```
This is a list of Mapping objects.

  incfns := [sc**i for i in 0..10]
   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0)]
                        Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

  [f 4 for f in incfns]
    [4,5,6,7,8,9,10,11,12,13,14]
                        Type: List Fraction Integer

Use the recur operation for recursion:

  g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).

  times(n:NNI, i:INT):INT == n*i
                        Type: Void

  r := recur(times)
    theMap(MAPPKG1;recur;2M;7!0,0)
                        Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

  fact := curryRight(r, 1)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
                        Type: (NonNegativeInteger -> Integer)

  fact 4
    24
                        Type: PositiveInteger

Constructed functions can be used within other functions.

  mto2ton(m, n) ==
    raiser := square^n
    raiser m
                        Type: Void

This is 3^(2^3).

  mto2ton(3, 3)
    6561
```

```
                        Type: Fraction Integer
```

Here shiftfib is a unary function that modifies its argument.

```
  shiftfib(r: List INT) : INT ==
    t := r.1
    r.1 := r.2
    r.2 := r.2 + t
    t
                        Type: Void
```

By currying over the argument we get a function with private state.

```
  fibinit: List INT := [0, 1]
    [0,1]
                        Type: List Integer

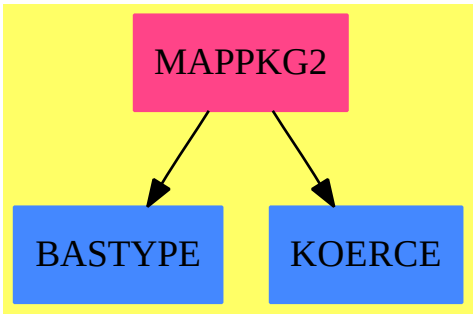
  fibs := curry(shiftfib, fibinit)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                        Type: (() -> Integer)

  [fibs() for i in 0..30]
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
    317811, 514229, 832040]
                        Type: List Integer
```

See Also:
o)help MappingPackage1
o)show MappingPackage2
o)help MappingPackage3
o)help MappingPackage4

———————-

14.10.1 MappingPackage2 (MAPPKG2)



Exports:
const constant curry diag

— package MAPPKG2 MappingPackage2 —

```
)abbrev package MAPPKG2 MappingPackage2
++ Description:
++ Various Currying operations.

MappingPackage2(A:SetCategory, C:SetCategory): MPcat == MPdef where
    NNI   ==>  NonNegativeInteger

    MPcat ==  with
        const:   C           -> (A ->C)
          ++\spad{const c} is a function which produces \spad{c} when
          ++ applied to its argument.

        curry:    (A ->C, A)    -> (()->C)
          ++\spad{cu(f,a)} is the function \spad{g}
          ++ such that \spad{g ()= f a}.
        constant:    (()->C)       -> (A ->C)
          ++\spad{vu(f)} is the function \spad{g}
          ++ such that \spad{g a= f ()}.

        diag:  ((A,A)->C)    -> (A->C)
          ++\spad{diag(f)} is the function \spad{g}
          ++ such that \spad{g a = f(a,a)}.


    MPdef == add

        MappingPackageInternalHacks2(A, C)

        a: A
        c: C
```

```
faa:  A -> A
f0c:  ()-> C
fac:  A -> C
faac: (A,A)->C

const c       == (a1:A):C +-> arg2(a1, c)
curry(fac, a) == fac a
constant f0c  == (a1:A):C +-> arg2(a1, f0c())
diag  faac    == (a1:A):C +-> faac(a1, a1)
```

———-

— **MAPPKG2.dotabb** —

```
"MAPPKG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPPKG2" -> "BASTYPE"
"MAPPKG2" -> "KOERCE"
```

———-

14.11 package MAPPKG3 MappingPackage3

— **MappingPackage3.input** —

```
)set break resume
)spool MappingPackage3.output
)set message test on
)set message auto off
)clear all
--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R   Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R      (Integer) has been added to workspace.
--R                                                                   Type: Void
--E 1

--S 2 of 26
power(2,3)
--R
--R   Compiling function power with type (Fraction(Integer),Integer) ->
--R      Fraction(Integer)
```

```
--R
--R   (2)  8
--R                                                    Type: Fraction(Integer)
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R   (3)  theMap(MAPPKG3;twist;MM;5!0)
--R                    Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 26
rewop(3, 2)
--R
--R
--R   (4)  8
--R                                                    Type: Fraction(Integer)
--E 4

--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R                                                                 Type: Void
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R   (6)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                              Type: (Fraction(Integer) -> Fraction(Integer))
--E 6

--S 7 of 26
square 4
--R
--R
--R   (7)  16
--R                                                    Type: Fraction(Integer)
--E 7

--S 8 of 26
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R   (8)  theMap(MAPPKG3;constantRight;MM;3!0)
--R          Type: ((Fraction(Integer),Fraction(Integer)) -> Fraction(Integer))
--E 8
```

```
--S 9 of 26
squirrel(1/2, 1/3)
--R
--R
--R        1
--R   (9)  -
--R        4
--R                                                    Type: Fraction(Integer)
--E 9

--S 10 of 26
sixteen := curry(square, 4/1)
--R
--R
--R   (10)  theMap(MAPPKG2;curry;MAM;2!0,0)
--R                                              Type: (() -> Fraction(Integer))
--E 10

--S 11 of 26
sixteen()
--R
--R
--R   (11)  16
--R                                                    Type: Fraction(Integer)
--E 11

--S 12 of 26
square2:=square*square
--R
--R
--R   (12)  theMap(MAPPKG3;*;MMM;6!0,0)
--R                               Type: (Fraction(Integer) -> Fraction(Integer))
--E 12

--S 13 of 26
square2 3
--R
--R
--R   (13)  81
--R                                                    Type: Fraction(Integer)
--E 13

--S 14 of 26
sc(x: FRAC INT): FRAC INT == x + 1
--R
--R   Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
--R      been added to workspace.
--R                                                                 Type: Void
--E 14
```

```
--S 15 of 26
incfns := [sc**i for i in 0..10]
--R
--R   Compiling function sc with type Fraction(Integer) -> Fraction(
--R      Integer)
--R
--R   (15)
--R   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R    theMap(MAPPKG1;**;MNniM;6!0,0)]
--R                          Type: List((Fraction(Integer) -> Fraction(Integer)))
--E 15

--S 16 of 26
[f 4 for f in incfns]
--R
--R
--R   (16)  [4,5,6,7,8,9,10,11,12,13,14]
--R                                                  Type: List(Fraction(Integer))
--E 16

--S 17 of 26
times(n:NNI, i:INT):INT == n*i
--R
--R   Function declaration times : (NonNegativeInteger,Integer) -> Integer
--R      has been added to workspace.
--R                                                                       Type: Void
--E 17

--S 18 of 26
r := recur(times)
--R
--R   Compiling function times with type (NonNegativeInteger,Integer) ->
--R      Integer
--R
--R   (18)  theMap(MAPPKG1;recur;2M;7!0,0)
--R                                Type: ((NonNegativeInteger,Integer) -> Integer)
--E 18

--S 19 of 26
fact := curryRight(r, 1)
--R
--R
--R   (19)  theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                                          Type: (NonNegativeInteger -> Integer)
--E 19
```

```
--S 20 of 26
fact 4
--R
--R
--R   (20)  24
--R                                                    Type: PositiveInteger
--E 20

--S 21 of 26
mto2ton(m, n) ==
  raiser := square^n
  raiser m
--R
--R                                                                Type: Void
--E 21

--S 22 of 26
mto2ton(3, 3)
--R
--R   Compiling function mto2ton with type (PositiveInteger,
--R      PositiveInteger) -> Fraction(Integer)
--R
--R   (22)  6561
--R                                                  Type: Fraction(Integer)
--E 22

--S 23 of 26
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R
--R   Function declaration shiftfib : List(Integer) -> Integer has been
--R      added to workspace.
--R                                                                Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R
--R   (24)  [0,1]
--R                                                     Type: List(Integer)
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
```

```
--R   Compiling function shiftfib with type List(Integer) -> Integer
--R
--R   (25)  theMap(MAPPKG2;curry;MAM;2!0,0)
--R                                                        Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R
--R   (26)
--R   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R    317811, 514229, 832040]
--R                                                          Type: List(Integer)
--E 26
)spool
)lisp (bye)
```

———-

— MappingPackage3.help —

```
========================================================================
MappingPackage examples
========================================================================

Function are objects of type Mapping.  In this section we demonstrate
some library operations from the packages MappingPackage1, MappingPackage2,
and MappingPackage3 that manipulate and create functions. Some terminology:
a nullary function takes no arguments, a unary function takes one argument,
and a binary function takes two arguments.

We begin by creating an example function that raises a rational number
to an integer exponent.

  power(q: FRAC INT, n: INT): FRAC INT == q**n
                     Type: Void

  power(2,3)
    8
                      Type: Fraction Integer

The twist operation transposes the arguments of a binary function.
Here rewop(a, b) is power(b, a).

  rewop := twist power
    theMap(MAPPKG3;twist;MM;5!0)
                      Type: ((Integer,Fraction Integer) -> Fraction Integer)
```

```
This is 2^3.

  rewop(3, 2)
    8
                        Type: Fraction Integer

Now we define square in terms of power.

  square: FRAC INT -> FRAC INT
                        Type: Void
The curryRight operation creates a unary function from a binary one by
providing a constant argument on the right.

  square:= curryRight(power, 2)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
                        Type: (Fraction Integer -> Fraction Integer)

Likewise, the curryLeft operation provides a constant argument on the
left.

  square 4
    16
                        Type: Fraction Integer

The constantRight operation creates (in a trivial way) a binary
function from a unary one: constantRight(f) is the function g such
that g(a,b)= f(a).

  squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
    theMap(MAPPKG3;constantRight;MM;3!0)
              Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)

Likewise, constantLeft(f) is the function g such that g(a,b)= f(b).

  squirrel(1/2, 1/3)
    1
    -
    4
                        Type: Fraction Integer

The curry operation makes a unary function nullary.

  sixteen := curry(square, 4/1)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                        Type: (() -> Fraction Integer)

  sixteen()
    16
                        Type: Fraction Integer
```

```
The * operation constructs composed functions.

  square2:=square*square
    theMap(MAPPKG3;*;MMM;6!0,0)
                        Type: (Fraction Integer -> Fraction Integer)

  square2 3
    81
                        Type: Fraction Integer

Use the ** operation to create functions that are n-fold iterations of
other functions.

  sc(x: FRAC INT): FRAC INT == x + 1
                        Type: Void

This is a list of Mapping objects.

  incfns := [sc**i for i in 0..10]
   [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
    theMap(MAPPKG1;**;MNniM;6!0,0)]
                        Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

  [f 4 for f in incfns]
    [4,5,6,7,8,9,10,11,12,13,14]
                        Type: List Fraction Integer

Use the recur operation for recursion:

  g := recur f means g(n,x) == f(n,f(n-1,...f(1,x))).

  times(n:NNI, i:INT):INT == n*i
                        Type: Void

  r := recur(times)
    theMap(MAPPKG1;recur;2M;7!0,0)
                        Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

  fact := curryRight(r, 1)
    theMap(MAPPKG3;curryRight;MBM;1!0,0)
                        Type: (NonNegativeInteger -> Integer)
```

```
  fact 4
    24
                        Type: PositiveInteger
```

Constructed functions can be used within other functions.

```
  mto2ton(m, n) ==
    raiser := square^n
    raiser m
                        Type: Void
```

This is 3^(2^3).

```
  mto2ton(3, 3)
    6561
                        Type: Fraction Integer
```

Here shiftfib is a unary function that modifies its argument.

```
  shiftfib(r: List INT) : INT ==
    t := r.1
    r.1 := r.2
    r.2 := r.2 + t
    t
                        Type: Void
```

By currying over the argument we get a function with private state.

```
  fibinit: List INT := [0, 1]
    [0,1]
                        Type: List Integer

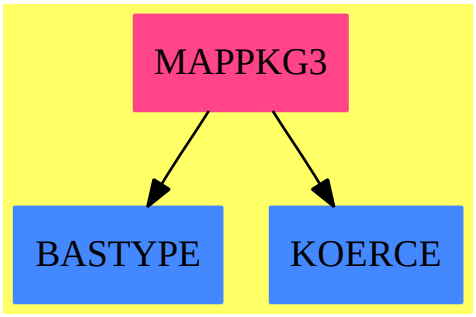
  fibs := curry(shiftfib, fibinit)
    theMap(MAPPKG2;curry;MAM;2!0,0)
                        Type: (() -> Integer)

  [fibs() for i in 0..30]
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
    2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
    317811, 514229, 832040]
                        Type: List Integer
```

See Also:
o)help MappingPackage1
o)help MappingPackage2
o)show MappingPackage3
o)help MappingPackage4

———-

14.11.1 MappingPackage3 (MAPPKG3)



Exports:
constantLeft constantRight curryLeft curryRight twist ?*?

— **package MAPPKG3 MappingPackage3** —

```
)abbrev package MAPPKG3 MappingPackage3
++ Description:
++ Various Currying operations.

MappingPackage3(A:SetCategory, B:SetCategory, C:SetCategory):_
  MPcat == MPdef where
    NNI   ==>  NonNegativeInteger

    MPcat ==  with
        curryRight:   ((A,B)->C, B) -> (A ->C)
          ++\spad{curryRight(f,b)} is the function \spad{g} such that
          ++ \spad{g a = f(a,b)}.
        curryLeft:   ((A,B)->C, A) -> (B ->C)
          ++\spad{curryLeft(f,a)} is the function \spad{g}
          ++ such that \spad{g b = f(a,b)}.

        constantRight:   (A -> C)      -> ((A,B)->C)
          ++\spad{constantRight(f)} is the function \spad{g}
          ++ such that \spad{g (a,b)= f a}.
        constantLeft:   (B -> C)      -> ((A,B)->C)
          ++\spad{constantLeft(f)} is the function \spad{g}
          ++ such that \spad{g (a,b)= f b}.

        twist: ((A,B)->C)    -> ((B,A)->C)
          ++\spad{twist(f)} is the function \spad{g}
```

```
      ++ such that \spad{g (a,b)= f(b,a)}.

    "*":   (B->C, A->B) -> (A->C)
      ++\spad{f*g} is the function \spad{h}
      ++ such that \spad{h x= f(g x)}.


MPdef == add

    MappingPackageInternalHacks3(A, B, C)

    a: A
    b: B
    c: C
    faa:  A -> A
    f0c:  ()-> C
    fac:  A -> C
    fbc:  B -> C
    fab:  A -> B
    fabc: (A,B)->C
    faac: (A,A)->C

    -- Fix left and right arguments as constants.
    curryRight(fabc,b) == (a:A):C +-> fabc(a,b)
    curryLeft(fabc,a)  == (b:B):C +-> fabc(a,b)

    -- Add left and right arguments which are ignored.
    constantRight fac == (a:A, b:B):C +-> fac a
    constantLeft fbc  == (a:A, b:B):C +-> fbc b

    -- Combinators to rearrange arguments.
    twist fabc == (b:B, a:A):C +-> fabc(a,b)

    -- Functional composition
    fbc*fab == (a:A):C +-> comp(fbc,fab,a)
```

———-

— MAPPKG3.dotabb —

```
"MAPPKG3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG3"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPPKG3" -> "BASTYPE"
"MAPPKG3" -> "KOERCE"
```

———-

14.12 package MAPPKG4 MappingPackage4

— MappingPackage4.input —

```
)set break resume
)spool MappingPackage4.output
)set message test on
)set message auto off
)clear all

--S 1 of 21
p:=(x:EXPR(INT)):EXPR(INT)+->3*x
--R
--R
--R   (1)  theMap(Closure)
--R                              Type: (Expression Integer -> Expression Integer)
--E 1

--S 2 of 21
q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
--R
--R
--R   (2)  theMap(Closure)
--R                              Type: (Expression Integer -> Expression Integer)
--E 2

--S 3 of 21
(p+q)(4)-(p(4)+q(4))
--R
--R
--R   (3)  0
--R                                                    Type: Expression Integer
--E 3

--S 4 of 21
(p+q)(x)-(p(x)+q(x))
--R
--R
--R   (4)  0
--R                                                    Type: Expression Integer
--E 4

--S 5 of 21
(p-q)(4)-(p(4)-q(4))
--R
--R
--R   (5)  0
--R                                                    Type: Expression Integer
```

```
--E 5

--S 6 of 21
(p-q)(x)-(p(x)-q(x))
--R
--R
--R   (6)  0
--R                                                  Type: Expression Integer
--E 6

--S 7 of 21
(p*q)(4)-(p(4)*q(4))
--R
--R
--R   (7)  0
--R                                                  Type: Expression Integer
--E 7

--S 8 of 21
(p*q)(x)-(p(x)*q(x))
--R
--R
--R   (8)  0
--R                                                  Type: Expression Integer
--E 8

--S 9 of 21
(p/q)(4)-(p(4)/q(4))
--R
--R
--R   (9)  0
--R                                                  Type: Expression Integer
--E 9

--S 10 of 21
(p/q)(x)-(p(x)/q(x))
--R
--R
--R   (10)  0
--R                                                  Type: Expression Integer
--E 10

--S 11 of 21
r:=(x:INT):INT+-> (x*x*x)
--R
--R
--R   (11)  theMap(Closure)
--R                                              Type: (Integer -> Integer)
--E 11
```

```
--S 12 of 21
s:=(y:INT):INT+-> (y*y+3)
--R
--R
--R   (12)  theMap(Closure)
--R                                                  Type: (Integer -> Integer)
--E 12

--S 13 of 21
(r+s)(4)-(r(4)+s(4))
--R
--R
--R   (13)  0
--R                                                   Type: NonNegativeInteger
--E 13

--S 14 of 21
(r-s)(4)-(r(4)-s(4))
--R
--R
--R   (14)  0
--R                                                   Type: NonNegativeInteger
--E 14

--S 15 of 21
(r*s)(4)-(r(4)*s(4))
--R
--R
--R   (15)  0
--R                                                   Type: NonNegativeInteger
--E 15

--S 16 of 21
t:=(x:INT):EXPR(INT)+-> (x*x*x)
--R
--R
--R   (16)  theMap(Closure)
--R                                       Type: (Integer -> Expression Integer)
--E 16

--S 17 of 21
u:=(y:INT):EXPR(INT)+-> (y*y+3)
--R
--R
--R   (17)  theMap(Closure)
--R                                       Type: (Integer -> Expression Integer)
--E 17

--S 18 of 21
(t/u)(4)-(t(4)/u(4))
```

```
--R
--R
--R   (18)  0
--R                                                    Type: Expression Integer
--E 18

--S 19 of 21
h:=(x:EXPR(INT)):EXPR(INT)+->1
--R
--R   (19)  theMap(Closure)
--R                              Type: (Expression Integer -> Expression Integer)
--E 19

--S 20 of 21
(p/h)(x)
--R
--R   (20)  3x
--R                                                    Type: Expression Integer
--E 20

--S 21 of 21
(q/h)(x)
--R
--R   (21)  2x + 3
--R                                                    Type: Expression Integer
--E 21

)spool
)lisp (bye)
```

———-

— MappingPackage4.help —

```
=====================================================================
MappingPackage examples
=====================================================================

We can construct some simple maps that take a variable x
into an equation:

  p:=(x:EXPR(INT)):EXPR(INT)+->3*x
  q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3

Now we can do the four arithmetic operations, +, -, *, / on these
newly constructed mappings. Since the maps are from the domain
Expression Integer to the same domain we can also use symbolic
values for the argument. All of the following will return 0,
```

```
showing that function composition is equivalent to the result
of doing the operations individually.

  (p+q)(4)-(p(4)+q(4))
  (p+q)(x)-(p(x)+q(x))

  (p-q)(4)-(p(4)-q(4))
  (p-q)(x)-(p(x)-q(x))

  (p*q)(4)-(p(4)*q(4))
  (p*q)(x)-(p(x)*q(x))

  (p/q)(4)-(p(4)/q(4))
  (p/q)(x)-(p(x)/q(x))

We can construct simple maps from Integer to Integer but this
limits our ability to do division.

  r:=(x:INT):INT+-> (x*x*x)
  s:=(y:INT):INT+-> (y*y+3)

Again, all of these will return 0:

  (r+s)(4)-(r(4)+s(4))
  (r-s)(4)-(r(4)-s(4))
  (r*s)(4)-(r(4)*s(4))

If we want to do division with Integer inputs we create the
appropriate map:

  t:=(x:INT):EXPR(INT)+-> (x*x*x)
  u:=(y:INT):EXPR(INT)+-> (y*y+3)

  (t/u)(4)-(t(4)/u(4))

We can even recover the original functions if we make a map
that always returns the constant 1:

  h:=(x:EXPR(INT)):EXPR(INT)+->1

    theMap(Closure)
             Type: (Expression Integer -> Expression Integer)

  (p/h)(x)

     3x
                                       Type: Expression Integer
  (q/h)(x)

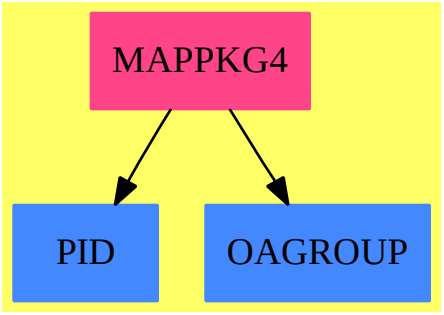
    2x + 3
```

```
                                        Type: Expression Integer

See Also:
o )show MappingPackage1
o )help MappingPackage2
o )help MappingPackage3
o )help MappingPackage4
```

———-

14.12.1 MappingPackage4 (MAPPKG4)



Exports:
?*? ?+? ?-? ?/?

— package MAPPKG4 MappingPackage4 —

```
)abbrev package MAPPKG4 MappingPackage4
++ Author: Timothy Daly
++ Description:
++ Functional Composition.
++ Given functions f and g, returns the applicable closure

MappingPackage4(A:SetCategory, B:Ring):
 with
   "+": (A->B, A->B) -> (A->B)
        ++ \spad(+) does functional addition
        ++
        ++X f:=(x:INT):INT +-> 3*x
        ++X g:=(x:INT):INT +-> 2*x+3
        ++X (f+g)(4)
   "-": (A->B, A->B) -> (A->B)
        ++ \spad(+) does functional addition
```

```
        ++
        ++X f:=(x:INT):INT +-> 3*x
        ++X g:=(x:INT):INT +-> 2*x+3
        ++X (f-g)(4)
   "*": (A->B, A->B) -> (A->B)
        ++ \spad(+) does functional addition
        ++
        ++X f:=(x:INT):INT +-> 3*x
        ++X g:=(x:INT):INT +-> 2*x+3
        ++X (f*g)(4)
   "/": (A->Expression(Integer), A->Expression(Integer)) -> (A->Expression(Integer))
        ++ \spad(+) does functional addition
        ++
        ++X p:=(x:EXPR(INT)):EXPR(INT)+->3*x
        ++X q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
        ++X (p/q)(4)
        ++X (p/q)(x)
  == add
    fab ==> (A -> B)
    faei ==> (A -> Expression(Integer))

    funcAdd(g:fab,h:fab,x:A):B == ((g x) + (h x))$B

    (a:fab)+(b:fab) == c +-> funcAdd(a,b,c)

    funcSub(g:fab,h:fab,x:A):B == ((g x) - (h x))$B

    (a:fab)-(b:fab) == c +-> funcSub(a,b,c)

    funcMul(g:fab,h:fab,x:A):B == ((g x) * (h x))$B

    (a:fab)*(b:fab) == c +-> funcMul(a,b,c)

    funcDiv(g:faei,h:faei,x:A):Expression(Integer)
           == ((g x) / (h x))$Expression(Integer)

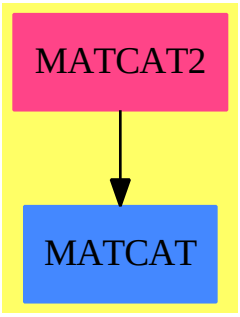
    (a:faei)/(b:faei) == c +-> funcDiv(a,b,c)
```

— MAPPKG4.dotabb —

```
"MAPPKG4" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG4"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"MAPPKG4" -> "PID"
"MAPPKG4" -> "OAGROUP"
```

14.13 package MATCAT2 MatrixCategoryFunctions2

14.13.1 MatrixCategoryFunctions2 (MATCAT2)



Exports:
map reduce

— **package MATCAT2 MatrixCategoryFunctions2** —

```
)abbrev package MATCAT2 MatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 21 March 1994
++ Basic Operations:
++ Related Domains: IndexedMatrix(R,minRow,minCol), Matrix(R),
++    RectangularMatrix(n,m,R), SquareMatrix(n,R)
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Keywords: matrix, map, reduce
++ Examples:
++ References:
++ Description:
++ \spadtype{MatrixCategoryFunctions2} provides functions between two matrix
++ domains.  The functions provided are \spadfun{map} and \spadfun{reduce}.

MatrixCategoryFunctions2(R1,Row1,Col1,M1,R2,Row2,Col2,M2):_
         Exports == Implementation where
  R1   : Ring
  Row1 : FiniteLinearAggregate R1
  Col1 : FiniteLinearAggregate R1
  M1   : MatrixCategory(R1,Row1,Col1)
  R2   : Ring
```

```
  Row2 : FiniteLinearAggregate R2
  Col2 : FiniteLinearAggregate R2
  M2   : MatrixCategory(R2,Row2,Col2)

  Exports ==> with
    map: (R1 -> R2,M1) -> M2
     ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
    map: (R1 -> Union(R2,"failed"),M1) -> Union(M2,"failed")
     ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
    reduce: ((R1,R2) -> R2,M1,R2) -> R2
     ++ \spad{reduce(f,m,r)} returns a matrix n where
     ++ \spad{n[i,j] = f(m[i,j],r)} for all indices i and j.

  Implementation ==> add
    minr ==> minRowIndex
    maxr ==> maxRowIndex
    minc ==> minColIndex
    maxc ==> maxColIndex

    map(f:(R1->R2),m:M1):M2 ==
      ans : M2 := new(nrows m,ncols m,0)
      for i in minr(m)..maxr(m) for k in minr(ans)..maxr(ans) repeat
        for j in minc(m)..maxc(m) for l in minc(ans)..maxc(ans) repeat
          qsetelt_!(ans,k,l,f qelt(m,i,j))
      ans

    map(f:(R1 -> (Union(R2,"failed"))),m:M1):Union(M2,"failed") ==
      ans : M2 := new(nrows m,ncols m,0)
      for i in minr(m)..maxr(m) for k in minr(ans)..maxr(ans) repeat
        for j in minc(m)..maxc(m) for l in minc(ans)..maxc(ans) repeat
          (r := f qelt(m,i,j)) = "failed" => return "failed"
          qsetelt_!(ans,k,l,r::R2)
      ans

    reduce(f,m,ident) ==
      s := ident
      for i in minr(m)..maxr(m) repeat
       for j in minc(m)..maxc(m) repeat
         s := f(qelt(m,i,j),s)
      s
```

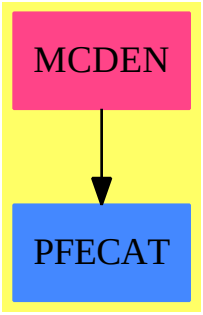
———-

— MATCAT2.dotabb —

```
"MATCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MATCAT2"]
"MATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MATCAT"]
"MATCAT2" -> "MATCAT"
```

14.14 package MCDEN MatrixCommonDenominator

14.14.1 MatrixCommonDenominator (MCDEN)



Exports:
commonDenominator clearDenominator splitDenominator

— **package MCDEN MatrixCommonDenominator** —

```
)abbrev package MCDEN MatrixCommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 20 Jul 1990
++ Keywords: gcd, quotient, matrix, common, denominator.
++ Description:
++ MatrixCommonDenominator provides functions to
++ compute the common denominator of a matrix of elements of the
++ quotient field of an integral domain.

MatrixCommonDenominator(R, Q): Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R

  VR ==> Vector R
  VQ ==> Vector Q

  Exports ==> with
    commonDenominator: Matrix Q -> R
      ++ commonDenominator(q) returns a common denominator d for
      ++ the elements of q.
    clearDenominator : Matrix Q -> Matrix R
```

```
      ++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
      ++ a common denominator for the elements of q.
    splitDenominator : Matrix Q -> Record(num: Matrix R, den: R)
      ++ splitDenominator(q) returns \spad{[p, d]} such that \spad{q = p/d} and d
      ++ is a common denominator for the elements of q.

  Implementation ==> add
    import ListFunctions2(Q, R)
    import MatrixCategoryFunctions2(Q,VQ,VQ,Matrix Q,R,VR,VR,Matrix R)

    clearDenominator m ==
      d := commonDenominator m
      map(x +-> numer(d*x), m)

    splitDenominator m ==
      d := commonDenominator m
      [map(x +-> numer(d*x), m), d]

    if R has GcdDomain then
      commonDenominator m == lcm map(denom, parts m)
    else
      commonDenominator m == reduce("*",map(denom, parts m),1)$List(R)
```

———-

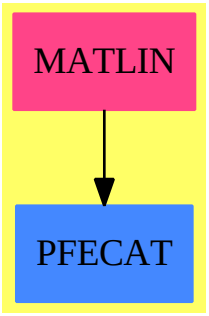
— **MCDEN.dotabb** —

```
"MCDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MCDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MCDEN" -> "PFECAT"
```

———-

14.15 package MATLIN MatrixLinearAlgebraFunctions

14.15.1 MatrixLinearAlgebraFunctions (MATLIN)



Exports:

adjoint	determinant	elColumn2!	elRow1!	elRow2!
fractionFreeGauss!	inverse	invertIfCan	minordet	normalizedDivide
nullSpace	nullity	rank	rowEchelon	

— package MATLIN MatrixLinearAlgebraFunctions —

```
)abbrev package MATLIN MatrixLinearAlgebraFunctions
++ Author: Clifton J. Williamson, P.Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: December 1992
++ Basic Operations:
++ Related Domains: IndexedMatrix(R,minRow,minCol), Matrix(R),
++    RectangularMatrix(n,m,R), SquareMatrix(n,R)
++ Also See:
++ AMS Classifications:
++ Keywords: matrix, canonical forms, linear algebra
++ Examples:
++ References:
++ Description:
++ \spadtype{MatrixLinearAlgebraFunctions} provides functions to compute
++ inverses and canonical forms.

MatrixLinearAlgebraFunctions(R,Row,Col,M):Exports == Implementation where
  R   : CommutativeRing
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M   : MatrixCategory(R,Row,Col)
  I ==> Integer

  Exports ==> with

    determinant: M -> R
```

```
    ++ \spad{determinant(m)} returns the determinant of the matrix m.
    ++ an error message is returned if the matrix is not square.
  minordet: M -> R
    ++ \spad{minordet(m)} computes the determinant of the matrix m using
    ++ minors. Error: if the matrix is not square.
  elRow1!       :   (M,I,I)         ->  M
    ++ elRow1!(m,i,j) swaps rows i and j of matrix m : elementary operation
    ++ of first kind
  elRow2!       :  (M,R,I,I)        ->  M
    ++ elRow2!(m,a,i,j)  adds to row i a*row(m,j) : elementary operation of
    ++ second kind. (i ^=j)
  elColumn2!    :  (M,R,I,I)        ->  M
    ++ elColumn2!(m,a,i,j)  adds to column i a*column(m,j) : elementary
    ++ operation of second kind. (i ^=j)

  if R has IntegralDomain then
    rank: M -> NonNegativeInteger
      ++ \spad{rank(m)} returns the rank of the matrix m.
    nullity: M -> NonNegativeInteger
      ++ \spad{nullity(m)} returns the mullity of the matrix m. This is
      ++ the dimension of the null space of the matrix m.
    nullSpace: M -> List Col
      ++ \spad{nullSpace(m)} returns a basis for the null space of the
      ++ matrix m.
    fractionFreeGauss! : M -> M
      ++ \spad{fractionFreeGauss(m)} performs the fraction
      ++ free gaussian  elimination on the matrix m.
    invertIfCan : M -> Union(M,"failed")
      ++ \spad{invertIfCan(m)} returns the inverse of m over R
    adjoint : M -> Record(adjMat:M, detMat:R)
      ++ \spad{adjoint(m)} returns the ajoint matrix of m (i.e. the matrix
      ++ n such that m*n = determinant(m)*id) and the detrminant of m.

  if R has EuclideanDomain then
    rowEchelon: M -> M
      ++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.

    normalizedDivide: (R, R) -> Record(quotient:R, remainder:R)
      ++ normalizedDivide(n,d) returns a normalized quotient and
      ++ remainder such that consistently unique representatives
      ++ for the residue class are chosen, e.g. positive remainders

  if R has Field then
    inverse: M -> Union(M,"failed")
      ++ \spad{inverse(m)} returns the inverse of the matrix.
      ++ If the matrix is not invertible, "failed" is returned.
      ++ Error: if the matrix is not square.

Implementation ==> add
```

```
rowAllZeroes?: (M,I) -> Boolean
rowAllZeroes?(x,i) ==
  -- determines if the ith row of x consists only of zeroes
  -- internal function: no check on index i
  for j in minColIndex(x)..maxColIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

colAllZeroes?: (M,I) -> Boolean
colAllZeroes?(x,j) ==
  -- determines if the ith column of x consists only of zeroes
  -- internal function: no check on index j
  for i in minRowIndex(x)..maxRowIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

minorDet:(M,I,List I,I,PrimitiveArray(Union(R,"uncomputed")))-> R
minorDet(x,m,l,i,v) ==
  z := v.m
  z case R => z
  ans : R := 0; rl : List I := nil()
  j := first l; l := rest l; pos := true
  minR := minRowIndex x; minC := minColIndex x;
  repeat
    if qelt(x,j + minR,i + minC) ^= 0 then
      ans :=
        md := minorDet(x,m - 2**(j :: NonNegativeInteger),_
                       concat_!(reverse rl,l),i + 1,v) *_
                       qelt(x,j + minR,i + minC)
        pos => ans + md
        ans - md
    null l =>
      v.m := ans
      return ans
    pos := not pos; rl := cons(j,rl); j := first l; l := rest l

minordet x ==
  (ndim := nrows x) ^= (ncols x) =>
    error "determinant: matrix must be square"
  -- minor expansion with (s---loads of) memory
  n1 : I := ndim - 1
  v : PrimitiveArray(Union(R,"uncomputed")) :=
       new((2**ndim - 1) :: NonNegativeInteger,"uncomputed")
  minR := minRowIndex x; maxC := maxColIndex x
  for i in 0..n1 repeat
    qsetelt_!(v,(2**i - 1),qelt(x,i + minR,maxC))
  minorDet(x, 2**ndim - 2, [i for i in 0..n1], 0, v)

   -- elementary operation of first kind: exchange two rows --
elRow1!(m:M,i:I,j:I) : M ==
```

```
    vec:=row(m,i)
    setRow!(m,i,row(m,j))
    setRow!(m,j,vec)
    m

          -- elementary operation of second kind: add to row i--
                     -- a*row j  (i^=j) --
  elRow2!(m : M,a:R,i:I,j:I) : M ==
    vec:= map((r1:R):R +-> a*r1,row(m,j))
    vec:=map("+",row(m,i),vec)
    setRow!(m,i,vec)
    m
          -- elementary operation of second kind: add to column i --
                      -- a*column j (i^=j) --
  elColumn2!(m : M,a:R,i:I,j:I) : M ==
    vec:= map((r1:R):R +-> a*r1,column(m,j))
    vec:=map("+",column(m,i),vec)
    setColumn!(m,i,vec)
    m

  if R has IntegralDomain then
    -- Fraction-Free Gaussian Elimination
    fractionFreeGauss! x  ==
      (ndim := nrows x) = 1 => x
      ans := b := 1$R
      minR := minRowIndex x; maxR := maxRowIndex x
      minC := minColIndex x; maxC := maxColIndex x
      i := minR
      for j in minC..maxC repeat
        if qelt(x,i,j) = 0 then -- candidate for pivot = 0
          rown := minR - 1
          for k in (i+1)..maxR repeat
            if qelt(x,k,j) ^= 0 then
               rown := k -- found a pivot
               leave
          if rown > minR - 1 then
             swapRows_!(x,i,rown)
             ans := -ans
        (c := qelt(x,i,j)) = 0 =>  "next j" -- try next column
        for k in (i+1)..maxR repeat
          if qelt(x,k,j) = 0 then
            for l in (j+1)..maxC repeat
              qsetelt_!(x,k,l,(c * qelt(x,k,l) exquo b) :: R)
          else
            pv := qelt(x,k,j)
            qsetelt_!(x,k,j,0)
            for l in (j+1)..maxC repeat
              val := c * qelt(x,k,l) - pv * qelt(x,i,l)
              qsetelt_!(x,k,l,(val exquo b) :: R)
        b := c
```

```
      (i := i+1)>maxR => leave
    if ans=-1 then
      lasti := i-1
      for j in 1..maxC repeat x(lasti, j) := -x(lasti,j)
    x

  --
  lastStep(x:M)  : M ==
    ndim := nrows x
    minR := minRowIndex x; maxR := maxRowIndex x
    minC := minColIndex x; maxC := minC+ndim -1
    exCol:=maxColIndex x
    det:=x(maxR,maxC)
    maxR1:=maxR-1
    maxC1:=maxC+1
    minC1:=minC+1
    iRow:=maxR
    iCol:=maxC-1
    for i in maxR1..1 by -1 repeat
      for j in maxC1..exCol repeat
        ss:=+/[x(i,iCol+k)*x(i+k,j) for k in 1..(maxR-i)]
        x(i,j) := _exquo((det * x(i,j) - ss),x(i,iCol))::R
      iCol:=iCol-1
    subMatrix(x,minR,maxR,maxC1,exCol)

  invertIfCan(y) ==
    (nr:=nrows y) ^= (ncols y) =>
        error "invertIfCan: matrix must be square"
    adjRec := adjoint y
    (den:=recip(adjRec.detMat)) case "failed" => "failed"
    den::R * adjRec.adjMat

  adjoint(y) ==
    (nr:=nrows y) ^= (ncols y) => error "adjoint: matrix must be square"
    maxR := maxRowIndex y
    maxC := maxColIndex y
    x := horizConcat(copy y,scalarMatrix(nr,1$R))
    ffr:= fractionFreeGauss!(x)
    det:=ffr(maxR,maxC)
    [lastStep(ffr),det]


if R has Field then

  VR      ==> Vector R
  IMATLIN ==> InnerMatrixLinearAlgebraFunctions(R,Row,Col,M)
  MMATLIN ==> InnerMatrixLinearAlgebraFunctions(R,VR,VR,Matrix R)
  FLA2    ==> FiniteLinearAggregateFunctions2(R, VR, R, Col)
  MAT2    ==> MatrixCategoryFunctions2(R,Row,Col,M,R,VR,VR,Matrix R)
```

```
    rowEchelon  y == rowEchelon(y)$IMATLIN
    rank        y == rank(y)$IMATLIN
    nullity     y == nullity(y)$IMATLIN
    determinant y == determinant(y)$IMATLIN
    inverse     y == inverse(y)$IMATLIN
    if Col has shallowlyMutable then
      nullSpace y == nullSpace(y)$IMATLIN
    else
      nullSpace y ==
        [map((r1:R):R +-> r1, v)$FLA2
          for v in nullSpace(map((r2:R):R +-> r2, y)$MAT2)$MMATLIN]

  else if R has IntegralDomain then
    QF     ==> Fraction R
    Row2   ==> Vector QF
    Col2   ==> Vector QF
    M2     ==> Matrix QF
    IMATQF ==> InnerMatrixQuotientFieldFunctions(R,Row,Col,M,QF,Row2,Col2,M2)

    nullSpace m == nullSpace(m)$IMATQF

    determinant y ==
      (nrows y) ^= (ncols y) => error "determinant: matrix must be square"
      fm:=fractionFreeGauss!(copy y)
      fm(maxRowIndex fm,maxColIndex fm)

    rank x ==
      y :=
        (rk := nrows x) > (rh := ncols x) =>
            rk := rh
            transpose x
        copy x
      y := fractionFreeGauss! y
      i := maxRowIndex y
      while rk > 0 and rowAllZeroes?(y,i) repeat
        i := i - 1
        rk := (rk - 1) :: NonNegativeInteger
      rk :: NonNegativeInteger

    nullity x == (ncols x - rank x) :: NonNegativeInteger

    if R has EuclideanDomain then

      if R has IntegerNumberSystem then
          normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
             qr := divide(n, d)
             qr.remainder >= 0 => qr
             d > 0 =>
                qr.remainder := qr.remainder + d
                qr.quotient := qr.quotient - 1
```

```
          qr
       qr.remainder := qr.remainder - d
       qr.quotient := qr.quotient + 1
       qr
else
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
       divide(n, d)

rowEchelon y ==
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  n := minR - 1
  i := minR
  for j in minC..maxC repeat
    if i > maxR then leave x
    n := minR - 1
    xnj: R
    for k in i..maxR repeat
      if not zero?(xkj:=qelt(x,k,j)) and ((n = minR - 1) _
             or sizeLess?(xkj,xnj)) then
        n := k
        xnj := xkj
    n = minR - 1 => "next j"
    swapRows_!(x,i,n)
    for k in (i+1)..maxR repeat
      qelt(x,k,j) = 0 => "next k"
      aa := extendedEuclidean(qelt(x,i,j),qelt(x,k,j))
      (a,b,d) := (aa.coef1,aa.coef2,aa.generator)
      b1 := (qelt(x,i,j) exquo d) :: R
      a1 := (qelt(x,k,j) exquo d) :: R
      -- a*b1+a1*b = 1
      for k1 in (j+1)..maxC repeat
        val1 := a * qelt(x,i,k1) + b * qelt(x,k,k1)
        val2 := -a1 * qelt(x,i,k1) + b1 * qelt(x,k,k1)
        qsetelt_!(x,i,k1,val1); qsetelt_!(x,k,k1,val2)
      qsetelt_!(x,i,j,d); qsetelt_!(x,k,j,0)

    un := unitNormal qelt(x,i,j)
    qsetelt_!(x,i,j,un.canonical)
    if un.associate ^= 1 then for jj in (j+1)..maxC repeat
        qsetelt_!(x,i,jj,un.associate * qelt(x,i,jj))

    xij := qelt(x,i,j)
    for k in minR..(i-1) repeat
      qelt(x,k,j) = 0 => "next k"
      qr := normalizedDivide(qelt(x,k,j), xij)
      qsetelt_!(x,k,j,qr.remainder)
      for k1 in (j+1)..maxC repeat
        qsetelt_!(x,k,k1,qelt(x,k,k1) - qr.quotient * qelt(x,i,k1))
```

```
            i := i + 1
          x

    else determinant x == minordet x
```

———-

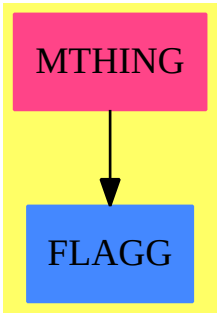
— **MATLIN.dotabb** —

```
"MATLIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MATLIN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MATLIN" -> "PFECAT"
```

———-

14.16 package MTHING MergeThing

14.16.1 MergeThing (MTHING)



Exports:
mergeDifference

— **package MTHING MergeThing** —

```
)abbrev package MTHING MergeThing
++ Description:
++ This package exports tools for merging lists

MergeThing(S:OrderedSet): Exports == Implementation where
  Exports == with
    mergeDifference: (List(S),List(S)) -> List(S)
      ++ mergeDifference(l1,l2) returns a list of elements in l1 not present
```

```
    ++ in l2. Assumes lists are ordered and all x in l2 are also in l1.
  Implementation == add
    mergeDifference1: (List S,S,List S) -> List S
    mergeDifference(x,y) ==
      null x or null y => x
      mergeDifference1(x,y.first,y.rest)
      x.first=y.first => x.rest
      x
    mergeDifference1(x,fy,ry) ==
      rx := x
      while not null rx repeat
        rx := rx.rest
        frx := rx.first
        while fy < frx repeat
          null ry => return x
          fy := first ry
          ry := rest ry
        frx = fy =>
          x.rest := rx.rest
          null ry => return x
          fy := ry.first
          ry := ry.rest
        x := rx
```

———-

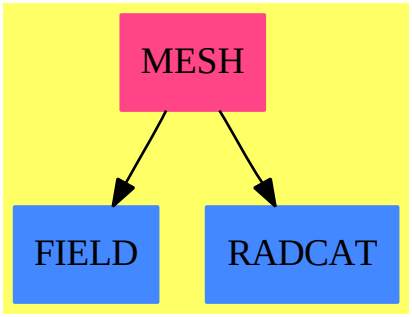
— **MTHING.dotabb** —

```
"MTHING" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MTHING"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"MTHING" -> "FLAGG"
```

———-

14.17 package MESH MeshCreationRoutinesForThreeDimensions

14.17.1 MeshCreationRoutinesForThreeDimensions (MESH)



Exports:
meshFun2Var meshPar1Var meshPar2Var ptFunc

— **package MESH MeshCreationRoutinesForThreeDimensions** —

```
)abbrev package MESH MeshCreationRoutinesForThreeDimensions
++ Author: Jim Wen
++ Date Created: ??
++ Date Last Updated: October 1991 by Jon Steinbach
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package has no description

MeshCreationRoutinesForThreeDimensions():Exports == Implementation where

  I       ==> Integer
  PI      ==> PositiveInteger
  SF      ==> DoubleFloat
  L       ==> List
  SEG     ==> Segment
  S       ==> String
  Fn1     ==> SF -> SF
  Fn2     ==> (SF,SF) -> SF
  Fn3     ==> (SF,SF,SF) -> SF
  FnPt    ==> (SF,SF) -> Point(SF)
  FnU     ==> Union(Fn3,"undefined")
  EX      ==> Expression
  DROP    ==> DrawOption
```

```
POINT   ==> Point(SF)
SPACE3  ==> ThreeSpace(SF)
COMPPROP ==> SubSpaceComponentProperty
TUBE    ==> TubePlot

Exports ==> with
  meshPar2Var: (Fn2,Fn2,Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
    ++ meshPar2Var(f,g,h,j,s1,s2,l) \undocumented
  meshPar2Var: (FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
    ++ meshPar2Var(f,s1,s2,l) \undocumented
  meshPar2Var: (SPACE3,FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
    ++ meshPar2Var(sp,f,s1,s2,l) \undocumented
  meshFun2Var: (Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
    ++ meshFun2Var(f,g,s1,s2,l) \undocumented
  meshPar1Var: (EX I,EX I,EX I,Fn1,SEG SF,L DROP) -> SPACE3
    ++ meshPar1Var(s,t,u,f,s1,l) \undocumented
  ptFunc: (Fn2,Fn2,Fn2,Fn3) -> ((SF,SF) -> POINT)
    ++ ptFunc(a,b,c,d) is an internal function exported in
    ++ order to compile packages.

Implementation ==> add
  import ViewDefaultsPackage()
  import SubSpaceComponentProperty()
  import DrawOptionFunctions0
  import SPACE3
  --import TUBE()

  -- local functions
  numberCheck(nums:Point SF):Void ==
    -- this function checks to see that the small floats are
    -- actually just that - rather than complex numbers or
    -- whatever (the whatever includes nothing presently
    -- since NaN, Not a Number, is not necessarily supported
    -- by common lisp). note that this function is dependent
    -- upon the fact that Common Lisp supports complex numbers.
    for i in minIndex(nums)..maxIndex(nums) repeat
      COMPLEXP(nums.(i::PositiveInteger))$Lisp =>
        error "An unexpected complex number was encountered in the calculations."

  makePt:(SF,SF,SF,SF) -> POINT
  makePt(x,y,z,c) == point(l : List SF := [x,y,z,c])
  ptFunc(f,g,h,c) ==
   (z1:SF,z2:SF):POINT +->
    x := f(z1,z2); y := g(z1,z2); z := h(z1,z2)
    makePt(x,y,z,c(x,y,z))

  -- parameterized equations of two variables
  meshPar2Var(sp,ptFun,uSeg,vSeg,opts) ==
    -- the issue of open and closed needs to be addressed, here, we are
    -- defaulting to open (which is probably the correct default)
```

```
      -- the user should be able to override that (optional argument?)
      llp : L L POINT := nil()
      uNum : PI  := var1Steps(opts,var1StepsDefault())
      vNum : PI  := var2Steps(opts,var2StepsDefault())
      ustep := (lo uSeg - hi uSeg)/uNum
      vstep := (lo vSeg - hi vSeg)/vNum
      someV := hi vSeg
      for iv in vNum..0 by -1 repeat
        if zero? iv then someV := lo vSeg
        -- hack: get last number in segment within segment
        lp : L POINT := nil()
        someU := hi uSeg
        for iu in uNum..0 by -1 repeat
          if zero? iu then someU := lo uSeg
          -- hack: get last number in segment within segment
          pt := ptFun(someU,someV)
          numberCheck pt
          lp := concat(pt,lp)
          someU := someU + ustep
        llp := concat(lp,llp)
        someV := someV + vstep
      -- now llp contains a list of lists of points
      -- for a surface that is a result of a function of 2 variables,
      -- the main component is open and each sublist is open as well
      lProp : L COMPPROP := [ new() for l in llp ]
      for aProp in lProp repeat
        close(aProp,false)
        solid(aProp,false)
      aProp : COMPPROP:= new()
      close(aProp,false)
      solid(aProp,false)
      space := sp
--      space := create3Space()
      mesh(space,llp,lProp,aProp)
      space

    meshPar2Var(ptFun,uSeg,vSeg,opts) ==
      sp := create3Space()
      meshPar2Var(sp,ptFun,uSeg,vSeg,opts)

    zCoord: (SF,SF,SF) -> SF
    zCoord(x,y,z) == z

    meshPar2Var(xFun,yFun,zFun,colorFun,uSeg,vSeg,opts) ==
      -- the color function should be parameterized by (u,v) as well,
      -- not (x,y,z) but we also want some sort of consistency and so
      -- changing this over would mean possibly changing the explicit
      -- stuff over and there, we probably do want the color function
      -- to be parameterized by (x,y,z) - not just (x,y) (this being
      -- for convinience only since z is also defined in terms of (x,y)).
```

```
      (colorFun case Fn3) =>
        meshPar2Var(ptFunc(xFun,yFun,zFun,colorFun :: Fn3),uSeg,vSeg,opts)
      meshPar2Var(ptFunc(xFun,yFun,zFun,zCoord),uSeg,vSeg,opts)

    -- explicit equations of two variables
    meshFun2Var(zFun,colorFun,xSeg,ySeg,opts) ==
      -- here, we construct the data for a function of two variables
      meshPar2Var((z1:SF,z2:SF):SF +-> z1,
                  (x1:SF,x2:SF):SF +-> x2,zFun,colorFun,xSeg,ySeg,opts)
```

———-

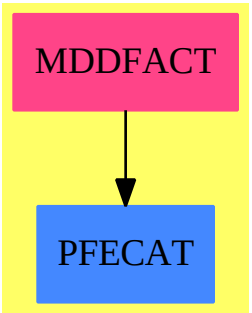
— **MESH.dotabb** —

```
"MESH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MESH"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"MESH" -> "FIELD"
"MESH" -> "RADCAT"
```

———-

14.18 package MDDFACT ModularDistinctDegreeFactorizer

14.18.1 ModularDistinctDegreeFactorizer (MDDFACT)



Exports:
factor gcd linears ddFact exptMod separateFactors

— **package MDDFACT ModularDistinctDegreeFactorizer** —

```
)abbrev package MDDFACT ModularDistinctDegreeFactorizer
++ Author: Barry Trager
++ Date Created:
++ Date Last Updated: 20.9.95 (JHD)
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package supports factorization and gcds
++ of univariate polynomials over the integers modulo different
++ primes. The inputs are given as polynomials over the integers
++ with the prime passed explicitly as an extra argument.

ModularDistinctDegreeFactorizer(U):C == T where
  U : UnivariatePolynomialCategory(Integer)
  I ==> Integer
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  V ==> Vector
  L ==> List
  DDRecord ==> Record(factor:EMR,degree:I)
  UDDRecord ==> Record(factor:U,degree:I)
  DDList ==> L DDRecord
  UDDList ==> L UDDRecord


  C == with
    gcd:(U,U,I) -> U
      ++ gcd(f1,f2,p) computes the gcd of the univariate polynomials
      ++ f1 and f2 modulo the integer prime p.
    linears: (U,I) -> U
      ++ linears(f,p) returns the product of all the linear factors
      ++ of f modulo p. Potentially incorrect result if f is not
      ++ square-free modulo p.
    factor:(U,I) -> L U
      ++ factor(f1,p) returns the list of factors of the univariate
      ++ polynomial f1 modulo the integer prime p.
      ++ Error: if f1 is not square-free modulo p.
    ddFact:(U,I) -> UDDList
      ++ ddFact(f,p) computes a distinct degree factorization of the
      ++ polynomial f modulo the prime p, i.e. such that each factor
      ++ is a product of irreducibles of the same degrees. The input
      ++ polynomial f is assumed to be square-free modulo p.
    separateFactors:(UDDList,I) -> L U
      ++ separateFactors(ddl, p) refines the distinct degree factorization
      ++ produced by ddFact to give a complete list of factors.
    exptMod:(U,I,U,I) -> U
```

```
    ++ exptMod(f,n,g,p) raises the univariate polynomial f to the nth
    ++ power modulo the polynomial g and the prime p.

T == add
  reduction(u:U,p:I):U ==
     zero? p => u
     map((i1:I):I +-> positiveRemainder(i1,p),u)
  merge(p:I,q:I):Union(I,"failed") ==
     p = q => p
     p = 0 => q
     q = 0 => p
     "failed"
  modInverse(c:I,p:I):I ==
      (extendedEuclidean(c,p,1)::Record(coef1:I,coef2:I)).coef1
  exactquo(u:U,v:U,p:I):Union(U,"failed") ==
      invlcv:=modInverse(leadingCoefficient v,p)
      r:=monicDivide(u,reduction(invlcv*v,p))
      reduction(r.remainder,p) ^=0 => "failed"
      reduction(invlcv*r.quotient,p)
  EMR := EuclideanModularRing(Integer,U,Integer,
                              reduction,merge,exactquo)

  probSplit2:(EMR,EMR,I) -> Union(List EMR,"failed")
  trace:(EMR,I,EMR) -> EMR
  ddfactor:EMR -> L EMR
  ddfact:EMR -> DDList
  sepFact1:DDRecord -> L EMR
  sepfact:DDList -> L EMR
  probSplit:(EMR,EMR,I) -> Union(L EMR,"failed")
  makeMonic:EMR -> EMR
  exptmod:(EMR,I,EMR) -> EMR

  lc(u:EMR):I == leadingCoefficient(u::U)
  degree(u:EMR):I == degree(u::U)
  makeMonic(u) == modInverse(lc(u),modulus(u)) * u

  i:I

  exptmod(u1,i,u2) ==
    i < 0 => error("negative exponentiation not allowed for exptMod")
    ans:= 1$EMR
    while i > 0 repeat
      if odd?(i) then ans:= (ans * u1) rem u2
      i:= i quo 2
      u1:= (u1 * u1) rem u2
    ans

  exptMod(a,i,b,q) ==
    ans:= exptmod(reduce(a,q),i,reduce(b,q))
    ans::U
```

```
ddfactor(u) ==
  if (c:= lc(u)) ^= 1$I then u:= makeMonic(u)
  ans:= sepfact(ddfact(u))
  cons(c::EMR,[makeMonic(f) for f in ans | degree(f) > 0])

gcd(u,v,q) == gcd(reduce(u,q),reduce(v,q))::U

factor(u,q) ==
  v:= reduce(u,q)
  dv:= reduce(differentiate(u),q)
  degree gcd(v,dv) > 0 =>
    error("Modular factor: polynomial must be squarefree")
  ans:= ddfactor v
  [f::U for f in ans]

ddfact(u) ==
  p:=modulus u
  w:= reduce(monomial(1,1)$U,p)
  m:= w
  d:I:= 1
  if (c:= lc(u)) ^= 1$I then u:= makeMonic u
  ans:DDList:= []
  repeat
    w:= exptmod(w,p,u)
    g:= gcd(w - m,u)
    if degree g > 0 then
      g:= makeMonic(g)
      ans:= [[g,d],:ans]
      u:= (u quo g)
    degree(u) = 0 => return [[c::EMR,0$I],:ans]
    d:= d+1
    d > (degree(u):I quo 2) =>
           return [[c::EMR,0$I],[u,degree(u)],:ans]

ddFact(u,q) ==
  ans:= ddfact(reduce(u,q))
  [[(dd.factor)::U,dd.degree]$UDDRecord for dd in ans]$UDDList

linears(u,q) ==
   uu:=reduce(u,q)
   m:= reduce(monomial(1,1)$U,q)
   gcd(exptmod(m,q,uu)-m,uu)::U

sepfact(factList) ==
  "append"/[sepFact1(f) for f in factList]

separateFactors(uddList,q) ==
  ans:= sepfact [[reduce(udd.factor,q),udd.degree]$DDRecord for
    udd in uddList]$DDList
```

```
      [f::U for f in ans]

    decode(s:Integer, p:Integer, x:U):U ==
      s<p => s::U
      qr := divide(s,p)
      qr.remainder :: U + x*decode(qr.quotient, p, x)

    sepFact1(f) ==
      u:= f.factor
      p:=modulus u
      (d := f.degree) = 0 => [u]
      if (c:= lc(u)) ^= 1$I then u:= makeMonic(u)
      d = (du := degree(u)) => [u]
      ans:L EMR:= []
      x:U:= monomial(1,1)
      -- for small primes find linear factors by exhaustion
      d=1 and p < 1000  =>
        for i in 0.. while du > 0 repeat
          if u(i::U) = 0 then
            ans := cons(reduce(x-(i::U),p),ans)
            du := du-1
        ans
      y:= x
      s:I:= 0
      ss:I := 1
      stack:L EMR:= [u]
      until null stack repeat
        t:= reduce(((s::U)+x),p)
        if not ((flist:= probSplit(first stack,t,d)) case "failed") then
          stack:= rest stack
          for fact in flist repeat
            f1:= makeMonic(fact)
            (df1:= degree(f1)) = 0 => nil
            df1 > d => stack:= [f1,:stack]
            ans:= [f1,:ans]
        p = 2 =>
          ss:= ss + 1
          x := y * decode(ss, p, y)
        s:= s+1
        s = p =>
          s:= 0
          ss := ss + 1
          x:= y * decode(ss, p, y)
--          not one? leadingCoefficient(x) =>
          not (leadingCoefficient(x) = 1) =>
              ss := p ** degree x
              x:= y ** (degree(x) + 1)
      [c * first(ans),:rest(ans)]

    probSplit(u,t,d) ==
```

```
      (p:=modulus(u)) = 2 => probSplit2(u,t,d)
      f1:= gcd(u,t)
      r:= ((p**(d:NNI)-1) quo 2):NNI
      n:= exptmod(t,r,u)
      f2:= gcd(u,n + 1)
      (g:= f1 * f2) = 1 => "failed"
      g = u => "failed"
      [f1,f2,(u quo g)]

    probSplit2(u,t,d) ==
      f:= gcd(u,trace(t,d,u))
      f = 1 => "failed"
      degree u = degree f => "failed"
      [1,f,u quo f]

    trace(t,d,u) ==
      p:=modulus(t)
      d:= d - 1
      tt:=t
      while d > 0 repeat
        tt:= (tt + (t:=exptmod(t,p,u))) rem u
        d:= d - 1
      tt
```

———-

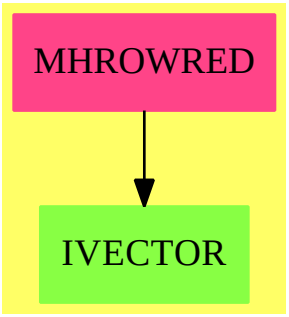
— MDDFACT.dotabb —

```
"MDDFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MDDFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MDDFACT" -> "PFECAT"
```

———-

14.19 package MHROWRED ModularHermitianRowReduction

14.19.1 ModularHermitianRowReduction (MHROWRED)



Exports:
rowEch normalizedDivide rowEchLocal rowEchelon rowEchelonLocal

— **package MHROWRED ModularHermitianRowReduction** —

```
)abbrev package MHROWRED ModularHermitianRowReduction
++ Author: Manuel Bronstein
++ Date Created: 22 February 1989
++ Date Last Updated: 24 November 1993
++ Keywords: matrix, reduction.
++ Description:
++ Modular hermitian row reduction.
-- should be moved into matrix whenever possible

ModularHermitianRowReduction(R): Exports == Implementation where
  R: EuclideanDomain

  Z   ==> Integer
  V   ==> Vector R
  M   ==> Matrix R
  REC ==> Record(val:R, cl:Z, rw:Z)

  Exports ==> with
    rowEch       : M -> M
      ++ rowEch(m) computes a modular row-echelon form of m, finding
      ++ an appropriate modulus.
    rowEchelon   : (M, R) -> M
      ++ rowEchelon(m, d) computes a modular row-echelon form mod d of
      ++    [d     ]
      ++    [  d   ]
```

```
      ++    [    . ]
      ++    [     d]
      ++    [   M  ]
      ++ where \spad{M = m mod d}.
    rowEchLocal    : (M, R) -> M
      ++ rowEchLocal(m,p) computes a modular row-echelon form of m, finding
      ++ an appropriate modulus over a local ring where p is the only prime.
    rowEchelonLocal: (M, R, R) -> M
      ++ rowEchelonLocal(m, d, p) computes the row-echelon form of m
      ++ concatenated with d times the identity matrix
      ++ over a local ring where p is the only prime.
    normalizedDivide: (R, R) -> Record(quotient:R, remainder:R)
      ++ normalizedDivide(n,d) returns a normalized quotient and
      ++ remainder such that consistently unique representatives
      ++ for the residue class are chosen, e.g. positive remainders



  Implementation ==> add
    order   : (R, R) -> Z
    vconc   : (M, R) -> M
    non0    : (V, Z) -> Union(REC, "failed")
    nonzero?: V -> Boolean
    mkMat   : (M, List Z) -> M
    diagSubMatrix: M -> Union(Record(val:R, mat:M), "failed")
    determinantOfMinor: M -> R
    enumerateBinomial: (List Z, Z, Z) -> List Z

    nonzero? v == any?(s +-> s ^= 0, v)

-- returns [a, i, rown] if v = [0,...,0,a,0,...,0]
-- where a <> 0 and i is the index of a, "failed" otherwise.
    non0(v, rown) ==
      ans:REC
      allZero:Boolean := true
      for i in minIndex v .. maxIndex v repeat
        if qelt(v, i) ^= 0 then
          if allZero then
            allZero := false
            ans := [qelt(v, i), i, rown]
          else return "failed"
      allZero => "failed"
      ans

-- returns a matrix made from the non-zero rows of x whose row number
-- is not in l
    mkMat(x, l) ==
      empty?(ll := [parts row(x, i)
         for i in minRowIndex x .. maxRowIndex x |
           (not member?(i, l)) and nonzero? row(x, i)]$List(List R)) =>
```

```
              zero(1, ncols x)
      matrix ll

-- returns [m, d] where m = x with the zero rows and the rows of
-- the diagonal of d removed, if x has a diagonal submatrix of d's,
-- "failed" otherwise.
    diagSubMatrix x ==
      l  := [u::REC for i in minRowIndex x .. maxRowIndex x |
                                    (u := non0(row(x, i), i)) case REC]
      for a in removeDuplicates([r.val for r in l]$List(R)) repeat
        {[r.cl for r in l | r.val = a]$List(Z)}$Set(Z) =
          {[z for z in minColIndex x .. maxColIndex x]$List(Z)}$Set(Z)
            => return [a, mkMat(x, [r.rw for r in l | a = r.val])]
      "failed"

-- returns a non-zero determinant of a minor of x of rank equal to
-- the number of columns of x, if there is one, 0 otherwise
    determinantOfMinor x ==
-- do not compute a modulus for square matrices, since this is as expensive
-- as the Hermite reduction itself
      (nr := nrows x) <= (nc := ncols x) => 0
      lc := [i for i in minColIndex x .. maxColIndex x]$List(Integer)
      lr := [i for i in minRowIndex x .. maxRowIndex x]$List(Integer)
      for i in 1..(n := binomial(nr, nc)) repeat
        (d := determinant x(enumerateBinomial(lr, nc, i), lc)) ^= 0 =>
          j := i + 1 + (random()$Z rem (n - i))
          return gcd(d, determinant x(enumerateBinomial(lr, nc, j), lc))
      0

-- returns the i-th selection of m elements of l = (a1,...,an),
--                 /n\
-- where 1 <= i <= | |
--                 \m/
    enumerateBinomial(l, m, i) ==
      m1 := minIndex l - 1
      zero?(m := m - 1) => [l(m1 + i)]
      for j in 1..(n := #l) repeat
        i <= (b := binomial(n - j, m)) =>
          return concat(l(m1 + j), enumerateBinomial(rest(l, j), m, i))
        i := i - b
      error "Should not happen"

    rowEch x ==
      (u := diagSubMatrix x) case "failed" =>
        zero?(d := determinantOfMinor x) => rowEchelon x
        rowEchelon(x, d)
      rowEchelon(u.mat, u.val)

    vconc(y, m) ==
      vertConcat(diagonalMatrix new(ncols y, m)$V, map(s +-> s rem m, y))
```

```
order(m, p) ==
  zero? m => -1
  for i in 0.. repeat
    (mm := m exquo p) case "failed" => return i
    m := mm::R

if R has IntegerNumberSystem then
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
        qr := divide(n, d)
        qr.remainder >= 0 => qr
        d > 0 =>
            qr.remainder := qr.remainder + d
            qr.quotient := qr.quotient - 1
            qr
        qr.remainder := qr.remainder - d
        qr.quotient := qr.quotient + 1
        qr
else
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
        divide(n, d)

rowEchLocal(x,p) ==
  (u := diagSubMatrix x) case "failed" =>
    zero?(d := determinantOfMinor x) => rowEchelon x
    rowEchelonLocal(x, d, p)
  rowEchelonLocal(u.mat, u.val, p)

rowEchelonLocal(y, m, p) ==
    m := p**(order(m,p)::NonNegativeInteger)
    x     := vconc(y, m)
    nrows := maxRowIndex x
    ncols := maxColIndex x
    minr  := i := minRowIndex x
    for j in minColIndex x .. ncols repeat
      if i > nrows then leave x
      rown := minr - 1
      pivord : Integer
      npivord : Integer
      for k in i .. nrows repeat
        qelt(x,k,j) = 0 => "next k"
        npivord := order(qelt(x,k,j),p)
        (rown = minr - 1) or (npivord  <  pivord) =>
                rown := k
                pivord := npivord
      rown = minr - 1 => "enuf"
      x := swapRows_!(x, i, rown)
      (a, b, d) := extendedEuclidean(qelt(x,i,j), m)
      qsetelt_!(x,i,j,d)
      pivot := d
```

```
      for k in j+1 .. ncols repeat
        qsetelt_!(x,i,k, a * qelt(x,i,k) rem m)
      for k in i+1 .. nrows repeat
        zero? qelt(x,k,j) => "next k"
        q := (qelt(x,k,j) exquo pivot) :: R
        for k1 in j+1 .. ncols repeat
          v2 := (qelt(x,k,k1) - q * qelt(x,i,k1)) rem m
          qsetelt_!(x, k, k1, v2)
        qsetelt_!(x, k, j, 0)
      for k in minr .. i-1 repeat
        zero? qelt(x,k,j) => "enuf"
        qr    := normalizedDivide(qelt(x,k,j), pivot)
        qsetelt_!(x,k,j, qr.remainder)
        for k1 in j+1 .. ncols x repeat
          qsetelt_!(x,k,k1,
                 (qelt(x,k,k1) - qr.quotient * qelt(x,i,k1)) rem m)
      i := i+1
    x

if R has Field then
  rowEchelon(y, m) == rowEchelon vconc(y, m)

else

  rowEchelon(y, m) ==
    x     := vconc(y, m)
    nrows := maxRowIndex x
    ncols := maxColIndex x
    minr  := i := minRowIndex x
    for j in minColIndex x .. ncols repeat
      if i > nrows then leave
      rown := minr - 1
      for k in i .. nrows repeat
        if (qelt(x,k,j) ^= 0) and ((rown = minr - 1) or
              sizeLess?(qelt(x,k,j), qelt(x,rown,j))) then rown := k
      rown = minr - 1 => "next j"
      x := swapRows_!(x, i, rown)
      for k in i+1 .. nrows repeat
        zero? qelt(x,k,j) => "next k"
        (a, b, d) := extendedEuclidean(qelt(x,i,j), qelt(x,k,j))
        (b1, a1) :=
           ((qelt(x,i,j) exquo d)::R, (qelt(x,k,j) exquo d)::R)
        -- a*b1+a1*b = 1
        for k1 in j+1 .. ncols repeat
          v1 := (a  * qelt(x,i,k1) +  b * qelt(x,k,k1)) rem m
          v2 := (b1 * qelt(x,k,k1) - a1 * qelt(x,i,k1)) rem m
          qsetelt_!(x, i, k1, v1)
          qsetelt_!(x, k, k1, v2)
        qsetelt_!(x, i, j, d)
        qsetelt_!(x, k, j, 0)
```

```
      un := unitNormal qelt(x,i,j)
      qsetelt_!(x,i,j,un.canonical)
      if un.associate ^= 1 then for jj in (j+1)..ncols repeat
          qsetelt_!(x,i,jj,un.associate * qelt(x,i,jj))

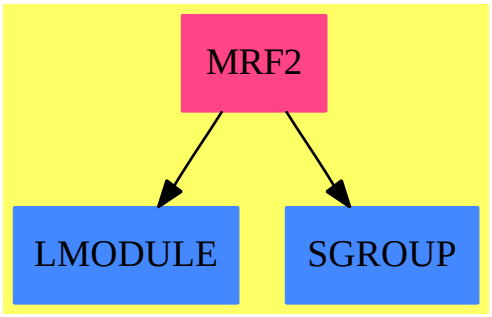
      xij := qelt(x,i,j)
      for k in minr .. i-1 repeat
        zero? qelt(x,k,j) => "next k"
        qr    := normalizedDivide(qelt(x,k,j), xij)
        qsetelt_!(x,k,j, qr.remainder)
        for k1 in j+1 .. ncols x repeat
          qsetelt_!(x,k,k1,
                  (qelt(x,k,k1) - qr.quotient * qelt(x,i,k1)) rem m)
      i := i+1
    x
```

— MHROWRED.dotabb —

```
"MHROWRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MHROWRED"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"MHROWRED" -> "IVECTOR"
```

14.20 package MRF2 MonoidRingFunctions2

14.20.1 MonoidRingFunctions2 (MRF2)



Exports:
map

— **package MRF2 MonoidRingFunctions2** —

```
)abbrev package MRF2 MonoidRingFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 14 May 1991
++ Date Last Updated: 14 May 1991
++ Basic Operations: map
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: monoid ring, group ring, change of coefficient domain
++ References:
++ Description:
++ MonoidRingFunctions2 implements functions between
++ two monoid rings defined with the same monoid over different rings.

MonoidRingFunctions2(R,S,M) : Exports == Implementation where
    R  : Ring
    S  : Ring
    M  : Monoid
    Exports ==> with
      map: (R -> S, MonoidRing(R,M)) -> MonoidRing(S,M)
        ++ map(f,u) maps f onto the coefficients f the element
        ++ u of the monoid ring to create an element of a monoid
        ++ ring with the same monoid b.
    Implementation ==> add
      map(fn, u) ==
        res : MonoidRing(S,M) := 0
        for te in terms u repeat
          res := res + monomial(fn(te.coef), te.monom)
        res
```

———-

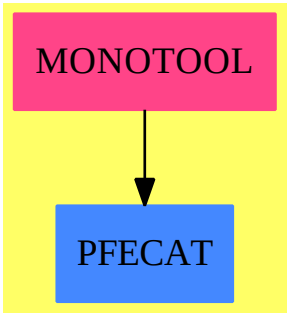
— **MRF2.dotabb** —

```
"MRF2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MRF2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"MRF2" -> "LMODULE"
"MRF2" -> "SGROUP"
```

———-

14.21 package MONOTOOL MonomialExtensionTools

14.21.1 MonomialExtensionTools (MONOTOOL)



Exports:
decompose normalDenom split splitSquarefree

— package MONOTOOL MonomialExtensionTools —

```
)abbrev package MONOTOOL MonomialExtensionTools
++ Author: Manuel Bronstein
++ Date Created: 18 August 1992
++ Date Last Updated: 3 June 1993
++ Description:
++ Tools for handling monomial extensions.

MonomialExtensionTools(F, UP): Exports == Implementation where
  F : Field
  UP: UnivariatePolynomialCategory F

  RF ==> Fraction UP
  FR ==> Factored UP

  Exports ==> with
    split      : (UP, UP -> UP) -> Record(normal:UP, special:UP)
      ++ split(p, D) returns \spad{[n,s]} such that \spad{p = n s},
      ++ all the squarefree factors of n are normal w.r.t. D,
      ++ and s is special w.r.t. D.
      ++ D is the derivation to use.
    splitSquarefree: (UP, UP -> UP) -> Record(normal:FR, special:FR)
      ++ splitSquarefree(p, D) returns
      ++ \spad{[n_1 n_2\^2 ... n_m\^m, s_1 s_2\^2 ... s_q\^q]} such that
      ++ \spad{p = n_1 n_2\^2 ... n_m\^m s_1 s_2\^2 ... s_q\^q}, each
      ++ \spad{n_i} is normal w.r.t. D and each \spad{s_i} is special
      ++ w.r.t D.
      ++ D is the derivation to use.
```

```
    normalDenom: (RF, UP -> UP) -> UP
      ++ normalDenom(f, D) returns the product of all the normal factors
      ++ of \spad{denom(f)}.
      ++ D is the derivation to use.
    decompose  : (RF, UP -> UP) -> Record(poly:UP, normal:RF, special:RF)
      ++ decompose(f, D) returns \spad{[p,n,s]} such that \spad{f = p+n+s},
      ++ all the squarefree factors of \spad{denom(n)} are normal w.r.t. D,
      ++ \spad{denom(s)} is special w.r.t. D,
      ++ and n and s are proper fractions (no pole at infinity).
      ++ D is the derivation to use.

  Implementation ==> add
    normalDenom(f, derivation) == split(denom f, derivation).normal

    split(p, derivation) ==
      pbar := (gcd(p, derivation p) exquo gcd(p, differentiate p))::UP
      zero? degree pbar => [p, 1]
      rec := split((p exquo pbar)::UP, derivation)
      [rec.normal, pbar * rec.special]

    splitSquarefree(p, derivation) ==
      s:Factored(UP) := 1
      n := s
      q := squareFree p
      for rec in factors q repeat
        r := rec.factor
        g := gcd(r, derivation r)
        if not ground? g then s := s * sqfrFactor(g, rec.exponent)
        h := (r exquo g)::UP
        if not ground? h then n := n * sqfrFactor(h, rec.exponent)
      [n, unit(q) * s]

    decompose(f, derivation) ==
      qr := divide(numer f, denom f)
-- rec.normal * rec.special = denom f
      rec := split(denom f, derivation)
-- eeu.coef1 * rec.normal + eeu.coef2 * rec.special = qr.remainder
-- and degree(eeu.coef1) < degree(rec.special)
-- and degree(eeu.coef2) < degree(rec.normal)
-- qr.remainder/denom(f) = eeu.coef1 / rec.special + eeu.coef2 / rec.normal
      eeu := extendedEuclidean(rec.normal, rec.special,
                               qr.remainder)::Record(coef1:UP, coef2:UP)
      [qr.quotient, eeu.coef2 / rec.normal, eeu.coef1 / rec.special]
```

———-

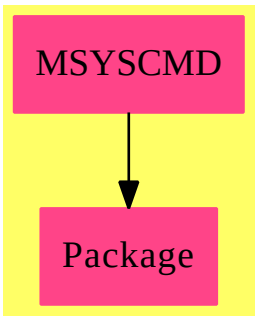
— MONOTOOL.dotabb —

```
"MONOTOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MONOTOOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MONOTOOL" -> "PFECAT"
```

———-

14.22 package MSYSCMD MoreSystemCommands

14.22.1 MoreSystemCommands (MSYSCMD)



Exports:
systemCommand

— **package MSYSCMD MoreSystemCommands** —

```
)abbrev package MSYSCMD MoreSystemCommands
++ Author:
++ Date Created:
++ Change History:
++ Basic Operations: systemCommand
++ Related Constructors:
++ Also See:
++ AMS Classification:
++ Keywords: command
++ Description:
++ \spadtype{MoreSystemCommands} implements an interface with the
++ system command facility. These are the commands that are issued
++ from source files or the system interpreter and they start with
++ a close parenthesis, e.g., the "what" commands.

MoreSystemCommands: public == private where

  public == with
```

```
    systemCommand: String -> Void
      ++ systemCommand(cmd) takes the string \spadvar{cmd} and passes
      ++ it to the runtime environment for execution as a system
      ++ command. Although various things may be printed, no usable
      ++ value is returned.

  private == add

    systemCommand cmd == doSystemCommand(cmd)$Lisp
```

———-

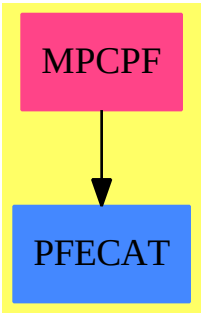
— **MSYSCMD.dotabb** —

```
"MSYSCMD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MSYSCMD"]
"Package" [color="#FF4488"]
"MSYSCMD" -> "Package"
```

———-

14.23 package MPCPF MPolyCatPolyFactorizer

14.23.1 MPolyCatPolyFactorizer (MPCPF)



Exports:
factor

— **package MPCPF MPolyCatPolyFactorizer** —

```
)abbrev package MPCPF MPolyCatPolyFactorizer
++ Author: P. Gianni
++ Date Created:
```

```
++ Date Last Updated: March 1995
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package exports a factor operation for multivariate polynomials
++ with coefficients which are polynomials over
++ some ring R over which we can factor. It is used internally by packages
++ such as the solve package which need to work with polynomials in a specific
++ set of variables with coefficients which are polynomials in all the other
++ variables.

MPolyCatPolyFactorizer(E,OV,R,PPR) : C == T
 where
  R     :   EuclideanDomain
  E     :   OrderedAbelianMonoidSup
    -- following type is required by PushVariables
  OV    :   OrderedSet  with
                convert : % -> Symbol
                  ++ convert(x) converts x to a symbol
                variable: Symbol -> Union(%, "failed")
                  ++ variable(s) makes an element from symbol s or fails.
  PR    ==> Polynomial R
  PPR   :   PolynomialCategory(PR,E,OV)
  NNI   ==> NonNegativeInteger
  ISY   ==> IndexedExponents Symbol
  SE    ==> Symbol
  UP    ==> SparseUnivariatePolynomial PR
  UPPR  ==> SparseUnivariatePolynomial PPR

  C  == with
     factor      :        PPR             ->   Factored PPR
       ++ factor(p) factors a polynomial with polynomial
       ++ coefficients.

             ---  Local Functions  ----
  T  == add

     import PushVariables(R,E,OV,PPR)

        ----  factorization of p ----
     factor(p:PPR) : Factored PPR ==
       ground? p => nilFactor(p,1)
       c := content p
       p := (p exquo c)::PPR
       vars:List OV :=variables p
       g:PR:=retract pushdown(p, vars)
```

```
flist := factor(g)$GeneralizedMultivariateFactorize(Symbol,ISY,R,R,PR)
ffact : List(Record(irr:PPR,pow:Integer))
ffact:=[[pushup(u.factor::PPR,vars),u.exponent] for u in factors flist]
fcont:=(unit flist)::PPR
nilFactor(c*fcont,1)*(_*/[primeFactor(ff.irr,ff.pow) for ff in ffact])
```

———-

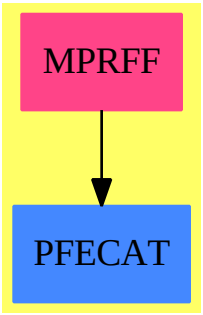
— **MPCPF.dotabb** —

```
"MPCPF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPCPF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPCPF" -> "PFECAT"
```

———-

14.24 package MPRFF MPolyCatRationalFunctionFactorizer

14.24.1 MPolyCatRationalFunctionFactorizer (MPRFF)



Exports:

factor	pushdown	pushup	pushdterm	pushucoef
pushuconst	totalfract			

— **package MPRFF MPolyCatRationalFunctionFactorizer** —

```
)abbrev package MPRFF MPolyCatRationalFunctionFactorizer
++ Author: P. Gianni
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
```

```
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package exports a factor operation for multivariate polynomials
++ with coefficients which are rational functions over
++ some ring R over which we can factor. It is used internally by packages
++ such as primary decomposition which need to work with polynomials
++ with rational function coefficients, i.e. themselves fractions of
++ polynomials.

MPolyCatRationalFunctionFactorizer(E,OV,R,PRF) : C == T
 where
  R     :   IntegralDomain
  F     ==> Fraction Polynomial R
  RN    ==> Fraction Integer
  E     :   OrderedAbelianMonoidSup
  OV    :   OrderedSet  with
                convert : % -> Symbol
                  ++ convert(x) converts x to a symbol
  PRF   :   PolynomialCategory(F,E,OV)
  NNI   ==> NonNegativeInteger
  P     ==> Polynomial R
  ISE   ==> IndexedExponents  SE
  SE    ==> Symbol
  UP    ==> SparseUnivariatePolynomial P
  UF    ==> SparseUnivariatePolynomial F
  UPRF  ==> SparseUnivariatePolynomial PRF
  QuoForm   ==> Record(sup:P,inf:P)

  C  == with
     totalfract  :        PRF             ->   QuoForm
       ++ totalfract(prf) takes a polynomial whose coefficients are
       ++ themselves fractions of polynomials and returns a record
       ++ containing the numerator and denominator resulting from
       ++ putting prf over a common denominator.
     pushdown    :      (PRF,OV)          ->   PRF
       ++ pushdown(prf,var) pushes all top level occurences of the
       ++ variable var into the coefficient domain for the polynomial prf.
     pushdterm   :      (UPRF,OV)         ->   PRF
       ++ pushdterm(monom,var) pushes all top level occurences of the
       ++ variable var into the coefficient domain for the monomial monom.
     pushup      :      (PRF,OV)          ->   PRF
       ++ pushup(prf,var) raises all occurences of the
       ++ variable var in the coefficients of the polynomial prf
       ++ back to the polynomial level.
     pushucoef   :       (UP,OV)          ->   PRF
       ++ pushucoef(upoly,var) converts the anonymous univariate
```

```
      ++ polynomial upoly to a polynomial in var over rational functions.
    pushuconst  :        (F,OV)          ->   PRF
      ++ pushuconst(r,var) takes a rational function and raises
      ++ all occurances of the variable var to the polynomial level.
    factor      :        PRF             ->   Factored PRF
      ++ factor(prf) factors a polynomial with rational function
      ++ coefficients.

            ---  Local Functions  ----
  T  == add

       ----  factorization of p ----
    factor(p:PRF) : Factored PRF ==
      truelist:List OV :=variables p
      tp:=totalfract(p)
      nump:P:= tp.sup
      denp:F:=inv(tp.inf ::F)
      ffact : List(Record(irr:PRF,pow:Integer))
      flist:Factored P
      if R is Fraction Integer then
        flist:=
          ((factor nump)$MRationalFactorize(ISE,SE,Integer,P))
                        pretend (Factored P)
      else
        if R has FiniteFieldCategory  then
           flist:= ((factor nump)$MultFiniteFactorize(SE,ISE,R,P))
                   pretend (Factored P)

        else
           if R has Field then error "not done yet"

           else
             if R has CharacteristicZero then
               flist:= ((factor nump)$MultivariateFactorize(SE,ISE,R,P))
                                              pretend (Factored P)
             else error "can't happen"
      ffact:=[[u.factor::F::PRF,u.exponent] for u in factors flist]
      fcont:=(unit flist)::F::PRF
      for x in truelist repeat
        fcont:=pushup(fcont,x)
        ffact:=[[pushup(ff.irr,x),ff.pow] for ff in ffact]
      (denp*fcont)*(_*/[primeFactor(ff.irr,ff.pow) for ff in ffact])


-- the following functions are used to "push" x in the coefficient ring -

       ----  push x in the coefficient domain for a polynomial ----
    pushdown(g:PRF,x:OV) : PRF ==
      ground? g => g
      rf:PRF:=0$PRF
```

```
    ug:=univariate(g,x)
    while ug^=0 repeat
      rf:=rf+pushdterm(ug,x)
      ug := reductum ug
    rf

   ----  push x in the coefficient domain for a term ----
  pushdterm(t:UPRF,x:OV):PRF ==
    n:=degree(t)
    cf:=monomial(1,convert x,n)$P :: F
    cf * leadingCoefficient t

            ----  push back the variable  ----
  pushup(f:PRF,x:OV) :PRF ==
    ground? f => pushuconst(retract f,x)
    v:=mainVariable(f)::OV
    g:=univariate(f,v)
    multivariate(map((y:PRF):PRF +-> pushup(y,x),g),v)

   ----  push x back from the coefficient domain ----
  pushuconst(r:F,x:OV):PRF ==
    xs:SE:=convert x
    degree(denom r,xs)>0 => error "bad polynomial form"
    inv((denom r)::F)*pushucoef(univariate(numer r,xs),x)


  pushucoef(c:UP,x:OV):PRF ==
    c = 0 => 0
    monomial((leadingCoefficient c)::F::PRF,x,degree c) +
              pushucoef(reductum c,x)


        ----  write p with a common denominator  ----

  totalfract(p:PRF) : QuoForm ==
    p=0 => [0$P,1$P]$QuoForm
    for x in variables p repeat p:=pushdown(p,x)
    g:F:=retract p
    [numer g,denom g]$QuoForm
```

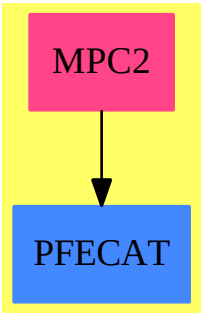
— MPRFF.dotabb —

```
"MPRFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPRFF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPRFF" -> "PFECAT"
```

———-

14.25 package MPC2 MPolyCatFunctions2

14.25.1 MPolyCatFunctions2 (MPC2)



Exports:
map reshape

— package MPC2 MPolyCatFunctions2 —

```
)abbrev package MPC2 MPolyCatFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 28 March 1990  (PG)
++ Description:
++ Utilities for MPolyCat

MPolyCatFunctions2(VarSet,E1,E2,R,S,PR,PS) : public == private where

  VarSet : OrderedSet
  E1     : OrderedAbelianMonoidSup
  E2     : OrderedAbelianMonoidSup
  R      : Ring
  S      : Ring
  PR     : PolynomialCategory(R,E1,VarSet)
  PS     : PolynomialCategory(S,E2,VarSet)
  SUPR   ==> SparseUnivariatePolynomial PR
  SUPS   ==> SparseUnivariatePolynomial PS

  public == with
    map:     (R -> S,PR) -> PS
      ++ map(f,p) \undocumented
    reshape: (List S, PR) -> PS
      ++ reshape(l,p) \undocumented
```

```
  private == add

    supMap:  (R -> S, SUPR) -> SUPS

    supMap(fn : R -> S, supr : SUPR): SUPS ==
      supr = 0 => monomial(fn(0$R) :: PS,0)$SUPS
      c : PS := map(fn,leadingCoefficient supr)$%
      monomial(c,degree supr)$SUPS + supMap(fn, reductum supr)

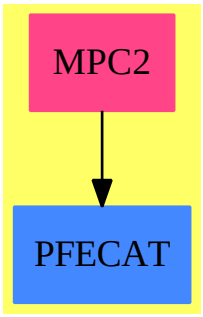
    map(fn : R -> S, pr : PR): PS ==
      varu : Union(VarSet,"failed") := mainVariable pr
      varu case "failed" =>  -- have a constant
        fn(retract pr) :: PS
      var : VarSet := varu :: VarSet
      supr : SUPR := univariate(pr,var)$PR
      multivariate(supMap(fn,supr),var)$PS
```

— **MPC2.dotabb** —

```
"MPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC2" -> "PFECAT"
```

14.26 package MPC3 MPolyCatFunctions3

14.26.1 MPolyCatFunctions3 (MPC3)



Exports:
map

— package MPC3 MPolyCatFunctions3 —

```
)abbrev package MPC3 MPolyCatFunctions3
++ Description:
++ This package has no description

MPolyCatFunctions3(Vars1,Vars2,E1,E2,R,PR1,PR2): C == T where
  E1   : OrderedAbelianMonoidSup
  E2   : OrderedAbelianMonoidSup
  Vars1: OrderedSet
  Vars2: OrderedSet
  R    : Ring
  PR1  : PolynomialCategory(R,E1,Vars1)
  PR2  : PolynomialCategory(R,E2,Vars2)

  C ==> with
    map: (Vars1 -> Vars2, PR1) -> PR2
      ++ map(f,x) \undocumented

  T ==> add

    map(f:Vars1 -> Vars2, p:PR1):PR2 ==
      (x1 := mainVariable p) case "failed" =>
        c:R:=(retract p)
        c::PR2
      up := univariate(p, x1::Vars1)
      x2 := f(x1::Vars1)
      ans:PR2 := 0
      while up ^= 0 repeat
        ans := ans + monomial(map(f,leadingCoefficient up),x2,degree up)
        up  := reductum up
      ans
```

———-

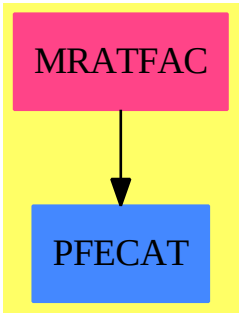
— MPC3.dotabb —

```
"MPC3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPC3"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC3" -> "PFECAT"
```

———-

14.27 package MRATFAC MRationalFactorize

14.27.1 MRationalFactorize (MRATFAC)



Exports:
factor

— package MRATFAC MRationalFactorize —

```
)abbrev package MRATFAC MRationalFactorize
++ Author: P. Gianni
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: MultivariateFactorize
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ MRationalFactorize contains the factor function for multivariate
++ polynomials over the quotient field of a ring R such that the package
++ MultivariateFactorize can factor multivariate polynomials over R.

MRationalFactorize(E,OV,R,P) : C == T
 where
  E   :   OrderedAbelianMonoidSup
  OV  :   OrderedSet
  R   :   Join(EuclideanDomain, CharacteristicZero)  -- with factor over R[x]
  FR  ==> Fraction R
  P   :   PolynomialCategory(FR,E,OV)
  MPR ==> SparseMultivariatePolynomial(R,OV)
  SUP ==> SparseUnivariatePolynomial

  C  == with
     factor      : P   ->  Factored P
```

```
      ++ factor(p) factors the multivariate polynomial p with coefficients
      ++ which are fractions of elements of R.

  T  == add
     IE     ==> IndexedExponents OV
     PCLFRR ==> PolynomialCategoryLifting(E,OV,FR,P,MPR)
     PCLRFR ==> PolynomialCategoryLifting(IE,OV,R,MPR,P)
     MFACT  ==> MultivariateFactorize(OV,IE,R,MPR)
     UPCF2  ==> UnivariatePolynomialCategoryFunctions2

     numer1(c:FR): MPR   == (numer c) :: MPR
     numer2(pol:P) : MPR == map(coerce,numer1,pol)$PCLFRR
     coerce1(d:R) : P == (d::FR)::P
     coerce2(pp:MPR) :P == map(coerce,coerce1,pp)$PCLRFR

     factor(p:P) : Factored P ==
       pden:R:=lcm([denom c for c in coefficients p])
       pol :P:= (pden::FR)*p
       ipol:MPR:= map(coerce,numer1,pol)$PCLFRR
       ffact:=(factor ipol)$MFACT
       (1/pden)*map(coerce,coerce1,(unit ffact))$PCLRFR *
           _*/[primeFactor(map(coerce,coerce1,u.factor)$PCLRFR,
                           u.exponent) for u in factors ffact]
```

———-

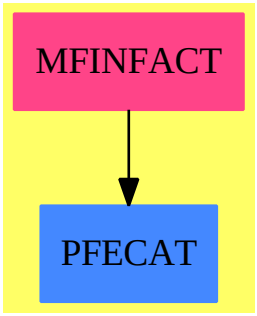
— MRATFAC.dotabb —

```
"MRATFAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MRATFAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MRATFAC" -> "PFECAT"
```

———-

14.28 package MFINFACT MultFiniteFactorize

14.28.1 MultFiniteFactorize (MFINFACT)



Exports:
factor

— package MFINFACT MultFiniteFactorize —

```
)abbrev package MFINFACT MultFiniteFactorize
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: 19 March 1992
++ Basic Functions:
++ Related Constructors: PrimeField, FiniteField, Polynomial
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Package for factorization of multivariate polynomials over finite fields.

MultFiniteFactorize(OV,E,F,PG) : C == T
 where
  F          :   FiniteFieldCategory
  OV         :   OrderedSet
  E          :   OrderedAbelianMonoidSup
  PG         :   PolynomialCategory(F,E,OV)
  SUP        ==> SparseUnivariatePolynomial
  R          ==> SUP F
  P          ==> SparseMultivariatePolynomial(R,OV)
  Z          ==> Integer
  FFPOLY     ==> FiniteFieldPolynomialPackage(F)
  MParFact   ==> Record(irr:P,pow:Z)
  MFinalFact ==> Record(contp:R,factors:List MParFact)
  SUParFact    ==> Record(irr:SUP P,pow:Z)
```

```
SUPFinalFact ==> Record(contp:R,factors:List SUParFact)

              --  contp   =  content,
              --  factors =  List of irreducible factors with exponent

C == with

  factor      : PG     ->  Factored PG
    ++ factor(p) produces the complete factorization of the multivariate
    ++ polynomial p over a finite field.
  factor      : SUP PG     ->  Factored SUP PG
    ++ factor(p) produces the complete factorization of the multivariate
    ++ polynomial p over a finite field. p is represented as a univariate
    ++ polynomial with multivariate coefficients over a finite field.

T == add

  import LeadingCoefDetermination(OV,IndexedExponents OV,R,P)
  import MultivariateLifting(IndexedExponents OV,OV,R,P)
  import FactoringUtilities(IndexedExponents OV,OV,R,P)
  import FactoringUtilities(E,OV,F,PG)
  import GenExEuclid(R,SUP R)

  NNI       ==> NonNegativeInteger
  L         ==> List
  UPCF2     ==> UnivariatePolynomialCategoryFunctions2
  LeadFact  ==> Record(polfac:L P,correct:R,corrfact:L SUP R)
  ContPrim  ==> Record(cont:P,prim:P)
  ParFact   ==> Record(irr:SUP R,pow:Z)
  FinalFact ==> Record(contp:R,factors:L ParFact)
  NewOrd    ==> Record(npol:SUP P,nvar:L OV,newdeg:L NNI)
  Valuf     ==> Record(inval:L L R,unvfact:L SUP R,lu:R,complead:L R)

               ----  Local Functions  ----
  ran       :                  Z             -> R
  mFactor   :               (P,Z)            -> MFinalFact
  supFactor :             (SUP P,Z)          -> SUPFinalFact
  mfconst   :       (SUP P,Z,L OV,L NNI)     -> L SUP P
  mfpol     :       (SUP P,Z,L OV,L NNI)     -> L SUP P
  varChoose :          (P,L OV,L NNI)        -> NewOrd
  simplify  :         (P,Z,L OV,L NNI)       -> MFinalFact
  intChoose :       (SUP P,L OV,R,L P,L L R)  -> Valuf
  pretest   :         (P,NNI,L OV,L R)       -> FinalFact
  checkzero :           (SUP P,SUP R)        -> Boolean
  pushdcoef :                PG              -> P
  pushdown  :               (PG,OV)          -> P
  pushupconst :              (R,OV)          -> PG
  pushup    :               (P,OV)          -> PG
  norm      :             L SUP R           -> Integer
  constantCase :       (P,L MParFact)        -> MFinalFact
```

```
pM          :             L SUP R            -> R
intfact     :     (SUP P,L OV,L NNI,MFinalFact,L L R) -> L SUP P

basicVar:OV:=NIL$Lisp pretend OV  -- variable for the basic step


convertPUP(lfg:MFinalFact): SUPFinalFact ==
  [lfg.contp,[[lff.irr ::SUP P,lff.pow]$SUParFact
               for lff in lfg.factors]]$SUPFinalFact

supFactor(um:SUP P,dx:Z) : SUPFinalFact ==
    degree(um)=0 => convertPUP(mFactor(ground um,dx))
    lvar:L OV:= "setUnion"/[variables cf for cf in coefficients um]
    lcont:SUP P
    lf:L SUP P

    flead : SUPFinalFact:=[0,empty()]$SUPFinalFact
    factorlist:L SUParFact :=empty()

    mdeg :=minimumDegree um     ---- is the Mindeg > 0? ----
    if mdeg>0 then
      f1:SUP P:=monomial(1,mdeg)
      um:=(um exquo f1)::SUP P
      factorlist:=cons([monomial(1,1),mdeg],factorlist)
      if degree um=0 then return
        lfg:=convertPUP mFactor(ground um, dx)
        [lfg.contp,append(factorlist,lfg.factors)]


    om:=map((p1:P):PG+->pushup(p1,basicVar),um)$UPCF2(P,SUP P,PG,SUP PG)
    sqfacs:=squareFree(om)
    lcont:=
      map((p1:PG):P+->pushdown(p1,basicVar),unit sqfacs)_
        $UPCF2(PG,SUP PG,P,SUP P)

                               ----   Factorize the content  ----
    if ground? lcont then
      flead:=convertPUP constantCase(ground lcont,empty())
    else
      flead:=supFactor(lcont,dx)

    factorlist:=flead.factors

                             ----  Make the polynomial square-free  ----
    sqqfact:=[[map((p:PG):P+->pushdown(p,basicVar),ff.factor),ff.exponent]
                 for ff in factors sqfacs]

                   ---  Factorize the primitive square-free terms ---
    for fact in sqqfact repeat
      ffactor:SUP P:=fact.irr
```

```
      ffexp:=fact.pow
      ffcont:=content ffactor
      coefs := coefficients ffactor
      ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]
      if ground?(leadingCoefficient ffactor) then
         lf:= mfconst(ffactor,dx,lvar,ldeg)
      else lf:=mfpol(ffactor,dx,lvar,ldeg)
      auxfl:=[[lfp,ffexp]$SUParFact  for lfp in lf]
      factorlist:=append(factorlist,auxfl)
    lcfacs :=
       */[leadingCoefficient leadingCoefficient(f.irr)**((f.pow)::NNI)
                        for f in factorlist]
    [(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R,
                  factorlist]$SUPFinalFact

factor(um:SUP PG):Factored SUP PG ==
    lv:List OV:=variables um
    ld:=degree(um,lv)
    dx:="min"/ld
    basicVar:=lv.position(dx,ld)
    cm:=map((p1:PG):P+->pushdown(p1,basicVar),um)$UPCF2(PG,SUP PG,P,SUP P)
    flist := supFactor(cm,dx)
    pushupconst(flist.contp,basicVar)::SUP(PG) *
     (*/[primeFactor(
       map((p1:P):PG+->pushup(p1,basicVar),u.irr)$UPCF2(P,SUP P,PG,SUP PG),
             u.pow) for u in flist.factors])

mFactor(m:P,dx:Z) : MFinalFact ==
  ground?(m) => constantCase(m,empty())
  lvar:L OV:= variables m
  lcont:P
  lf:L SUP P
  flead : MFinalFact:=[1,empty()]$MFinalFact
  factorlist:L MParFact :=empty()
                             ---- is the Mindeg > 0? ----
  lmdeg :=minimumDegree(m,lvar)
  or/[n>0 for n in lmdeg] => simplify(m,dx,lvar,lmdeg)
                          ----  Make the polynomial square-free  ----
  om:=pushup(m,basicVar)
  sqfacs:=squareFree(om)
  lcont := pushdown(unit sqfacs,basicVar)

                             ----  Factorize the content  ----
  if ground? lcont then
    flead:=constantCase(lcont,empty())
  else
    flead:=mFactor(lcont,dx)
  factorlist:=flead.factors
  sqqfact:List Record(factor:P,exponent:Integer)
  sqqfact:=[[pushdown(ff.factor,basicVar),ff.exponent]
```

```
                                        for ff in factors sqfacs]
                  ---  Factorize the primitive square-free terms ---
for fact in sqqfact repeat
  ffactor:P:=fact.factor
  ffexp := fact.exponent
  ground? ffactor =>
    for lterm in constantCase(ffactor,empty()).factors repeat
      factorlist:=cons([lterm.irr,lterm.pow * ffexp], factorlist)
  lvar := variables ffactor
  x:OV:=lvar.1
  ldeg:=degree(ffactor,lvar)
       ---  Is the polynomial linear in one of the variables ? ---
  member?(1,ldeg) =>
    x:OV:=lvar.position(1,ldeg)
    lcont:= gcd coefficients(univariate(ffactor,x))
    ffactor:=(ffactor exquo lcont)::P
    factorlist:=cons([ffactor,ffexp]$MParFact,factorlist)
    for lcterm in mFactor(lcont,dx).factors repeat
     factorlist:=cons([lcterm.irr,lcterm.pow * ffexp], factorlist)

  varch:=varChoose(ffactor,lvar,ldeg)
  um:=varch.npol


  ldeg:=ldeg.rest
  lvar:=lvar.rest
  if varch.nvar.1 ^= x then
    lvar:= varch.nvar
    x := lvar.1
    lvar:=lvar.rest
    pc:= gcd coefficients um
    if pc^=1 then
      um:=(um exquo pc)::SUP P
      ffactor:=multivariate(um,x)
      for lcterm in mFactor(pc,dx).factors repeat
        factorlist:=cons([lcterm.irr,lcterm.pow*ffexp],factorlist)
    ldeg:= degree(ffactor,lvar)

  -- should be unitNormal if unified, but for now it is easier
  lcum:F:= leadingCoefficient leadingCoefficient
          leadingCoefficient um
  if lcum ^=1  then
    um:=((inv lcum)::R::P) * um
    flead.contp := (lcum::R) *flead.contp

  if ground?(leadingCoefficient um)
  then lf:= mfconst(um,dx,lvar,ldeg)
  else lf:=mfpol(um,dx,lvar,ldeg)
  auxfl:=[[multivariate(lfp,x),ffexp]$MParFact  for lfp in lf]
  factorlist:=append(factorlist,auxfl)
```

```
  flead.factors:= factorlist
  flead


pM(lum:L SUP R) : R ==
  x := monomial(1,1)$R
  for i in 1..size()$F repeat
     p := x + (index(i::PositiveInteger)$F) ::R
     testModulus(p,lum) => return p
  for e in 2.. repeat
      p :=  (createIrreduciblePoly(e::PositiveInteger))$FFPOLY
      testModulus(p,lum) => return p
      while not((q := nextIrreduciblePoly(p)$FFPOLY) case "failed") repeat
         p := q::SUP F
         if testModulus(p, lum)$GenExEuclid(R, SUP R) then return p

  ----  push x in the coefficient domain for a term ----
pushdcoef(t:PG):P ==
   map((f1:F):R+->coerce(f1)$R,t)$MPolyCatFunctions2(OV,E,
                                      IndexedExponents OV,F,R,PG,P)


          ----  internal function, for testing bad cases  ----
intfact(um:SUP P,lvar: L OV,ldeg:L NNI,
        tleadpol:MFinalFact,ltry:L L R):  L SUP P ==
  polcase:Boolean:=(not empty? tleadpol.factors )
  vfchoo:Valuf:=
    polcase =>
      leadpol:L P:=[ff.irr for ff in tleadpol.factors]
      intChoose(um,lvar,tleadpol.contp,leadpol,ltry)
    intChoose(um,lvar,1,empty(),empty())
  unifact:List SUP R := vfchoo.unvfact
  nfact:NNI := #unifact
  nfact=1 => [um]
  ltry:L L R:= vfchoo.inval
  lval:L R:=first ltry
  dd:= vfchoo.lu
  lpol:List P:=empty()
  leadval:List R:=empty()
  if polcase then
    leadval := vfchoo.complead
    distf := distFact(vfchoo.lu,unifact,tleadpol,leadval,lvar,lval)
    distf case "failed" =>
         return intfact(um,lvar,ldeg,tleadpol,ltry)
    dist := distf :: LeadFact
      -- check the factorization of leading coefficient
    lpol:= dist.polfac
    dd := dist.correct
    unifact:=dist.corrfact
  if dd^=1 then
```

```
        unifact := [dd*unifact.i for i in 1..nfact]
        um := ((dd**(nfact-1)::NNI)::P)*um
      (ffin:= lifting(um,lvar,unifact,lval,lpol,ldeg,pM(unifact)))
          case "failed" => intfact(um,lvar,ldeg,tleadpol,ltry)
      factfin: L SUP P:=ffin :: L SUP P
      if dd^=1 then
        factfin:=[primitivePart ff  for ff in  factfin]
      factfin

-- the following functions are used to "push" x in the coefficient ring -
              ----  push back the variable  ----
    pushup(f:P,x:OV) :PG ==
       ground? f => pushupconst((retract f)@R,x)
       rr:PG:=0
       while f^=0 repeat
         lf:=leadingMonomial f
         cf:=pushupconst(leadingCoefficient f,x)
         lvf:=variables lf
         rr:=rr+monomial(cf,lvf, degree(lf,lvf))$PG
         f:=reductum f
       rr

        ----  push x in the coefficient domain for a polynomial ----
    pushdown(g:PG,x:OV) : P ==
       ground? g => ((retract g)@F)::R::P
       rf:P:=0$P
       ug:=univariate(g,x)
       while ug^=0 repeat
         cf:=monomial(1,degree ug)$R
         rf:=rf+cf*pushdcoef(leadingCoefficient ug)
         ug := reductum ug
       rf

      ----  push x back from the coefficient domain ----
    pushupconst(r:R,x:OV):PG ==
       ground? r => (retract r)@F ::PG
       rr:PG:=0
       while r^=0 repeat
         rr:=rr+monomial((leadingCoefficient r)::PG,x,degree r)$PG
         r:=reductum r
       rr

    -- This function has to be added to Eucliden domain
    ran(k1:Z) : R ==
      --if R case Integer then random()$R rem (2*k1)-k1
      --else
      +/[monomial(random()$F,i)$R for i in 0..k1]

    checkzero(u:SUP P,um:SUP R) : Boolean ==
      u=0 => um =0
```

```
    um = 0 => false
    degree u = degree um => checkzero(reductum u, reductum um)
    false

            ---  Choose the variable of least degree  ---
  varChoose(m:P,lvar:L OV,ldeg:L NNI) : NewOrd ==
    k:="min"/[d for d in ldeg]
    k=degree(m,first lvar) =>
                          [univariate(m,first lvar),lvar,ldeg]$NewOrd
    i:=position(k,ldeg)
    x:OV:=lvar.i
    ldeg:=cons(k,delete(ldeg,i))
    lvar:=cons(x,delete(lvar,i))
    [univariate(m,x),lvar,ldeg]$NewOrd


  norm(lum: L SUP R): Integer == "max"/[degree lup for lup in lum]

        ---  Choose the values to reduce to the univariate case  ---
  intChoose(um:SUP P,lvar:L OV,clc:R,plist:L P,ltry:L L R) : Valuf ==
    -- declarations
    degum:NNI := degree um
    nvar1:=#lvar
    range:NNI:=0
    unifact:L SUP R
    ctf1 : R := 1
    testp:Boolean :=             -- polynomial leading coefficient
      plist = empty() => false
      true
    leadcomp,leadcomp1 : L R
    leadcomp:=leadcomp1:=empty()
    nfatt:NNI := degum+1
    lffc:R:=1
    lffc1:=lffc
    newunifact : L SUP R:=empty()
    leadtest:=true --- the lc test with polCase has to be performed
    int:L R:=empty()

  --  New sets of values are chosen until we find twice the
  --  same number of "univariate" factors:the set smaller in modulo is
  --  is chosen.
    while true repeat
     lval := [ ran(range) for i in 1..nvar1]
     member?(lval,ltry) => range:=1+range
     ltry := cons(lval,ltry)
     leadcomp1:=[retract eval(pol,lvar,lval) for pol in plist]
     testp and or/[unit? epl for epl in leadcomp1] => range:=range+1
     newm:SUP R:=completeEval(um,lvar,lval)
     degum ^= degree newm or minimumDegree newm ^=0 => range:=range+1
     lffc1:=content newm
```

```
      newm:=(newm exquo lffc1)::SUP R
      testp and leadtest and ^ polCase(lffc1*clc,#plist,leadcomp1)
                          => range:=range+1
      Dnewm := differentiate newm
      D2newm := map(differentiate, newm)
      degree(gcd [newm,Dnewm,D2newm])^=0 => range:=range+1
     -- if R has Integer then luniv:=henselFact(newm,false)$
     -- else
      lcnm:F:=1
       -- should be unitNormal if unified, but for now it is easier
      if (lcnm:=leadingCoefficient leadingCoefficient newm)^=1 then
        newm:=((inv lcnm)::R)*newm
      dx:="max"/[degree uc  for uc in coefficients newm]
      luniv:=generalTwoFactor(newm)$TwoFactorize(F)
      lunivf:= factors luniv
      nf:= #lunivf

      nf=0 or nf>nfatt => "next values"      ---  pretest failed ---

                      --- the univariate polynomial is irreducible ---
      if nf=1 then leave (unifact:=[newm])

      lffc1:=lcnm * retract(unit luniv)@R * lffc1

  --  the new integer give the same number of factors
      nfatt = nf =>
      -- if this is the first univariate factorization with polCase=true
      -- or if the last factorization has smaller norm and satisfies
      -- polCase
        if leadtest or
          ((norm unifact > norm [ff.factor for ff in lunivf]) and
            (^testp or polCase(lffc1*clc,#plist,leadcomp1))) then
               unifact:=[uf.factor for uf in lunivf]
               int:=lval
               lffc:=lffc1
               if testp then leadcomp:=leadcomp1
        leave "foundit"

  --  the first univariate factorization, inizialize
      nfatt > degum =>
        unifact:=[uf.factor for uf in lunivf]
        lffc:=lffc1
        if testp then leadcomp:=leadcomp1
        int:=lval
        leadtest := false
        nfatt := nf

      nfatt>nf =>  -- for the previous values there were more factors
        if testp then leadtest:=^polCase(lffc*clc,#plist,leadcomp)
        else leadtest:= false
```

```
      -- if polCase=true we can consider the univariate decomposition
      if ^leadtest then
        unifact:=[uf.factor for uf in lunivf]
        lffc:=lffc1
        if testp then leadcomp:=leadcomp1
        int:=lval
      nfatt := nf
    [cons(int,ltry),unifact,lffc,leadcomp]$Valuf


  constantCase(m:P,factorlist:List MParFact) : MFinalFact ==
  --if R case Integer then [const m,factorlist]$MFinalFact
  --else
    lunm:=distdfact((retract m)@R,false)$DistinctDegreeFactorize(F,R)
    [(lunm.cont)::R, append(factorlist,
         [[(pp.irr)::P,pp.pow] for pp in lunm.factors])]$MFinalFact

              ----  The polynomial has mindeg>0   ----

  simplify(m:P,dm:Z,lvar:L OV,lmdeg:L NNI):MFinalFact ==
    factorlist:L MParFact:=empty()
    pol1:P:= 1$P
    for x in lvar repeat
      i := lmdeg.(position(x,lvar))
      i=0 => "next value"
      pol1:=pol1*monomial(1$P,x,i)
      factorlist:=cons([x::P,i]$MParFact,factorlist)
    m := (m exquo pol1)::P
    ground? m => constantCase(m,factorlist)
    flead:=mFactor(m,dm)
    flead.factors:=append(factorlist,flead.factors)
    flead

              ----  m square-free,primitive,lc constant  ----
  mfconst(um:SUP P,dm:Z,lvar:L OV,ldeg:L NNI):L SUP P ==
    nsign:Boolean
    factfin:L SUP P:=empty()
    empty? lvar =>
        um1:SUP R:=map(ground,
            um)$UPCF2(P,SUP P,R,SUP R)
        lum:= generalTwoFactor(um1)$TwoFactorize(F)
        [map(coerce,lumf.factor)$UPCF2(R,SUP R,P,SUP P)
              for lumf in factors lum]
    intfact(um,lvar,ldeg,[0,empty()]$MFinalFact,empty())

            --- m is square-free,primitive,lc is a polynomial  ---
  mfpol(um:SUP P,dm:Z,lvar:L OV,ldeg:L NNI):L SUP P ==
    dist : LeadFact
    tleadpol:=mFactor(leadingCoefficient um,dm)
    intfact(um,lvar,ldeg,tleadpol,empty())
```

```
    factor(m:PG):Factored PG ==
       lv:=variables m
       lv=empty() => makeFR(m,empty() )
    -- reduce to multivariate over SUP
       ld:=[degree(m,x) for x in lv]
       dx:="min"/ld
       basicVar:=lv(position(dx,ld))
       cm:=pushdown(m,basicVar)
       flist := mFactor(cm,dx)
       pushupconst(flist.contp,basicVar) *
          (*/[primeFactor(pushup(u.irr,basicVar),u.pow)
                                                  for u in flist.factors])
```

———-

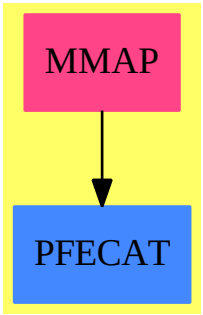
— **MFINFACT.dotabb** —

```
"MFINFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MFINFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MFINFACT" -> "PFECAT"
```

———-

14.29 package MMAP MultipleMap

14.29.1 MultipleMap (MMAP)



Exports:
map

— **package MMAP MultipleMap** —

```
)abbrev package MMAP MultipleMap
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Lifting of a map through 2 levels of polynomials;

MultipleMap(R1,UP1,UPUP1,R2,UP2,UPUP2): Exports == Implementation where
  R1   : IntegralDomain
  UP1  : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  R2   : IntegralDomain
  UP2  : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2

  Q1 ==> Fraction UP1
  Q2 ==> Fraction UP2

  Exports ==> with
    map: (R1 -> R2, UPUP1) -> UPUP2
      ++ map(f, p) lifts f to the domain of p then applies it to p.

  Implementation ==> add
    import UnivariatePolynomialCategoryFunctions2(R1, UP1, R2, UP2)

    rfmap: (R1 -> R2, Q1) -> Q2

    rfmap(f, q) == map(f, numer q) / map(f, denom q)

    map(f, p) ==
      map(x +-> rfmap(f,x),
          p)$UnivariatePolynomialCategoryFunctions2(Q1, UPUP1, Q2, UPUP2)
```

———-

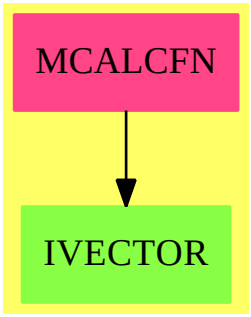
— **MMAP.dotabb** —

```
"MMAP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MMAP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MMAP" -> "PFECAT"
```

———-

14.30 package MCALCFN MultiVariableCalculusFunctions

14.30.1 MultiVariableCalculusFunctions (MCALCFN)



Exports:

bandedHessian	bandedJacobian	divergence	gradient	hessian
jacobian	laplacian			

— package MCALCFN MultiVariableCalculusFunctions —

```
)abbrev package MCALCFN MultiVariableCalculusFunctions
++ Author: Themos Tsikas, Grant Keady
++ Date Created: December 1992
++ Date Last Updated: June 1993
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{MultiVariableCalculusFunctions} Package provides several
++ functions for multivariable calculus.
++ These include gradient, hessian and jacobian, divergence and laplacian.
++ Various forms for banded and sparse storage of matrices are included.

MultiVariableCalculusFunctions(S,F,FLAF,FLAS) : Exports == Implementation where
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger

  S: SetCategory
  F: PartialDifferentialRing(S)
  FLAS: FiniteLinearAggregate(S)
    with finiteAggregate
  FLAF: FiniteLinearAggregate(F)
```

```
Exports ==> with
  gradient: (F,FLAS) -> Vector F
   ++ \spad{gradient(v,xlist)}
   ++ computes the gradient, the vector of first partial derivatives,
   ++ of the scalar field v,
   ++ v a function of the variables listed in xlist.
  divergence: (FLAF,FLAS) ->  F
   ++ \spad{divergence(vf,xlist)}
   ++ computes the divergence of the vector field vf,
   ++ vf a vector function of the variables listed in xlist.
  laplacian: (F,FLAS) -> F
   ++ \spad{laplacian(v,xlist)}
   ++ computes the laplacian of the scalar field v,
   ++ v a function of the variables listed in xlist.
  hessian: (F,FLAS) -> Matrix F
   ++ \spad{hessian(v,xlist)}
   ++ computes the hessian, the matrix of second partial derivatives,
   ++ of the scalar field v,
   ++ v a function of the variables listed in xlist.
  bandedHessian: (F,FLAS,NNI) -> Matrix F
   ++ \spad{bandedHessian(v,xlist,k)}
   ++ computes the hessian, the matrix of second partial derivatives,
   ++ of the scalar field v,
   ++ v a function of the variables listed in xlist,
   ++ k is the semi-bandwidth, the number of nonzero subdiagonals,
   ++ 2*k+1 being actual bandwidth.
   ++ Stores the nonzero band in lower triangle in a matrix,
   ++ dimensions k+1 by #xlist,
   ++ whose rows are the vectors formed by diagonal, subdiagonal, etc.
   ++ of the real, full-matrix, hessian.
   ++ (The notation conforms to LAPACK/NAG-F07 conventions.)
  -- At one stage it seemed a good idea to help the ASP<n> domains
  -- with the types of their input arguments and this led to the
  -- standard Gradient|Hessian|Jacobian functions.
  --standardJacobian: (Vector(F),List(S)) -> Matrix F
  -- ++ \spad{jacobian(vf,xlist)}
  -- ++ computes the jacobian, the matrix of first partial derivatives,
  -- ++ of the vector field vf,
  -- ++ vf a vector function of the variables listed in xlist.
  jacobian: (FLAF,FLAS) -> Matrix F
   ++ \spad{jacobian(vf,xlist)}
   ++ computes the jacobian, the matrix of first partial derivatives,
   ++ of the vector field vf,
   ++ vf a vector function of the variables listed in xlist.
  bandedJacobian: (FLAF,FLAS,NNI,NNI) -> Matrix F
   ++ \spad{bandedJacobian(vf,xlist,kl,ku)}
   ++ computes the jacobian, the matrix of first partial derivatives,
   ++ of the vector field vf,
   ++ vf a vector function of the variables listed in xlist,
```

```
    ++ kl is the number of nonzero subdiagonals,
    ++ ku is the number of nonzero superdiagonals,
    ++ kl+ku+1 being actual bandwidth.
    ++ Stores the nonzero band in a matrix,
    ++ dimensions kl+ku+1 by #xlist.
    ++ The upper triangle is in the top ku rows,
    ++ the diagonal is in row ku+1,
    ++ the lower triangle in the last kl rows.
    ++ Entries in a column in the band store correspond to entries
    ++ in same column of full store.
    ++ (The notation conforms to LAPACK/NAG-F07 conventions.)

  Implementation ==> add
    localGradient(v:F,xlist:List(S)):Vector(F) ==
       vector([D(v,x) for x in xlist])
    gradient(v,xflas) ==
       --xlist:List(S) := [xflas(i) for i in 1 .. maxIndex(xflas)]
       xlist:List(S) := parts(xflas)
       localGradient(v,xlist)
    localDivergence(vf:Vector(F),xlist:List(S)):F ==
       i: PI
       n: NNI
       ans: F
       -- Perhaps should report error if two args of min different
       n:= min(#(xlist),((maxIndex(vf))::NNI))$NNI
       ans:= 0
       for i in 1 .. n repeat ans := ans + D(vf(i),xlist(i))
       ans
    divergence(vf,xflas) ==
       xlist:List(S) := parts(xflas)
       i: PI
       n: NNI
       ans: F
       -- Perhaps should report error if two args of min different
       n:= min(#(xlist),((maxIndex(vf))::NNI))$NNI
       ans:= 0
       for i in 1 .. n repeat ans := ans + D(vf(i),xlist(i))
       ans
    laplacian(v,xflas) ==
       xlist:List(S) := parts(xflas)
       gv:Vector(F) := localGradient(v,xlist)
       localDivergence(gv,xlist)
    hessian(v,xflas) ==
       xlist:List(S) := parts(xflas)
       matrix([[D(v,[x,y]) for x in xlist] for y in xlist])
    --standardJacobian(vf,xlist) ==
    --   i: PI
    --   matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
    jacobian(vf,xflas) ==
       xlist:List(S) := parts(xflas)
```

```
      i: PI
      matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
   bandedHessian(v,xflas,k) ==
      xlist:List(S) := parts(xflas)
      j,iw: PI
      n: NNI
      bandM: Matrix F
      n:= #(xlist)
      bandM:= new(k+1,n,0)
      for j in 1 .. n repeat setelt(bandM,1,j,D(v,xlist(j),2))
      for iw in 2 .. (k+1) repeat (_
        for j in 1 .. (n-iw+1) repeat (_
          setelt(bandM,iw,j,D(v,[xlist(j),xlist(j+iw-1)])) ) )
      bandM
   jacobian(vf,xflas) ==
      xlist:List(S) := parts(xflas)
      i: PI
      matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
   bandedJacobian(vf,xflas,kl,ku) ==
      xlist:List(S) := parts(xflas)
      j,iw: PI
      n: NNI
      bandM: Matrix F
      n:= #(xlist)
      bandM:= new(kl+ku+1,n,0)
      for j in 1 .. n repeat setelt(bandM,ku+1,j,D(vf(j),xlist(j)))
      for iw in (ku+2) .. (ku+kl+1) repeat (_
        for j in 1 .. (n-iw+ku+1) repeat (_
          setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j))) ) )
      for iw in 1 .. ku repeat (_
        for j in (ku+2-iw) .. n repeat (_
          setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j))) ) )
      bandM
```

———-

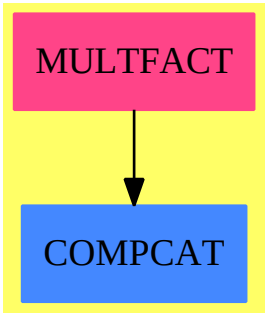
— **MCALCFN.dotabb** —

```
"MCALCFN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MCALCFN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"MCALCFN" -> "IVECTOR"
```

———-

14.31 package MULTFACT MultivariateFactorize

14.31.1 MultivariateFactorize (MULTFACT)



Exports:
factor

— package MULTFACT MultivariateFactorize —

```
)abbrev package MULTFACT MultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Basic Functions:
++ Related Constructors: MultFiniteFactorize, AlgebraicMultFact, UnivariateFactorize
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \spadtype{Integer} or \spadtype{Fraction Integer}.

MultivariateFactorize(OV,E,R,P) : C == T
 where
  R          :   Join(EuclideanDomain, CharacteristicZero)
                    -- with factor on R[x]
  OV         :   OrderedSet
  E          :   OrderedAbelianMonoidSup
  P          :   PolynomialCategory(R,E,OV)
  Z            ==> Integer
  MParFact     ==> Record(irr:P,pow:Z)
  USP          ==> SparseUnivariatePolynomial P
  SUParFact    ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact   ==> Record(contp:R,factors:List MParFact)
```

```
                --  contp   =  content,
                --  factors =  List of irreducible factors with exponent
  L          ==> List

  C == with
    factor      :      P  ->  Factored P
      ++ factor(p) factors the multivariate polynomial p over its coefficient
      ++ domain
    factor      :      USP  ->  Factored USP
      ++ factor(p) factors the multivariate polynomial p over its coefficient
      ++ domain where p is represented as a univariate polynomial with
      ++ multivariate coefficients
  T == add
    factor(p:P) : Factored P ==
      R is Fraction Integer =>
         factor(p)$MRationalFactorize(E,OV,Integer,P)
      R is Fraction Complex Integer =>
         factor(p)$MRationalFactorize(E,OV,Complex Integer,P)
      R is Fraction Polynomial Integer and OV has convert: % -> Symbol =>
         factor(p)$MPolyCatRationalFunctionFactorizer(E,OV,Integer,P)
      factor(p,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)

    factor(up:USP) : Factored USP ==
      factor(up,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)
```

———-

— MULTFACT.dotabb —

```
"MULTFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MULTFACT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"MULTFACT" -> "COMPCAT"
```

———-

14.32 package MLIFT MultivariateLifting

— package MLIFT MultivariateLifting —

```
)abbrev package MLIFT MultivariateLifting
++ Author : P.Gianni.
++ Description:
++ This package provides the functions for the multivariate "lifting", using
```

```
++ an algorithm of Paul Wang.
++ This package will work for every euclidean domain R which has property
++ F, i.e. there exists a factor operation in \spad{R[x]}.

MultivariateLifting(E,OV,R,P) : C == T
 where
  OV        :   OrderedSet
  E         :   OrderedAbelianMonoidSup
  R         :   EuclideanDomain  -- with property "F"
  Z         ==> Integer
  BP        ==> SparseUnivariatePolynomial R
  P         :   PolynomialCategory(R,E,OV)
  SUP       ==> SparseUnivariatePolynomial P
  NNI       ==> NonNegativeInteger
  Term      ==> Record(expt:NNI,pcoef:P)
  VTerm     ==> List Term
  Table     ==> Vector List BP
  L         ==> List

  C == with
    corrPoly:      (SUP,L OV,L R,L NNI,L SUP,Table,R) -> Union(L SUP,"failed")
      ++ corrPoly(u,lv,lr,ln,lu,t,r) \undocumented
    lifting:       (SUP,L OV,L BP,L R,L P,L NNI,R) -> Union(L SUP,"failed")
      ++ lifting(u,lv,lu,lr,lp,ln,r) \undocumented
    lifting1:      (SUP,L OV,L SUP,L R,L P,L VTerm,L NNI,Table,R) ->
                                                 Union(L SUP,"failed")
      ++ lifting1(u,lv,lu,lr,lp,lt,ln,t,r) \undocumented

  T == add
    GenExEuclid(R,BP)
    NPCoef(BP,E,OV,R,P)
    IntegerCombinatoricFunctions(Z)

    SUPF2   ==> SparseUnivariatePolynomialFunctions2

    DetCoef ==> Record(deter:L SUP,dterm:L VTerm,nfacts:L BP,
                       nlead:L P)

              ---   local functions   ---
    normalDerivM    :    (P,Z,OV)     ->  P
    normalDeriv     :     (SUP,Z)     ->  SUP
    subslead        :     (SUP,P)     ->  SUP
    subscoef        :   (SUP,L Term)  ->  SUP
    maxDegree       :   (SUP,OV)      ->  NonNegativeInteger


    corrPoly(m:SUP,lvar:L OV,fval:L R,ld:L NNI,flist:L SUP,
             table:Table,pmod:R):Union(L SUP,"failed") ==
      --  The correction coefficients are evaluated recursively.
      --   Extended Euclidean algorithm for the multivariate case.
```

```
  -- the polynomial is univariate  --
  #lvar=0 =>
    lp:=solveid(map(ground,m)$SUPF2(P,R),pmod,table)
    if lp case "failed" then return "failed"
    lcoef:= [map(coerce,mp)$SUPF2(R,P) for mp in lp::L BP]


  diff,ddiff,pol,polc:SUP
  listpolv,listcong:L SUP
  deg1:NNI:= ld.first
  np:NNI:= #flist
  a:P:= fval.first ::P
  y:OV:=lvar.first
  lvar:=lvar.rest
  listpolv:L SUP := [map((p1:P):P +-> eval(p1,y,a),f1) for f1 in flist]
  um:=map((p1:P):P +-> eval(p1,y,a),m)
  flcoef:=corrPoly(um,lvar,fval.rest,ld.rest,listpolv,table,pmod)
  if flcoef case "failed" then return "failed"
  else lcoef:=flcoef :: L SUP
  listcong:=[*/[flist.i for i in 1..np | i^=l] for l in 1..np]
  polc:SUP:= (monomial(1,y,1) - a)::SUP
  pol := 1$SUP
  diff:=m- +/[lcoef.i*listcong.i for i in 1..np]
  for l in 1..deg1 repeat
    if diff=0 then return lcoef
    pol := pol*polc
    (ddiff:=map((p:P):P+->eval(normalDerivM(p,l,y),y,a),diff)) = 0
          => "next l"
    fbeta := corrPoly(ddiff,lvar,fval.rest,ld.rest,listpolv,table,pmod)
    if fbeta case "failed" then return "failed"
    else beta:=fbeta :: L SUP
    lcoef := [lcoef.i+beta.i*pol  for i in 1..np]
    diff:=diff- +/[listcong.i*beta.i for i in 1..np]*pol
  lcoef

lifting1(m:SUP,lvar:L OV,plist:L SUP,vlist:L R,tlist:L P,_
  coeflist:L VTerm,listdeg:L NNI,table:Table,pmod:R) :Union(L SUP,"failed") ==
-- The factors of m (multivariate) are determined ,
-- We suppose to know the true univariate factors
-- some coefficients are determined
  conglist:L SUP:=empty()
  nvar : NNI:= #lvar
  pol,polc:P
  mc,mj:SUP
  testp:Boolean:= (not empty?(tlist))
  lalpha : L SUP := empty()
  tlv:L P:=empty()
  subsvar:L OV:=empty()
  subsval:L R:=empty()
```

```
    li:L OV := lvar
    ldeg:L NNI:=empty()
    clv:L VTerm:=empty()
    --j =#variables, i=#factors
    for j in 1..nvar repeat
      x  := li.first
      li := rest li
      conglist:= plist
      v := vlist.first
      vlist := rest vlist
      degj := listdeg.j
      ldeg := cons(degj,ldeg)
      subsvar:=cons(x,subsvar)
      subsval:=cons(v,subsval)

    --substitute the determined coefficients
      if testp then
        if j<nvar then
          tlv:=[eval(p,li,vlist) for p in tlist]
          clv:=[[[term.expt,eval(term.pcoef,li,vlist)]$Term
                 for term in clist] for clist  in coeflist]
        else (tlv,clv):=(tlist,coeflist)
        plist :=[subslead(p,lcp) for p in plist for lcp in tlv]
        if not(empty? coeflist) then
          plist:=[subscoef(tpol,clist)
                 for tpol in plist for clist in clv]
      mj := map((p1:P):P+->eval(p1,li,vlist),m)  --m(x1,..,xj,aj+1,..,an
      polc := x::P - v::P  --(xj-aj)
      pol:= 1$P
    --Construction of Rik, k in 1..right degree for xj+1
      for k in 1..degj repeat  --I can exit before
       pol := pol*polc
       mc := */[term for term in plist]-mj
       if mc=0 then leave "next var"
       --Modulus Dk
       mc:=map((p1:P):P +-> normalDerivM(p1,k,x),mc)
       (mc := map((p1:P):P +-> eval(p1,[x],[v]),mc))=0 => "next k"
       flalpha:=corrPoly(mc,subsvar.rest,subsval.rest,
                         ldeg.rest,conglist,table,pmod)
       if flalpha case "failed" then return "failed"
       else lalpha:=flalpha :: L SUP
       plist:=[term-alpha*pol for term in plist for alpha in lalpha]
      -- PGCD may call with a smaller valure of degj
      idegj:Integer:=maxDegree(m,x)
      for term in plist repeat idegj:=idegj -maxDegree(term,x)
      idegj < 0 => return "failed"
    plist
      --There are not extraneous factors

  maxDegree(um:SUP,x:OV):NonNegativeInteger ==
```

```
    ans:NonNegativeInteger:=0
    while um ^= 0 repeat
       ans:=max(ans,degree(leadingCoefficient um,x))
       um:=reductum um
    ans

  lifting(um:SUP,lvar:L OV,plist:L BP,vlist:L R,
          tlist:L P,listdeg:L NNI,pmod:R):Union(L SUP,"failed") ==
  -- The factors of m (multivariate) are determined, when the
  --  univariate true factors are known and some coefficient determined
    nplist:List SUP:=[map(coerce,pp)$SUPF2(R,P) for pp in plist]
    empty? tlist =>
      table:=tablePow(degree um,pmod,plist)
      table case "failed" => error "Table construction failed in MLIFT"
      lifting1(um,lvar,nplist,vlist,tlist,empty(),listdeg,table,pmod)
    ldcoef:DetCoef:=npcoef(um,plist,tlist)
    if not empty?(listdet:=ldcoef.deter) then
      if #listdet = #plist  then return listdet
      plist:=ldcoef.nfacts
      nplist:=[map(coerce,pp)$SUPF2(R,P) for pp in plist]
      um:=(um exquo */[pol for pol in listdet])::SUP
      tlist:=ldcoef.nlead
      tab:=tablePow(degree um,pmod,plist.rest)
    else tab:=tablePow(degree um,pmod,plist)
    tab case "failed" => error "Table construction failed in MLIFT"
    table:Table:=tab
    ffl:=lifting1(um,lvar,nplist,vlist,tlist,ldcoef.dterm,listdeg,table,pmod)
    if ffl case "failed" then return "failed"
    append(listdet,ffl:: L SUP)

  -- normalDerivM(f,m,x) = the normalized (divided by m!) m-th
  -- derivative with respect to x of the multivariate polynomial f
  normalDerivM(g:P,m:Z,x:OV) : P ==
   multivariate(normalDeriv(univariate(g,x),m),x)

  normalDeriv(f:SUP,m:Z) : SUP ==
   (n1:Z:=degree f) < m => 0$SUP
   n1=m => leadingCoefficient f :: SUP
   k:=binomial(n1,m)
   ris:SUP:=0$SUP
   n:Z:=n1
   while n>= m repeat
     while n1>n repeat
       k:=(k*(n1-m)) quo n1
       n1:=n1-1
     ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
     f:=reductum f
     n:=degree f
   ris
```

```
    subslead(m:SUP,pol:P):SUP ==
      dm:NNI:=degree m
      monomial(pol,dm)+reductum m

    subscoef(um:SUP,lterm:L Term):SUP ==
      dm:NNI:=degree um
      new:=monomial(leadingCoefficient um,dm)
      for k in dm-1..0 by -1 repeat
        i:NNI:=k::NNI
        empty?(lterm) or lterm.first.expt^=i =>
                              new:=new+monomial(coefficient(um,i),i)
        new:=new+monomial(lterm.first.pcoef,i)
        lterm:=lterm.rest
      new
```

———-

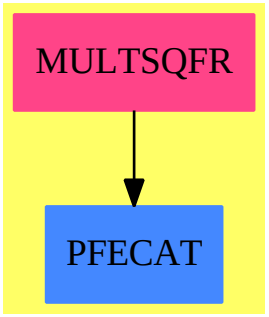
— **MLIFT.dotabb** —

```
"MLIFT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MLIFT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MLIFT" -> "PFECAT"
```

———-

14.33 package MULTSQFR MultivariateSquareFree

14.33.1 MultivariateSquareFree (MULTSQFR)



Exports:

check	coefChoose	compdegd	consnewpol	intChoose
lift	myDegree	normDeriv2	nsqfree	squareFree
squareFreePrim	univcase			

— package MULTSQFR MultivariateSquareFree —

```
)abbrev package MULTSQFR MultivariateSquareFree
++ Author : P.Gianni
++ Description:
++ This package provides the functions for the computation of the square
++ free decomposition of a multivariate polynomial.
++ It uses the package GenExEuclid for the resolution of
++ the equation \spad{Af + Bg = h} and its generalization to n polynomials
++ over an integral domain and the package \spad{MultivariateLifting}
++ for the "multivariate" lifting.

MultivariateSquareFree (E,OV,R,P) : C == T where
 Z   ==> Integer
 NNI ==> NonNegativeInteger
 R   : EuclideanDomain
 OV  : OrderedSet
 E   : OrderedAbelianMonoidSup
 P   : PolynomialCategory(R,E,OV)
 SUP ==> SparseUnivariatePolynomial P

 BP          ==> SparseUnivariatePolynomial(R)
 fUnion      ==> Union("nil","sqfr","irred","prime")
 ffSUP       ==> Record(flg:fUnion,fctr:SUP,xpnt:Integer)
 ffP         ==> Record(flg:fUnion,fctr:P,xpnt:Integer)
 FFE         ==> Record(factor:BP,exponent:Z)
 FFEP        ==> Record(factor:P,exponent:Z)
 FFES        ==> Record(factor:SUP,exponent:Z)
 Choice      ==> Record(upol:BP,Lval:List(R),Lfact:List FFE,ctpol:R)
 squareForm  ==> Record(unitPart:P,suPart:List FFES)
 Twopol      ==> Record(pol:SUP,polval:BP)
 UPCF2       ==> UnivariatePolynomialCategoryFunctions2


 C == with
   squareFree      :      P     -> Factored P
     ++ squareFree(p) computes the square free
     ++ decomposition of a multivariate polynomial p.
   squareFree      :      SUP     -> Factored SUP
     ++ squareFree(p) computes the square free
     ++ decomposition of a multivariate polynomial p presented as
     ++ a univariate polynomial with multivariate coefficients.
   squareFreePrim :      P     -> Factored P
     ++ squareFreePrim(p) compute the square free decomposition
     ++ of a primitive multivariate polynomial p.



                  ----  local functions  ----
```

```
   compdegd   :             List FFE                  -> Z
     ++ compdegd should be local
   univcase   :               (P,OV)                  -> Factored(P)
     ++ univcase should be local
   consnewpol :            (SUP,BP,Z)                 -> Twopol
     ++ consnewpol should be local
   nsqfree    :           (SUP,List(OV), List List R)  -> squareForm
     ++ nsqfree should be local
   intChoose  :           (SUP,List(OV),List List R)   -> Choice
     ++ intChoose should be local
   coefChoose :           (Z,Factored P)               -> P
     ++ coefChoose should be local
   check      :     (List(FFE),List(FFE))              -> Boolean
     ++ check should be local
   lift : (SUP,BP,BP,P,List(OV),List(NNI),List(R)) -> Union(List(SUP),"failed")
     ++ lift should be local
   myDegree   :       (SUP,List OV,NNI)              -> List NNI
     ++ myDegree should be local
   normDeriv2 :            (BP,Z)                    ->  BP
     ++ normDeriv2 should be local



 T == add

   pmod:R   :=  (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R


   import GenExEuclid()
   import MultivariateLifting(E,OV,R,P)
   import PolynomialGcdPackage(E,OV,R,P)
   import FactoringUtilities(E,OV,R,P)
   import IntegerCombinatoricFunctions(Z)


    ----  Are the univariate square-free decompositions consistent?  ----

     ----  new square-free algorithm for primitive polynomial  ----
   nsqfree(oldf:SUP,lvar:List(OV),ltry:List List R) : squareForm ==
     f:=oldf
     univPol := intChoose(f,lvar,ltry)
--     debug msg
--     if not empty? ltry then output("ltry =", (ltry::OutputForm))$OutputPackage
     f0:=univPol.upol
     --the polynomial is square-free
     f0=1$BP => [1$P,[[f,1]$FFES]]$squareForm
     lfact:List(FFE):=univPol.Lfact
     lval:=univPol.Lval
     ctf:=univPol.ctpol
     leadpol:Boolean:=false
```

```
sqdec:List FFES := empty()
exp0:Z:=0
unitsq:P:=1
lcf:P:=leadingCoefficient f
if ctf^=1 then
  f0:=ctf*f0
  f:=(ctf::P)*f
  lcf:=ctf*lcf
sqlead:List FFEP:= empty()
sqlc:Factored P:=1
if lcf^=1$P then
  leadpol:=true
  sqlc:=squareFree lcf
  unitsq:=unitsq*(unit sqlc)
  sqlead:= factors sqlc
lfact:=sort((z1:FFE,z2:FFE):Boolean +-> z1.exponent > z2.exponent,lfact)
while lfact^=[] repeat
  pfact:=lfact.first
  (g0,exp0):=(pfact.factor,pfact.exponent)
  lfact:=lfact.rest
  lfact=[] and exp0 =1 =>
    f := (f exquo (ctf::P))::SUP
    gg := unitNormal leadingCoefficient f
    sqdec:=cons([gg.associate*f,exp0],sqdec)
    return  [gg.unit, sqdec]$squareForm
  if ctf^=1 then g0:=ctf*g0
  npol:=consnewpol(f,f0,exp0)
  (d,d0):=(npol.pol,npol.polval)
  if leadpol then lcoef:=coefChoose(exp0,sqlc)
  else lcoef:=1$P
  ldeg:=myDegree(f,lvar,exp0::NNI)
  result:=lift(d,g0,(d0 exquo g0)::BP,lcoef,lvar,ldeg,lval)
  result case "failed" => return nsqfree(oldf,lvar,ltry)
  result0:SUP:= (result::List SUP).1
  r1:SUP:=result0**(exp0:NNI)
  if (h:=f exquo r1) case "failed" then return nsqfree(oldf,lvar,empty())
  sqdec:=cons([result0,exp0],sqdec)
  f:=h::SUP
  f0:=completeEval(h,lvar,lval)
  lcr:P:=leadingCoefficient result0
  if leadpol and lcr^=1$P then
    for lpfact in sqlead  while lcr^=1 repeat
      ground? lcr =>
        unitsq:=(unitsq exquo lcr)::P
        lcr:=1$P
      (h1:=lcr exquo lpfact.factor) case "failed" => "next"
      lcr:=h1::P
      lpfact.exponent:=(lpfact.exponent)-exp0
[((retract f) exquo ctf)::P,sqdec]$squareForm
```

```
squareFree(f:SUP) : Factored SUP ==
  degree f =0 =>
    fu:=squareFree retract f
    makeFR((unit fu)::SUP,[["sqfr",ff.fctr::SUP,ff.xpnt]
             for ff in factorList fu])
  lvar:= "setUnion"/[variables cf for cf in coefficients f]
  empty? lvar =>  -- the polynomial is univariate
    upol:=map(ground,f)$UPCF2(P,SUP,R,BP)
    usqfr:=squareFree upol
    makeFR(map(coerce,unit usqfr)$UPCF2(R,BP,P,SUP),
           [["sqfr",map(coerce,ff.fctr)$UPCF2(R,BP,P,SUP),ff.xpnt]
              for ff in factorList usqfr])

  lcf:=content f
  f:=(f exquo lcf) ::SUP
  lcSq:=squareFree lcf
  lfs:List ffSUP:=[["sqfr",ff.fctr ::SUP,ff.xpnt]
                    for ff in factorList lcSq]
  partSq:=nsqfree(f,lvar,empty())

  lfs:=append([["sqfr",fu.factor,fu.exponent]$ffSUP
                for fu in partSq.suPart],lfs)
  makeFR((unit lcSq * partSq.unitPart) ::SUP,lfs)

squareFree(f:P) : Factored P ==
  ground? f => makeFR(f,[])      ---   the polynomial is constant  ---

  lvar:List(OV):=variables(f)
  result1:List ffP:= empty()

  lmdeg :=minimumDegree(f,lvar)     ---       is the mindeg > 0 ? ---
  p:P:=1$P
  for im in 1..#lvar repeat
    (n:=lmdeg.im)=0 => "next im"
    y:=lvar.im
    p:=p*monomial(1$P,y,n)
    result1:=cons(["sqfr",y::P,n],result1)
  if p^=1$P then
    f := (f exquo p)::P
    if ground? f then return makeFR(f, result1)
    lvar:=variables(f)


  #lvar=1 =>                    ---  the polynomial is univariate ---
    result:=univcase(f,lvar.first)
    makeFR(unit result,append(result1,factorList result))

  ldeg:=degree(f,lvar)          ---  general case ---
  m:="min"/[j for j in ldeg|j^=0]
```

```
    i:Z:=1
    for j in ldeg while j>m repeat i:=i+1
    x:=lvar.i
    lvar:=delete(lvar,i)
    f0:=univariate (f,x)
    lcont:P:= content f0
    nsqfftot:=nsqfree((f0 exquo lcont)::SUP,lvar,empty())
    nsqff:List ffP:=[["sqfr",multivariate(fu.factor,x),fu.exponent]$ffP
                        for fu in nsqfftot.suPart]
    result1:=append(result1,nsqff)
    ground? lcont => makeFR(lcont*nsqfftot.unitPart,result1)
    sqlead:=squareFree(lcont)
    makeFR(unit sqlead*nsqfftot.unitPart,append(result1,factorList sqlead))

  -- Choose the integer for the evaluation.                          --
  -- If the polynomial is square-free the function returns upol=1. --

   intChoose(f:SUP,lvar:List(OV),ltry:List List R):Choice ==
     degf:= degree f
     try:NNI:=0
     nvr:=#lvar
     range:Z:=10
     lfact1:List(FFE):=[]
     lval1:List R := []
     lfact:List(FFE)
     ctf1:R:=1
     f1:BP:=1$BP
     d1:Z
     while range<10000000000 repeat
       range:=2*range
       lval:= [ran(range) for i in 1..nvr]
       member?(lval,ltry) => "new integer"
       ltry:=cons(lval,ltry)
       f0:=completeEval(f,lvar,lval)
       degree f0 ^=degf  => "new integer"
       ctf:=content f0
       lfact:List(FFE):=factors(squareFree((f0 exquo (ctf:R)::BP)::BP))

            ----  the univariate polynomial is square-free  ----
       if #lfact=1 and (lfact.1).exponent=1 then
         return [1$BP,lval,lfact,1$R]$Choice

       d0:=compdegd lfact
                 ----      inizialize lfact1      ----
       try=0 =>
         f1:=f0
         lfact1:=lfact
         ctf1:=ctf
         lval1:=lval
         d1:=d0
```

```
      try:=1
    d0=d1 =>
      return [f1,lval1,lfact1,ctf1]$Choice
    d0 < d1 =>
      try:=1
      f1:=f0
      lfact1:=lfact
      ctf1:=ctf
      lval1:=lval
      d1:=d0


     ----  Choose the leading coefficient for the lifting  ----
coefChoose(exp:Z,sqlead:Factored(P)) : P ==
  lcoef:P:=unit(sqlead)
  for term in factors(sqlead) repeat
    texp:=term.exponent
    texp<exp => "next term"
    texp=exp => lcoef:=lcoef*term.factor
    lcoef:=lcoef*(term.factor)**((texp quo exp)::NNI)
  lcoef

     ----  Construction of the polynomials for the lifting  ----
consnewpol(g:SUP,g0:BP,deg:Z):Twopol ==
  deg=1 => [g,g0]$Twopol
  deg:=deg-1
  [normalDeriv(g,deg),normDeriv2(g0,deg)]$Twopol

      ----  lift the univariate square-free factor  ----
lift(ud:SUP,g0:BP,g1:BP,lcoef:P,lvar:List(OV),
                    ldeg:List(NNI),lval:List(R)) : Union(List SUP,"failed") ==
  leadpol:Boolean:=false
  lcd:P:=leadingCoefficient ud
  leadlist:List(P):=empty()

  if ^ground?(leadingCoefficient ud) then
    leadpol:=true
    ud:=lcoef*ud
    lcg0:R:=leadingCoefficient g0
    if ground? lcoef then g0:=retract(lcoef) quo lcg0 *g0
    else g0:=(retract(eval(lcoef,lvar,lval)) quo lcg0) * g0
    g1:=lcg0*g1
    leadlist:=[lcoef,lcd]
  plist:=lifting(ud,lvar,[g0,g1],lval,leadlist,ldeg,pmod)
  plist case "failed" => "failed"
  (p0:SUP,p1:SUP):=((plist::List SUP).1,(plist::List SUP).2)
  if completeEval(p0,lvar,lval) ^= g0 then (p0,p1):=(p1,p0)
  [primitivePart p0,primitivePart p1]

            ----  the polynomial is univariate  ----
```

```
   univcase(f:P,x:OV) : Factored(P) ==
     uf := univariate f
     cf:=content uf
     uf :=(uf exquo cf)::BP
     result:Factored BP:=squareFree uf
     makeFR(multivariate(cf*unit result,x),
         [["sqfr",multivariate(term.factor,x),term.exponent]
           for term in factors result])

--   squareFreePrim(p:P) : Factored P ==
--     -- p is content free
--     ground? p => makeFR(p,[])      ---   the polynomial is constant  ---
--
--     lvar:List(OV):=variables p
--     #lvar=1 =>                     ---  the polynomial is univariate ---
--       univcase(p,lvar.first)
--     nsqfree(p,lvar,1)

   compdegd(lfact:List(FFE)) : Z ==
     ris:Z:=0
     for pfact in lfact repeat
       ris:=ris+(pfact.exponent -1)*degree pfact.factor
     ris

   normDeriv2(f:BP,m:Z) : BP ==
     (n1:Z:=degree f) < m => 0$BP
     n1=m => (leadingCoefficient f)::BP
     k:=binomial(n1,m)
     ris:BP:=0$BP
     n:Z:=n1
     while n>= m repeat
       while n1>n repeat
         k:=(k*(n1-m)) quo n1
         n1:=n1-1
       ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
       f:=reductum f
       n:=degree f
     ris

   myDegree(f:SUP,lvar:List OV,exp:NNI) : List NNI==
     [n quo exp for n in degree(f,lvar)]
```

———-

— **MULTSQFR.dotabb** —

```
"MULTSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MULTSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
```

```
"MULTSQFR" -> "PFECAT"
```

Chapter 15

Chapter N

15.1 package NAGF02 NagEigenPackage

— NagEigenPackage.help —

```
F02 -- Eigenvalues and Eigenvectors          Introduction -- F02
                          Chapter F02
                Eigenvalues and Eigenvectors

1. Scope of the Chapter

This chapter is concerned with computing

--    eigenvalues and eigenvectors of a matrix

--    eigenvalues and eigenvectors of generalized matrix
      eigenvalue problems

--    singular values and singular vectors of a matrix.

2. Background to the Problems

2.1. Eigenvalue Problems

In the most usual form of eigenvalue problem we are given a
square n by n matrix A and wish to compute (lambda) (an
eigenvalue) and x/=0 (an eigenvector) which satisfy the equation

                          Ax=(lambda)x

Such problems are called 'standard' eigenvalue problems in
contrast to 'generalized' eigenvalue problems where we wish to
```

satisfy the equation

Ax=(lambda)Bx

B also being a square n by n matrix.

Section 2.1.1 and Section 2.1.2 discuss, respectively, standard and generalized eigenvalue problems where the matrices involved are dense; Section 2.1.3 discusses both types of problem in the case where A and B are sparse (and symmetric).

2.1.1. Standard eigenvalue problems

Some of the routines in this chapter find all the n eigenvalues, some find all the n eigensolutions (eigenvalues and corresponding eigenvectors), and some find a selected group of eigenvalues and/or eigenvectors. The matrix A may be:

(i) general (real or complex)

(ii) real symmetric, or

(iii) complex Hermitian (so that if a_{ij} =(alpha)+i(beta) then a_{ji} =(alpha)-i(beta)).

In all cases the computation starts with a similarity transformation $S^{-1}AS=T$, where S is non-singular and is the product of fairly simple matrices, and T has an 'easier form' than A so that its eigensolutions are easily determined. The matrices A and T, of course, have the same eigenvalues, and if y is an eigenvector of T then Sy is the corresponding eigenvector of A.

In case (i) (general real or complex A), the selected form of T is an upper Hessenberg matrix (t_{ij} =0 if i-j>1) and S is the product of n-2 stabilised elementary transformation matrices. There is no easy method of computing selected eigenvalues of a Hessenberg matrix, so that all eigenvalues are always calculated. In the real case this computation is performed via the Francis QR algorithm with double shifts, and in the complex case by means of the LR algorithm. If the eigenvectors are required they are computed by back-substitution following the QR and LR algorithm.

In case (ii) (real and symmetric A) the selected simple form of T is a tridiagonal matrix (t_{ij} =0 if |i-j|>1), and S is the product

of n-2 orthogonal Householder transformation matrices. If only selected eigenvalues are required, they are obtained by the method of bisection using the Sturm sequence property, and the corresponding eigenvectors of T are computed by inverse iteration. If all eigenvalues are required, they are computed from T via the QL algorithm (an adaptation of the QR algorithm), and the corresponding eigenvectors of T are the product of the transformations for the QL reduction. In all cases the corresponding eigenvectors of A are recovered from the computation of x=Sy.

In case (iii) (complex Hermitian A) analogous transformations as in case (ii) are used. T has complex elements in off-diagonal positions, but a simple diagonal similarity transformation is then used to produce a real tridiagonal form, after which the QL algorithm and succeeding methods described in the previous paragraph are used to complete the solution.

2.1.2. Generalized eigenvalue problems

Here we distinguish as a special case those problems in which both A and B are symmetric and B is positive-definite and well-conditioned with respect to inversion (i.e., all the eigenvalues of B are significantly greater than zero). Such problems can be satisfactorily treated by first reducing them to case (ii) of Section 2.1.1 and then using the methods described there to
                                                        T
compute the eigensolutions. If B is factorized as LL (L lower triangular), then Ax=(lambda)Bx is equivalent to the standard
                                          -1   T -1   T
symmetric problem Ry=(lambda)y, where R=L A(L)   and y=L x. After finding an eigenvector y of R, the required x is computed
                           T
by back-substitution in y=L x.

For generalized problems of the form Ax=(lambda)Bx which do not fall into the special case, the QZ algorithm is provided.

In order to appreciate the domain in which this algorithm is appropriate we remark first that when B is non-singular the problem Ax=(lambda)Bx is fully equivalent to the problem
  -1
(B A)x=(lambda)x; both the eigenvalues and eigenvectors being the same. When A is non-singular Ax=(lambda)Bx is equivalent to
             -1
the problem (A B)x=(mu)x; the eigenvalues (mu) being the reciprocals of the required eigenvalues and the eigenvectors remaining the same. In theory then, provided at least one of the matrices A and B is non-singular, the generalized problem Ax=(lambda)Bx could be solved via the standard problem

Cx=(lambda)x with an appropriate matrix C, and as far as economy of effort is concerned this is quite satisfactory. However, in practice, for this reduction to be satisfactory from the standpoint of numerical stability, one requires more than the mere non-singularity of A or B. It is necessary that $B^{-1}A$ (or $A^{-1}B$) should not only exist but that B (or A) should be well-conditioned with respect to inversion. The nearer B (or A) is to singularity the more unsatisfactory $B^{-1}A$ (or $A^{-1}B$) will be as a vehicle for determining the required eigenvalues. Unfortunately one cannot counter ill-conditioning in B (or A) by computing $B^{-1}A$ (or $A^{-1}B$) accurately to single precision using iterative refinement. Well-determined eigenvalues of the original Ax=(lambda)Bx may be poorly determined even by the correctly rounded version of $B^{-1}A$ (or $A^{-1}B$). The situation may in some instances be saved by the observation that if Ax=(lambda)Bx then (A-kB)x=((lambda)-k)Bx. Hence if A-kB is non-singular we may solve the standard problem $[(A-kB)^{-1}B]x=(mu)x$ and for numerical stability we require only that (A-kB) be well-conditioned with respect to inversion.

In practice one may well be in a situation where no k is known for which (A-kB) is well-conditioned with respect to inversion and indeed (A-kB) may be singular for all k. The QZ algorithm is designed to deal directly with the problem Ax=(lambda)Bx itself and its performance is unaffected by singularity or near-singularity of A, B or A-kB.

2.1.3. Sparse symmetric problems

If the matrices A and B are large and sparse (i.e., only a small proportion of the elements are non-zero), then the methods described in the previous Section are unsuitable, because in reducing the problem to a simpler form, much of the sparsity of the problem would be lost; hence the computing time and the storage required would be very large. Instead, for symmetric problems, the method of simultaneous iteration may be used to determine selected eigenvalues and the corresponding eigenvectors. The routine provided has been designed to handle both symmetric and generalized symmetric problems.

2.2. Singular Value Problems

The singular value decomposition of an m by n real matrix A is

given by

$$A=QDP^{T},$$

where Q is an m by m orthogonal matrix, P is an n by n orthogonal matrix and D is an m by n diagonal matrix with non-negative diagonal elements. The first k==min(m,n) columns of Q and P are the left- and right-hand singular vectors of A and the k diagonal elements of D are the singular values.

When A is complex then the singular value decomposition is given by

$$A=QDP^{H},$$

where Q and P are unitary, P^{H} denotes the complex conjugate of P^{T} and D is as above for the real case.

If the matrix A has column means of zero, then AP is the matrix of principal components of A and the singular values are the square roots of the sample variances of the observations with respect to the principal components. (See also Chapter G03.)

Routines are provided to return the singular values and vectors of a general real or complex matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Discussion

There is one routine, F02FJF, which is designed for sparse symmetric eigenvalue problems, either standard or generalized. The remainder of the routines are designed for dense matrices.

3.2. Eigenvalue and Eigenvector Routines

These reduce the matrix A to a simpler form by a similarity transformation $S^{-1}AS=T$ where T is an upper Hessenberg or tridiagonal matrix, compute the eigensolutions of T, and then recover the eigenvectors of A via the matrix S. The eigenvectors are normalised so that

$$\sum_{r=1}^{n} |x_r|^2 = 1$$

r=1

x being the rth component of the eigenvector x, and so that the
 r
element of largest modulus is real if x is complex. For problems
of the type Ax=(lambda)Bx with A and B symmetric and B positive-
                                                      T
definite, the eigenvectors are normalised so that x Bx=1, x
always being real for such problems.

3.3. Singular Value and Singular Vector Routines

These reduce the matrix A to real bidiagonal form, B say, by
                          T
orthogonal transformations Q AP=B in the real case, and by
                       H
unitary transformations Q AP=B in the complex case, and the
singular values and vectors are computed via this bidiagonal
form. The singular values are returned in descending order.

3.4. Decision Trees

(i) Eigenvalues and Eigenvectors

Please see figure in printed Reference Manual

(ii) Singular Values and Singular Vectors

Please see figure in printed Reference Manual

F02 -- Eigenvalues and Eigenvectors Contents -- F02
Chapter F02

Eigenvalues and Eigenvectors

F02AAF All eigenvalues of real symmetric matrix

F02ABF All eigenvalues and eigenvectors of real symmetric matrix

F02ADF All eigenvalues of generalized real symmetric-definite
        eigenproblem

F02AEF All eigenvalues and eigenvectors of generalized real
        symmetric-definite eigenproblem

F02AFF All eigenvalues of real matrix

```
F02AGF  All eigenvalues and eigenvectors of real matrix

F02AJF  All eigenvalues of complex matrix

F02AKF  All eigenvalues and eigenvectors of complex matrix

F02AWF  All eigenvalues of complex Hermitian matrix

F02AXF  All eigenvalues and eigenvectors of complex Hermitian
        matrix

F02BBF  Selected eigenvalues and eigenvectors of real symmetric
        matrix

F02BJF  All eigenvalues and optionally eigenvectors of
        generalized eigenproblem by QZ algorithm, real matrices

F02FJF  Selected eigenvalues and eigenvectors of sparse symmetric
        eigenproblem

F02WEF  SVD of real matrix

F02XEF  SVD of complex matrix
```

%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AAF
        F02AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AAF calculates all the eigenvalues of a real symmetric matrix.

2. Specification

       SUBROUTINE F02AAF (A, IA, N, R, E, IFAIL)
       INTEGER          IA, N, IFAIL
       DOUBLE PRECISION A(IA,N), R(N), E(N)

3. Description

This routine reduces the real symmetric matrix A to a real
symmetric tridiagonal matrix using Householder's method. The
eigenvalues of the tridiagonal matrix are then determined using
the QL algorithm.
```

```
4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array              Input/Output
     On entry: the lower triangle of the n by n symmetric matrix
     A. The elements of the array above the diagonal need not be
     set. On exit: the elements of A below the diagonal are
     overwritten, and the rest of the array is unchanged.

 2:  IA -- INTEGER                                        Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02AAF is called.
     Constraint: IA >= N.

 3:  N -- INTEGER                                         Input
     On entry: n, the order of the matrix A.

 4:  R(N) -- DOUBLE PRECISION array                      Output
     On exit: the eigenvalues in ascending order.

 5:  E(N) -- DOUBLE PRECISION array                   Workspace

 6:  IFAIL -- INTEGER                              Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     Failure in F02AVF(*) indicating that more than 30*N
     iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the eigenvalues depends on the sensitivity of the
matrix to rounding errors produced in tridiagonalisation. For a
detailed error analysis see Wilkinson and Reinsch [1] pp 222 and
235.
```

```
8. Further Comments

                                                                    3
The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the real symmetric matrix:

                    ( 0.5  0.0  2.3 -2.6)
                    ( 0.0  0.5 -1.4 -0.7)
                    ( 2.3 -1.4  0.5  0.0).
                    (-2.6 -0.7  0.0  0.5)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F02 -- Eigenvalue and Eigenvectors                          F02ABF
        F02ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02ABF calculates all the eigenvalues and eigenvectors of a real
symmetric matrix.

2. Specification

       SUBROUTINE F02ABF (A, IA, N, R, V, IV, E, IFAIL)
       INTEGER          IA, N, IV, IFAIL
       DOUBLE PRECISION A(IA,N), R(N), V(IV,N), E(N)

3. Description

This routine reduces the real symmetric matrix A to a real
symmetric tridiagonal matrix by Householder's method. The
eigenvalues and eigenvectors are calculated using the QL
algorithm.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
```

```
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array                          Input
     On entry: the lower triangle of the n by n symmetric matrix
     A. The elements of the array above the diagonal need not be
     set. See also Section 8.

 2:  IA -- INTEGER                                              Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02ABF is called.
     Constraint: IA >= N.

 3:  N -- INTEGER                                               Input
     On entry: n, the order of the matrix A.

 4:  R(N) -- DOUBLE PRECISION array                            Output
     On exit: the eigenvalues in ascending order.

 5:  V(IV,N) -- DOUBLE PRECISION array                         Output
     On exit: the normalised eigenvectors, stored by columns;
     the ith column corresponds to the ith eigenvalue. The
     eigenvectors are normalised so that the sum of squares of
     the elements is equal to 1.

 6:  IV -- INTEGER                                              Input
     On entry:
     the first dimension of the array V as declared in the
     (sub)program from which F02ABF is called.
     Constraint: IV >= N.

 7:  E(N) -- DOUBLE PRECISION array                          Workspace

 8:  IFAIL -- INTEGER                                     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     Failure in F02AMF(*) indicating that more than 30*N
     iterations are required to isolate all the eigenvalues.
```

```
7. Accuracy

The eigenvectors are always accurately orthogonal but the
accuracy of the individual eigenvectors is dependent on their
inherent sensitivity to changes in the original matrix. For a
detailed error analysis see Wilkinson and Reinsch [1] pp 222 and
235.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters A and V, in which case the
eigenvectors will overwrite the original matrix. However this is
not standard Fortran 77, and may not work on all systems.

9. Example

To calculate all the eigenvalues and eigenvectors of the real
symmetric matrix:

                    ( 0.5  0.0  2.3 -2.6)
                    ( 0.0  0.5 -1.4 -0.7)
                    ( 2.3 -1.4  0.5  0.0).
                    (-2.6 -0.7  0.0  0.5)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors                          F02ADF
        F02ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02ADF calculates all the eigenvalues of  Ax=(lambda)Bx,  where A
is a real symmetric matrix and B is a real symmetric positive-
definite matrix.
```

2. Specification

```
      SUBROUTINE F02ADF (A, IA, B, IB, N, R, DE, IFAIL)
      INTEGER          IA, IB, N, IFAIL
      DOUBLE PRECISION A(IA,N), B(IB,N), R(N), DE(N)
```

3. Description

The problem is reduced to the standard symmetric eigenproblem using Cholesky's method to decompose B into triangular matrices, $B=LL^T$, where L is lower triangular. Then $Ax=(\lambda)Bx$ implies $(L^{-1}AL^{-T})(L^T x)=(\lambda)(L^T x)$; hence the eigenvalues of $Ax=(\lambda)Bx$ are those of $Py=(\lambda)y$ where P is the symmetric matrix $L^{-1}AL^{-T}$. Householder's method is used to tridiagonalise the matrix P and the eigenvalues are then found using the QL algorithm.

4. References

[1] Wilkinson J H and Reinsch C (1971) Handbook for Automatic Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
On entry: the upper triangle of the n by n symmetric matrix A. The elements of the array below the diagonal need not be set. On exit: the lower triangle of the array is overwritten. The rest of the array is unchanged.

2: IA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F02ADF is called.
Constraint: IA >= N.

3: B(IB,N) -- DOUBLE PRECISION array Input/Output
On entry: the upper triangle of the n by n symmetric positive-definite matrix B. The elements of the array below the diagonal need not be set. On exit: the elements below the diagonal are overwritten. The rest of the array is unchanged.

4: IB -- INTEGER Input
On entry:
the first dimension of the array B as declared in the (sub)program from which F02ADF is called.

```
     Constraint: IB >= N.

 5:  N -- INTEGER                                              Input
     On entry: n, the order of the matrices A and B.

 6:  R(N) -- DOUBLE PRECISION array                            Output
     On exit: the eigenvalues in ascending order.

 7:  DE(N) -- DOUBLE PRECISION array                        Workspace

 8:  IFAIL -- INTEGER                                     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     Failure in F01AEF(*); the matrix B is not positive-definite
     possibly due to rounding errors.

IFAIL= 2
     Failure in F02AVF(*), more than 30*N iterations are required
     to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-
conditioned with respect to inversion, the eigenvalues could be
inaccurately determined. For a detailed error analysis see
Wilkinson and Reinsch [1] pp 310, 222 and 235.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the general symmetric
eigenproblem Ax=(lambda) Bx where A is the symmetric matrix:

                    (0.5  1.5  6.6   4.8)
                    (1.5  6.5 16.2   8.6)
                    (6.6 16.2 37.6   9.8)
```

```
                          (4.8  8.6  9.8 -17.1)

and B is the symmetric positive-definite matrix:

                             (1  3  4  1)
                             (3 13 16 11)
                             (4 16 24 18).
                             (1 11 18 27)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AEF
       F02AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AEF calculates all the eigenvalues and eigenvectors of
Ax=(lambda)Bx,  where A is a real symmetric matrix and B is a
real symmetric positive-definite matrix.

2. Specification

       SUBROUTINE F02AEF (A, IA, B, IB, N, R, V, IV, DL, E, IFAIL)
       INTEGER          IA, IB, N, IV, IFAIL
       DOUBLE PRECISION A(IA,N), B(IB,N), R(N), V(IV,N), DL(N), E
      1                 (N)

3. Description

The problem is reduced to the standard symmetric eigenproblem
using Cholesky's method to decompose B into triangular matrices
    T
B=LL , where L is lower triangular. Then Ax=(lambda)Bx implies
  -1  -T   T             T
(L  AL  )(L x)=(lambda)(L x); hence the eigenvalues of
Ax=(lambda)Bx are those of Py=(lambda)y, where P is the symmetric
        -1  -T
matrix L  AL  . Householder's method is used to tridiagonalise
the matrix P and the eigenvalues are found using the QL
algorithm. An eigenvector z of the derived problem is related to
                                                      T
```

```
an eigenvector x of the original problem by z=L x. The
eigenvectors z are determined using the QL algorithm and are
                    T
normalised so that z z=1; the eigenvectors of the original
                                          T
problem are then determined by solving L x=z, and are normalised
         T
so that x Bx=1.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array              Input/Output

     On entry: the upper triangle of the n by n symmetric matrix
     A. The elements of the array below the diagonal need not be
     set. On exit: the lower triangle of the array is
     overwritten. The rest of the array is unchanged. See also
     Section 8.

 2:  IA -- INTEGER                                        Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02AEF is called.
     Constraint: IA >= N.

 3:  B(IB,N) -- DOUBLE PRECISION array              Input/Output
     On entry: the upper triangle of the n by n symmetric
     positive-definite matrix B. The elements of the array below
     the diagonal need not be set. On exit: the elements below
     the diagonal are overwritten. The rest of the array is
     unchanged.

 4:  IB -- INTEGER                                        Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F02AEF is called.
     Constraint: IB >= N.

 5:  N -- INTEGER                                         Input
     On entry: n, the order of the matrices A and B.

 6:  R(N) -- DOUBLE PRECISION array                      Output
     On exit: the eigenvalues in ascending order.

 7:  V(IV,N) -- DOUBLE PRECISION array                   Output
```

```
     On exit: the normalised eigenvectors, stored by columns;
     the ith column corresponds to the ith eigenvalue. The
                                                T
     eigenvectors x are normalised so that x Bx=1. See also
     Section 8.

 8:  IV -- INTEGER                                          Input
     On entry:
     the first dimension of the array V as declared in the
     (sub)program from which F02AEF is called.
     Constraint: IV >= N.

 9:  DL(N) -- DOUBLE PRECISION array                     Workspace

10:  E(N) -- DOUBLE PRECISION array                      Workspace

11:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     Failure in F01AEF(*); the matrix B is not positive-definite,
     possibly due to rounding errors.

IFAIL= 2
     Failure in F02AMF(*); more than 30*N iterations are required
     to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-
conditioned with respect to inversion, the eigenvectors could be
inaccurately determined. For a detailed error analysis see
Wilkinson and Reinsch [1] pp 310, 222 and 235.

8. Further Comments

                                                                3
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters A and V, in which case the
```

```
eigenvectors will overwrite the original matrix A. However this
is not standard Fortran 77, and may not work on all systems.

9. Example

To calculate all the eigenvalues and eigenvectors of the general
symmetric eigenproblem Ax=(lambda) Bx where A is the symmetric
matrix:

                      (0.5  1.5  6.6   4.8)
                      (1.5  6.5 16.2   8.6)
                      (6.6 16.2 37.6   9.8)
                      (4.8  8.6  9.8 -17.1)

and B is the symmetric positive-definite matrix:

                          (1  3  4  1)
                          (3 13 16 11)
                          (4 16 24 18).
                          (1 11 18 27)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AFF
       F02AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AFF calculates all the eigenvalues of a real unsymmetric
matrix.

2. Specification

      SUBROUTINE F02AFF (A, IA, N, RR, RI, INTGER, IFAIL)
      INTEGER          IA, N, INTGER(N), IFAIL
      DOUBLE PRECISION A(IA,N), RR(N), RI(N)

3. Description

The matrix A is first balanced and then reduced to upper
Hessenberg form using stabilised elementary similarity
```

```
transformations. The eigenvalues are then found using the QR
algorithm for real Hessenberg matrices.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array                Input/Output
     On entry: the n by n matrix A. On exit: the array is
     overwritten.

 2:  IA -- INTEGER                                          Input
     On entry:
     the dimension of the array A as declared in the (sub)program
     from which F02AFF is called.
     Constraint: IA >= N.

 3:  N -- INTEGER                                           Input
     On entry: n, the order of the matrix A.

 4:  RR(N) -- DOUBLE PRECISION array                       Output
     On exit: the real parts of the eigenvalues.

 5:  RI(N) -- DOUBLE PRECISION array                       Output
     On exit: the imaginary parts of the eigenvalues.

 6:  INTGER(N) -- INTEGER array                            Output
     On exit: INTGER(i) contains the number of iterations used
     to find the ith eigenvalue. If INTGER(i) is negative, the i
     th eigenvalue is the second of a pair found simultaneously.

     Note that the eigenvalues are found in reverse order,
     starting with the nth.

 7:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
```

```
     More than 30*N iterations are required to isolate all the
     eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and
the multiplicity of the roots. For a detailed error analysis see
Wilkinson and Reinsch [1] pp 352 and 367.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the real matrix:

                    (  1.5 0.1  4.5 -1.5)
                    (-22.5 3.5 12.5 -2.5)
                    ( -2.5 0.3  4.5 -2.5).
                    ( -2.5 0.1  4.5  2.5)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AGF
        F02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AGF calculates all the eigenvalues and eigenvectors of a real
unsymmetric matrix.

2. Specification

       SUBROUTINE F02AGF (A, IA, N, RR, RI, VR, IVR, VI, IVI,
      1                   INTGER, IFAIL)
       INTEGER          IA, N, IVR, IVI, INTGER(N), IFAIL
       DOUBLE PRECISION A(IA,N), RR(N), RI(N), VR(IVR,N), VI
      1                 (IVI,N)
```

```
3. Description

The matrix A is first balanced and then reduced to upper
Hessenberg form using real stabilised elementary similarity
transformations. The eigenvalues and eigenvectors of the
Hessenberg matrix are calculated using the QR algorithm. The
eigenvectors of the Hessenberg matrix are back-transformed to
give the eigenvectors of the original matrix A.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array              Input/Output
     On entry: the n by n matrix A. On exit: the array is
     overwritten.

 2:  IA -- INTEGER                                        Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02AGF is called.
     Constraint: IA >= N.

 3:  N -- INTEGER                                         Input
     On entry: n, the order of the matrix A.

 4:  RR(N) -- DOUBLE PRECISION array                     Output
     On exit: the real parts of the eigenvalues.

 5:  RI(N) -- DOUBLE PRECISION array                     Output
     On exit: the imaginary parts of the eigenvalues.

 6:  VR(IVR,N) -- DOUBLE PRECISION array                 Output
     On exit: the real parts of the eigenvectors, stored by
     columns. The ith column corresponds to the ith eigenvalue.
     The eigenvectors are normalised so that the sum of the
     squares of the moduli of the elements is equal to 1 and the
     element of largest modulus is real. This ensures that real
     eigenvalues have real eigenvectors.

 7:  IVR -- INTEGER                                       Input
     On entry:
     the first dimension of the array VR as declared in the
     (sub)program from which F02AGF is called.
     Constraint: IVR >= N.

 8:  VI(IVI,N) -- DOUBLE PRECISION array                 Output
```

```
     On exit: the imaginary parts of the eigenvectors, stored by
     columns. The ith column corresponds to the ith eigenvalue.

 9:  IVI -- INTEGER                                          Input
     On entry:
     the first dimension of the array VI as declared in the
     (sub)program from which F02AGF is called.
     Constraint: IVI >= N.

10:  INTGER(N) -- INTEGER array                             Output
     On exit: INTGER(i) contains the number of iterations used
     to find the ith eigenvalue. If INTGER(i) is negative, the i
     th eigenvalue is the second of a pair found simultaneously.

     Note that the eigenvalues are found in reverse order,
     starting with the nth.

11:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     More than 30*N iterations are required to isolate all the
     eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and
the multiplicity of the roots. For a detailed error analysis see
Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

                                                                3
The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the real
matrix:

                    (  1.5 0.1  4.5 -1.5)
```

```
                              (-22.5 3.5 12.5 -2.5)
                              ( -2.5 0.3  4.5 -2.5).
                              ( -2.5 0.1  4.5  2.5)

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          F02 -- Eigenvalue and Eigenvectors                        F02AJF
                 F02AJF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          F02AJF calculates all the eigenvalues of a complex matrix.

          2. Specification

                 SUBROUTINE F02AJF (AR, IAR, AI, IAI, N, RR, RI, INTGER,
                1                   IFAIL)
                 INTEGER          IAR, IAI, N, INTGER(N), IFAIL
                 DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N)

          3. Description

          The complex matrix A is first balanced and then reduced to upper
          Hessenberg form using stabilised elementary similarity
          transformations. The eigenvalues are then found using the
          modified LR algorithm for complex Hessenberg matrices.

          4. References

          [1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
                Computation II, Linear Algebra. Springer-Verlag.

          5. Parameters

           1:  AR(IAR,N) -- DOUBLE PRECISION array             Input/Output
               On entry: the real parts of the elements of the n by n
               complex matrix A. On exit: the array is overwritten.

           2:  IAR -- INTEGER                                        Input
               On entry:
               the first dimension of the array AR as declared in the
```

```
     (sub)program from which F02AJF is called.
     Constraint: IAR >= N.

 3:  AI(IAI,N) -- DOUBLE PRECISION array             Input/Output
     On entry: the imaginary parts of the elements of the n by n
     complex matrix A. On exit: the array is overwritten.

 4:  IAI -- INTEGER                                        Input
     On entry:
     the first dimension of the array AI as declared in the
     (sub)program from which F02AJF is called.
     Constraint: IAI >= N.

 5:  N -- INTEGER                                          Input
     On entry: n, the order of the matrix A.

 6:  RR(N) -- DOUBLE PRECISION array                      Output
     On exit: the real parts of the eigenvalues.

 7:  RI(N) -- DOUBLE PRECISION array                      Output
     On exit: the imaginary parts of the eigenvalues.

 8:  INTGER(N) -- INTEGER array                         Workspace

 9:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     More than 30*N iterations are required to isolate all the
     eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and
the multiplicity of the roots. For a detailed error analysis see
Wilkinson and Reinsch [1] pp 352 and 401.

8. Further Comments

                                                              3
The time taken by the routine is approximately proportional to n
```

```
          9. Example

          To calculate all the eigenvalues of the complex matrix:

                    (-21.0-5.0i      24.60i 13.6+10.2i     4.0i)
                    (     22.5i 26.00-5.00i  7.5-10.0i 2.5     )
                    ( -2.0+1.5i  1.68+2.24i  4.5-5.0i  1.5+2.0i).
                    (     -2.5i -2.60       -2.7+3.6i  2.5-5.0i)

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          F02 -- Eigenvalue and Eigenvectors                          F02AKF
                  F02AKF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          F02AKF calculates all the eigenvalues and eigenvectors of a
          complex matrix.

          2. Specification

                 SUBROUTINE F02AKF (AR, IAR, AI, IAI, N, RR, RI, VR, IVR,
                1                   VI, IVI, INTGER, IFAIL)
                 INTEGER          IAR, IAI, N, IVR, IVI, INTGER(N), IFAIL
                 DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N), VR
                1                 (IVR,N), VI(IVI,N)

          3. Description

          The complex matrix A is first balanced and then reduced to upper
          Hessenberg form by stabilised elementary similarity
          transformations. The eigenvalues and eigenvectors of the
          Hessenberg matrix are calculated using the LR algorithm. The
          eigenvectors of the Hessenberg matrix are back-transformed to
          give the eigenvectors of the original matrix.

          4. References

          [1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
                Computation II, Linear Algebra. Springer-Verlag.
```

```
5. Parameters

 1:  AR(IAR,N) -- DOUBLE PRECISION array             Input/Output
     On entry: the real parts of the elements of the n by n
     complex matrix A. On exit: the array is overwritten.

 2:  IAR -- INTEGER                                          Input
     On entry:
     the first dimension of the array AR as declared in the
     (sub)program from which F02AKF is called.
     Constraint: IAR >= N.

 3:  AI(IAI,N) -- DOUBLE PRECISION array             Input/Output
     On entry: the imaginary parts of the elements of the n by n
     complex matrix A. On exit: the array is overwritten.

 4:  IAI -- INTEGER                                          Input
     On entry:
     the first dimension of the array AI as declared in the
     (sub)program from which F02AKF is called.
     Constraint: IAI >= N.

 5:  N -- INTEGER                                            Input
     On entry: n, the order of the matrix A.

 6:  RR(N) -- DOUBLE PRECISION array                        Output
     On exit: the real parts of the eigenvalues.

 7:  RI(N) -- DOUBLE PRECISION array                        Output
     On exit: the imaginary parts of the eigenvalues.

 8:  VR(IVR,N) -- DOUBLE PRECISION array                    Output
     On exit: the real parts of the eigenvectors, stored by
     columns. The ith column corresponds to the ith eigenvalue.
     The eigenvectors are normalised so that the sum of squares
     of the moduli of the elements is equal to 1 and the element
     of largest modulus is real.

 9:  IVR -- INTEGER                                          Input
     On entry:
     the first dimension of the array VR as declared in the
     (sub)program from which F02AKF is called.
     Constraint: IVR >= N.

10:  VI(IVI,N) -- DOUBLE PRECISION array                    Output
     On exit: the imaginary parts of the eigenvectors, stored by
     columns. The ith column corresponds to the ith eigenvalue.

11:  IVI -- INTEGER                                         Input
```

```
     On entry:
     the first dimension of the array VI as declared in the
     (sub)program from which F02AKF is called.
     Constraint: IVI >= N.

12:  INTGER(N) -- INTEGER array                          Workspace

13:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     More than 30*N iterations are required to isolate all the
     eigenvalues.

7. Accuracy

The accuracy of the results depends on the conditioning of the
original matrix and the multiplicity of the roots. For a detailed
error analysis see Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the complex
matrix:

           (-21.0-5.0i      24.60i 13.6+10.2i     4.0i)
           (     22.5i 26.00-5.00i  7.5-10.0i 2.5     )
           ( -2.0+1.5i  1.68+2.24i  4.5-5.0i  1.5+2.0i).
           (     -2.5i -2.60       -2.7+3.6i  2.5-5.0i)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AWF
       F02AWF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AWF calculates all the eigenvalues of a complex Hermitian
matrix.

2. Specification

      SUBROUTINE F02AWF (AR, IAR, AI, IAI, N, R, WK1, WK2, WK3,
     1                   IFAIL)
      INTEGER          IAR, IAI, N, IFAIL
      DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), WK1(N),
     1                 WK2(N), WK3(N)

3. Description

The complex Hermitian matrix A is first reduced to a real
tridiagonal matrix by n-2 unitary transformations, and a
subsequent diagonal transformation. The eigenvalues are then
derived using the QL algorithm, an adaptation of the QR
algorithm.

4. References

[1]   Peters G (1967) NPL Algorithms Library. Document No.
      F1/04/A.

[2]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  AR(IAR,N) -- DOUBLE PRECISION array             Input/Output
     On entry: the real parts of the elements of the lower
     triangle of the n by n complex Hermitian matrix A. Elements
     of the array above the diagonal need not be set. On exit:
     the array is overwritten.

 2:  IAR -- INTEGER                                        Input
     On entry:
     the first dimension of the array AR as declared in the
     (sub)program from which F02AWF is called.
     Constraint: IAR >= N.
```

```
 3:  AI(IAI,N) -- DOUBLE PRECISION array             Input/Output
     On entry: the imaginary parts of the elements of the lower
     triangle of the n by n complex Hermitian matrix A. Elements
     of the array above the diagonal need not be set. On exit:
     the array is overwritten.

 4:  IAI -- INTEGER                                        Input
     On entry:
     the first dimension of the array AI as declared in the
     (sub)program from which F02AWF is called.
     Constraint: IAI >= N.

 5:  N -- INTEGER                                          Input
     On entry: n, the order of the complex Hermitian matrix, A.

 6:  R(N) -- DOUBLE PRECISION array                       Output
     On exit: the eigenvalues in ascending order.

 7:  WK1(N) -- DOUBLE PRECISION array                  Workspace

 8:  WK2(N) -- DOUBLE PRECISION array                  Workspace

 9:  WK3(N) -- DOUBLE PRECISION array                  Workspace

10:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     More than 30*N iterations are required to isolate all the
     eigenvalues.

7. Accuracy

For a detailed error analysis see Peters [1] page 3 and Wilkinson
and Reinsch [2] page 235.

8. Further Comments

                                                              3
The time taken by the routine is approximately proportional to n
```

```
9. Example

To calculate all the eigenvalues of the complex Hermitian matrix:

            (0.50        0.00       1.84+1.38i  2.08-1.56i)
            (0.00        0.50       1.12+0.84i -0.56+0.42i)
            (1.84-1.38i  1.12-0.84i 0.50        0.00      ).
            (2.08+1.56i -0.56-0.42i 0.00        0.50      )

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F02 -- Eigenvalue and Eigenvectors                          F02AXF
        F02AXF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02AXF calculates all the eigenvalues and eigenvectors of a
complex Hermitian matrix.

2. Specification

       SUBROUTINE F02AXF (AR, IAR, AI, IAI, N, R, VR, IVR, VI,
      1                   IVI, WK1, WK2, WK3, IFAIL)
       INTEGER          IAR, IAI, N, IVR, IVI, IFAIL
       DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), VR(IVR,N), VI
      1                 (IVI,N), WK1(N), WK2(N), WK3(N)

3. Description

The complex Hermitian matrix is first reduced to a real
tridiagonal matrix by n-2 unitary transformations and a
subsequent diagonal transformation. The eigenvalues and
eigenvectors are then derived using the QL algorithm, an
adaptation of the QR algorithm.

4. References

[1]   Peters G (1967) NPL Algorithms Library. Document No.
      F2/03/A.
```

[2] Peters G (1967) NPL Algorithms Library. Document No. F1/04/A.

5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input
On entry: the real parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

2: IAR -- INTEGER Input
On entry:
the first dimension of the array AR as declared in the (sub)program from which F02AXF is called.
Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input
On entry: the imaginary parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

4: IAI -- INTEGER Input
On entry:
the first dimension of the array AI as declared in the (sub)program from which F02AXF is called.
Constraint: IAI >= N.

5: N -- INTEGER Input
On entry: n, the order of the matrix, A.

6: R(N) -- DOUBLE PRECISION array Output
On exit: the eigenvalues in ascending order.

7: VR(IVR,N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvectors, stored by columns. The ith column corresponds to the ith eigenvector. The eigenvectors are normalised so that the sum of the squares of the moduli of the elements is equal to 1 and the element of largest modulus is real. See also Section 8.

8: IVR -- INTEGER Input
On entry:
the first dimension of the array VR as declared in the (sub)program from which F02AXF is called.
Constraint: IVR >= N.

9: VI(IVI,N) -- DOUBLE PRECISION array Output
On exit: the imaginary parts of the eigenvectors, stored by

```
     columns. The ith column corresponds to the ith eigenvector.
     See also Section 8.

10:  IVI -- INTEGER                                          Input
     On entry:
     the first dimension of the array VI as declared in the
     (sub)program from which F02AXF is called.
     Constraint: IVI >= N.

11:  WK1(N) -- DOUBLE PRECISION array                     Workspace

12:  WK2(N) -- DOUBLE PRECISION array                     Workspace

13:  WK3(N) -- DOUBLE PRECISION array                     Workspace

14:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     More than 30*N iterations are required to isolate all the
     eigenvalues.

IFAIL= 2
     The diagonal elements of AI are not all zero, i.e., the
     complex matrix is not Hermitian.

7. Accuracy

The eigenvectors are always accurately orthogonal but the
accuracy of the individual eigenvalues and eigenvectors is
dependent on their inherent sensitivity to small changes in the
original matrix. For a detailed error analysis see Peters [1]
page 3 and [2] page 3.

8. Further Comments

                                                                3
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the implementation document, the
routine may be called with the same actual array supplied for
```

```
parameters AR and VR, and for AI and VI, in which case the
eigenvectors will overwrite the original matrix A. However this
is not standard Fortran 77, and may not work on all systems.

9. Example

To calculate the eigenvalues and eigenvectors of the complex
Hermitian matrix:

            (0.50        0.00       1.84+1.38i  2.08-1.56i)
            (0.00        0.50       1.12+0.84i -0.56+0.42i)
            (1.84-1.38i  1.12-0.84i 0.50        0.00      ).
            (2.08+1.56i -0.56-0.42i 0.00        0.50      )

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F02 -- Eigenvalue and Eigenvectors                          F02BBF
       F02BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02BBF calculates selected eigenvalues and eigenvectors of a real
symmetric matrix by reduction to tridiagonal form, bisection and
inverse iteration, where the selected eigenvalues lie within a
given interval.

2. Specification

      SUBROUTINE F02BBF (A, IA, N, ALB, UB, M, MM, R, V, IV, D,
     1                   E, E2, X, G, C, ICOUNT, IFAIL)
      INTEGER          IA, N, M, MM, IV, ICOUNT(M), IFAIL
      DOUBLE PRECISION A(IA,N), ALB, UB, R(M), V(IV,M), D(N), E
     1                 (N), E2(N), X(N,7), G(N)
      LOGICAL          C(N)

3. Description

The real symmetric matrix A is reduced to a symmetric tridiagonal
matrix T by Householder's method. The eigenvalues which lie
within a given interval [l,u], are calculated by the method of
bisection. The corresponding eigenvectors of T are calculated by
```

```
inverse iteration. A back-transformation is then performed to
obtain the eigenvectors of the original matrix A.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,N) -- DOUBLE PRECISION array              Input/Output
     On entry: the lower triangle of the n by n symmetric matrix
     A. The elements of the array above the diagonal need not be
     set. On exit: the elements of A below the diagonal are
     overwritten, and the rest of the array is unchanged.

 2:  IA -- INTEGER                                        Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02BBF is called.
     Constraint: IA >= N.

 3:  N -- INTEGER                                         Input
     On entry: n, the order of the matrix A.

 4:  ALB -- DOUBLE PRECISION                              Input

 5:  UB -- DOUBLE PRECISION                               Input
     On entry: l and u, the lower and upper end-points of the
     interval within which eigenvalues are to be calculated.

 6:  M -- INTEGER                                         Input
     On entry: an upper bound for the number of eigenvalues
     within the interval.

 7:  MM -- INTEGER                                       Output
     On exit: the actual number of eigenvalues within the
     interval.

 8:  R(M) -- DOUBLE PRECISION array                      Output
     On exit: the eigenvalues, not necessarily in ascending
     order.

 9:  V(IV,M) -- DOUBLE PRECISION array                   Output
     On exit: the eigenvectors, stored by columns. The ith
     column corresponds to the ith eigenvalue. The eigenvectors
     are normalised so that the sum of the squares of the
     elements are equal to 1.

10:  IV -- INTEGER                                       Input
```

```
     On entry:
     the first dimension of the array V as declared in the
     (sub)program from which F02BBF is called.
     Constraint: IV >= N.

11:  D(N) -- DOUBLE PRECISION array                        Workspace

12:  E(N) -- DOUBLE PRECISION array                        Workspace

13:  E2(N) -- DOUBLE PRECISION array                       Workspace

14:  X(N,7) -- DOUBLE PRECISION array                      Workspace

15:  G(N) -- DOUBLE PRECISION array                        Workspace

16:  C(N) -- LOGICAL array                                 Workspace

17:  ICOUNT(M) -- INTEGER array                               Output
     On exit: ICOUNT(i) contains the number of iterations for
     the ith eigenvalue.

18:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     M is less than the number of eigenvalues in the given
     interval. On exit MM contains the number of eigenvalues in
     the interval. Rerun with this value for M.

IFAIL= 2
     More than 5 iterations are required to determine any one
     eigenvector.

7. Accuracy

There is no guarantee of the accuracy of the eigenvectors as the
results depend on the original matrix and the multiplicity of the
roots. For a detailed error analysis see Wilkinson and Reinsch
[1] pp 222 and 436.

8. Further Comments
```

```
                                                                      3
The time taken by the routine is approximately proportional to n

This subroutine should only be used when less than 25% of the
eigenvalues and the corresponding eigenvectors are required. Also
this subroutine is less efficient with matrices which have
multiple eigenvalues.

9. Example

To calculate the eigenvalues lying between -2.0 and 3.0, and the
corresponding eigenvectors of the real symmetric matrix:

                    ( 0.5  0.0  2.3 -2.6)
                    ( 0.0  0.5 -1.4 -0.7)
                    ( 2.3 -1.4  0.5  0.0).
                    (-2.6 -0.7  0.0  0.5)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors                          F02BJF
       F02BJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02BJF calculates all the eigenvalues and, if required, all the
eigenvectors of the generalized eigenproblem  Ax=(lambda)Bx
where A and B are real, square matrices, using the QZ algorithm.

2. Specification

      SUBROUTINE F02BJF (N, A, IA, B, IB, EPS1, ALFR, ALFI,
     1                   BETA, MATV, V, IV, ITER, IFAIL)
      INTEGER          N, IA, IB, IV, ITER(N), IFAIL
      DOUBLE PRECISION A(IA,N), B(IB,N), EPS1, ALFR(N), ALFI(N),
     1                 BETA(N), V(IV,N)
      LOGICAL          MATV

3. Description
```

All the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem Ax=(lambda)Bx where A and B are real, square matrices, are determined using the QZ algorithm. The QZ algorithm consists of 4 stages:

(a) A is reduced to upper Hessenberg form and at the same time B is reduced to upper triangular form.

(b) A is further reduced to quasi-triangular form while the triangular form of B is maintained.

(c) The quasi-triangular form of A is reduced to triangular form and the eigenvalues extracted. This routine does not actually produce the eigenvalues (lambda) , but instead
$$(\lambda)_j$$
returns (alpha) and (beta) such that
$$(\lambda)_j = (\alpha)_j / (\beta)_j, \quad j=1,2,\ldots,n$$
The division by (beta) becomes the responsibility of the
$$(\beta)_j$$
user's program, since (beta) may be zero indicating an
$$(\beta)_j$$
infinite eigenvalue. Pairs of complex eigenvalues occur with (alpha) /(beta) and (alpha) /(beta) complex
$$(\alpha)_j / (\beta)_j \quad (\alpha)_{j+1} / (\beta)_{j+1}$$
conjugates, even though (alpha) and (alpha) are not
$$(\alpha)_j \quad (\alpha)_{j+1}$$
conjugate.

(d) If the eigenvectors are required (MATV = .TRUE.), they are obtained from the triangular matrices and then transformed back into the original co-ordinate system.

4. References

[1] Moler C B and Stewart G W (1973) An Algorithm for Generalized Matrix Eigenproblems. SIAM J. Numer. Anal. 10 241--256.

[2] Ward R C (1975) The Combination Shift QZ Algorithm. SIAM J. Numer. Anal. 12 835--853.

[3] Wilkinson J H (1979) Kronecker's Canonical Form and the QZ Algorithm. Linear Algebra and Appl. 28 285--303.

5. Parameters

```
 1:  N -- INTEGER                                              Input
     On entry: n, the order of the matrices A and B.

 2:  A(IA,N) -- DOUBLE PRECISION array                 Input/Output
     On entry: the n by n matrix A. On exit: the array is
     overwritten.

 3:  IA -- INTEGER                                             Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02BJF is called.
     Constraint: IA >= N.

 4:  B(IB,N) -- DOUBLE PRECISION array                 Input/Output
     On entry: the n by n matrix B. On exit: the array is
     overwritten.

 5:  IB -- INTEGER                                             Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F02BJF is called.
     Constraint: IB >= N.

 6:  EPS1 -- DOUBLE PRECISION                                  Input
     On entry: the tolerance used to determine negligible
     elements. If EPS1 > 0.0, an element will be considered
     negligible if it is less than EPS1 times the norm of its
     matrix. If EPS1 <= 0.0, machine precision is used in place
     of EPS1. A positive value of EPS1 may result in faster
     execution but less accurate results.

 7:  ALFR(N) -- DOUBLE PRECISION array                        Output

 8:  ALFI(N) -- DOUBLE PRECISION array                        Output
     On exit: the real and imaginary parts of (alpha) , for
                                                            j
     j=1,2,...,n.

 9:  BETA(N) -- DOUBLE PRECISION array                        Output
     On exit: (beta) , for j=1,2,...,n.
                     j

10:  MATV -- LOGICAL                                          Input
     On entry: MATV must be set .TRUE. if the eigenvectors are
     required, otherwise .FALSE..

11:  V(IV,N) -- DOUBLE PRECISION array                        Output
     On exit: if MATV = .TRUE., then
          (i)if the jth eigenvalue is real, the jth column of V
          contains its eigenvector;
```

```
          (ii) if the jth and (j+1)th eigenvalues form a complex
          pair, the jth and (j+1)th columns of V contain the
          real and imaginary parts of the eigenvector associated
          with the first eigenvalue of the pair. The conjugate
          of this vector is the eigenvector for the conjugate
          eigenvalue.
     Each eigenvector is normalised so that the component of
     largest modulus is real and the sum of squares of the moduli
     equal one.

     If MATV = .FALSE., V is not used.

12:  IV -- INTEGER                                          Input
     On entry:
     the first dimension of the array V as declared in the
     (sub)program from which F02BJF is called.
     Constraint: IV >= N.

13:  ITER(N) -- INTEGER array                               Output
     On exit: ITER(j) contains the number of iterations needed
     to obtain the jth eigenvalue. Note that the eigenvalues are
     obtained in reverse order, starting with the nth.

14:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= i
     More than 30*N iterations are required to determine all the
     diagonal 1 by 1 or 2 by 2 blocks of the quasi-triangular
     form in the second step of the QZ algorithm. IFAIL is set to
     the index i of the eigenvalue at which this failure occurs.
     If the soft failure option is used, (alpha)  and (beta)  are
                                              j           j
     correct for j=i+1,i+2,...,n, but V does not contain any
     correct eigenvectors.

7. Accuracy

The computed eigenvalues are always exact for a problem
(A+E)x=(lambda)(B+F)x where ||E||/||A|| and ||F||/||B||
```

```
are both of the order of max(EPS1,(epsilon)), EPS1 being defined
as in Section 5 and (epsilon) being the machine precision.

Note: interpretation of results obtained with the QZ algorithm
often requires a clear understanding of the effects of small
changes in the original data. These effects are reviewed in
Wilkinson [3], in relation to the significance of small values of
(alpha)  and (beta) . It should be noted that if (alpha)  and
       j           j                                    j
(beta)  are both small for any j, it may be that no reliance can
      j
be placed on any of the computed eigenvalues
(lambda) =(alpha) /(beta) . The user is recommended to study [3]
        i        i       i
and, if in difficulty, to seek expert advice on determining the
sensitivity of the eigenvalues to perturbations in the data.

8. Further Comments

                                                                    3
The time taken by the routine is approximately proportional to n
and also depends on the value chosen for parameter EPS1.

9. Example

To find all the eigenvalues and eigenvectors of Ax=(lambda) Bx
where

                      (3.9 12.5 -34.5 -0.5)
                      (4.3 21.5 -47.5  7.5)
                    A=(4.3 21.5 -43.5  3.5)
                      (4.4 26.0 -46.0  6.0)

and

                           (1 2 -3 1)
                           (1 3 -5 4)
                         B=(1 3 -4 3).
                           (1 3 -4 4)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors                          F02FJF
       F02FJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
```

your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

To find eigenvalues and eigenvectors of a real sparse symmetric or generalized symmetric eigenvalue problem.

2. Specification

```
      SUBROUTINE F02FJF (N, M, K, NOITS, TOL, DOT, IMAGE, MONIT,
     1                   NOVECS, X, NRX, D, WORK, LWORK, RWORK,
     2                   LRWORK, IWORK, LIWORK, IFAIL)
      INTEGER           N, M, K, NOITS, NOVECS, NRX, LWORK,
     1                  LRWORK, IWORK(LIWORK), LIWORK, IFAIL
      DOUBLE PRECISION TOL, DOT, X(NRX,K), D(K), WORK(LWORK),
     1                  RWORK(LRWORK)
      EXTERNAL          DOT, IMAGE, MONIT
```

3. Description

F02FJF finds the m eigenvalues of largest absolute value and the corresponding eigenvectors for the real eigenvalue problem

$$Cx=(\lambda)x \tag{1}$$

where C is an n by n matrix such that

$$BC=C^{T}B \tag{2}$$

for a given positive-definite matrix B. C is said to be B-symmetric. Different specifications of C allow for the solution of a variety of eigenvalue problems. For example, when

$$C=A \text{ and } B=I \text{ where } A=A^{T}$$

the routine finds the m eigenvalues of largest absolute magnitude for the standard symmetric eigenvalue problem

$$Ax=(\lambda)x. \tag{3}$$

The routine is intended for the case where A is sparse.

As a second example, when

$$C=B^{-1}A$$

where

$$A=A^{T}$$

the routine finds the m eigenvalues of largest absolute magnitude for the generalized symmetric eigenvalue problem

$$Ax=(\lambda)Bx. \qquad (4)$$

The routine is intended for the case where A and B are sparse.

The routine does not require C explicitly, but C is specified via a user-supplied routine IMAGE which, given an n element vector z, computes the image w given by

$$w=Cz.$$

For instance, in the above example, where $C=B^{-1}A$, routine IMAGE will need to solve the positive-definite system of equations Bw=Az for w.

To find the m eigenvalues of smallest absolute magnitude of (3) we can choose $C=A^{-1}$ and hence find the reciprocals of the required eigenvalues, so that IMAGE will need to solve Aw=z for w, and correspondingly for (4) we can choose $C=A^{-1}B$ and solve Aw=Bz for w.

A table of examples of choice of IMAGE is given in Table 3.1. It should be remembered that the routine also returns the corresponding eigenvectors and that B is positive-definite. Throughout A is assumed to be symmetric and, where necessary, non-singularity is also assumed.

Eigenvalues Required	Problem		
	$Ax=(\lambda)x$	$(B=I)Ax=(\lambda)Bx$	$ABx=(\lambda)x$
Largest	Compute w=Az	Solve Bw=Az	Compute w=ABz
Smallest (Find 1/(lambda))	Solve Aw=z	Solve Aw=Bz	Solve Av=z, Bw=(nu)

Furthest from (sigma) (Find (lambda)-(sigma))	Compute w=(A-(sigma)I)z	Solve Bw=(A-(sigma)B)z	Compute w=(AB-(sigma)I)z
Closest to (sigma) (Find 1/((lambda)-(sigma)))	Solve (A-(sigma)I)w=z	Solve (A-(sigma)B)w=Bz	Solve (AB-(sigma)I)w=z

Table 3.1
The Requirement of IMAGE for Various Problems

The matrix B also need not be supplied explicitly, but is specified via a user-supplied routine DOT which, given n element vectors z and w, computes the generalized dot product $w^T Bz$.

F02FJF is based upon routine SIMITZ (see Nikolai [1]), which is itself a derivative of the Algol procedure ritzit (see Rutishauser [4]), and uses the method of simultaneous (subspace) iteration. (See Parlett [2] for description, analysis and advice on the use of the method.)

The routine performs simultaneous iteration on k>m vectors. Initial estimates to p<=k eigenvectors, corresponding to the p eigenvalues of C of largest absolute value, may be supplied by the user to F02FJF. When possible k should be chosen so that the kth eigenvalue is not too close to the m required eigenvalues, but if k is initially chosen too small then F02FJF may be re-entered, supplying approximations to the k eigenvectors found so far and with k then increased.

At each major iteration F02FJF solves an r by r (r<=k) eigenvalue sub-problem in order to obtain an approximation to the eigenvalues for which convergence has not yet occurred. This approximation is refined by Chebyshev acceleration.

4. References

[1] Nikolai P J (1979) Algorithm 538: Eigenvectors and eigenvalues of real generalized symmetric matrices by simultaneous iteration. ACM Trans. Math. Softw. 5 118--125.

[2] Parlett B N (1980) The Symmetric Eigenvalue Problem. Prentice-Hall.

[3] Rutishauser H (1969) Computational aspects of F L Bauer's simultaneous iteration method. Num. Math. 13 4--13.

[4] Rutishauser H (1970) Simultaneous iteration method for symmetric matrices. Num. Math. 16 205--223.

5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix C. Constraint: N >= 1.

2: M -- INTEGER Input/Output
On entry: m, the number of eigenvalues required.
Constraint: M >= 1. On exit: m, the number of eigenvalues actually found. It is equal to m if IFAIL = 0 on exit, and is less than m if IFAIL = 2, 3 or 4. See Section 6 and Section 8 for further information.

3: K -- INTEGER Input
On entry: the number of simultaneous iteration vectors to be used. Too small a value of K may inhibit convergence, while a larger value of K incurs additional storage and additional work per iteration. Suggested value: K = M + 4 will often be a reasonable choice in the absence of better information.
Constraint: M < K <= N.

4: NOITS -- INTEGER Input/Output
On entry: the maximum number of major iterations (eigenvalue sub-problems) to be performed. If NOITS <= 0, then the value 100 is used in place of NOITS. On exit: the number of iterations actually performed.

5: TOL -- DOUBLE PRECISION Input
On entry: a relative tolerance to be used in accepting eigenvalues and eigenvectors. If the eigenvalues are required to about t significant figures, then TOL should be set to about $10^{-t} \cdot d_i$ is accepted as an eigenvalue as soon as two successive approximations to d_i differ by less than $(|\tilde{d}_i| * TOL)/10$, where $\tilde{d}_i$ is the latest aproximation to d_i. Once an eigenvalue has been accepted, then an eigenvector is accepted as soon as $(d_i f_i)/(d_i - d_k) < TOL$, where f_i is the normalised residual of the current approximation to the

```
eigenvector (see Section 8 for further information). The
values of the f  and d  can be printed from routine MONIT.
              i      i
If TOL is supplied outside the range ((epsilon), 1.0), where
(epsilon) is the machine precision, then the value (epsilon)
is used in place of TOL.

6:  DOT -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                                 External Procedure
                                T
DOT must return the value w Bz for given vectors w and z.
For the standard eigenvalue problem, where B=I, DOT must
                           T
return the dot product w z.

Its specification is:

       DOUBLE PRECISION FUNCTION DOT (IFLAG, N, Z, W,
      1                              RWORK, LRWORK,
      2                              IWORK, LIWORK)
       INTEGER          IFLAG, N, LRWORK, IWORK(LIWORK),
      1                 LIWORK
       DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)

 1:  IFLAG -- INTEGER                             Input/Output
     On entry: IFLAG is always non-negative. On exit: IFLAG
     may be used as a flag to indicate a failure in the
                    T
     computation of w Bz. If IFLAG is negative on exit from
     DOT, then FO2FJF will exit immediately with IFAIL set
     to IFLAG. Note that in this case DOT must still be
     assigned a value.

 2:  N -- INTEGER                                     Input
     On entry: the number of elements in the vectors z and w
     and the order of the matrix B.

 3:  Z(N) -- DOUBLE PRECISION array                   Input
                                       T
     On entry: the vector z for which w Bz is required.

 4:  W(N) -- DOUBLE PRECISION array                   Input
                                       T
     On entry: the vector w for which w Bz is required.

 5:  RWORK(LRWORK) -- DOUBLE PRECISION array  User Workspace


 6:  LRWORK -- INTEGER                                Input
```

```
 7:  IWORK(LIWORK) -- INTEGER array          User Workspace


 8:  LIWORK -- INTEGER                                  Input
     DOT is called from F02FJF with the parameters RWORK,
     LRWORK, IWORK and LIWORK as supplied to F02FJF. The
     user is free to use the arrays RWORK and IWORK to
     supply information to DOT and to IMAGE as an
     alternative to using COMMON.
DOT must be declared as EXTERNAL in the (sub)program
from which F02FJF is called. Parameters denoted as
Input must not be changed by this procedure.

7:  IMAGE -- SUBROUTINE, supplied by the user.
                                          External Procedure
IMAGE must return the vector w=Cz for a given vector z.

Its specification is:

       SUBROUTINE IMAGE (IFLAG, N, Z, W, RWORK, LRWORK,
      1                  IWORK, LIWORK)
       INTEGER          IFLAG, N, LRWORK, IWORK(LIWORK),
      1                 LIWORK
       DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)

 1:  IFLAG -- INTEGER                           Input/Output
     On entry: IFLAG is always non-negative. On exit: IFLAG
     may be used as a flag to indicate a failure in the
     computation of w. If IFLAG is negative on exit from
     IMAGE, then F02FJF will exit immediately with IFAIL set
     to IFLAG.

 2:  N -- INTEGER                                       Input
     On entry: n, the number of elements in the vectors w
     and z, and the order of the matrix C.

 3:  Z(N) -- DOUBLE PRECISION array                     Input
     On entry: the vector z for which Cz is required.

 4:  W(N) -- DOUBLE PRECISION array                    Output
     On exit: the vector w=Cz.

 5:  RWORK(LRWORK) -- DOUBLE PRECISION array  User Workspace

 6:  LRWORK -- INTEGER                                  Input

 7:  IWORK(LIWORK) -- INTEGER array          User Workspace

 8:  LIWORK -- INTEGER                                  Input
```

```
     IMAGE is called from F02FJF with the parameters RWORK,
     LRWORK, IWORK and LIWORK as supplied to F02FJF. The
     user is free to use the arrays RWORK and IWORK to
     supply information to IMAGE and DOT as an alternative
     to using COMMON.
IMAGE must be declared as EXTERNAL in the (sub)program
from which F02FJF is called. Parameters denoted as
Input must not be changed by this procedure.

8:  MONIT -- SUBROUTINE, supplied by the user.
                                              External Procedure
MONIT is used to monitor the progress of F02FJF. MONIT may
be the dummy subroutine F02FJZ if no monitoring is actually
required. (F02FJZ is included in the NAG Foundation Library
and so need not be supplied by the user. The routine name
F02FJZ may be implementation dependent: see the Users' Note
for your implementation for details.) MONIT is called after
the solution of each eigenvalue sub-problem and also just
prior to return from F02FJF. The parameters ISTATE and
NEXTIT allow selective printing by MONIT.

Its specification is:

      SUBROUTINE MONIT (ISTATE, NEXTIT, NEVALS,
     1                  NEVECS, K, F, D)
      INTEGER          ISTATE, NEXTIT, NEVALS, NEVECS,
     1                 K
      DOUBLE PRECISION F(K), D(K)

 1:  ISTATE -- INTEGER                                  Input
On entry: ISTATE specifies the state of F02FJF and will
have values as follows:
ISTATE = 0
      No eigenvalue or eigenvector has just been
      accepted.

ISTATE = 1
      One or more eigenvalues have been accepted since
      the last call to MONIT.

ISTATE = 2
      One or more eigenvectors have been accepted since
      the last call to MONIT.

ISTATE = 3
      One or more eigenvalues and eigenvectors have
      been accepted since the last call to MONIT.

ISTATE = 4
      Return from F02FJF is about to occur.
```

```
 2:  NEXTIT -- INTEGER                                    Input
     On entry: the number of the next iteration.

 3:  NEVALS -- INTEGER                                    Input
     On entry: the number of eigenvalues accepted so far.

 4:  NEVECS -- INTEGER                                    Input
     On entry: the number of eigenvectors accepted so far.

 5:  K -- INTEGER                                         Input
     On entry: k, the number of simultaneous iteration
     vectors.

 6:  F(K) -- DOUBLE PRECISION array                       Input
     On entry: a vector of error quantities measuring the
     state of convergence of the simultaneous iteration
     vectors. See the parameter TOL of F02FJF above and
     Section 8 for further details. Each element of F is
     initially set to the value 4.0 and an element remains
     at 4.0 until the corresponding vector is tested.

 7:  D(K) -- DOUBLE PRECISION array                       Input
     On entry: D(i) contains the latest approximation to the
     absolute value of the ith eigenvalue of C.
MONIT must be declared as EXTERNAL in the (sub)program
from which F02FJF is called. Parameters denoted as
Input must not be changed by this procedure.

 9:  NOVECS -- INTEGER                                    Input
     On entry: the number of approximate vectors that are being
     supplied in X. If NOVECS is outside the range (0,K), then
     the value 0 is used in place of NOVECS.

10:  X(NRX,K) -- DOUBLE PRECISION array             Input/Output
     On entry: if 0 < NOVECS <= K, the first NOVECS columns of X
     must contain approximations to the eigenvectors
     corresponding to the NOVECS eigenvalues of largest absolute
     value of C. Supplying approximate eigenvectors can be useful
     when reasonable approximations are known, or when the
     routine is being restarted with a larger value of K.
     Otherwise it is not necessary to supply approximate vectors,
     as simultaneous iteration vectors will be generated randomly
     by the routine. On exit: if IFAIL = 0, 2, 3 or 4, the first
     m' columns contain the eigenvectors corresponding to the
     eigenvalues returned in the first m' elements of D (see
     below); and the next k-m'-1 columns contain approximations
     to the eigenvectors corresponding to the approximate
     eigenvalues returned in the next k-m'-1 elements of D. Here
     m' is the value returned in M (see above), the number of
```

```
     eigenvalues actually found. The kth column is used as
     workspace.

11:  NRX -- INTEGER                                          Input
     On entry:
     the first dimension of the array X as declared in the
     (sub)program from which F02FJF is called.
     Constraint: NRX >= N.

12:  D(K) -- DOUBLE PRECISION array                          Output
     On exit: if IFAIL = 0, 2, 3 or 4, the first m' elements
     contain the first m' eigenvalues in decreasing order of
     magnitude; and the next k-m'-1 elements contain
     approximations to the next k-m'-1 eigenvalues. Here m' is
     the value returned in M (see above), the number of
     eigenvalues actually found. D(k) contains the value e where
     (-e,e) is the latest interval over which Chebyshev
     acceleration is performed.

13:  WORK(LWORK) -- DOUBLE PRECISION array                Workspace

14:  LWORK -- INTEGER                                        Input
     On entry: the length of the array WORK, as declared in the
     (sub)program from which F02FJF is called. Constraint:
     LWORK>=3*K+max(K*K,2*N).

15:  RWORK(LRWORK) -- DOUBLE PRECISION array       User Workspace
     RWORK is not used by F02FJF, but is passed directly to
     routines DOT and IMAGE and may be used to supply information
     to these routines.

16:  LRWORK -- INTEGER                                       Input
     On entry: the length of the array RWORK, as declared in the
     (sub)program from which F02FJF is called. Constraint: LRWORK
     >= 1.

17:  IWORK(LIWORK) -- INTEGER array               User Workspace
     IWORK is not used by F02FJF, but is passed directly to
     routines DOT and IMAGE and may be used to supply information
     to these routines.

18:  LIWORK -- INTEGER                                       Input
     On entry: the length of the array IWORK, as declared in the
     (sub)program from which F02FJF is called. Constraint: LIWORK
     >= 1.

19:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.
```

```
     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:


IFAIL< 0
     A negative value of IFAIL indicates an exit from F02FJF
     because the user has set IFLAG negative in DOT or IMAGE. The
     value of IFAIL will be the same as the user's setting of
     IFLAG.

IFAIL= 1
     On entry N < 1,

     or       M < 1,

     or       M >= K,

     or       K > N,

     or       NRX < N,

     or       LWORK <3*K+max(K*K*N),

     or       LRWORK < 1,

     or       LIWORK < 1.

IFAIL= 2
     Not all the requested eigenvalues and vectors have been
     obtained. Approximations to the rth eigenvalue are
     oscillating rapidly indicating that severe cancellation is
     occurring in the rth eigenvector and so M is returned as (r-
     1). A restart with a larger value of K may permit
     convergence.

IFAIL= 3
     Not all the requested eigenvalues and vectors have been
     obtained. The rate of convergence of the remaining
```

eigenvectors suggests that more than NOITS iterations would be required and so the input value of M has been reduced. A restart with a larger value of K may permit convergence.

IFAIL= 4
Not all the requested eigenvalues and vectors have been obtained. NOITS iterations have been performed. A restart, possibly with a larger value of K, may permit convergence.

IFAIL= 5
This error is very unlikely to occur, but indicates that convergence of the eigenvalue sub-problem has not taken place. Restarting with a different set of approximate vectors may allow convergence. If this error occurs the user should check carefully that F02FJF is being called correctly.

7. Accuracy

Eigenvalues and eigenvectors will normally be computed to the accuracy requested by the parameter TOL, but eigenvectors corresponding to small or to close eigenvalues may not always be computed to the accuracy requested by the parameter TOL. Use of the routine MONIT to monitor acceptance of eigenvalues and eigenvectors is recommended.

8. Further Comments

The time taken by the routine will be principally determined by the time taken to solve the eigenvalue sub-problem and the time taken by the routines DOT and IMAGE. The time taken to solve an eigenvalue sub-problem is approximately proportional to nk^2. It is important to be aware that several calls to DOT and IMAGE may occur on each major iteration.

As can be seen from Table 3.1, many applications of F02FJF will require routine IMAGE to solve a system of linear equations. For example, to find the smallest eigenvalues of Ax=(lambda)Bx, IMAGE needs to solve equations of the form Aw=Bz for w and routines from Chapters F01 and F04 of the NAG Foundation Library will frequently be useful in this context. In particular, if A is a positive-definite variable band matrix, F04MCF may be used after A has been factorized by F01MCF. Thus factorization need be performed only once prior to calling F02FJF. An illustration of this type of use is given in the example program in Section 9.

An approximation $\tilde{d}_h$, to the ith eigenvalue, is accepted as soon

~
as d and the previous approximation differ by less than
h
~
|d |*TOL/10. Eigenvectors are accepted in groups corresponding to
h
clusters of eigenvalues that are equal, or nearly equal, in
absolute value and that have already been accepted. If d is the
r
last eigenvalue in such a group and we define the residual r as
j

$$r_j = Cx_j - y_j$$

where y is the projection of Cx , with respect to B, onto the
r j
space spanned by x ,x ,...,x and x is the current approximation
1 2 r j
to the jth eigenvector, then the value f returned in MONIT is
i
given by

$$f_i = \max\left(\frac{||r_j||_B}{||Cx_j||_B}, ||x_j||_B^2 = x_j^T B x_j\right)$$

and each vector in the group is accepted as an eigenvector if

$$(|d_r| - |f_r|)/(|d_r| - e) < TOL$$

~
where e is the current approximation to |d |. The values of the
k
f are systematically increased if the convergence criteria
i
appear to be too strict. See Rutishauser [4] for further details.

The algorithm implemented by F02FJF differs slightly from SIMITZ
(Nikolai [1]) in that the eigenvalue sub-problem is solved using
the singular value decomposition of the upper triangular matrix R
T
of the Gram-Schmidt factorization of Cx , rather than forming R R
r

9. Example

To find the four eigenvalues of smallest absolute value and
corresponding eigenvectors for the generalized symmetric

eigenvalue problem Ax=(lambda)Bx, where A and B are the 16 by 16 matrices

```
          (1 a     a                    )
          (a 1 a     a                  )
          (  a 1 a     a                )
          (    a 1 a     a              )
          (a     a 1 a     a            )
          (  a     a 1 a     a          )
          (    a     a 1 a     a        )
          (      a     a 1 a     a      )
        A=(        a     a 1 a     a    )
          (          a     a 1 a     a  )
          (            a     a 1 a     a)
          (              a     a 1 a    )
          (                a     a 1 a  )
          (                  a     a 1 a)
          (                    a     a 1)
```

```
          1
where a=- -
          4
```

```
          (1 b                         )
          (b 1 b                       )
          (  b 1 b                     )
          (    b 1 b                   )
          (      b 1 b                 )
          (        b 1 b               )
          (          b 1 b             )
        B=(            b 1 b           )
          (              b 1 b         )
          (                b 1 b       )
          (                  b 1 b     )
          (                    b 1 b   )
          (                      b 1 b )
          (                        b 1 b)
```

```
          1
where b=- -
          2
```

TOL is taken as 0.0001 and 6 iteration vectors are used. F01MAF is used to factorize the matrix A, prior to calling F02FJF, and F04MAF is used within IMAGE to solve the equations Aw=Bz for w. Details of the factorization of A are passed from F01MAF to F04MAF by means of the COMMON block BLOCK1.

```
Output from MONIT occurs each time ISTATE is non-zero. Note that
the required eigenvalues are the reciprocals of the eigenvalues
returned by F02FJF.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F02 -- Eigenvalue and Eigenvectors                          F02WEF
        F02WEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02WEF returns all, or part, of the singular value decomposition
of a general real matrix.

2. Specification

       SUBROUTINE F02WEF (M, N, A, LDA, NCOLB, B, LDB, WANTQ, Q,
      1                   LDQ, SV, WANTP, PT, LDPT, WORK, IFAIL)
       INTEGER          M, N, LDA, NCOLB, LDB, LDQ, LDPT, IFAIL
       DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), SV(*), PT
      1                 (LDPT,*), WORK(*)
       LOGICAL          WANTQ, WANTP

3. Description

The m by n matrix A is factorized as

                                 T
                              A=QDP ,

where

      (S)
    D=(0),          m>n,

    D=S,            m=n,

    D=(S 0),        m<n,

Q is an m by m orthogonal matrix, P is an n by n orthogonal
matrix and S is a min(m,n) by min(m,n) diagonal matrix with non-
```

negative diagonal elements, $sv_1, sv_2, \ldots, sv_{min(m,n)}$, ordered such that

$$sv_1 >= sv_2 >= \ldots >= sv_{min(m,n)} >= 0.$$

The first min(m,n) columns of Q are the left-hand singular vectors of A, the diagonal elements of S are the singular values of A and the first min(m,n) columns of P are the right-hand singular vectors of A.

Either or both of the left-hand and right-hand singular vectors of A may be requested and the matrix C given by

$$C = Q^T B,$$

where B is an m by ncolb given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing A to upper triangular form by means of Householder transformations, from the left when m>=n and from the right when m<n. The upper triangular form is then reduced to bidiagonal form by Givens plane rotations and finally the QR algorithm is used to obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] DSVDC.

Note that if K is any orthogonal diagonal matrix so that

$$KK^T = I,$$

(so that K has elements +1 or -1 on the diagonal)

then

$$A = (QK)D(PK)^T$$

is also a singular value decomposition of A.

4. References

[1] Dongarra J J, Moler C B, Bunch J R and Stewart G W (1979) LINPACK Users' Guide. SIAM, Philadelphia.

[2] Hammarling S (1985) The Singular Value Decomposition in Multivariate Statistics. ACM Signum Newsletter. 20, 3 2--25.

[3] Wilkinson J H (1978) Singular Value Decomposition -- Basic Aspects. Numerical Software -- Needs and Availability. (ed D A H Jacobs) Academic Press.

5. Parameters

1: M -- INTEGER Input
On entry: the number of rows, m, of the matrix A.
Constraint: M >= 0.

When M = 0 then an immediate return is effected.

2: N -- INTEGER Input
On entry: the number of columns, n, of the matrix A.
Constraint: N >= 0.

When N = 0 then an immediate return is effected.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array A must be at least max(1,N).
On entry: the leading m by n part of the array A must contain the matrix A whose singular value decomposition is required. On exit: if M >= N and WANTQ = .TRUE., then the leading m by n part of A will contain the first n columns of the orthogonal matrix Q.

If M < N and WANTP = .TRUE., then the leading m by n part of A will contain the first m rows of the orthogonal matrix P^T.

If M >= N and WANTQ = .FALSE. and WANTP = .TRUE., then the leading n by n part of A will contain the first n rows of the orthogonal matrix P^T.

Otherwise the leading m by n part of A is used as internal workspace.

4: LDA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F02WEF is called.
Constraint: LDA >= max(1,M).

5: NCOLB -- INTEGER Input
On entry: ncolb, the number of columns of the matrix B.

```
     When NCOLB = 0 the array B is not referenced. Constraint:
     NCOLB >= 0.

 6:  B(LDB,*) -- DOUBLE PRECISION array              Input/Output
     Note: the second dimension of the array B must be at least
     max(1,ncolb) On entry: if NCOLB > 0, the leading m by ncolb
     part of the array B must contain the matrix to be
     transformed. On exit: B is overwritten by the m by ncolb
              T
     matrix  Q B.

 7:  LDB -- INTEGER                                        Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F02WEF is called.
     Constraint: if NCOLB > 0 then LDB >= max(1,M).

 8:  WANTQ -- LOGICAL                                      Input
     On entry: WANTQ must be .TRUE., if the left-hand singular
     vectors are required. If WANTQ = .FALSE., then the array Q
     is not referenced.

 9:  Q(LDQ,*) -- DOUBLE PRECISION array                    Output
     Note: the second dimension of the array Q must be at least
     max(1,M).
     On exit: if M < N and WANTQ = .TRUE., the leading m by m
     part of the array Q will contain the orthogonal matrix Q.
     Otherwise the array Q is not referenced.

10:  LDQ -- INTEGER                                       Input
     On entry:
     the first dimension of the array Q as declared in the
     (sub)program from which F02WEF is called.
     Constraint: if M < N and WANTQ = .TRUE., LDQ >= max(1,M).

11:  SV(*) -- DOUBLE PRECISION array                      Output
     Note: the length of SV must be at least min(M,N). On exit:
     the min(M,N) diagonal elements of the matrix S.

12:  WANTP -- LOGICAL                                     Input
     On entry: WANTP must be .TRUE. if the right-hand singular
     vectors are required. If WANTP = .FALSE., then the array PT
     is not referenced.

13:  PT(LDPT,*) -- DOUBLE PRECISION array                 Output
     Note: the second dimension of the array PT must be at least
     max(1,N).
     On exit: if M >= N and WANTQ and WANTP are .TRUE., the
     leading n by n part of the array PT will contain the
                       T
```

```
     orthogonal matrix P . Otherwise the array PT is not
     referenced.

14:  LDPT -- INTEGER                                          Input
     On entry:
     the first dimension of the array PT as declared in the
     (sub)program from which F02WEF is called.
     Constraint: if M >= N and WANTQ and WANTP are .TRUE., LDPT
     >= max(1,N).

15:  WORK(*) -- DOUBLE PRECISION array                        Output
     Note: the length of WORK must be at least max(1,lwork),
     where lwork must be as given in the following table:

     M >= N
     WANTQ is .TRUE. and WANTP = .TRUE.
                     2
           lwork=max(N +5*(N-1),N+NCOLB,4)

     WANTQ = .TRUE. and WANTP = .FALSE.
                     2
           lwork=max(N +4*(N-1),N+NCOLB,4)

     WANTQ = .FALSE. and WANTP = .TRUE.
           lwork=max(3*(N-1),2) when NCOLB = 0

           lwork=max(5*(N-1),2) when NCOLB > 0

     WANTQ = .FALSE. and WANTP = .FALSE.
           lwork=max(2*(N-1),2) when NCOLB = 0

           lwork=max(3*(N-1),2) when NCOLB > 0

     M < N
     WANTQ = .TRUE. and WANTP = .TRUE.
                     2
           lwork=max(M +5*(M-1),2)

     WANTQ = .TRUE. and WANTP = .FALSE.
           lwork=max(3*(M-1),1)

     WANTQ = .FALSE. and WANTP = .TRUE.
                     2
           lwork=max(M +3*(M-1),2) when NCOLB = 0

                     2
           lwork=max(M +5*(M-1),2) when NCOLB > 0

     WANTQ = .FALSE. and WANTP = .FALSE.
           lwork=max(2*(M-1),1) when NCOLB = 0
```

```
          lwork=max(3*(M-1),1) when NCOLB > 0
     On exit: WORK(min(M,N)) contains the total number of
     iterations taken by the R algorithm.

     The rest of the array is used as workspace.

16:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     One or more of the following conditions holds:
          M < 0,

          N < 0,

          LDA < M,

          NCOLB < 0,

          LDB < M and NCOLB > 0,

          LDQ < M and M < N and WANTQ = .TRUE.,

          LDPT < N and M >= N and WANTQ = .TRUE., and WANTP = .
          TRUE..

IFAIL> 0
     The QR algorithm has failed to converge in 50*min(m,n)
     iterations. In this case SV(1), SV(2),..., SV(IFAIL) may not
     have been found correctly and the remaining singular values
     may not be the smallest. The matrix A will nevertheless have
                              T
     been factorized as  A=QEP ,  where the leading min(m,n) by
     min(m,n) part of E is a bidiagonal matrix with SV(1), SV(2),
     ..., SV(min(m,n)) as the diagonal elements and WORK(1), WORK
     (2),..., WORK(min(m,n)-1) as the super-diagonal elements.
```

```
     This failure is not likely to occur.

7. Accuracy

The computed factors Q, D and P satisfy the relation

                                T
                              QDP =A+E,

where

                        ||E||<=c(epsilon)||A||,

(epsilon) being the machine precision, c is a modest function of
m and n and ||.|| denotes the spectral (two) norm. Note that
||A||=sv .
        1

8. Further Comments

Following the use of this routine the rank of A may be estimated
by a call to the INTEGER FUNCTION F06KLF(*). The statement:

       IRANK = F06KLF(MIN(M, N), SV, 1, TOL)

returns the value (k-1) in IRANK, where k is the smallest integer
for which SV(k)<tol*SV(1), where tol is the tolerance supplied in
TOL, so that IRANK is an estimate of the rank of S and thus also
of A. If TOL is supplied as negative then the machine precision
is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

                            (2.0  2.5  2.5)
                            (2.0  2.5  2.5)
                          A=(1.6 -0.4  2.8)
                            (2.0 -0.5  0.5)
                            (1.2 -0.3 -2.9)

                             T
together with the vector Q b for the vector

                               ( 1.1)
                               ( 0.9)
                             b=( 0.6)
                               ( 0.0)
```

```
                                  (-0.8)

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

          9.2. Example 2

          To find the singular value decomposition of the 3 by 5 matrix

                            (2.0 2.0  1.6  2.0  1.2)
                          A=(2.5 2.5 -0.4 -0.5 -0.3)
                            (2.5 2.5  2.8  0.5 -2.9)

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          F02 -- Eigenvalue and Eigenvectors                      F02XEF
                 F02XEF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          F02XEF returns all, or part, of the singular value decomposition
          of a general complex matrix.

          2. Specification

                SUBROUTINE F02XEF (M, N, A, LDA, NCOLB, B, LDB, WANTQ, Q,
               1                   LDQ, SV, WANTP, PH, LDPH, RWORK, CWORK,
               2                   IFAIL)
                INTEGER                   M, N, LDA, NCOLB, LDB, LDQ, LDPH,
               1                          IFAIL
                DOUBLE PRECISION          SV(*), RWORK(*)
                COMPLEX(KIND=KIND(1.0D0)) A(LDA,*), B(LDB,*), Q(LDQ,*),
               1                          PH(LDPH,*), CWORK(*)
                LOGICAL                   WANTQ, WANTP

          3. Description

          The m by n matrix A is factorized as

                                          H
```

$$A=QDP\ ,$$

where

$$D=\begin{pmatrix}(S)\\(0)\end{pmatrix}\quad m>n,$$

$$D=S,\qquad m=n,$$

$$D=(S\ 0),\qquad m<n,$$

Q is an m by m unitary matrix, P is an n by n unitary matrix and S is a min(m,n) by min(m,n) diagonal matrix with real non-negative diagonal elements, $sv_1, sv_2, \ldots, sv_{\min(m,n)}$, ordered such that

$$sv_1 \geq sv_2 \geq \ldots \geq sv_{\min(m,n)} \geq 0.$$

The first min(m,n) columns of Q are the left-hand singular vectors of A, the diagonal elements of S are the singular values of A and the first min(m,n) columns of P are the right-hand singular vectors of A.

Either or both of the left-hand and right-hand singular vectors of A may be requested and the matrix C given by

$$C=Q^H B,$$

where B is an m by ncolb given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing A to upper triangular form by means of Householder transformations, from the left when m>=n and from the right when m<n. The upper triangular form is then reduced to bidiagonal form by Givens plane rotations and finally the QR algorithm is used to obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] ZSVDC.

Note that if K is any unitary diagonal matrix so that

$$KK^H=I,$$

then

```
                                   H
                          A=(QK)D(PK)
```

is also a singular value decomposition of A.

4. References

[1] Dongarra J J, Moler C B, Bunch J R and Stewart G W (1979) LINPACK Users' Guide. SIAM, Philadelphia.

[2] Hammarling S (1985) The Singular Value Decomposition in Multivariate Statistics. ACM Signum Newsletter. 20, 3 2--25.

[3] Wilkinson J H (1978) Singular Value Decomposition -- Basic Aspects. Numerical Software -- Needs and Availability. (ed D A H Jacobs) Academic Press.

5. Parameters

1: M -- INTEGER Input
On entry: the number of rows, m, of the matrix A.
Constraint: M >= 0.

When M = 0 then an immediate return is effected.

2: N -- INTEGER Input
On entry: the number of columns, n, of the matrix A.
Constraint: N >= 0.

When N = 0 then an immediate return is effected.

3: A(LDA,*) -- COMPLEX(KIND(1.0D)) array Input/Output
Note: the second dimension of the array A must be at least max(1,N).
On entry: the leading m by n part of the array A must contain the matrix A whose singular value decomposition is required. On exit: if M >= N and WANTQ = .TRUE., then the leading m by n part of A will contain the first n columns of the unitary matrix Q.
If M < N and WANTP = .TRUE., then the leading m by n part of
H
A will contain the first m rows of the unitary matrix P .
will contain the first m rows of the unitary matrix P If M >= N and WANTQ = .FALSE. and WANTP = .TRUE., then the leading n by n part of A will contain the first n
H
rows of the unitary matrix P . Otherwise the leading m by n part of A is used as internal workspace.

```
 4:  LDA -- INTEGER                                              Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F02XEF is called.
     Constraint: LDA >= max(1,M).

 5:  NCOLB -- INTEGER                                            Input
     On entry: ncolb, the number of columns of the matrix B.
     When NCOLB = 0 the array B is not referenced. Constraint:
     NCOLB >= 0.

 6:  B(LDB,*) -- COMPLEX(KIND(1.0D)) array           Input/Output
     Note: the second dimension of the array B must be at least
     max(1,NCOLB).
     On entry: if NCOLB > 0, the leading m by ncolb part of the
     array B must contain the matrix to be transformed. On exit:
                                                        H
     B is overwritten by the m by ncolb matrix Q B.

 7:  LDB -- INTEGER                                              Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F02XEF is called.
     Constraint: if NCOLB > 0, then LDB >= max(1,M).

 8:  WANTQ -- LOGICAL                                            Input
     On entry: WANTQ must be .TRUE. if the left-hand singular
     vectors are required. If WANTQ = .FALSE. then the array Q is
     not referenced.

 9:  Q(LDQ,*) -- COMPLEX(KIND(1.0D)) array                      Output
     Note: the second dimension of the array Q must be at least
     max(1,M).
     On exit: if M < N and WANTQ = .TRUE., the leading m by m
     part of the array Q will contain the unitary matrix Q.
     Otherwise the array Q is not referenced.

10:  LDQ -- INTEGER                                             Input
     On entry:
     the first dimension of the array Q as declared in the
     (sub)program from which F02XEF is called.
     Constraint: if M < N and WANTQ = .TRUE., LDQ >= max(1,M).

11:  SV(*) -- DOUBLE PRECISION array                           Output
     Note: the length of SV must be at least min(M,N). On exit:
     the min(m,n) diagonal elements of the matrix S.

12:  WANTP -- LOGICAL                                           Input
     On entry: WANTP must be .TRUE. if the right-hand singular
     vectors are required. If WANTP = .FALSE. then the array PH
```

```
     is not referenced.

13:  PH(LDPH,*) -- DOUBLE PRECISION array                    Output
     Note: the second dimension of the array PH must be at least
     max(1,N).
     On exit: if M >= N and WANTQ and WANTP are .TRUE., the
     leading n by n part of the array PH will contain the unitary
                H
     matrix P . Otherwise the array PH is not referenced.

14:  LDPH -- INTEGER                                         Input
     On entry:
     the first dimension of the array PH as declared in the
     (sub)program from which F02XEF is called.
     Constraint: if M >= N and WANTQ and WANTP are .TRUE.,  LDPH
     >= max(1,N).

15:  RWORK(*) -- DOUBLE PRECISION array                      Output
     Note: the length of RWORK must be at least max(1,lrwork),
     where lrwork must satisfy:
     lrwork=2*(min(M,N)-1) when
           NCOLB = 0 and WANTQ and WANTP are .FALSE.,

     lrwork=3*(min(M,N)-1) when
           either NCOLB = 0 and WANTQ = .FALSE. and WANTP = .
           TRUE., or WANTP = .FALSE. and one or both of NCOLB > 0
           and WANTQ = .TRUE.

     lrwork=5*(min(M,N)-1)
           otherwise.
     On exit: RWORK(min(M,N)) contains the total number of
     iterations taken by the QR algorithm.

     The rest of the array is used as workspace.

16:  CWORK(*) -- COMPLEX(KIND(1.0D)) array                Workspace
     Note: the length of CWORK must be at least max(1,lcwork),
     where lcwork must satisfy:
                    2
     lcwork=N+max(N ,NCOLB) when
           M >= N and WANTQ and WANTP are both .TRUE.

                    2
     lcwork=N+max(N +N,NCOLB) when
           M >= N and WANTQ = .TRUE., but WANTP = .FALSE.

     lcwork=N+max(N,NCOLB) when
           M >= N and WANTQ = .FALSE.

             2
```

```
     lcwork=M +M when
           M < N and WANTP = .TRUE.

     lcwork = M when
           M < N and WANTP = .FALSE.

17:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     One or more of the following conditions holds:
          M < 0,

          N < 0,

          LDA < M,

          NCOLB < 0,

          LDB < M and NCOLB > 0,

          LDQ < M and M < N and WANTQ = .TRUE.,

          LDPH < N and M >= N and WANTQ = .TRUE. and WANTP = .
          TRUE..

IFAIL> 0
     The QR algorithm has failed to converge in 50*min(m,n)
     iterations. In this case SV(1), SV(2),..., SV(IFAIL) may not
     have been found correctly and the remaining singular values
     may not be the smallest. The matrix A will nevertheless have
                              H
     been factorized as  A=QEP   where the leading min(m,n) by
     min(m,n) part of E is a bidiagonal matrix with SV(1), SV(2),
     ..., SV(min(m,n)) as the diagonal elements and RWORK(1),
     RWORK(2),..., RWORK(min(m,n)-1) as the super-diagonal
     elements.
```

This failure is not likely to occur.

7. Accuracy

The computed factors Q, D and P satisfy the relation

$$QDP^H = A+E,$$

where

$$||E|| <= c(epsilon)||A||,$$

(epsilon) being the machine precision, c is a modest function of m and n and ||.|| denotes the spectral (two) norm. Note that $||A|| = sv_1$.

8. Further Comments

Following the use of this routine the rank of A may be estimated by a call to the INTEGER FUNCTION F06KLF(*). The statement:

```
IRANK = F06KLF(MIN(M, N), SV, 1, TOL)
```

returns the value (k-1) in IRANK, where k is the smallest integer for which SV(k)<tol*SV(1), where tol is the tolerance supplied in TOL, so that IRANK is an estimate of the rank of S and thus also of A. If TOL is supplied as negative then the machine precision is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

$$A = \begin{pmatrix} 0.5i & -0.5+1.5i & -1.0+1.0i \\ 0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\ 0.4 & -0.4+0.4i & 1.8 \\ 0.3-0.4i & 0.1+0.7i & 0.0 \\ -0.3i & 0.3+0.3i & 2.4i \end{pmatrix}$$

together with the vector $Q^H b$ for the vector

$$b = \begin{pmatrix} -0.55+1.05i \\ 0.49+0.93i \\ 0.56-0.16i \end{pmatrix}$$

```
                                ( 0.39+0.23i)
                                ( 1.13+0.83i)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

9.2. Example 2

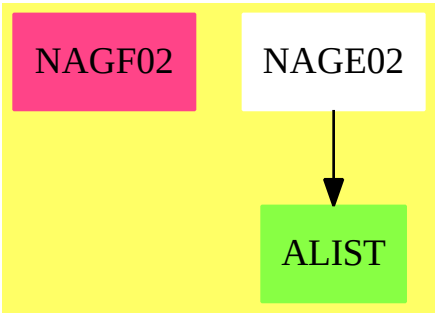
To find the singular value decompostition of the 3 by 5 matrix

          (     0.5i 0.4-0.3i  0.4      0.3+0.4i     0.3i)
       A=(-0.5-1.5i 0.9-1.3i -0.4-0.4i 0.1-0.7i 0.3-0.3i)
          (-1.0-1.0i 0.2-1.4i  1.8      0.0         -2.4i)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

———-

15.1.1 NagEigenPackage (NAGF02)



Exports:

f02aaf	f02abf	f02adf	f02aef	f02aff
f02agf	f02ajf	f02akf	f02awf	f02axf
f02bbf	f02bjf	f02fjf	f02fjf	f02wef
f02xef				

— package NAGF02 NagEigenPackage —

```
)abbrev package NAGF02 NagEigenPackage
++ Author: Godfrey Nolan and Mike Dewar
```

```
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:20 1994
++ Description:
++ This package uses the NAG Library to compute\br
++ \tab{5}eigenvalues and eigenvectors of a matrix\br
++ \tab{5} eigenvalues and eigenvectors of generalized matrix
++ eigenvalue problems\br
++ \tab{5}singular values and singular vectors of a matrix.

NagEigenPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    f02aaf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
     ++ f02aaf(ia,n,a,ifail)
     ++ calculates all the eigenvalue.
     ++ See \downlink{Manual Page}{manpageXXf02aaf}.
    f02abf : (Matrix DoubleFloat,Integer,Integer,Integer,Integer) -> Result
     ++ f02abf(a,ia,n,iv,ifail)
     ++ calculates all the eigenvalues of a real
     ++ symmetric matrix.
     ++ See \downlink{Manual Page}{manpageXXf02abf}.
    f02adf : (Integer,Integer,Integer,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
     ++ f02adf(ia,ib,n,a,b,ifail)
     ++ calculates all the eigenvalues of  Ax=(lambda)Bx,  where A
     ++ is a real symmetric matrix and B is a real symmetric positive-
     ++ definite matrix.
     ++ See \downlink{Manual Page}{manpageXXf02adf}.
    f02aef : (Integer,Integer,Integer,Integer,_
        Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
     ++ f02aef(ia,ib,n,iv,a,b,ifail)
     ++ calculates all the eigenvalues of
     ++ Ax=(lambda)Bx,  where A is a real symmetric matrix and B is a
     ++ real symmetric positive-definite matrix.
     ++ See \downlink{Manual Page}{manpageXXf02aef}.
    f02aff : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
     ++ f02aff(ia,n,a,ifail)
     ++ calculates all the eigenvalues of a real unsymmetric
     ++ matrix.
     ++ See \downlink{Manual Page}{manpageXXf02aff}.
    f02agf : (Integer,Integer,Integer,Integer,_
        Matrix DoubleFloat,Integer) -> Result
     ++ f02agf(ia,n,ivr,ivi,a,ifail)
     ++ calculates all the eigenvalues of a real
     ++ unsymmetric matrix.
     ++ See \downlink{Manual Page}{manpageXXf02agf}.
    f02ajf : (Integer,Integer,Integer,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
```

```
 ++ f02ajf(iar,iai,n,ar,ai,ifail)
 ++ calculates all the eigenvalue.
 ++ See \downlink{Manual Page}{manpageXXf02ajf}.
f02akf : (Integer,Integer,Integer,Integer,_
    Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ f02akf(iar,iai,n,ivr,ivi,ar,ai,ifail)
 ++ calculates all the eigenvalues of a
 ++ complex matrix.
 ++ See \downlink{Manual Page}{manpageXXf02akf}.
f02awf : (Integer,Integer,Integer,Matrix DoubleFloat,_
    Matrix DoubleFloat,Integer) -> Result
 ++ f02awf(iar,iai,n,ar,ai,ifail)
 ++ calculates all the eigenvalues of a complex Hermitian
 ++ matrix.
 ++ See \downlink{Manual Page}{manpageXXf02awf}.
f02axf : (Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,_
    Integer,Integer,Integer,Integer) -> Result
 ++ f02axf(ar,iar,ai,iai,n,ivr,ivi,ifail)
 ++ calculates all the eigenvalues of a
 ++ complex Hermitian matrix.
 ++ See \downlink{Manual Page}{manpageXXf02axf}.
f02bbf : (Integer,Integer,DoubleFloat,DoubleFloat,_
    Integer,Integer,Matrix DoubleFloat,Integer) -> Result
 ++ f02bbf(ia,n,alb,ub,m,iv,a,ifail)
 ++ calculates selected eigenvalues of a real
 ++ symmetric matrix by reduction to tridiagonal form, bisection and
 ++ inverse iteration, where the selected eigenvalues lie within a
 ++ given interval.
 ++ See \downlink{Manual Page}{manpageXXf02bbf}.
f02bjf : (Integer,Integer,Integer,DoubleFloat,_
  Boolean,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ f02bjf(n,ia,ib,eps1,matv,iv,a,b,ifail)
 ++ calculates all the eigenvalues and, if required, all the
 ++ eigenvectors of the generalized eigenproblem  Ax=(lambda)Bx
 ++ where A and B are real, square matrices, using the QZ algorithm.
 ++ See \downlink{Manual Page}{manpageXXf02bjf}.
f02fjf : (Integer,Integer,DoubleFloat,Integer,_
    Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,_
    Integer,Union(fn:FileName,fp:Asp27(DOT)),_
    Union(fn:FileName,fp:Asp28(IMAGE))) -> Result
 ++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
 ++        liwork,m,noits,x,ifail,dot,image)
 ++ finds eigenvalues of a real sparse symmetric
 ++ or generalized symmetric eigenvalue problem.
 ++ See \downlink{Manual Page}{manpageXXf02fjf}.
f02fjf : (Integer,Integer,DoubleFloat,Integer,_
    Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,_
    Integer,Union(fn:FileName,fp:Asp27(DOT)),_
    Union(fn:FileName,fp:Asp28(IMAGE)),FileName) -> Result
 ++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
```

```
    ++        liwork,m,noits,x,ifail,dot,image,monit)
    ++ finds eigenvalues of a real sparse symmetric
    ++ or generalized symmetric eigenvalue problem.
    ++ See \downlink{Manual Page}{manpageXXf02fjf}.
   f02wef : (Integer,Integer,Integer,Integer,_
       Integer,Boolean,Integer,Boolean,Integer,Matrix DoubleFloat,_
       Matrix DoubleFloat,Integer) -> Result
    ++ f02wef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldpt,a,b,ifail)
    ++ returns all, or part, of the singular value decomposition
    ++ of a general real matrix.
    ++ See \downlink{Manual Page}{manpageXXf02wef}.
   f02xef : (Integer,Integer,Integer,Integer,_
       Integer,Boolean,Integer,Boolean,Integer,Matrix Complex DoubleFloat,_
       Matrix Complex DoubleFloat,Integer) -> Result
    ++ f02xef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldph,a,b,ifail)
    ++ returns all, or part, of the singular value decomposition
    ++ of a general complex matrix.
    ++ See \downlink{Manual Page}{manpageXXf02xef}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Any
    import Record
    import Integer
    import Matrix DoubleFloat
    import Boolean
    import NAGLinkSupportPackage
    import FortranPackage
    import AnyFunctions1(Integer)
    import AnyFunctions1(Boolean)
    import AnyFunctions1(Matrix DoubleFloat)
    import AnyFunctions1(Matrix Complex DoubleFloat)
    import AnyFunctions1(DoubleFloat)


    f02aaf(iaArg:Integer,nArg:Integer,aArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f02aaf",_
        ["ia"::S,"n"::S,"ifail"::S,"r"::S,"a"::S,"e"::S]$Lisp,_
        ["r"::S,"e"::S]$Lisp,_
        [["double"::S,["r"::S,"n"::S]$Lisp,["a"::S,"ia"::S,"n"::S]$Lisp_
        ,["e"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"ia"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["r"::S,"a"::S,"ifail"::S]$Lisp,_
        [([iaArg::Any,nArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

```
f02abf(aArg:Matrix DoubleFloat,iaArg:Integer,nArg:Integer,_
    ivArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02abf",_
    ["ia"::S,"n"::S,"iv"::S,"ifail"::S,"a"::S,"r"::S,"v"::S,"e"::S]$Lisp,_
    ["r"::S,"v"::S,"e"::S]$Lisp,_
    [["double"::S,["a"::S,"ia"::S,"n"::S]$Lisp_
    ,["r"::S,"n"::S]$Lisp,["v"::S,"iv"::S,"n"::S]$Lisp,_
    ["e"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"n"::S,"iv"::S,"ifail"::S_
    ]$Lisp_
    ]$Lisp,_
    ["r"::S,"v"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,nArg::Any,ivArg::Any,ifailArg::Any,aArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02adf(iaArg:Integer,ibArg:Integer,nArg:Integer,_
    aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02adf",_
    ["ia"::S,"ib"::S,"n"::S,"ifail"::S,"r"::S,"a"::S,"b"::S,"de"::S]$Lisp,_
    ["r"::S,"de"::S]$Lisp,_
    [["double"::S,["r"::S,"n"::S]$Lisp,["a"::S,"ia"::S,"n"::S]$Lisp_
    ,["b"::S,"ib"::S,"n"::S]$Lisp,["de"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"ib"::S,"n"::S,"ifail"::S_
    ]$Lisp_
    ]$Lisp,_
    ["r"::S,"a"::S,"b"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,ibArg::Any,nArg::Any,ifailArg::Any,aArg::Any,bArg::Any])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02aef(iaArg:Integer,ibArg:Integer,nArg:Integer,_
    ivArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02aef",_
    ["ia"::S,"ib"::S,"n"::S,"iv"::S,"ifail"::S_
    ,"r"::S,"v"::S,"a"::S,"b"::S,"dl"::S_
    ,"e"::S]$Lisp,_
    ["r"::S,"v"::S,"dl"::S,"e"::S]$Lisp,_
    [["double"::S,["r"::S,"n"::S]$Lisp,["v"::S,"iv"::S,"n"::S]$Lisp_
    ,["a"::S,"ia"::S,"n"::S]$Lisp,["b"::S,"ib"::S,"n"::S]$Lisp,_
    ["dl"::S,"n"::S]$Lisp,["e"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"ia"::S,"ib"::S,"n"::S,"iv"::S_
    ,"ifail"::S]$Lisp_
```

```
    ]$Lisp,_
    ["r"::S,"v"::S,"a"::S,"b"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,ibArg::Any,nArg::Any,ivArg::Any,_
    ifailArg::Any,aArg::Any,bArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02aff(iaArg:Integer,nArg:Integer,aArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02aff",_
    ["ia"::S,"n"::S,"ifail"::S,"rr"::S,"ri"::S,"intger"::S,"a"::S]$Lisp,_
    ["rr"::S,"ri"::S,"intger"::S]$Lisp,_
    [["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_
    ,["a"::S,"ia"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"n"::S,["intger"::S,"n"::S]$Lisp_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["rr"::S,"ri"::S,"intger"::S,"a"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,nArg::Any,ifailArg::Any,aArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02agf(iaArg:Integer,nArg:Integer,ivrArg:Integer,_
    iviArg:Integer,aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02agf",_
    ["ia"::S,"n"::S,"ivr"::S,"ivi"::S,"ifail"::S_
    ,"rr"::S,"ri"::S,"vr"::S,"vi"::S,"intger"::S_
    ,"a"::S]$Lisp,_
    ["rr"::S,"ri"::S,"vr"::S,"vi"::S,"intger"::S]$Lisp,_
    [["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_
    ,["vr"::S,"ivr"::S,"n"::S]$Lisp,["vi"::S,"ivi"::S,"n"::S]$Lisp,_
    ["a"::S,"ia"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"n"::S,"ivr"::S,"ivi"::S_
    ,["intger"::S,"n"::S]$Lisp,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["rr"::S,"ri"::S,"vr"::S,"vi"::S,"intger"::S,"a"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
    ifailArg::Any,aArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02ajf(iarArg:Integer,iaiArg:Integer,nArg:Integer,_
    arArg:Matrix DoubleFloat,aiArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02ajf",_
    ["iar"::S,"iai"::S,"n"::S,"ifail"::S,"rr"::S,"ri"::S,_
    "ar"::S,"ai"::S,"intger"::S_
```

```
    ]$Lisp,_
    ["rr"::S,"ri"::S,"intger"::S]$Lisp,_
    [["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_
    ,["ar"::S,"iar"::S,"n"::S]$Lisp,["ai"::S,"iai"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"iar"::S,"iai"::S,"n"::S,"ifail"::S_
    ,["intger"::S,"n"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["rr"::S,"ri"::S,"ar"::S,"ai"::S,"ifail"::S]$Lisp,_
    [([iarArg::Any,iaiArg::Any,nArg::Any,ifailArg::Any,_
    arArg::Any,aiArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02akf(iarArg:Integer,iaiArg:Integer,nArg:Integer,_
    ivrArg:Integer,iviArg:Integer,arArg:Matrix DoubleFloat,_
    aiArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02akf",_
    ["iar"::S,"iai"::S,"n"::S,"ivr"::S,"ivi"::S_
    ,"ifail"::S,"rr"::S,"ri"::S,"vr"::S,"vi"::S,"ar"::S_
    ,"ai"::S,"intger"::S]$Lisp,_
    ["rr"::S,"ri"::S,"vr"::S,"vi"::S,"intger"::S]$Lisp,_
    [["double"::S,["rr"::S,"n"::S]$Lisp,["ri"::S,"n"::S]$Lisp_
    ,["vr"::S,"ivr"::S,"n"::S]$Lisp,["vi"::S,"ivi"::S,"n"::S]$Lisp,_
    ["ar"::S,"iar"::S,"n"::S]$Lisp,["ai"::S,"iai"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"iar"::S,"iai"::S,"n"::S,"ivr"::S_
    ,"ivi"::S,"ifail"::S,["intger"::S,"n"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["rr"::S,"ri"::S,"vr"::S,"vi"::S,"ar"::S,"ai"::S,"ifail"::S]$Lisp,_
    [([iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
    ifailArg::Any,arArg::Any,aiArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02awf(iarArg:Integer,iaiArg:Integer,nArg:Integer,_
    arArg:Matrix DoubleFloat,aiArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02awf",_
    ["iar"::S,"iai"::S,"n"::S,"ifail"::S,"r"::S,"ar"::S,"ai"::S,_
    "wk1"::S,"wk2"::S_
    ,"wk3"::S]$Lisp,_
    ["r"::S,"wk1"::S,"wk2"::S,"wk3"::S]$Lisp,_
    [["double"::S,["r"::S,"n"::S]$Lisp,["ar"::S,"iar"::S,"n"::S]$Lisp_
    ,["ai"::S,"iai"::S,"n"::S]$Lisp,["wk1"::S,"n"::S]$Lisp,_
    ["wk2"::S,"n"::S]$Lisp,["wk3"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"iar"::S,"iai"::S,"n"::S,"ifail"::S_
    ]$Lisp_
```

```
    ]$Lisp,_
    ["r"::S,"ar"::S,"ai"::S,"ifail"::S]$Lisp,_
    [([iarArg::Any,iaiArg::Any,nArg::Any,ifailArg::Any,arArg::Any,_
    aiArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02axf(arArg:Matrix DoubleFloat,iarArg:Integer,aiArg:Matrix DoubleFloat,_
    iaiArg:Integer,nArg:Integer,ivrArg:Integer,_
    iviArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02axf",_
    ["iar"::S,"iai"::S,"n"::S,"ivr"::S,"ivi"::S_
    ,"ifail"::S,"ar"::S,"ai"::S,"r"::S,"vr"::S,"vi"::S_
    ,"wk1"::S,"wk2"::S,"wk3"::S]$Lisp,_
    ["r"::S,"vr"::S,"vi"::S,"wk1"::S,"wk2"::S,"wk3"::S]$Lisp,_
    [["double"::S,["ar"::S,"iar"::S,"n"::S]$Lisp_
    ,["ai"::S,"iai"::S,"n"::S]$Lisp,["r"::S,"n"::S]$Lisp,_
    ["vr"::S,"ivr"::S,"n"::S]$Lisp,["vi"::S,"ivi"::S,"n"::S]$Lisp,_
    ["wk1"::S,"n"::S]$Lisp_
    ,["wk2"::S,"n"::S]$Lisp,["wk3"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"iar"::S,"iai"::S,"n"::S,"ivr"::S_
    ,"ivi"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["r"::S,"vr"::S,"vi"::S,"ifail"::S]$Lisp,_
    [([iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
    ifailArg::Any,arArg::Any,aiArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02bbf(iaArg:Integer,nArg:Integer,albArg:DoubleFloat,_
    ubArg:DoubleFloat,mArg:Integer,ivArg:Integer,_
    aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f02bbf",_
    ["ia"::S,"n"::S,"alb"::S,"ub"::S,"m"::S_
    ,"iv"::S,"mm"::S,"ifail"::S,"r"::S,"v"::S,"icount"::S,"a"::S,"d"::S_
    ,"e"::S,"e2"::S,"x"::S,"g"::S,"c"::S_
    ]$Lisp,_
    ["mm"::S,"r"::S,"v"::S,"icount"::S,"d"::S,"e"::S,"e2"::S,"x"::S,_
    "g"::S,"c"::S]$Lisp,_
    [["double"::S,"alb"::S,"ub"::S,["r"::S,"m"::S]$Lisp_
    ,["v"::S,"iv"::S,"m"::S]$Lisp,["a"::S,"ia"::S,"n"::S]$Lisp,_
    ["d"::S,"n"::S]$Lisp,["e"::S,"n"::S]$Lisp,["e2"::S,"n"::S]$Lisp_
    ,["x"::S,"n"::S,7$Lisp]$Lisp,["g"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"n"::S,"m"::S,"iv"::S_
    ,"mm"::S,["icount"::S,"m"::S]$Lisp,"ifail"::S]$Lisp_
    ,["logical"::S,["c"::S,"n"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["mm"::S,"r"::S,"v"::S,"icount"::S,"a"::S,"ifail"::S]$Lisp,_
```

```
        [([iaArg::Any,nArg::Any,albArg::Any,ubArg::Any,mArg::Any,_
        ivArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f02bjf(nArg:Integer,iaArg:Integer,ibArg:Integer,_
        eps1Arg:DoubleFloat,matvArg:Boolean,ivArg:Integer,_
        aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f02bjf",_
        ["n"::S,"ia"::S,"ib"::S,"eps1"::S,"matv"::S_
        ,"iv"::S,"ifail"::S,"alfr"::S,"alfi"::S,"beta"::S,"v"::S,"iter"::S_
        ,"a"::S,"b"::S]$Lisp,_
        ["alfr"::S,"alfi"::S,"beta"::S,"v"::S,"iter"::S]$Lisp,_
        [["double"::S,"eps1"::S,["alfr"::S,"n"::S]$Lisp_
        ,["alfi"::S,"n"::S]$Lisp,["beta"::S,"n"::S]$Lisp,_
        ["v"::S,"iv"::S,"n"::S]$Lisp,["a"::S,"ia"::S,"n"::S]$Lisp,_
        ["b"::S,"ib"::S,"n"::S]$Lisp_
        ]$Lisp_
        ,["integer"::S,"n"::S,"ia"::S,"ib"::S,"iv"::S_
        ,["iter"::S,"n"::S]$Lisp,"ifail"::S]$Lisp_
        ,["logical"::S,"matv"::S]$Lisp_
        ]$Lisp,_
        ["alfr"::S,"alfi"::S,"beta"::S,"v"::S,"iter"::S,"a"::S,"b"::S,_
        "ifail"::S]$Lisp,_
        [([nArg::Any,iaArg::Any,ibArg::Any,eps1Arg::Any,matvArg::Any,_
        ivArg::Any,ifailArg::Any,aArg::Any,bArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f02fjf(nArg:Integer,kArg:Integer,tolArg:DoubleFloat,_
        novecsArg:Integer,nrxArg:Integer,lworkArg:Integer,_
        lrworkArg:Integer,liworkArg:Integer,mArg:Integer,_
        noitsArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
        dotArg:Union(fn:FileName,fp:Asp27(DOT)),_
        imageArg:Union(fn:FileName,fp:Asp28(IMAGE))): Result ==
        pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
        if dotArg case fn
                  then outputAsFortran(dotArg.fn)
                  else outputAsFortran(dotArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
        if imageArg case fn
                  then outputAsFortran(imageArg.fn)
                  else outputAsFortran(imageArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
        outputAsFortran()$Asp29(MONIT)
        popFortranOutputStack()$FOP
```

```
    [(invokeNagman([dotFilename,imageFilename,monitFilename]$Lisp,_
    "f02fjf",_
    ["n"::S,"k"::S,"tol"::S,"novecs"::S,"nrx"::S_
    ,"lwork"::S,"lrwork"::S,"liwork"::S,"m"::S,"noits"::S_
    ,"ifail"::S,"dot"::S,"image"::S,"monit"::S,"d"::S,"x"::S,_
    "work"::S,"rwork"::S,"iwork"::S_
    ]$Lisp,_
    ["d"::S,"work"::S,"rwork"::S,"iwork"::S,"dot"::S,"image"::S,_
    "monit"::S]$Lisp,_
    [["double"::S,"tol"::S,["d"::S,"k"::S]$Lisp_
    ,["x"::S,"nrx"::S,"k"::S]$Lisp,["work"::S,"lwork"::S]$Lisp,_
    ["rwork"::S,"lrwork"::S]$Lisp,"dot"::S,"image"::S,"monit"::S_
    ]$Lisp_
    ,["integer"::S,"n"::S,"k"::S,"novecs"::S,"nrx"::S_
    ,"lwork"::S,"lrwork"::S,"liwork"::S,"m"::S,"noits"::S,"ifail"::S,_
    ["iwork"::S,"liwork"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["d"::S,"m"::S,"noits"::S,"x"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,kArg::Any,tolArg::Any,novecsArg::Any,nrxArg::Any,_
    lworkArg::Any,lrworkArg::Any,liworkArg::Any,mArg::Any,_
    noitsArg::Any,ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02fjf(nArg:Integer,kArg:Integer,tolArg:DoubleFloat,_
    novecsArg:Integer,nrxArg:Integer,lworkArg:Integer,_
    lrworkArg:Integer,liworkArg:Integer,mArg:Integer,_
    noitsArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
    dotArg:Union(fn:FileName,fp:Asp27(DOT)),_
    imageArg:Union(fn:FileName,fp:Asp28(IMAGE)),_
    monitArg:FileName): Result ==
    pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
    if dotArg case fn
              then outputAsFortran(dotArg.fn)
              else outputAsFortran(dotArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
    if imageArg case fn
              then outputAsFortran(imageArg.fn)
              else outputAsFortran(imageArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
    outputAsFortran(monitArg)
    [(invokeNagman([dotFilename,imageFilename,monitFilename]$Lisp,_
    "f02fjf",_
    ["n"::S,"k"::S,"tol"::S,"novecs"::S,"nrx"::S_
    ,"lwork"::S,"lrwork"::S,"liwork"::S,"m"::S,"noits"::S_
    ,"ifail"::S,"dot"::S,"image"::S,"monit"::S,"d"::S,"x"::S,_
    "work"::S,"rwork"::S,"iwork"::S_
    ]$Lisp,_
```

```
    ["d"::S,"work"::S,"rwork"::S,"iwork"::S,"dot"::S,"image"::S,_
    "monit"::S]$Lisp,_
    [["double"::S,"tol"::S,["d"::S,"k"::S]$Lisp_
    ,["x"::S,"nrx"::S,"k"::S]$Lisp,["work"::S,"lwork"::S]$Lisp,_
    ["rwork"::S,"lrwork"::S]$Lisp,"dot"::S,"image"::S,"monit"::S_
    ]$Lisp_
    ,["integer"::S,"n"::S,"k"::S,"novecs"::S,"nrx"::S_
    ,"lwork"::S,"lrwork"::S,"liwork"::S,"m"::S,"noits"::S,"ifail"::S,_
    ["iwork"::S,"liwork"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["d"::S,"m"::S,"noits"::S,"x"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,kArg::Any,tolArg::Any,novecsArg::Any,nrxArg::Any,_
    lworkArg::Any,lrworkArg::Any,liworkArg::Any,mArg::Any,_
    noitsArg::Any,ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f02wef(mArg:Integer,nArg:Integer,ldaArg:Integer,_
    ncolbArg:Integer,ldbArg:Integer,wantqArg:Boolean,_
    ldqArg:Integer,wantpArg:Boolean,ldptArg:Integer,_
    aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    workLength : Integer :=
      mArg >= nArg =>
        wantqArg and wantpArg =>
          max(max(nArg**2 + 5*(nArg - 1),nArg + ncolbArg),4)
        wantqArg =>
          max(max(nArg**2 + 4*(nArg - 1),nArg + ncolbArg),4)
        wantpArg =>
          zero? ncolbArg => max(3*(nArg - 1),2)
          max(5*(nArg - 1),2)
        zero? ncolbArg => max(2*(nArg - 1),2)
        max(3*(nArg - 1),2)
      wantqArg and wantpArg =>
        max(mArg**2 + 5*(mArg - 1),2)
      wantqArg =>
        max(3*(mArg - 1),1)
      wantpArg =>
        zero? ncolbArg => max(mArg**2+3*(mArg - 1),2)
        max(mArg**2+5*(mArg - 1),2)
      zero? ncolbArg => max(2*(mArg - 1),1)
      max(3*(mArg - 1),1)

    [(invokeNagman(NIL$Lisp,_
    "f02wef",_
    ["m"::S,"n"::S,"lda"::S,"ncolb"::S,"ldb"::S_
    ,"wantq"::S,"ldq"::S,"wantp"::S,"ldpt"::S,"ifail"::S_
    ,"q"::S,"sv"::S,"pt"::S,"work"::S,"a"::S_
    ,"b"::S]$Lisp,_
    ["q"::S,"sv"::S,"pt"::S,"work"::S]$Lisp,_
```

```
      [["double"::S,["q"::S,"ldq"::S,"m"::S]$Lisp_
      ,["sv"::S,"m"::S]$Lisp,["pt"::S,"ldpt"::S,"n"::S]$Lisp,_
      ["work"::S,workLength]$Lisp,["a"::S,"lda"::S,"n"::S]$Lisp,_
      ["b"::S,"ldb"::S,"ncolb"::S]$Lisp_
      ]$Lisp_
      ,["integer"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S_
      ,"ldb"::S,"ldq"::S,"ldpt"::S,"ifail"::S]$Lisp_
      ,["logical"::S,"wantq"::S,"wantp"::S]$Lisp_
      ]$Lisp,_
      ["q"::S,"sv"::S,"pt"::S,"work"::S,"a"::S,"b"::S,"ifail"::S]$Lisp,_
      [([mArg::Any,nArg::Any,ldaArg::Any,ncolbArg::Any,ldbArg::Any,_
      wantqArg::Any,ldqArg::Any,wantpArg::Any,ldptArg::Any,_
      ifailArg::Any,aArg::Any,bArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result

  f02xef(mArg:Integer,nArg:Integer,ldaArg:Integer,_
      ncolbArg:Integer,ldbArg:Integer,wantqArg:Boolean,_
      ldqArg:Integer,wantpArg:Boolean,ldphArg:Integer,_
      aArg:Matrix Complex DoubleFloat,bArg:Matrix Complex DoubleFloat,_
      ifailArg:Integer): Result ==
      -- This segment added by hand, to deal with an assumed size array GDN
      tem : Integer := (min(mArg,nArg)-1)
      rLen : Integer :=
        zero? ncolbArg and not wantqArg and not wantpArg => 2*tem
        zero? ncolbArg and wantpArg and not wantqArg => 3*tem
        not wantpArg =>
          ncolbArg >0 or wantqArg => 3*tem
        5*tem
      cLen : Integer :=
        mArg >= nArg =>
          wantqArg and wantpArg => 2*(nArg + max(nArg**2,ncolbArg))
          wantqArg and not wantpArg => 2*(nArg + max(nArg**2+nArg,ncolbArg))
          2*(nArg + max(nArg,ncolbArg))
        wantpArg => 2*(mArg**2 + mArg)
        2*mArg
      svLength : Integer :=
        min(mArg,nArg)
      [(invokeNagman(NIL$Lisp,_
      "f02xef",_
      ["m"::S,"n"::S,"lda"::S,"ncolb"::S,"ldb"::S_
      ,"wantq"::S,"ldq"::S,"wantp"::S,"ldph"::S,"ifail"::S_
      ,"q"::S,"sv"::S,"ph"::S,"rwork"::S,"a"::S_
      ,"b"::S,"cwork"::S]$Lisp,_
      ["q"::S,"sv"::S,"ph"::S,"rwork"::S,"cwork"::S]$Lisp,_
      [["double"::S,["sv"::S,svLength]$Lisp,["rwork"::S,rLen]$Lisp_
      ]$Lisp_
      ,["integer"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S_
      ,"ldb"::S,"ldq"::S,"ldph"::S,"ifail"::S]$Lisp_
      ,["logical"::S,"wantq"::S,"wantp"::S]$Lisp_
```

```
,["double complex"::S,["q"::S,"ldq"::S,"m"::S]$Lisp,_
["ph"::S,"ldph"::S,"n"::S]$Lisp,["a"::S,"lda"::S,"n"::S]$Lisp,_
["b"::S,"ldb"::S,"ncolb"::S]$Lisp,["cwork"::S,cLen]$Lisp]$Lisp_
]$Lisp,_
["q"::S,"sv"::S,"ph"::S,"rwork"::S,"a"::S,"b"::S,"ifail"::S]$Lisp,_
[([mArg::Any,nArg::Any,ldaArg::Any,ncolbArg::Any,ldbArg::Any,_
wantqArg::Any,ldqArg::Any,wantpArg::Any,ldphArg::Any,_
ifailArg::Any,aArg::Any,bArg::Any ])_
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGF02.dotabb** —

```
"NAGF02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE02" -> "ALIST"
```

———-

15.2 package NAGE02 NagFittingPackage

— **NagFittingPackage.help** —

```
E02 -- Curve and Surface Fitting              Introduction -- E02
                          Chapter E02
                  Curve and Surface Fitting

Contents of this Introduction:

1.     Scope of the Chapter

2.     Background to the Problems

2.1.   Preliminary Considerations

2.1.1. Fitting criteria: norms

2.1.2. Weighting of data points

2.2.   Curve Fitting

2.2.1. Representation of polynomials
```

2.2.2. Representation of cubic splines

2.3. Surface Fitting

2.3.1. Bicubic splines: definition and representation

2.4. General Linear and Nonlinear Fitting Functions

2.5. Constrained Problems

2.6. References

3. Recommendations on Choice and Use of Routines

3.1. General

3.1.1. Data considerations

3.1.2. Transformation of variables

3.2. Polynomial Curves

3.2.1. Least-squares polynomials: arbitrary data points

3.2.2. Least-squares polynomials: selected data points

3.3. Cubic Spline Curves

3.3.1. Least-squares cubic splines

3.3.2. Automatic fitting with cubic splines

3.4. Spline Surfaces

3.4.1. Least-squares bicubic splines

3.4.2. Automatic fitting with bicubic splines

3.5. General Linear and Nonlinear Fitting Functions

3.5.1. General linear functions

3.5.2. Nonlinear functions

3.6. Constraints

3.7. Evaluation, Differentiation and Integration

3.8. Index

1. Scope of the Chapter

The main aim of this chapter is to assist the user in finding a function which approximates a set of data points. Typically the data contain random errors, as of experimental measurement, which need to be smoothed out. To seek an approximation to the data, it is first necessary to specify for the approximating function a mathematical form (a polynomial, for example) which contains a number of unspecified coefficients: the appropriate fitting routine then derives for the coefficients the values which provide the best fit of that particular form. The chapter deals mainly with curve and surface fitting (i.e., fitting with functions of one and of two variables) when a polynomial or a cubic spline is used as the fitting function, since these cover the most common needs. However, fitting with other functions and/or more variables can be undertaken by means of general linear or nonlinear routines (some of which are contained in other chapters) depending on whether the coefficients in the function occur linearly or nonlinearly. Cases where a graph rather than a set of data points is given can be treated simply by first reading a suitable set of points from the graph.

The chapter also contains routines for evaluating, differentiating and integrating polynomial and spline curves and surfaces, once the numerical values of their coefficients have been determined.

2. Background to the Problems

2.1. Preliminary Considerations

In the curve-fitting problems considered in this chapter, we have a dependent variable y and an independent variable x, and we are given a set of data points (x_r, y_r), for r=1,2,...,m. The preliminary matters to be considered in this section will, for simplicity, be discussed in this context of curve-fitting problems. In fact, however, these considerations apply equally well to surface and higher-dimensional problems. Indeed, the discussion presented carries over essentially as it stands if, for these cases, we interpret x as a vector of several independent variables and correspondingly each x_r as a vector containing the rth data value of each independent variable.

We wish, then, to approximate the set of data points as closely as possible with a specified function, f(x) say, which is as

smooth as possible -- f(x) may, for example, be a polynomial. The requirements of smoothness and closeness conflict, however, and a balance has to be struck between them. Most often, the smoothness requirement is met simply by limiting the number of coefficients allowed in the fitting function -- for example, by restricting the degree in the case of a polynomial. Given a particular number of coefficients in the function in question, the fitting routines of this chapter determine the values of the coefficients such that the 'distance' of the function from the data points is as small as possible. The necessary balance is struck by the user comparing a selection of such fits having different numbers of coefficients. If the number of coefficients is too low, the approximation to the data will be poor. If the number is too high, the fit will be too close to the data, essentially following the random errors and tending to have unwanted fluctuations between the data points. Between these extremes, there is often a group of fits all similarly close to the data points and then, particularly when least-squares polynomials are used, the choice is clear: it is the fit from this group having the smallest number of coefficients.

The above process can be seen as the user minimizing the smoothness measure (i.e., the number of coefficients) subject to the distance from the data points being acceptably small. Some of the routines, however, do this task themselves. They use a different measure of smoothness (in each case one that is continuous) and minimize it subject to the distance being less than a threshold specified by the user. This is a much more automatic process, requiring only some experimentation with the threshold.

2.1.1. Fitting criteria: norms

A measure of the above 'distance' between the set of data points and the function f(x) is needed. The distance from a single data point (x_r, y_r) to the function can simply be taken as

$$(\text{epsilon})_r = y_r - f(x_r), \qquad (1)$$

and is called the residual of the point. (With this definition, the residual is regarded as a function of the coefficients contained in f(x); however, the term is also used to mean the particular value of $(\text{epsilon})_r$ which corresponds to the fitted values of the coefficients.) However, we need a measure of distance for the set of data points as a whole. Three different measures are used in the different routines (which measure to select, according to circumstances, is discussed later in this

sub-section). With (epsilon) defined in (1), these measures, or
r
norms, are

$$\sum_{r=1}^{m} |(\text{epsilon})_r|, \tag{2}$$

$$\left(\sum_{r=1}^{m} (\text{epsilon})_r^2 \right)^{1/2}, \text{ and} \tag{3}$$

$$\max_r |(\text{epsilon})_r|, \tag{4}$$

respectively the l norm, the l norm and the l norm.
1 2 infty

Minimization of one or other of these norms usually provides the
fitting criterion, the minimization being carried out with
respect to the coefficients in the mathematical form used for
f(x): with respect to the b for example if the mathematical form
i
is the power series in (8) below. The fit which results from
minimizing (2) is known as the l fit, or the fit in the l norm:
1 1
that which results from minimizing (3) is the l fit, the well-
2
known least-squares fit (minimizing (3) is equivalent to
minimizing the square of (3), i.e., the sum of squares of
residuals, and it is the latter which is used in practice), and
that from minimizing (4) is the l , or minimax, fit.
infty

Strictly speaking, implicit in the use of the above norms are the
statistical assumptions that the random errors in the y are
r
independent of one another and that any errors in the x are
r
negligible by comparison. From this point of view, the use of the
l norm is appropriate when the random errors in the y have a
2 r
normal distribution, and the l norm is appropriate when they
infty

have a rectangular distribution, as when fitting a table of values rounded to a fixed number of decimal places. The l_1 norm is appropriate when the error distribution has its frequency function proportional to the negative exponential of the modulus of the normalised error -- not a common situation.

However, the user is often indifferent to these statistical considerations, and simply seeks a fit which he can assess by inspection, perhaps visually from a graph of the results. In this event, the l_1 norm is particularly appropriate when the data are thought to contain some 'wild' points (since fitting in this norm tends to be unaffected by the presence of a small number of such points), though of course in simple situations the user may prefer to identify and reject these points. The l_∞ norm should be used only when the maximum residual is of particular concern, as may be the case for example when the data values have been obtained by accurate computation, as of a mathematical function. Generally, however, a routine based on least-squares should be preferred, as being computationally faster and usually providing more information on which to assess the results. In many problems the three fits will not differ significantly for practical purposes.

Some of the routines based on the l_2 norm do not minimize the norm itself but instead minimize some (intuitively acceptable) measure of smoothness subject to the norm being less than a user-specified threshold. These routines fit with cubic or bicubic splines (see (10) and (14) below) and the smoothing measures relate to the size of the discontinuities in their third derivatives. A much more automatic fitting procedure follows from this approach.

2.1.2. Weighting of data points

The use of the above norms also assumes that the data values y_r are of equal (absolute) accuracy. Some of the routines enable an allowance to be made to take account of differing accuracies. The allowance takes the form of 'weights' applied to the y-values so that those values known to be more accurate have a greater influence on the fit than others. These weights, to be supplied by the user, should be calculated from estimates of the absolute accuracies of the y-values, these estimates being expressed as standard deviations, probable errors or some other measure which has the same dimensions as y. Specifically, for each y_r the

corresponding weight w_r should be inversely proportional to the accuracy estimate of y_r. For example, if the percentage accuracy is the same for all y_r, then the absolute accuracy of y_r is proportional to y_r (assuming y_r to be positive, as it usually is in such cases) and so $w_r = K/y_r$, for r=1,2,...,m, for an arbitrary positive constant K. (This definition of weight is stressed because often weight is defined as the square of that used here.) The norms (2), (3) and (4) above are then replaced respectively by

$$\sum_{r=1}^{m} |w_r(\epsilon)_r|, \tag{5}$$

$$\left(\sum_{r=1}^{m} w_r^2(\epsilon)_r^2 \right)^{1/2}, \text{ and} \tag{6}$$

$$\max_r |w_r(\epsilon)_r|. \tag{7}$$

Again it is the square of (6) which is used in practice rather than (6) itself.

2.2. Curve Fitting

When, as is commonly the case, the mathematical form of the fitting function is immaterial to the problem, polynomials and cubic splines are to be preferred because their simplicity and ease of handling confer substantial benefits. The cubic spline is the more versatile of the two. It consists of a number of cubic polynomial segments joined end to end with continuity in first and second derivatives at the joins. The third derivative at the joins is in general discontinuous. The x-values of the joins are called knots, or, more precisely, interior knots. Their number determines the number of coefficients in the spline, just as the degree determines the number of coefficients in a polynomial.

2.2.1. Representation of polynomials

Rather than using the power-series form

$$f(x)==b_0+b_1 x+b_2 x^2+\ldots+b_k x^k \qquad (8)$$

to represent a polynomial, the routines in this chapter use the Chebyshev series form

$$f(x)==\tfrac{1}{2}a_0 T_0(x)+a_1 T_1(x)+a_2 T_2(x)+\ldots+a_k T_k(x), \qquad (9)$$

where $T_i(x)$ is the Chebyshev polynomial of the first kind of degree i (see Cox and Hayes [1], page 9), and where the range of x has been normalised to run from -1 to +1. The use of either form leads theoretically to the same fitted polynomial, but in practice results may differ substantially because of the effects of rounding error. The Chebyshev form is to be preferred, since it leads to much better accuracy in general, both in the computation of the coefficients and in the subsequent evaluation of the fitted polynomial at specified points. This form also has other advantages: for example, since the later terms in (9) generally decrease much more rapidly from left to right than do those in (8), the situation is more often encountered where the last terms are negligible and it is obvious that the degree of the polynomial can be reduced (note that on the interval -1<=x<=1 for all i, $T_i(x)$ attains the value unity but never exceeds it, so that the coefficient a_i gives directly the maximum value of the term containing it).

2.2.2. Representation of cubic splines

A cubic spline is represented in the form

$$f(x)==c_1 N_1(x)+c_2 N_2(x)+\ldots+c_p N_p(x), \qquad (10)$$

where $N_i(x)$, for i=1,2,...,p, is a normalised cubic B-spline (see Hayes [2]). This form, also, has advantages of computational speed and accuracy over alternative representations.

2.3. Surface Fitting

There are now two independent variables, and we shall denote

these by x and y. The dependent variable, which was denoted by y in the curve-fitting case, will now be denoted by f. (This is a rather different notation from that indicated for the general-dimensional problem in the first paragraph of Section 2.1 , but it has some advantages in presentation.)

Again, in the absence of contrary indications in the particular application being considered, polynomials and splines are the approximating functions most commonly used. Only splines are used by the surface-fitting routines in this chapter.

2.3.1. Bicubic splines: definition and representation

The bicubic spline is defined over a rectangle R in the (x,y) plane, the sides of R being parallel to the x- and y-axes. R is divided into rectangular panels, again by lines parallel to the axes. Over each panel the bicubic spline is a bicubic polynomial, that is it takes the form

$$\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}\, x^i y^j . \qquad (13)$$

Each of these polynomials joins the polynomials in adjacent panels with continuity up to the second derivative. The constant x-values of the dividing lines parallel to the y-axis form the set of interior knots for the variable x, corresponding precisely to the set of interior knots of a cubic spline. Similarly, the constant y-values of dividing lines parallel to the x-axis form the set of interior knots for the variable y. Instead of representing the bicubic spline in terms of the above set of bicubic polynomials, however, it is represented, for the sake of computational speed and accuracy, in the form

$$f(x,y) = \sum_{i=1}^{p}\sum_{j=1}^{q} c_{ij} M_i(x) N_j(y), \qquad (14)$$

where $M_i(x)$, for i=1,2,...,p, and $N_j(y)$, for j=1,2,...,q, are normalised B-splines (see Hayes and Halliday [4] for further details of bicubic splines and Hayes [2] for normalised B-splines).

2.4. General Linear and Nonlinear Fitting Functions

We have indicated earlier that, unless the data-fitting application under consideration specifically requires some other type of fitting function, a polynomial or a spline is usually to be preferred. Special routines for these functions, in one and in two variables, are provided in this chapter. When the application does specify some other fitting function, however, it may be treated by a routine which deals with a general linear function, or by one for a general nonlinear function, depending on whether the coefficients in the given function occur linearly or nonlinearly.

The general linear fitting function can be written in the form

$$f(x)=c_1(\phi)_1(x)+c_2(\phi)_2(x)+\ldots+c_p(\phi)_p(x), \qquad (15)$$

where x is a vector of one or more independent variables, and the $(\phi)_i$ are any given functions of these variables (though they must be linearly independent of one another if there is to be the possibility of a unique solution to the fitting problem). This is not intended to imply that each $(\phi)_i$ is necessarily a function of all the variables: we may have, for example, that each $(\phi)_i$ is a function of a different single variable, and even that one of the $(\phi)_i$ is a constant. All that is required is that a value of each $(\phi)_i(x)$ can be computed when a value of each independent variable is given.

When the fitting function f(x) is not linear in its coefficients, no more specific representation is available in general than f(x) itself. However, we shall find it helpful later on to indicate the fact that f(x) contains a number of coefficients (to be determined by the fitting process) by using instead the notation f(x;c), where c denotes the vector of coefficients. An example of a nonlinear fitting function is

$$f(x;c)=c_1+c_2\exp(-c_4x)+c_3\exp(-c_5x), \qquad (16)$$

which is in one variable and contains five coefficients. Note that here, as elsewhere in this Chapter Introduction, we use the term 'coefficients' to include all the quantities whose values are to be determined by the fitting process, not just those which occur linearly. We may observe that it is only the presence of the coefficients c_4 and c_5 which makes the form (16) nonlinear.

4 5
If the values of these two coefficients were known beforehand, (16) would instead be a linear function which, in terms of the general linear form (15), has p=3 and

(phi) (x)==1, (phi) (x)==exp(-c x), and (phi) (x)==exp(-c x).
1 2 4 3 5

We may note also that polynomials and splines, such as (9) and (14), are themselves linear in their coefficients. Thus if, when fitting with these functions, a suitable special routine is not available (as when more than two independent variables are involved or when fitting in the l norm), it is appropriate to
1
use a routine designed for a general linear function.

2.5. Constrained Problems

So far, we have considered only fitting processes in which the values of the coefficients in the fitting function are determined by an unconstrained minimization of a particular norm. Some fitting problems, however, require that further restrictions be placed on the determination of the coefficient values. Sometimes these restrictions are contained explicitly in the formulation of the problem in the form of equalities or inequalities which the coefficients, or some function of them, must satisfy. For example, if the fitting function contains a term Aexp(-kx), it may be required that k>=0. Often, however, the equality or inequality constraints relate to the value of the fitting function or its derivatives at specified values of the independent variable(s), but these too can be expressed in terms of the coefficients of the fitting function, and it is appropriate to do this if a general linear or nonlinear routine is being used. For example, if the fitting function is that given in (10), the requirement that the first derivative of the function at x=x be non-negative can be expressed as
0

c N '(x)+c N '(x)+...+c N '(x)>=0, (17)
1 1 0 2 2 0 p p 0

where the prime denotes differentiation with respect to x and each derivative is evaluated at x=x . On the other hand, if the
0
requirement had been that the derivative at x=x be exactly zero,
0
the inequality sign in (17) would be replaced by an equality.

Routines which provide a facility for minimizing the appropriate norm subject to such constraints are discussed in Section 3.6.

2.6. References

[1] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[2] Hayes J G (1974) Numerical Methods for Curve and Surface Fitting. Bull Inst Math Appl. 10 144--152.

(For definition of normalised B-splines and details of numerical methods.)

[3] Hayes J G (1970) Curve Fitting by Polynomials in One Variable. Numerical Approximation to Functions and Data. (ed J G Hayes) Athlone Press, London.

[4] Hayes J G and Halliday J (1974) The Least-squares Fitting of Cubic Spline Surfaces to General Data Sets. J. Inst. Math. Appl. 14 89--103.

3. Recommendations on Choice and Use of Routines

3.1. General

The choice of a routine to treat a particular fitting problem will depend first of all on the fitting function and the norm to be used. Unless there is good reason to the contrary, the fitting function should be a polynomial or a cubic spline (in the appropriate number of variables) and the norm should be the l_2 norm (leading to the least-squares fit). If some other function is to be used, the choice of routine will depend on whether the function is nonlinear (in which case see Section 3.5.2) or linear in its coefficients (see Section 3.5.1), and, in the latter case, on whether the l_1 or l_2 norm is to be used. The latter section is appropriate for polynomials and splines, too, if the l_1 norm is preferred.

In the case of a polynomial or cubic spline, if there is only one independent variable, the user should choose a spline (Section 3.3) when the curve represented by the data is of complicated form, perhaps with several peaks and troughs. When the curve is of simple form, first try a polynomial (see Section 3.2) of low degree, say up to degree 5 or 6, and then a spline if the polynomial fails to provide a satisfactory fit. (Of course, if third-derivative discontinuities are unacceptable to the user, a polynomial is the only choice.) If the problem is one of surface

fitting, one of the spline routines should be used (Section 3.4). If the problem has more than two independent variables, it may be treated by the general linear routine in Section 3.5.1, again using a polynomial in the first instance.

Another factor which affects the choice of routine is the presence of constraints, as previously discussed in Section 2.5. Indeed this factor is likely to be overriding at present, because of the limited number of routines which have the necessary facility. See Section 3.6.

3.1.1. Data considerations

A satisfactory fit cannot be expected by any means if the number and arrangement of the data points do not adequately represent the character of the underlying relationship: sharp changes in behaviour, in particular, such as sharp peaks, should be well covered. Data points should extend over the whole range of interest of the independent variable(s): extrapolation outside the data ranges is most unwise. Then, with polynomials, it is advantageous to have additional points near the ends of the ranges, to counteract the tendency of polynomials to develop fluctuations in these regions. When, with polynomial curves, the user can precisely choose the x-values of the data, the special points defined in Section 3.2.2 should be selected. With splines the choice is less critical as long as the character of the relationship is adequately represented. All fits should be tested graphically before accepting them as satisfactory.

For this purpose it should be noted that it is not sufficient to plot the values of the fitted function only at the data values of the independent variable(s); at the least, its values at a similar number of intermediate points should also be plotted, as unwanted fluctuations may otherwise go undetected. Such fluctuations are the less likely to occur the lower the number of coefficients chosen in the fitting function. No firm guide can be given, but as a rough rule, at least initially, the number of coefficients should not exceed half the number of data points (points with equal or nearly equal values of the independent variable, or both independent variables in surface fitting, counting as a single point for this purpose). However, the situation may be such, particularly with a small number of data points, that a satisfactorily close fit to the data cannot be achieved without unwanted fluctuations occurring. In such cases, it is often possible to improve the situation by a transformation of one or more of the variables, as discussed in the next paragraph: otherwise it will be necessary to provide extra data points. Further advice on curve fitting is given in Cox and Hayes [1] and, for polynomials only, in Hayes [3] of Section 2.7. Much of the advice applies also to surface fitting; see also the

Routine Documents.

3.1.2. Transformation of variables

Before starting the fitting, consideration should be given to the choice of a good form in which to deal with each of the variables: often it will be satisfactory to use the variables as they stand, but sometimes the use of the logarithm, square root, or some other function of a variable will lead to a better-behaved relationship. This question is customarily taken into account in preparing graphs and tables of a relationship and the same considerations apply when curve or surface fitting. The practical context will often give a guide. In general, it is best to avoid having to deal with a relationship whose behaviour in one region is radically different from that in another. A steep rise at the left-hand end of a curve, for example, can often best be treated by curve fitting in terms of log(x+c) with some suitable value of the constant c. A case when such a transformation gave substantial benefit is discussed in Hayes [3] page 60. According to the features exhibited in any particular case, transformation of either dependent variable or independent variable(s) or both may be beneficial. When there is a choice it is usually better to transform the independent variable(s): if the dependent variable is transformed, the weights attached to the data points must be adjusted. Thus (denoting the dependent variable by y, as in the notation for curves) if the y_r to be fitted have been obtained by a transformation y=g(Y) from original data values Y_r, with weights W_r, for r=1,2,...,m, we must take

$$w_r = W_r / (dy/dY), \qquad (18)$$

where the derivative is evaluated at Y_r. Strictly, the transformation of Y and the adjustment of weights are valid only when the data errors in the Y_r are small compared with the range spanned by the Y_r, but this is usually the case.

3.2. Polynomial Curves

3.2.1. Least-squares polynomials: arbitrary data points

E02ADF fits to arbitrary data points, with arbitrary weights, polynomials of all degrees up to a maximum degree k, which is at choice. If the user is seeking only a low degree polynomial, up

to degree 5 or 6 say, k=10 is an appropriate value, providing
there are about 20 data points or more. To assist in deciding the
degree of polynomial which satisfactorily fits the data, the
routine provides the root-mean-square-residual s_i for all degrees
i=1,2,...,k. In a satisfactory case, these s_i will decrease
steadily as i increases and then settle down to a fairly constant
value, as shown in the example

i	s_i
0	3.5215
1	0.7708
2	0.1861
3	0.0820
4	0.0554
5	0.0251
6	0.0264
7	0.0280
8	0.0277
9	0.0297
10	0.0271

If the s_i values settle down in this way, it indicates that the
closest polynomial approximation justified by the data has been
achieved. The degree which first gives the approximately constant
value of s_i (degree 5 in the example) is the appropriate degree
to select. (Users who are prepared to accept a fit higher than
sixth degree, should simply find a high enough value of k to
enable the type of behaviour indicated by the example to be
detected: thus they should seek values of k for which at least 4
or 5 consecutive values of s_i are approximately the same.) If the
degree were allowed to go high enough, s_i would, in most cases,
eventually start to decrease again, indicating that the data

points are being fitted too closely and that undesirable fluctuations are developing between the points. In some cases, particularly with a small number of data points, this final decrease is not distinguishable from the initial decrease in s_i. In such cases, users may seek an acceptable fit by examining the graphs of several of the polynomials obtained. Failing this, they may (a) seek a transformation of variables which improves the behaviour, (b) try fitting a spline, or (c) provide more data points. If data can be provided simply by drawing an approximating curve by hand and reading points from it, use the points discussed in Section 3.2.2.

3.2.2. Least-squares polynomials: selected data points

When users are at liberty to choose the x-values of data points, such as when the points are taken from a graph, it is most advantageous when fitting with polynomials to use the values $x_r = \cos((\pi)r/n)$, for r=0,1,...,n for some value of n, a suitable value for which is discussed at the end of this section. Note that these x_r relate to the variable x after it has been normalised so that its range of interest is -1 to +1. E02ADF may then be used as in Section 3.2.1 to seek a satisfactory fit.

3.3. Cubic Spline Curves

3.3.1. Least-squares cubic splines

E02BAF fits to arbitrary data points, with arbitrary weights, a cubic spline with interior knots specified by the user. The choice of these knots so as to give an acceptable fit must largely be a matter of trial and error, though with a little experience a satisfactory choice can often be made after one or two trials. It is usually best to start with a small number of knots (too many will result in unwanted fluctuations in the fit, or even in there being no unique solution) and, examining the fit graphically at each stage, to add a few knots at a time at places where the fit is particularly poor. Moving the existing knots towards these places will also often improve the fit. In regions where the behaviour of the curve underlying the data is changing rapidly, closer knots will be needed than elsewhere. Otherwise, positioning is not usually very critical and equally-spaced knots are often satisfactory. See also the next section, however.

A useful feature of the routine is that it can be used in applications which require the continuity to be less than the normal continuity of the cubic spline. For example, the fit may be required to have a discontinuous slope at some point in the

range. This can be achieved by placing three coincident knots at the given point. Similarly a discontinuity in the second derivative at a point can be achieved by placing two knots there. Analogy with these discontinuous cases can provide guidance in more usual cases: for example, just as three coincident knots can produce a discontinuity in slope, so three close knots can produce a rapid change in slope. The closer the knots are, the more rapid can the change be.

Figure 1
Please see figure in printed Reference Manual

An example set of data is given in Figure 1. It is a rather tricky set, because of the scarcity of data on the right, but it will serve to illustrate some of the above points and to show some of the dangers to be avoided. Three interior knots (indicated by the vertical lines at the top of the diagram) are chosen as a start. We see that the resulting curve is not steep enough in the middle and fluctuates at both ends, severely on the right. The spline is unable to cope with the shape and more knots are needed.

In Figure 2, three knots have been added in the centre, where the data shows a rapid change in behaviour, and one further out at each end, where the fit is poor. The fit is still poor, so a further knot is added in this region and, in Figure 3, disaster ensues in rather spectacular fashion.

Figure 2
Please see figure in printed Reference Manual

Figure 3
Please see figure in printed Reference Manual

The reason is that, at the right-hand end, the fits in Figure 1 and Figure 2 have been interpreted as poor simply because of the fluctuations about the curve underlying the data (or what it is naturally assumed to be). But the fitting process knows only about the data and nothing else about the underlying curve, so it is important to consider only closeness to the data when deciding goodness of fit.

Thus, in Figure 1, the curve fits the last two data points quite well compared with the fit elsewhere, so no knot should have been added in this region. In Figure 2, the curve goes exactly through the last two points, so a further knot is certainly not needed here.

Figure 4

Please see figure in printed Reference Manual

Figure 4 shows what can be achieved without the extra knot on each of the flat regions. Remembering that within each knot interval the spline is a cubic polynomial, there is really no need to have more than one knot interval covering each flat region.

What we have, in fact, in Figure 2 and Figure 3 is a case of too many knots (so too many coefficients in the spline equation) for the number of data points. The warning in the second paragraph of Section 2.1 was that the fit will then be too close to the data, tending to have unwanted fluctuations between the data points. The warning applies locally for splines, in the sense that, in localities where there are plenty of data points, there can be a lot of knots, as long as there are few knots where there are few points, especially near the ends of the interval. In the present example, with so few data points on the right, just the one extra knot in Figure 2 is too many! The signs are clearly present, with the last two points fitted exactly (at least to the graphical accuracy and actually much closer than that) and fluctuations within the last two knot-intervals (cf. Figure 1, where only the final point is fitted exactly and one of the wobbles spans several data points).

The situation in Figure 3 is different. The fit, if computed exactly, would still pass through the last two data points, with even more violent fluctuations. However, the problem has become so ill-conditioned that all accuracy has been lost. Indeed, if the last interior knot were moved a tiny amount to the right, there would be no unique solution and an error message would have been caused. Near-singularity is, sadly, not picked up by the routine, but can be spotted readily in a graph, as Figure 3. B-spline coefficients becoming large, with alternating signs, is another indication. However, it is better to avoid such situations, firstly by providing, whenever possible, data adequately covering the range of interest, and secondly by placing knots only where there is a reasonable amount of data.

The example here could, in fact, have utilised from the start the observation made in the second paragraph of this section, that three close knots can produce a rapid change in slope. The example has two such rapid changes and so requires two sets of three close knots (in fact, the two sets can be so close that one knot can serve in both sets, so only five knots prove sufficient in Figure 4). It should be noted, however, that the rapid turn occurs within the range spanned by the three knots. This is the reason that the six knots in Figure 2 are not satisfactory as they do not quite span the two turns.

Some more examples to illustrate the choice of knots are given in Cox and Hayes [1].

3.3.2. Automatic fitting with cubic splines

E02BEF also fits cubic splines to arbitrary data points with arbitrary weights but itself chooses the number and positions of the knots. The user has to supply only a threshold for the sum of squares of residuals. The routine first builds up a knot set by a series of trial fits in the l_2 norm. Then, with the knot set decided, the final spline is computed to minimize a certain smoothing measure subject to satisfaction of the chosen threshold. Thus it is easier to use than E02BAF (see previous section), requiring only some experimentation with this threshold. It should therefore be first choice unless the user has a preference for the ordinary least-squares fit or, for example, wishes to experiment with knot positions, trying to keep their number down (E02BEF aims only to be reasonably frugal with knots).

3.4. Spline Surfaces

3.4.1. Least-squares bicubic splines

E02DAF fits to arbitrary data points, with arbitrary weights, a bicubic spline with its two sets of interior knots specified by the user. For choosing these knots, the advice given for cubic splines, in Section 3.3.1 above, applies here too. (See also the next section, however.) If changes in the behaviour of the surface underlying the data are more marked in the direction of one variable than of the other, more knots will be needed for the former variable than the latter. Note also that, in the surface case, the reduction in continuity caused by coincident knots will extend across the whole spline surface: for example, if three knots associated with the variable x are chosen to coincide at a value L, the spline surface will have a discontinuous slope across the whole extent of the line x=L.

With some sets of data and some choices of knots, the least-squares bicubic spline will not be unique. This will not occur, with a reasonable choice of knots, if the rectangle R is well covered with data points: here R is defined as the smallest rectangle in the (x,y) plane, with sides parallel to the axes, which contains all the data points. Where the least-squares solution is not unique, the minimal least-squares solution is computed, namely that least-squares solution which has the smallest value of the sum of squares of the B-spline coefficients c_{ij} (see the end of Section 2.3.2 above). This choice of least-

squares solution tends to minimize the risk of unwanted fluctuations in the fit. The fit will not be reliable, however, in regions where there are few or no data points.

3.4.2. Automatic fitting with bicubic splines

E02DDF also fits bicubic splines to arbitrary data points with arbitrary weights but chooses the knot sets itself. The user has to supply only a threshold for the sum of squares of residuals. Just like the automatic curve E02BEF (Section 3.3.2), E02DDF then builds up the knot sets and finally fits a spline minimizing a smoothing measure subject to satisfaction of the threshold. Again, this easier to use routine is normally to be preferred, at least in the first instance.

E02DCF is a very similar routine to E02DDF but deals with data points of equal weight which lie on a rectangular mesh in the (x,y) plane. This kind of data allows a very much faster computation and so is to be preferred when applicable. Substantial departures from equal weighting can be ignored if the user is not concerned with statistical questions, though the quality of the fit will suffer if this is taken too far. In such cases, the user should revert to E02DDF.

3.5. General Linear and Nonlinear Fitting Functions

3.5.1. General linear functions

For the general linear function (15), routines are available for fitting in the l_1 and l_2 norms. The least-squares routines (which are to be preferred unless there is good reason to use another norm -- see Section 2.1.1) are in Chapter F04. The l_1 routine is E02GAF.

All the above routines are essentially linear algebra routines, and in considering their use we need to view the fitting process in a slightly different way from hitherto. Taking y to be the dependent variable and x the vector of independent variables, we have, as for equation (1) but with each x_r now a vector,

$$\epsilon_r = y_r - f(x_r) \quad r=1,2,\ldots,m.$$

Substituting for f(x) the general linear form (15), we can write this as

$$c_1\phi_1(x_r)+c_2\phi_2(x_r)+\ldots+c_n\phi_n(x_r)=y_r-\epsilon_r ,$$

```
 1     1  r   2     2  r       p     p  r   r          r
 r=1,2,...,m                                                  (19)
```

Thus we have a system of linear equations in the coefficients c_j.
Usually, in writing these equations, the (epsilon) ϵ_r are omitted
and simply taken as implied. The system of equations is then
described as an overdetermined system (since we must have m>=p if
there is to be the possibility of a unique solution to our
fitting problem), and the fitting process of computing the c_j to
minimize one or other of the norms (2), (3) and (4) can be
described, in relation to the system of equations, as solving the
overdetermined system in that particular norm. In matrix
notation, the system can be written as

```
                          (Phi)c=y,                           (20)
```

where (Phi) is the m by p matrix whose element in row r and
column j is $(\phi)_j(x_r)$, for r=1,2,...,m; j=1,2,...,p. The vectors
c and y respectively contain the coefficients c_j and the data
values y_r.

The routines, however, use the standard notation of linear
algebra, the overdetermined system of equations being denoted by

```
                             Ax=b                             (21)
```

The correspondence between this notation and that which we have
used for the data-fitting problem (equation (20)) is therefore
given by

```
                  A==(Phi),   x==c   b==y                     (22)
```

Note that the norms used by these routines are the unweighted
norms (2) and (3). If the user wishes to apply weights to the
data points, that is to use the norms (5) or (6), the
equivalences (22) should be replaced by

```
                    A==D(Phi),   x==c   b==Dy
```

where D is a diagonal matrix with w_r as the rth diagonal element.
Here w_r, for r=1,2,...,m, is the weight of the rth data point as

defined in Section 2.1.2.

3.5.2. Nonlinear functions

Routines for fitting with a nonlinear function in the l_2 norm are provided in Chapter E04, and that chapter's Introduction should be consulted for the appropriate choice of routine. Again, however, the notation adopted is different from that we have used for data fitting. In the latter, we denote the fitting function by f(x;c), where x is the vector of independent variables and c is the vector of coefficients, whose values are to be determined. The squared l_2 norm, to be minimized with respect to the elements of c, is then

$$\sum_{r=1}^{m} w_r [y_r - f(x_r;c)]^2 \qquad (23)$$

where y_r is the rth data value of the dependent variable, x_r is the vector containing the rth values of the independent variables, and w_r is the corresponding weight as defined in Section 2.1.2.

On the other hand, in the nonlinear least-squares routines of Chapter E04, the function to be minimized is denoted by

$$\sum_{i=1}^{m} f_i^2(x), \qquad (24)$$

the minimization being carried out with respect to the elements of the vector x. The correspondence between the two notations is given by

x==c and

$f_i(x) == w_r [y_r - f(x_r;c)]$, i=r=1,2,...,m.

Note especially that the vector x of variables of the nonlinear least-squares routines is the vector c of coefficients of the data-fitting problem, and in particular that, if the selected

routine requires derivatives of the f (x) to be provided, these
i
are derivatives of w [y -f(x ;c)] with respect to the
r r r
coefficients of the data-fitting problem.

3.6. Constraints

At present, there are only a limited number of routines which fit subject to constraints. Chapter E04 contains a routine, E04UCF, which can be used for fitting with a nonlinear function in the l
2
norm subject to equality or inequality constraints. This routine, unlike those in that chapter suited to the unconstrained case, is not designed specifically for minimizing functions which are sums of squares, and so the function (23) has to be treated as a general nonlinear function. The E04 Chapter Introduction should be consulted.

The remaining constraint routine relates to fitting with polynomials in the l norm. E02AGF deals with polynomial curves
2
and allows precise values of the fitting function and (if required) all its derivatives up to a given order to be prescribed at one or more values of the independent variable.

3.7. Evaluation, Differentiation and Integration

Routines are available to evaluate, differentiate and integrate polynomials in Chebyshev-series form and cubic or bicubic splines in B-spline form. These polynomials and splines may have been produced by the various fitting routines or, in the case of polynomials, from prior calls of the differentiation and integration routines themselves.

E02AEF and E02AKF evaluate polynomial curves: the latter has a longer parameter list but does not require the user to normalise the values of the independent variable and can accept coefficients which are not stored in contiguous locations. E02BBF evaluates cubic spline curves, and E02DEF and E02DFF bicubic spline surfaces.

Differentiation and integration of polynomial curves are carried out by E02AHF and E02AJF respectively. The results are provided in Chebyshev-series form and so repeated differentiation and integration are catered for. Values of the derivative or integral can then be computed using the appropriate evaluation routine.

For splines the differentiation and integration routines provided are of a different nature from those for polynomials. E02BCF

provides values of a cubic spline curve and its first three derivatives (the rest, of course, are zero) at a given value of x spline over its whole range. These routines can also be applied to surfaces of the form (14). For example, if, for each value of j in turn, the coefficients c_{ij}, for i=1,2,...,p are supplied to E02BCF with $x=x_0$ and on each occasion we select from the output the value of the second derivative, d_j say, and if the whole set of d_j are then supplied to the same routine with $x=y_0$, the output will contain all the values at (x_0,y_0) of

$$\frac{\partial^2 f}{\partial x^2} \quad \text{and} \quad \frac{\partial^{r+2} f}{\partial x^2 \partial y^r}, \quad r=1,2,3.$$

Equally, if after each of the first p calls of E02BCF we had selected the function value (E02BBF would also provide this) instead of the second derivative and we had supplied these values to E02BDF, the result obtained would have been the value of

$$\int_A^B f(x_0,y)\,dy,$$

where A and B are the end-points of the y interval over which the spline was defined.

3.8. Index

Automatic fitting,
- with bicubic splines — E02DCF, E02DDF
- with cubic splines — E02BEF

Data on rectangular mesh — E02DCF

Differentiation,
- of cubic splines — E02BCF
- of polynomials — E02AHF

Evaluation,
- of bicubic splines — E02DEF, E02DFF
- of cubic splines — E02BBF
- of cubic splines and derivatives — E02BCF

```
     of definite integral of cubic splines                E02BDF
     of polynomials                                       E02AEF
                                                          E02AKF
Integration,
     of cubic splines (definite integral)                 E02BDF
     of polynomials                                       E02AJF
Least-squares curve fit,
     with cubic splines                                   E02BAF
     with polynomials,
          arbitrary data points                           E02ADF
          with constraints                                E02AGF
Least-squares surface fit with bicubic splines            E02DAF
l  fit with general linear function,                      E02GAF
 1
Sorting,
     2-D data into panels                                 E02ZAF


E02 -- Curve and Surface Fitting                Contents -- E02
Chapter E02

Curve and Surface Fitting

E02ADF  Least-squares curve fit, by polynomials, arbitrary data
        points

E02AEF  Evaluation of fitted polynomial in one variable from
        Chebyshev series form (simplified parameter list)

E02AGF  Least-squares polynomial fit, values and derivatives may
        be constrained, arbitrary data points,

E02AHF  Derivative of fitted polynomial in Chebyshev series form

E02AJF  Integral of fitted polynomial in Chebyshev series form

E02AKF  Evaluation of fitted polynomial in one variable, from
        Chebyshev series form

E02BAF  Least-squares curve cubic spline fit (including
        interpolation)

E02BBF  Evaluation of fitted cubic spline, function only

E02BCF  Evaluation of fitted cubic spline, function and
        derivatives

E02BDF  Evaluation of fitted cubic spline, definite integral

E02BEF  Least-squares cubic spline curve fit, automatic knot
```

```
        placement

E02DAF  Least-squares surface fit, bicubic splines

E02DCF  Least-squares surface fit by bicubic splines with
        automatic knot placement, data on rectangular grid

E02DDF  Least-squares surface fit by bicubic splines with
        automatic knot placement, scattered data

E02DEF  Evaluation of a fitted bicubic spline at a vector of
        points

E02DFF  Evaluation of a fitted bicubic spline at a mesh of points

E02GAF  L -approximation by general linear function
         1

E02ZAF  Sort 2-D data into panels for fitting bicubic splines
```

%%

```
E02 -- Curve and Surface Fitting                          E02ADF
        E02ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02ADF computes weighted least-squares polynomial approximations
to an arbitrary set of data points.

2. Specification

       SUBROUTINE E02ADF (M, KPLUS1, NROWS, X, Y, W, WORK1,
      1                   WORK2, A, S, IFAIL)
       INTEGER          M, KPLUS1, NROWS, IFAIL
       DOUBLE PRECISION X(M), Y(M), W(M), WORK1(3*M), WORK2
      1                 (2*KPLUS1), A(NROWS,KPLUS1), S(KPLUS1)

3. Description

This routine determines least-squares polynomial approximations
of degrees 0,1,...,k to the set of data points (x ,y ) with
                                                r  r
weights w , for r=1,2,...,m.
         r
```

The approximation of degree i has the property that it minimizes
(sigma) the sum of squares of the weighted residuals (epsilon) ,
i r
where

(epsilon) =w (y -f)
r r r r r

and f is the value of the polynomial of degree i at the rth data
r
point.

Each polynomial is represented in Chebyshev-series form with

normalised argument x. This argument lies in the range -1 to +1
and is related to the original variable x by the linear
transformation

(2x-x -x)
max min
x= --------------.
(x -x)
max min

Here x and x are respectively the largest and smallest
max min
values of x . The polynomial approximation of degree i is
r
represented as

1
-a T (x)+a T (x)+a T (x)+...+a T (x),
2 i+1,1 0 i+1,2 1 i+1,3 2 i+1,i+1 i

where T (x) is the Chebyshev polynomial of the first kind of
j

degree j with argument (x).

For i=0,1,...,k, the routine produces the values of a , for
i+1,j+1
j=0,1,...,i, together with the value of the root mean square

/ (sigma)

```
                /         i
residual s =   /   --------. In the case m=i+1 the routine sets
          i  \/     m-i-1
the value of s  to zero.
             i
```

The method employed is due to Forsythe [4] and is based upon the generation of a set of polynomials orthogonal with respect to summation over the normalised data set. The extensions due to Clenshaw [1] to represent these polynomials as well as the approximating polynomials in their Chebyshev-series forms are incorporated. The modifications suggested by Reinsch and Gentleman (see [5]) to the method originally employed by Clenshaw for evaluating the orthogonal polynomials from their Chebyshev-series representations are used to give greater numerical stability.

For further details of the algorithm and its use see Cox [2] and [3].

Subsequent evaluation of the Chebyshev-series representations of the polynomial approximations should be carried out using E02AEF.

4. References

[1] Clenshaw C W (1960) Curve Fitting with a Digital Computer. Comput. J. 2 170--173.

[2] Cox M G (1974) A Data-fitting Package for the Non-specialist User. Software for Numerical Mathematics. (ed D J Evans) Academic Press.

[3] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[4] Forsythe G E (1957) Generation and use of orthogonal polynomials for data fitting with a digital computer. J. Soc. Indust. Appl. Math. 5 74--88.

[5] Gentlemen W M (1969) An Error Analysis of Goertzel's (Watt's) Method for Computing Fourier Coefficients. Comput. J. 12 160--165.

[6] Hayes J G (1970) Curve Fitting by Polynomials in One Variable. Numerical Approximation to Functions and Data. (ed J G Hayes) Athlone Press, London.

5. Parameters

1: M -- INTEGER Input
On entry: the number m of data points. Constraint: M >= MDIST >= 2, where MDIST is the number of distinct x values in the data.

2: KPLUS1 -- INTEGER Input
On entry: k+1, where k is the maximum degree required. Constraint: 0 < KPLUS1 <= MDIST, where MDIST is the number of distinct x values in the data.

3: NROWS -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which E02ADF is called.
Constraint: NROWS >= KPLUS1.

4: X(M) -- DOUBLE PRECISION array Input
On entry: the values x_r of the independent variable, for r=1,2,...,m. Constraint: the values must be supplied in non-decreasing order with X(M) > X(1).

5: Y(M) -- DOUBLE PRECISION array Input
On entry: the values y_r of the dependent variable, for r=1,2,...,m.

6: W(M) -- DOUBLE PRECISION array Input
On entry: the set of weights, w_r, for r=1,2,...,m. For advice on the choice of weights, see Section 2.1.2 of the Chapter Introduction. Constraint: W(r) > 0.0, for r=1,2,...m.

7: WORK1(3*M) -- DOUBLE PRECISION array Workspace

8: WORK2(2*KPLUS1) -- DOUBLE PRECISION array Workspace

9: A(NROWS,KPLUS1) -- DOUBLE PRECISION array Output

On exit: the coefficients of $T_j(x)$ in the approximating polynomial of degree i. A(i+1,j+1) contains the coefficient $a_{i+1,j+1}$, for i=0,1,...,k; j=0,1,...,i.

10: S(KPLUS1) -- DOUBLE PRECISION array Output
On exit: S(i+1) contains the root mean square residual s_i, for i=0,1,...,k, as described in Section 3. For the

```
     interpretation of the values of the s  and their use in
                                         i
     selecting an appropriate degree, see Section 3.1 of the
     Chapter Introduction.

11:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     The weights are not all strictly positive.

IFAIL= 2
     The values of X(r), for r=1,2,...,M are not in non-
     decreasing order.

IFAIL= 3
     All X(r) have the same value: thus the normalisation of X is
     not possible.

IFAIL= 4
     On entry KPLUS1 < 1 (so the maximum degree required is
               negative)

     or        KPLUS1 > MDIST, where MDIST is the number of
               distinct x values in the data (so there cannot be a
               unique solution for degree k=KPLUS1-1).

IFAIL= 5
     NROWS < KPLUS1.

7. Accuracy

No error analysis for the method has been published. Practical
experience with the method, however, is generally extremely
satisfactory.

8. Further Comments

The time taken by the routine is approximately proportional to
m(k+1)(k+11).
```

```
The approximating polynomials may exhibit undesirable
oscillations (particularly near the ends of the range) if the
maximum degree k exceeds a critical value which depends on the
number of data points m and their relative positions. As a rough
guide, for equally-spaced data, this critical value is about


2*\/m. For further details see Hayes [6] page 60.

9. Example

Determine weighted least-squares polynomial approximations of
degrees 0, 1, 2 and 3 to a set of 11 prescribed data points. For
the approximation of degree 3, tabulate the data and the
corresponding values of the approximating polynomial, together
with the residual errors, and also the values of the
approximating polynomial at points half-way between each pair of
adjacent data points.

The example program supplied is written in a general form that
will enable polynomial approximations of degrees 0,1,...,k to be
obtained to m data points, with arbitrary positive weights, and
the approximation of degree k to be tabulated. E02AEF is used to
evaluate the approximating polynomial. The program is self-
starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting                          E02AEF
        E02AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02AEF evaluates a polynomial from its Chebyshev-series
representation.

2. Specification

       SUBROUTINE E02AEF (NPLUS1, A, XCAP, P, IFAIL)
       INTEGER          NPLUS1, IFAIL
       DOUBLE PRECISION A(NPLUS1), XCAP, P
```

3. Description

This routine evaluates the polynomial

$$-a_2 T_1(x) + a_1 T_0(x) + a_2 T_1(x) + a_3 T_2(x) + \ldots + a_{n+1} T_n(x)$$

for any value of x satisfying -1<=x<=1. Here $T_j(x)$ denotes the Chebyshev polynomial of the first kind of degree j with argument x. The value of n is prescribed by the user.

In practice, the variable x will usually have been obtained from an original variable x, where $x_{min} <= x <= x_{max}$ and

$$x = \frac{(x - x_{min}) - (x_{max} - x)}{(x_{max} - x_{min})}$$

Note that this form of the transformation should be used computationally rather than the mathematical equivalent

$$x = \frac{(2x - x_{min} - x_{max})}{(x_{max} - x_{min})}$$

since the former guarantees that the computed value of x differs from its true value by at most 4(epsilon), where (epsilon) is the machine precision, whereas the latter has no such guarantee.

The method employed is based upon the three-term recurrence relation due to Clenshaw [1], with modifications to give greater numerical stability due to Reinsch and Gentleman (see [4]).

For further details of the algorithm and its use see Cox [2] and

[3].

4. References

[1] Clenshaw C W (1955) A Note on the Summation of Chebyshev Series. Math. Tables Aids Comput. 9 118--120.

[2] Cox M G (1974) A Data-fitting Package for the Non-specialist User. Software for Numerical Mathematics. (ed D J Evans) Academic Press.

[3] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[4] Gentlemen W M (1969) An Error Analysis of Goertzel's (Watt's) Method for Computing Fourier Coefficients. Comput. J. 12 160--165.

5. Parameters

1: NPLUS1 -- INTEGER Input
On entry: the number n+1 of terms in the series (i.e., one greater than the degree of the polynomial). Constraint: NPLUS1 >= 1.

2: A(NPLUS1) -- DOUBLE PRECISION array Input
On entry: A(i) must be set to the value of the ith coefficient in the series, for i=1,2,...,n+1.

3: XCAP -- DOUBLE PRECISION Input

On entry: x, the argument at which the polynomial is to be evaluated. It should lie in the range -1 to +1, but a value just outside this range is permitted (see Section 6) to allow for possible rounding errors committed in the

transformation from x to x discussed in Section 3. Provided the recommended form of the transformation is used, a successful exit is thus assured whenever the value of x lies in the range x to x .
min max

4: P -- DOUBLE PRECISION Output
On exit: the value of the polynomial.

5: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not

```
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     ABS(XCAP) > 1.0 + 4(epsilon), where (epsilon) is the
     machine precision. In this case the value of P is set
     arbitrarily to zero.

IFAIL= 2
     On entry NPLUS1 < 1.

7. Accuracy

The rounding errors committed are such that the computed value of
the polynomial is exact for a slightly perturbed set of
coefficients a +(delta)a . The ratio of the sum of the absolute
               i         i
values of the (delta)a  to the sum of the absolute values of the
                     i
a  is less than a small multiple of (n+1) times machine
 i
precision.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

It is expected that a common use of E02AEF will be the evaluation
of the polynomial approximations produced by E02ADF and E02AFF(*)

9. Example



Evaluate at 11 equally-spaced points in the interval -1<=x<=1 the
polynomial of degree 4 with Chebyshev coefficients, 2.0, 0.5, 0.
25, 0.125, 0.0625.

The example program is written in a general form that will enable
a polynomial of degree n in its Chebyshev-series form to be
```

```
evaluated at m equally-spaced points in the interval -1<=x<=1.
The program is self-starting in that any number of data sets can
be supplied.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                              E02AGF
        E02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02AGF computes constrained weighted least-squares polynomial
approximations in Chebyshev-series form to an arbitrary set of
data points. The values of the approximations and any number of
their derivatives can be specified at selected points.

2. Specification

       SUBROUTINE E02AGF (M, KPLUS1, NROWS, XMIN, XMAX, X, Y, W,
      1                   MF, XF, YF, LYF, IP, A, S, NP1, WRK,
      2                   LWRK, IWRK, LIWRK, IFAIL)
       INTEGER          M, KPLUS1, NROWS, MF, LYF, IP(MF), NP1,
      1                 LWRK, IWRK(LIWRK), LIWRK, IFAIL
       DOUBLE PRECISION XMIN, XMAX, X(M), Y(M), W(M), XF(MF), YF
      1                 (LYF), A(NROWS,KPLUS1), S(KPLUS1), WRK
      2                 (LWRK)

3. Description

This routine determines least-squares polynomial approximations
of degrees up to k to the set of data points (x ,y ) with weights
                                               r  r
w , for r=1,2,...,m. The value of k, the maximum degree required,
 r
is prescribed by the user. At each of the values XF , for r =
                                                    r
1,2,...,MF, of the independent variable x, the approximations and
their derivatives up to order p  are constrained to have one of
                               r
                                                          MF
                                                          --
```

the user-specified values YF , for s=1,2,...,n, where n=MF+ > p
                           s   -- r
                                                            r=1

The approximation of degree i has the property that, subject to
the imposed contraints, it minimizes (Sigma) , the sum of the
                                           i
squares of the weighted residuals (epsilon) for r=1,2,...,m
                                          r
where

                  (epsilon) =w (y -f (x))
                          r r r i r

and f (x) is the value of the polynomial approximation of degree
     i r
i at the rth data point.

Each polynomial is represented in Chebyshev-series form with

normalised argument x. This argument lies in the range -1 to +1
and is related to the original variable x by the linear
transformation

                        2x-(x   +x   )
                             max min
                    x= --------------
                        (x   -x   )
                          max min

where x   and x   , specified by the user, are respectively the
       min   max
lower and upper end-points of the interval of x over which the
polynomials are to be defined.

The polynomial approximation of degree i can be written as

          1
          -a   +a   T (x)+...+a T (x)+...+a T (x)
          2 i,0 i,1 1   ij j   ii i

where T (x) is the Chebyshev polynomial of the first kind of
       j

degree j with argument x. For i=n,n+1,...,k, the routine produces
the values of the coefficients a , for j=0,1,...,i, together

ij
with the value of the root mean square residual, S , defined as
i

```
      /     --
     /      >
    /       --
   /        i
  /     ----------, where m' is the number of data points with
\/      (m'+n-i-1)
```

non-zero weight.

Values of the approximations may subsequently be computed using E02AEF or E02AKF.

First E02AGF determines a polynomial (mu)(x), of degree n-1, which satisfies the given constraints, and a polynomial (nu)(x), of degree n, which has value (or derivative) zero wherever a constrained value (or derivative) is specified. It then fits y -(mu)(x), for r=1,2,...,m with polynomials of the required
 r   r

degree in x each with factor (nu)(x). Finally the coefficients of

(mu)(x) are added to the coefficients of these fits to give the coefficients of the constrained polynomial approximations to the data points (x ,y), for r=1,2,...,m. The method employed is
r r
given in Hayes [3]: it is an extension of Forsythe's orthogonal polynomials method [2] as modified by Clenshaw [1].

4. References

[1] Clenshaw C W (1960) Curve Fitting with a Digital Computer. Comput. J. 2 170--173.

[2] Forsythe G E (1957) Generation and use of orthogonal polynomials for data fitting with a digital computer. J. Soc. Indust. Appl. Math. 5 74--88.

[3] Hayes J G (1970) Curve Fitting by Polynomials in One Variable. Numerical Approximation to Functions and Data. (ed J G Hayes) Athlone Press, London.

5. Parameters

1: M -- INTEGER Input
On entry: the number m of data points to be fitted.
Constraint: M >= 1.

2: KPLUS1 -- INTEGER Input
On entry: k+1, where k is the maximum degree required.
Constraint: n+1<=KPLUS1<=m''+n, where n is the total number of constraints and m'' is the number of data points with non-zero weights and distinct abscissae which do not coincide with any of the XF(r).

3: NROWS -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which E02AGF is called.
Constraint: NROWS >= KPLUS1.

4: XMIN -- DOUBLE PRECISION Input

5: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points, respectively, of the interval $[x_{min}, x_{max}]$. Unless there are specific reasons to the contrary, it is recommended that XMIN and XMAX be set respectively to the lowest and highest value among the x_r and XF(r). This avoids the danger of extrapolation provided there is a constraint point or data point with non-zero weight at each end-point. Constraint: XMAX > XMIN.

6: X(M) -- DOUBLE PRECISION array Input
On entry: the value x_r of the independent variable at the rth data point, for r=1,2,...,m. Constraint: the X(r) must be in non-decreasing order and satisfy XMIN <= X(r) <= XMAX.

7: Y(M) -- DOUBLE PRECISION array Input
On entry: Y(r) must contain y_r, the value of the dependent variable at the rth data point, for r=1,2,...,m.

8: W(M) -- DOUBLE PRECISION array Input
On entry: the weights w_r to be applied to the data points x_r, for r=1,2...,m. For advice on the choice of weights see the Chapter Introduction. Negative weights are treated as positive. A zero weight causes the corresponding data point to be ignored. Zero weight should be given to any data point

whose x and y values both coincide with those of a
constraint (otherwise the denominators involved in the root-
mean-square residuals s_i will be slightly in error).

9: MF -- INTEGER Input
On entry: the number of values of the independent variable
at which a constraint is specified. Constraint: MF >= 1.

10: XF(MF) -- DOUBLE PRECISION array Input
On entry: the rth value of the independent variable at
which a constraint is specified, for r = 1,2,...,MF.
Constraint: these values need not be ordered but must be
distinct and satisfy XMIN <= XF(r) <= XMAX.

11: YF(LYF) -- DOUBLE PRECISION array Input
On entry: the values which the approximating polynomials
and their derivatives are required to take at the points
specified in XF. For each value of XF(r), YF contains in
successive elements the required value of the approximation,
its first derivative, second derivative,..., p_r th
derivative, for r = 1,2,...,MF. Thus the value which the kth
derivative of each approximation (k=0 referring to the
approximation itself) is required to take at the point XF(r)
must be contained in YF(s), where
$$s=r+k+p_1+p_2+\ldots+p_{r-1},$$
for k=0,1,...,p_r and r = 1,2,...,MF. The derivatives are
with respect to the user's variable x.

12: LYF -- INTEGER Input
On entry:
the dimension of the array YF as declared in the
(sub)program from which E02AGF is called.
Constraint: LYF>=n, where $n=MF+p_1+p_2+\ldots+p_{MF}$.

13: IP(MF) -- INTEGER array Input
On entry: IP(r) must contain p_r, the order of the highest-
order derivative specified at XF(r), for r = 1,2,...,MF.
p_r =0 implies that the value of the approximation at XF(r) is
specified, but not that of any derivative. Constraint: IP(r)
>= 0, for r=1,2,...,MF.

14: A(NROWS,KPLUS1) -- DOUBLE PRECISION array Output
On exit: A(i+1,j+1) contains the coefficient a_{ij} in the

```
                                                        ij
     approximating polynomial of degree i, for i=n,n+1,...,k;
     j=0,1,...,i.

15:  S(KPLUS1) -- DOUBLE PRECISION array                    Output
     On exit: S(i+1) contains s , for i=n,n+1,...,k, the root-
                               i
     mean-square residual corresponding to the approximating
     polynomial of degree i. In the case where the number of data
     points with non-zero weight is equal to k+1-n, s  is
                                                     i
     indeterminate: the routine sets it to zero. For the
     interpretation of the values of s  and their use in
                                      i
     selecting an appropriate degree, see Section 3.1 of the
     Chapter Introduction.

16:  NP1 -- INTEGER                                        Output
     On exit: n+1, where n is the total number of constraint
     conditions imposed: n=MF+p +p +...+p  .
                             1  2      MF

17:  WRK(LWRK) -- DOUBLE PRECISION array                   Output
     On exit: WRK contains weighted residuals of the highest
     degree of fit determined (k). The residual at x  is in
     element 2(n+1)+3(m+k+1)+r, for r=1,2,...,m. The rest of the
     array is used as workspace.

18:  LWRK -- INTEGER                                        Input
     On entry:
     the dimension of the array WRK as declared in the
     (sub)program from which E02AGF is called.
     Constraint: LWRK>=max(4*M+3*KPLUS1, 8*n+5*IPMAX+MF+10)+2*n+2
     , where IPMAX = max(IP(R)).

19:  IWRK(LIWRK) -- INTEGER array                        Workspace

20:  LIWRK -- INTEGER                                       Input
     On entry:
     the dimension of the array IWRK as declared in the
     (sub)program from which E02AGF is called.
     Constraint: LIWRK>=2*MF+2.

21:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).
```

```
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry M < 1,

     or       KPLUS1 < n + 1,

     or       NROWS < KPLUS1,

     or       MF < 1,

     or       LYF < n,

     or       LWRK is too small (see Section 5),

     or       LIWRK<2*MF+2.
     (Here n is the total number of constraint conditions.)

IFAIL= 2
     IP(r) < 0 for some r = 1,2,...,MF.

IFAIL= 3
     XMIN >= XMAX, or XF(r) is not in the interval XMIN to XMAX
     for some r = 1,2,...,MF, or the XF(r) are not distinct.

IFAIL= 4
     X(r) is not in the interval XMIN to XMAX for some
     r=1,2,...,M.

IFAIL= 5
     X(r) < X(r-1) for some r=2,3,...,M.

IFAIL= 6
     KPLUS1>m''+n, where m'' is the number of data points with
     non-zero weight and distinct abscissae which do not coincide
     with any XF(r). Thus there is no unique solution.

IFAIL= 7
     The polynomials (mu)(x) and/or (nu)(x) cannot be determined.
     The problem supplied is too ill-conditioned. This may occur
     when the constraint points are very close together, or large
     in number, or when an attempt is made to constrain high-
     order derivatives.

7. Accuracy

No complete error analysis exists for either the interpolating
```

```
algorithm or the approximating algorithm. However, considerable
experience with the approximating algorithm shows that it is
generally extremely satisfactory. Also the moderate number of
constraints, of low order, which are typical of data fitting
applications, are unlikely to cause difficulty with the
interpolating routine.

8. Further Comments

The time taken by the routine to form the interpolating
                                                    3
polynomial is approximately proportional to n , and that to form
the approximating polynomials is very approximately proportional
to m(k+1)(k+1-n).

To carry out a least-squares polynomial fit without constraints,
use E02ADF. To carry out polynomial interpolation only, use
E01AEF(*).

9. Example

The example program reads data in the following order, using the
notation of the parameter list above:

     MF

     IP(i), XF(i), Y-value and derivative values (if any) at
     XF(i), for i= 1,2,...,MF

     M

     X(i), Y(i), W(i), for i=1,2,...,M

     k, XMIN, XMAX

The output is:

     the root-mean-square residual for each degree from n to k;

     the Chebyshev coefficients for the fit of degree k;

     the data points, and the fitted values and residuals for
     the fit of degree k.

The program is written in a generalized form which will read any
number of data sets.

The data set supplied specifies 5 data points in the interval [0.
0,4.0] with unit weights, to which are to be fitted polynomials,
p, of degrees up to 4, subject to the 3 constraints:
```

```
     p(0.0)=1.0,  p'(0.0)=-2.0,  p(4.0)=9.0.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting                              E02AHF
        E02AHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02AHF determines the coefficients in the Chebyshev-series
representation of the derivative of a polynomial given in
Chebyshev-series form.

2. Specification

       SUBROUTINE E02AHF (NP1, XMIN, XMAX, A, IA1, LA, PATM1,
      1                   ADIF, IADIF1, LADIF, IFAIL)
       INTEGER          NP1, IA1, LA, IADIF1, LADIF, IFAIL
       DOUBLE PRECISION XMIN, XMAX, A(LA), PATM1, ADIF(LADIF)

3. Description

This routine forms the polynomial which is the derivative of a
given polynomial. Both the original polynomial and its derivative
are represented in Chebyshev-series form. Given the coefficients
a , for i=0,1,...,n, of a polynomial p(x) of degree n, where
 i

                          1
                p(x)= -a +a T (x)+...+a T (x)
                      2 0  1 1         n n



the routine returns the coefficients a , for i=0,1,...,n-1, of
                                      i
the polynomial q(x) of degree n-1, where

                     dp(x)  1
              q(x)= ------= -a +a T (x)+...+a   T   (x).
```

dx 2 0 1 1 n-1 n-1

Here T (x) denotes the Chebyshev polynomial of the first kind of
j

degree j with argument x. It is assumed that the normalised

variable x in the interval [-1,+1] was obtained from the user's original variable x in the interval [x ,x] by the linear
min max
transformation

$$x = \frac{2x-(x_{max}+x_{min})}{x_{max}-x_{min}}$$

and that the user requires the derivative to be with respect to

the variable x. If the derivative with respect to x is required, set x =1 and x =-1.
max min

Values of the derivative can subsequently be computed, from the coefficients obtained, by using E02AKF.

The method employed is that of [1] modified to obtain the

derivative with respect to x. Initially setting a =a =0, the
n+1 n
routine forms successively

$$a_{i-1} = a_{i+1} + \frac{2}{x_{max}-x_{min}}2ia_i, \quad i=n,n-1,\ldots,1.$$

4. References

[1] Unknown (1961) Chebyshev-series. Modern Computing Methods, Chapter 8. NPL Notes on Applied Science (2nd Edition). 16 HMSO.

5. Parameters

1: NP1 -- INTEGER Input
On entry: n+1, where n is the degree of the given polynomial p(x). Thus NP1 is the number of coefficients in this polynomial. Constraint: NP1 >= 1.

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points respectively of the interval [x , x]. The Chebyshev-series
min max

representation is in terms of the normalised variable x, where

$$x = \frac{2x - (x_{max} + x_{min})}{x_{max} - x_{min}}.$$

Constraint: XMAX > XMIN.

4: A(LA) -- DOUBLE PRECISION array Input
On entry: the Chebyshev coefficients of the polynomial p(x). Specifically, element 1 + i*IA1 of A must contain the coefficient a_i, for i=0,1,...,n. Only these n+1 elements will be accessed.

Unchanged on exit, but see ADIF, below.

5: IA1 -- INTEGER Input
On entry: the index increment of A. Most frequently the Chebyshev coefficients are stored in adjacent elements of A, and IA1 must be set to 1. However, if, for example, they are stored in A(1),A(4),A(7),..., then the value of IA1 must be 3. See also Section 8. Constraint: IA1 >= 1.

6: LA -- INTEGER Input
On entry:
the dimension of the array A as declared in the (sub)program from which E02AHF is called.
Constraint: LA>=1+(NP1-1)*IA1.

7: PATM1 -- DOUBLE PRECISION Output
On exit: the value of $p(x_{min})$. If this value is passed to the integration routine E02AJF with the coefficients of q(x)

, then the original polynomial p(x) is recovered, including its constant coefficient.

8: ADIF(LADIF) -- DOUBLE PRECISION array Output
On exit: the Chebyshev coefficients of the derived polynomial q(x). (The differentiation is with respect to the variable x). Specifically, element 1+i*IADIF1 of ADIF

contains the coefficient a_i, i=0,1,...n-1. Additionally element 1+n*IADIF1 is set to zero. A call of the routine may have the array name ADIF the same as A, provided that note is taken of the order in which elements are overwritten, when choosing the starting elements and increments IA1 and IADIF1: i.e., the coefficients $a_0, a_1, \ldots, a_{i-1}$ must be intact after coefficient a_i is stored. In particular, it is possible to overwrite the a_i completely by having IA1 = IADIF1, and the actual arrays for A and ADIF identical.

9: IADIF1 -- INTEGER Input
On entry: the index increment of ADIF. Most frequently the Chebyshev coefficients are required in adjacent elements of ADIF, and IADIF1 must be set to 1. However, if, for example, they are to be stored in ADIF(1),ADIF(4),ADIF(7),..., then the value of IADIF1 must be 3. See Section 8. Constraint: IADIF1 >= 1.

10: LADIF -- INTEGER Input
On entry:
the dimension of the array ADIF as declared in the (sub)program from which E02AHF is called.
Constraint: LADIF>=1+(NP1-1)*IADIF1.

11: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

```
IFAIL= 1
     On entry NP1 < 1,

     or       XMAX <= XMIN,

     or       IA1 < 1,

     or       LA<=(NP1-1)*IA1,

     or       IADIF1 < 1,

     or       LADIF<=(NP1-1)*IADIF1.

7. Accuracy

There is always a loss of precision in numerical differentiation,
in this case associated with the multiplication by 2i in the
formula quoted in Section 3.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

The increments IA1, IADIF1 are included as parameters to give a
degree of flexibility which, for example, allows a polynomial in
two variables to be differentiated with respect to either
variable without rearranging the coefficients.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the 1st and 2nd derivatives of this polynomial at 4
equally spaced points over the interval. (For the purposes of
this example, XMIN, XMAX and the Chebyshev coefficients are
simply supplied in DATA statements. Normally a program would
first read in or generate data and compute the fitted
polynomial.)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                          E02AJF
        E02AJF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AJF determines the coefficients in the Chebyshev-series representation of the indefinite integral of a polynomial given in Chebyshev-series form.

2. Specification

```
      SUBROUTINE E02AJF (NP1, XMIN, XMAX, A, IA1, LA, QATM1,
     1                   AINT, IAINT1, LAINT, IFAIL)
      INTEGER          NP1, IA1, LA, IAINT1, LAINT, IFAIL
      DOUBLE PRECISION XMIN, XMAX, A(LA), QATM1, AINT(LAINT)
```

3. Description

This routine forms the polynomial which is the indefinite integral of a given polynomial. Both the original polynomial and its integral are represented in Chebyshev-series form. If supplied with the coefficients a_i, for $i=0,1,\ldots,n$, of a polynomial $p(x)$ of degree n, where

$$p(x) = \tfrac{1}{2}a_0 T_0(x) + a_1 T_1(x) + \ldots + a_n T_n(x),$$

the routine returns the coefficients a'_i, for $i=0,1,\ldots,n+1$, of the polynomial $q(x)$ of degree $n+1$, where

$$q(x) = \tfrac{1}{2}a'_0 T_0(x) + a'_1 T_1(x) + \ldots + a'_{n+1} T_{n+1}(x),$$

and

$$q(x) = \int p(x)\,dx.$$

Here $T_j(x)$ denotes the Chebyshev polynomial of the first kind of degree j with argument x. It is assumed that the normalised

variable x in the interval [-1,+1] was obtained from the user's original variable x in the interval [x_{min},x_{max}] by the linear transformation

$$x = \frac{2x-(x_{max}+x_{min})}{x_{max}-x_{min}}$$

and that the user requires the integral to be with respect to the variable x. If the integral with respect to x is required, set $x_{max}=1$ and $x_{min}=-1$.

Values of the integral can subsequently be computed, from the coefficients obtained, by using E02AKF.

The method employed is that of Chebyshev-series [1] modified for integrating with respect to x. Initially taking $a_{n+1}=a_{n+2}=0$, the routine forms successively

$$a'_i = \frac{a_{i-1}-a_{i+1}}{2i}*\frac{x_{max}-x_{min}}{2}, \quad i=n+1,n,\ldots,1.$$

The constant coefficient a'_0 is chosen so that q(x) is equal to a specified value, QATM1, at the lower end-point of the interval on which it is defined, i.e., x=-1, which corresponds to $x=x_{min}$.

4. References

[1] Unknown (1961) Chebyshev-series. Modern Computing Methods, Chapter 8. NPL Notes on Applied Science (2nd Edition). 16 HMSO.

5. Parameters

1: NP1 -- INTEGER Input

On entry: n+1, where n is the degree of the given polynomial p(x). Thus NP1 is the number of coefficients in this polynomial. Constraint: NP1 >= 1.

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points respectively of the interval [x_{min} ,x_{max}]. The Chebyshev-series

representation is in terms of the normalised variable x, where

$$x = \frac{2x-(x_{max}+x_{min})}{x_{max}-x_{min}}.$$

Constraint: XMAX > XMIN.

4: A(LA) -- DOUBLE PRECISION array Input
On entry: the Chebyshev coefficients of the polynomial p(x). Specifically, element 1+i*IA1 of A must contain the coefficient a_i, for i=0,1,...,n. Only these n+1 elements will be accessed.

Unchanged on exit, but see AINT, below.

5: IA1 -- INTEGER Input
On entry: the index increment of A. Most frequently the Chebyshev coefficients are stored in adjacent elements of A, and IA1 must be set to 1. However, if for example, they are stored in A(1),A(4),A(7),..., then the value of IA1 must be 3. See also Section 8. Constraint: IA1 >= 1.

6: LA -- INTEGER Input
On entry:
the dimension of the array A as declared in the (sub)program from which E02AJF is called.
Constraint: LA>=1+(NP1-1)*IA1.

7: QATM1 -- DOUBLE PRECISION Input
On entry: the value that the integrated polynomial is required to have at the lower end-point of its interval of

definition, i.e., at x=-1 which corresponds to x=x_{min} . Thus,

```
     QATM1 is a constant of integration and will normally be set
     to zero by the user.

 8:  AINT(LAINT) -- DOUBLE PRECISION array                Output
     On exit: the Chebyshev coefficients of the integral q(x).
     (The integration is with respect to the variable x, and the
     constant coefficient is chosen so that q(x    ) equals QATM1)
                                              min
     Specifically, element 1+i*IAINT1 of AINT contains the
     coefficient a' , for i=0,1,...,n+1. A call of the routine
                  i
     may have the array name AINT the same as A, provided that
     note is taken of the order in which elements are overwritten
     when choosing starting elements and increments IA1 and
     IAINT1: i.e., the coefficients, a ,a ,...,a    must be
                                      0  1      i-2
     intact after coefficient a'  is stored. In particular it is
                               i
     possible to overwrite the a  entirely by having IA1 =
                               i
     IAINT1, and the actual array for A and AINT identical.

 9:  IAINT1 -- INTEGER                                    Input
     On entry: the index increment of AINT. Most frequently the
     Chebyshev coefficients are required in adjacent elements of
     AINT, and IAINT1 must be set to 1. However, if, for example,
     they are to be stored in AINT(1),AINT(4),AINT(7),..., then
     the value of IAINT1 must be 3. See also Section 8.
     Constraint: IAINT1 >= 1.

10:  LAINT -- INTEGER                                    Input
     On entry:
     the dimension of the array AINT as declared in the
     (sub)program from which E02AJF is called.
     Constraint: LAINT>=1+NP1*IAINT1.

11:  IFAIL -- INTEGER                              Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry NP1 < 1,
```

```
     or       XMAX <= XMIN,

     or       IA1 < 1,

     or       LA<=(NP1-1)*IA1,

     or       IAINT1 < 1,

     or       LAINT<=NP1*IAINT1.

7. Accuracy

In general there is a gain in precision in numerical integration,
in this case associated with the division by 2i in the formula
quoted in Section 3.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

The increments IA1, IAINT1 are included as parameters to give a
degree of flexibility which, for example, allows a polynomial in
two variables to be integrated with respect to either variable
without rearranging the coefficients.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the integral of the polynomial from 0.0 to 2.0. (For
the purpose of this example, XMIN, XMAX and the Chebyshev
coefficients are simply supplied in DATA statements. Normally a
program would read in or generate data and compute the fitted
polynomial).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting                          E02AKF
        E02AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.
```

1. Purpose

E02AKF evaluates a polynomial from its Chebyshev-series representation, allowing an arbitrary index increment for accessing the array of coefficients.

2. Specification

```
      SUBROUTINE E02AKF (NP1, XMIN, XMAX, A, IA1, LA, X, RESULT,
     1                   IFAIL)
      INTEGER          NP1, IA1, LA, IFAIL
      DOUBLE PRECISION XMIN, XMAX, A(LA), X, RESULT
```

3. Description

If supplied with the coefficients a_i, for i=0,1,...,n, of a

polynomial p(x) of degree n, where

$$p(x) = \tfrac{1}{2}a_0 T_0(x) + a_1 T_1(x) + \ldots + a_n T_n(x),$$

this routine returns the value of p(x) at a user-specified value

of the variable x. Here $T_j(x)$ denotes the Chebyshev polynomial of

the first kind of degree j with argument x. It is assumed that

the independent variable x in the interval [-1,+1] was obtained from the user's original variable x in the interval $[x_{min}, x_{max}]$ by the linear transformation

$$x = \frac{2x - (x_{max} + x_{min})}{x_{max} - x_{min}}.$$

The coefficients a_i may be supplied in the array A, with any

i
increment between the indices of array elements which contain successive coefficients. This enables the routine to be used in surface fitting and other applications, in which the array might have two or more dimensions.

The method employed is based upon the three-term recurrence relation due to Clenshaw [1], with modifications due to Reinsch and Gentleman (see [4]). For further details of the algorithm and its use see Cox [2] and Cox and Hayes [3].

4. References

[1] Clenshaw C W (1955) A Note on the Summation of Chebyshev-series. Math. Tables Aids Comput. 9 118--120.

[2] Cox M G (1973) A data-fitting package for the non-specialist user. Report NAC40. National Physical Laboratory.

[3] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[4] Gentlemen W M (1969) An Error Analysis of Goertzel's (Watt's) Method for Computing Fourier Coefficients. Comput. J. 12 160--165.

5. Parameters

1: NP1 -- INTEGER Input
On entry: n+1, where n is the degree of the given

polynomial p(x). Constraint: NP1 >= 1.

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points respectively of the interval [x , x]. The Chebyshev-series
min max

representation is in terms of the normalised variable x, where

$$x = \frac{2x - (x_{max} + x_{min})}{x_{max} - x_{min}}.$$

```
     Constraint: XMIN < XMAX.

 4:  A(LA) -- DOUBLE PRECISION array                          Input
     On entry: the Chebyshev coefficients of the polynomial p(x).
     Specifically, element 1+i*IA1 must contain the coefficient
     a , for i=0,1,...,n. Only these n+1 elements will be
      i
     accessed.

 5:  IA1 -- INTEGER                                           Input
     On entry: the index increment of A. Most frequently, the
     Chebyshev coefficients are stored in adjacent elements of A,
     and IA1 must be set to 1. However, if, for example, they are
     stored in A(1),A(4),A(7),..., then the value of IA1 must be
     3. Constraint: IA1 >= 1.

 6:  LA -- INTEGER                                            Input
     On entry:
     the dimension of the array A as declared in the (sub)program
     from which E02AKF is called.
     Constraint: LA>=(NP1-1)*IA1+1.

 7:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x at which the polynomial is to be
     evaluated. Constraint: XMIN <= X <= XMAX.

 8:  RESULT -- DOUBLE PRECISION                              Output


     On exit: the value of the polynomial p(x).

 9:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry NP1 < 1,

     or       IA1 < 1,

     or       LA<=(NP1-1)*IA1,
```

```
     or       XMIN >= XMAX.

IFAIL= 2
     X does not satisfy the restriction XMIN <= X <= XMAX.

7. Accuracy

The rounding errors are such that the computed value of the
polynomial is exact for a slightly perturbed set of coefficients
a +(delta)a . The ratio of the sum of the absolute values of the
 i         i
(delta)a  to the sum of the absolute values of the a  is less
        i                                          i
than a small multiple of (n+1)*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to
n+1.

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the polynomial at 4 equally spaced points over the
interval. (For the purposes of this example, XMIN, XMAX and the
Chebyshev coefficients are supplied in DATA statements. Normally
a program would first read in or generate data and compute the
fitted polynomial.)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                          E02BAF
        E02BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02BAF computes a weighted least-squares approximation to an
arbitrary set of data points by a cubic spline with knots
prescribed by the user. Cubic spline interpolation can also be
carried out.
```

```
2. Specification

       SUBROUTINE E02BAF (M, NCAP7, X, Y, W, LAMDA, WORK1, WORK2,
      1                   C, SS, IFAIL)
       INTEGER          M, NCAP7, IFAIL
       DOUBLE PRECISION X(M), Y(M), W(M), LAMDA(NCAP7), WORK1(M),
      1                 WORK2(4*NCAP7), C(NCAP7), SS

3. Description

This routine determines a least-squares cubic spline
approximation s(x) to the set of data points (x ,y ) with weights
                                               r  r


w , for r=1,2,...,m. The value of NCAP7 = n+7, where n is the
 r
number of intervals of the spline (one greater than the number of
interior knots), and the values of the knots
(lambda) ,(lambda) ,...,(lambda)   , interior to the data
        5         6             n+3
interval, are prescribed by the user.

s(x) has the property that it minimizes (theta), the sum of
squares of the weighted residuals (epsilon) , for r=1,2,...,m,
                                           r
where

                   (epsilon) =w (y -s(x )).
                           r  r  r    r

The routine produces this minimizing value of (theta) and the


coefficients c ,c ,...,c , where q=n+3, in the B-spline
             1  2      q
representation

                              q
                              --
                        s(x)= >  c N (x).
                              --  i i
                              i=1

Here N (x) denotes the normalised B-spline of degree 3 defined
      i
upon the knots (lambda) ,(lambda)   ,...,(lambda)   .
                       i         i+1             i+4
```

In order to define the full set of B-splines required, eight additional knots (lambda) ,(lambda) ,(lambda) ,(lambda) and
                          1   2   3   4
(lambda)   ,(lambda)- ,(lambda)   ,(lambda)   are inserted
       n+4   n+5   n+6   n+7
automatically by the routine. The first four of these are set equal to the smallest x and the last four to the largest x .
                      r   r

The representation of s(x) in terms of B-splines is the most

compact form possible in that only n+3 coefficients, in addition

to the n+7 knots, fully define s(x).

The method employed involves forming and then computing the least-squares solution of a set of m linear equations in the

coefficients c (i=1,2,...,n+3). The equations are formed using a
             i
recurrence relation for B-splines that is unconditionally stable (Cox [1], de Boor [5]), even for multiple (coincident) knots. The least-squares solution is also obtained in a stable manner by using orthogonal transformations, viz. a variant of Givens rotations (Gentleman [6] and [7]). This requires only one equation to be stored at a time. Full advantage is taken of the structure of the equations, there being at most four non-zero values of N (x) for any value of x and hence at most four
          i
coefficients in each equation.

For further details of the algorithm and its use see Cox [2], [3] and [4].

Subsequent evaluation of s(x) from its B-spline representation may be carried out using E02BBF. If derivatives of s(x) are also required, E02BCF may be used. E02BDF can be used to compute the definite integral of s(x).

4. References

[1] Cox M G (1972) The Numerical Evaluation of B-splines. J. Inst. Math. Appl. 10 134--149.

[2] Cox M G (1974) A Data-fitting Package for the Non-specialist User. Software for Numerical Mathematics. (ed D J Evans) Academic Press.

[3] Cox M G (1975) Numerical methods for the interpolation and approximation of data by spline functions. PhD Thesis. City University, London.

[4] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[5] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

[6] Gentleman W M (1974) Algorithm AS 75. Basic Procedures for Large Sparse or Weighted Linear Least-squares Problems. Appl. Statist. 23 448--454.

[7] Gentleman W M (1973) Least-squares Computations by Givens Transformations without Square Roots. J. Inst. Math. Applic. 12 329--336.

[8] Schoenberg I J and Whitney A (1953) On Polya Frequency Functions III. Trans. Amer. Math. Soc. 74 246--259.

5. Parameters

1: M -- INTEGER Input
On entry: the number m of data points. Constraint: M >= MDIST >= 4, where MDIST is the number of distinct x values in the data.

2: NCAP7 -- INTEGER Input

On entry: n+7, where n is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range x_1 to x_m) over which the spline is defined. Constraint: 8 <= NCAP7 <= MDIST + 4, where MDIST is the number of distinct x values in the data.

3: X(M) -- DOUBLE PRECISION array Input
On entry: the values x_r of the independent variable (abscissa), for r=1,2,...,m. Constraint: $x_1 <= x_2 <= ... <= x_m$.

4: Y(M) -- DOUBLE PRECISION array Input
On entry: the values y_r of the dependent variable (ordinate), for r=1,2,...,m.

```
 5:  W(M) -- DOUBLE PRECISION array                          Input
     On entry: the values w  of the weights, for r=1,2,...,m.
                           r
     For advice on the choice of weights, see the Chapter
     Introduction. Constraint: W(r) > 0, for r=1,2,...,m.

 6:  LAMDA(NCAP7) -- DOUBLE PRECISION array          Input/Output
     On entry: LAMDA(i) must be set to the (i-4)th (interior)

     knot, (lambda) , for i=5,6,...,n+3. Constraint: X(1) < LAMDA
                  i
     (5) <= LAMDA(6) <=... <= LAMDA(NCAP7-4) < X(M). On exit: the
     input values are unchanged, and LAMDA(i), for i = 1, 2, 3,
     4, NCAP7-3, NCAP7-2, NCAP7-1, NCAP7 contains the additional
     (exterior) knots introduced by the routine. For advice on
     the choice of knots, see Section 3.3 of the Chapter
     Introduction.

 7:  WORK1(M) -- DOUBLE PRECISION array                 Workspace

 8:  WORK2(4*NCAP7) -- DOUBLE PRECISION array           Workspace

 9:  C(NCAP7) -- DOUBLE PRECISION array                    Output
     On exit: the coefficient c  of the B-spline N (x), for
                               i                  i


     i=1,2,...,n+3. The remaining elements of the array are not
     used.

10:  SS -- DOUBLE PRECISION                               Output
     On exit: the residual sum of squares, (theta).

11:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     The knots fail to satisfy the condition

     X(1) < LAMDA(5) <= LAMDA(6) <=... <= LAMDA(NCAP7-4) < X(M).
```

```
     Thus the knots are not in correct order or are not interior
     to the data interval.

IFAIL= 2
     The weights are not all strictly positive.

IFAIL= 3
     The values of X(r), for r = 1,2,...,M are not in non-
     decreasing order.

IFAIL= 4
     NCAP7 < 8 (so the number of interior knots is negative) or
     NCAP7 > MDIST + 4, where MDIST is the number of distinct x
     values in the data (so there cannot be a unique solution).

IFAIL= 5
     The conditions specified by Schoenberg and Whitney [8] fail
     to hold for at least one subset of the distinct data
     abscissae. That is, there is no subset of NCAP7-4 strictly
     increasing values, X(R(1)),X(R(2)),...,X(R(NCAP7-4)), among
     the abscissae such that
          X(R(1)) < LAMDA(1) < X(R(5)),

          X(R(2)) < LAMDA(2) < X(R(6)),

          ...

          X(R(NCAP7-8)) < LAMDA(NCAP7-8) < X(R(NCAP7-4)).
     This means that there is no unique solution: there are
     regions containing too many knots compared with the number
     of data points.

7. Accuracy

The rounding errors committed are such that the computed
coefficients are exact for a slightly perturbed set of ordinates
y +(delta)y . The ratio of the root-mean-square value for the
 r         r
(delta)y  to the root-mean-square value of the y  can be expected
        r                                      r
to be less than a small multiple of (kappa)*m*machine precision,
where (kappa) is a condition number for the problem. Values of
(kappa) for 20-30 practical data sets all proved to lie between
4.5 and 7.8 (see Cox [3]). (Note that for these data sets,
replacing the coincident end knots at the end-points x  and x
                                                      1      m
used in the routine by various choices of non-coincident exterior
knots gave values of (kappa) between 16 and 180. Again see Cox
[3] for further details.) In general we would not expect (kappa)
to be large unless the choice of knots results in near-violation
```

```
of the Schoenberg-Whitney conditions.

A cubic spline which adequately fits the data and is free from
spurious oscillations is more likely to be obtained if the knots
are chosen to be grouped more closely in regions where the
function (underlying the data) or its derivatives change more
rapidly than elsewhere.

8. Further Comments



The time taken by the routine is approximately C*(2m+n+7)
seconds, where C is a machine-dependent constant.

Multiple knots are permitted as long as their multiplicity does
not exceed 4, i.e., the complete set of knots must satisfy


(lambda) <(lambda)   , for i=1,2,...,n+3, (cf. Section 6). At a
        i         i+4
knot of multiplicity one (the usual case), s(x) and its first two
derivatives are continuous. At a knot of multiplicity two, s(x)
and its first derivative are continuous. At a knot of
multiplicity three, s(x) is continuous, and at a knot of
multiplicity four, s(x) is generally discontinous.

The routine can be used efficiently for cubic spline


interpolation, i.e.,if m=n+3. The abscissae must then of course
satisfy x <x <...<x . Recommended values for the knots in this
         1  2      m


case are (lambda) =x   , for i=5,6,...,n+3.
                  i  i-2

9. Example

Determine a weighted least-squares cubic spline approximation
with five intervals (four interior knots) to a set of 14 given
data points. Tabulate the data and the corresponding values of
the approximating spline, together with the residual errors, and
also the values of the approximating spline at points half-way
between each pair of adjacent data points.

The example program is written in a general form that will enable
```

```
a cubic spline approximation with n intervals (n-1 interior
knots) to be obtained to m data points, with arbitrary positive
weights, and the approximation to be tabulated. Note that E02BBF
is used to evaluate the approximating spline. The program is
self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E02 -- Curve and Surface Fitting                          E02BBF
        E02BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02BBF evaluates a cubic spline from its B-spline representation.

2. Specification

       SUBROUTINE E02BBF (NCAP7, LAMDA, C, X, S, IFAIL)
       INTEGER          NCAP7, IFAIL
       DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S

3. Description

This routine evaluates the cubic spline s(x) at a prescribed
argument x from its augmented knot set (lambda) , for
                                                i


i=1,2,...,n+7, (see E02BAF) and from the coefficients c , for
                                                       i
i=1,2,...,q in its B-spline representation

                              q
                              --
                     s(x)= >  c N (x)
                              --  i i
                              i=1


Here q=n+3, where n is the number of intervals of the spline, and
```

N (x) denotes the normalised B-spline of degree 3 defined upon
 i
the knots (lambda) ,(lambda)   ,...,(lambda)   . The prescribed
                  i   i+1   i+4
argument x must satisfy (lambda) <=x<=(lambda)   .
                                4   n+4

It is assumed that (lambda) >=(lambda)   , for j=2,3,...,n+7, and
                           j   j-1
(lambda)   >(lambda) .
                  4
       n+4

The method employed is that of evaluation by taking convex combinations due to de Boor [4]. For further details of the algorithm and its use see Cox [1] and [3].

It is expected that a common use of E02BBF will be the evaluation of the cubic spline approximations produced by E02BAF. A generalization of E02BBF which also forms the derivative of s(x) is E02BCF. E02BCF takes about 50% longer than E02BBF.

4. References

[1] Cox M G (1972) The Numerical Evaluation of B-splines. J. Inst. Math. Appl. 10 134--149.

[2] Cox M G (1978) The Numerical Evaluation of a Spline from its B-spline Representation. J. Inst. Math. Appl. 21 135--143.

[3] Cox M G and Hayes J G (1973) Curve fitting: a guide and suite of algorithms for the non-specialist user. Report NAC26. National Physical Laboratory.

[4] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

5. Parameters

1: NCAP7 -- INTEGER Input

On entry: n+7, where n is the number of intervals (one greater than the number of interior knots, i.e., the knots strictly within the range (lambda) to (lambda)   ) over
                                    4
                                                  n+4
which the spline is defined. Constraint: NCAP7 >= 8.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array Input
On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, (lambda) for
j

j=1,2,...,n+7. Constraint: the LAMDA(j) must be in non-decreasing order with LAMDA(NCAP7-3) > LAMDA(4).

3: C(NCAP7) -- DOUBLE PRECISION array Input
On entry: the coefficient c of the B-spline N (x), for
i i

i=1,2,...,n+3. The remaining elements of the array are not used.

4: X -- DOUBLE PRECISION Input
On entry: the argument x at which the cubic spline is to be evaluated. Constraint: LAMDA(4) <= X <= LAMDA(NCAP7-3).

5: S -- DOUBLE PRECISION Output
On exit: the value of the spline, s(x).

6: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The argument X does not satisfy LAMDA(4) <= X <= LAMDA(NCAP7-3).

In this case the value of S is set arbitrarily to zero.

IFAIL= 2
NCAP7 < 8, i.e., the number of interior knots is negative.

7. Accuracy

The computed value of s(x) has negligible error in most practical situations. Specifically, this value has an absolute error

```
bounded in modulus by 18*c   * machine precision, where c    is
                           max                              max
the largest in modulus of c ,c   ,c    and c   , and j is an
                            j  j+1  j+2      j+3
integer such that (lambda)   <=x<=(lambda)   . If c ,c   ,c
                           j+3             j+4      j  j+1  j+2
and c    are all of the same sign, then the computed value of
     j+3
s(x) has a relative error not exceeding 20*machine precision in
modulus. For further details see Cox [2].

8. Further Comments



The time taken by the routine is approximately C*(1+0.1*log(n+7))
seconds, where C is a machine-dependent constant.

Note: the routine does not test all the conditions on the knots
given in the description of LAMDA in Section 5, since to do this
would result in a computation time approximately linear in n+7
instead of log(n+7). All the conditions are tested in E02BAF,
however.

9. Example

Evaluate at 9 equally-spaced points in the interval 1.0<=x<=9.0
the cubic spline with (augmented) knots 1.0, 1.0, 1.0, 1.0, 3.0,
6.0, 8.0, 9.0, 9.0, 9.0, 9.0 and normalised cubic B-spline
coefficients 1.0, 2.0, 4.0, 7.0, 6.0, 4.0, 3.0.

The example program is written in a general form that will enable


a cubic spline with n intervals, in its normalised cubic B-spline
form, to be evaluated at m equally-spaced points in the interval


LAMDA(4) <= x <= LAMDA(n+4). The program is self-starting in that
any number of data sets may be supplied.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                          E02BCF
        E02BCF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BCF evaluates a cubic spline and its first three derivatives from its B-spline representation.

2. Specification

```
SUBROUTINE E02BCF (NCAP7, LAMDA, C, X, LEFT, S, IFAIL)
INTEGER          NCAP7, LEFT, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S(4)
```

3. Description

This routine evaluates the cubic spline s(x) and its first three derivatives at a prescribed argument x. It is assumed that s(x) is represented in terms of its B-spline coefficients c_i, for $i=1,2,\ldots,n+3$ and (augmented) ordered knot set $(\lambda)_i$, for $i=1,2,\ldots,n+7$, (see E02BAF), i.e.,

$$s(x)=\sum_{i=1}^{q} c_i N_i(x)$$

Here q=n+3, n is the number of intervals of the spline and $N_i(x)$ denotes the normalised B-spline of degree 3 (order 4) defined upon the knots $(\lambda)_i, (\lambda)_{i+1}, \ldots, (\lambda)_{i+4}$. The prescribed argument x must satisfy

$$(\lambda)_4 \le x \le (\lambda)_{n+4}$$

At a simple knot $(\lambda)_i$ (i.e., one satisfying

(lambda) <(lambda) <(lambda)), the third derivative of the
i-1 i i+1
spline is in general discontinuous. At a multiple knot (i.e., two or more knots with the same value), lower derivatives, and even the spline itself, may be discontinuous. Specifically, at a point x=u where (exactly) r knots coincide (such a point is termed a knot of multiplicity r), the values of the derivatives of order 4-j, for j=1,2,...,r, are in general discontinuous. (Here 1<=r<=4;r>4 is not meaningful.) The user must specify whether the value at such a point is required to be the left- or right-hand derivative.

The method employed is based upon:

(i) carrying out a binary search for the knot interval containing the argument x (see Cox [3]),

(ii) evaluating the non-zero B-splines of orders 1,2,3 and 4 by recurrence (see Cox [2] and [3]),

(iii) computing all derivatives of the B-splines of order 4 by applying a second recurrence to these computed B-spline values (see de Boor [1]),

(iv) multiplying the 4th-order B-spline values and their derivative by the appropriate B-spline coefficients, and summing, to yield the values of s(x) and its derivatives.

E02BCF can be used to compute the values and derivatives of cubic spline fits and interpolants produced by E02BAF.

If only values and not derivatives are required, E02BBF may be used instead of E02BCF, which takes about 50% longer than E02BBF.

4. References

[1] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

[2] Cox M G (1972) The Numerical Evaluation of B-splines. J. Inst. Math. Appl. 10 134--149.

[3] Cox M G (1978) The Numerical Evaluation of a Spline from its B-spline Representation. J. Inst. Math. Appl. 21 135--143.

5. Parameters

1: NCAP7 -- INTEGER Input

On entry: n+7, where n is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range (lambda) to (lambda) over which the spline is defined).
Constraint: NCAP7 >= 8.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array Input
On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, (lambda) , for j

j=1,2,...,n+7. Constraint: the LAMDA(j) must be in non-decreasing order with

LAMDA(NCAP7-3) > LAMDA(4).

3: C(NCAP7) -- DOUBLE PRECISION array Input
On entry: the coefficient c of the B-spline N (x), for i i

i=1,2,...,n+3. The remaining elements of the array are not used.

4: X -- DOUBLE PRECISION Input
On entry: the argument x at which the cubic spline and its derivatives are to be evaluated. Constraint: LAMDA(4) <= X <= LAMDA(NCAP7-3).

5: LEFT -- INTEGER Input
On entry: specifies whether left- or right-hand values of the spline and its derivatives are to be computed (see Section 3). Left- or right-hand values are formed according to whether LEFT is equal or not equal to 1. If x does not coincide with a knot, the value of LEFT is immaterial. If x = LAMDA(4), right-hand values are computed, and if x = LAMDA (NCAP7-3), left-hand values are formed, regardless of the value of LEFT.

6: S(4) -- DOUBLE PRECISION array Output
On exit: S(j) contains the value of the (j-1)th derivative of the spline at the argument x, for j = 1,2,3,4. Note that S(1) contains the value of the spline.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

```
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL= 2
     Either LAMDA(4) >= LAMDA(NCAP7-3), i.e., the range over
     which s(x) is defined is null or negative in length, or X is
     an invalid argument, i.e., X < LAMDA(4) or X >
     LAMDA(NCAP7-3).

7. Accuracy

The computed value of s(x) has negligible error in most practical
situations. Specifically, this value has an absolute error
bounded in modulus by 18*c   * machine precision, where c    is
                          max                             max
the largest in modulus of c ,c   ,c    and c   , and j is an
                           j  j+1  j+2      j+3
integer such that (lambda)   <=x<=(lambda)   . If c ,c   ,c
                          j+3             j+4      j  j+1  j+2
and c    are all of the same sign, then the computed value of
     j+3
s(x) has relative error bounded by 18*machine precision. For full
details see Cox [3].

No complete error analysis is available for the computation of
the derivatives of s(x). However, for most practical purposes the
absolute errors in the computed derivatives should be small.

8. Further Comments

The time taken by this routine is approximately linear in


log(n+7).

Note: the routine does not test all the conditions on the knots
given in the description of LAMDA in Section 5, since to do this


would result in a computation time approximately linear in n+7
```

```
instead of log(n+7). All the conditions are tested in E02BAF,
however.

9. Example

Compute, at the 7 arguments x = 0, 1, 2, 3, 4, 5, 6, the left-
and right-hand values and first 3 derivatives of the cubic spline
defined over the interval 0<=x<=6 having the 6 interior knots x =
1, 3, 3, 3, 4, 4, the 8 additional knots 0, 0, 0, 0, 6, 6, 6, 6,
and the 10 B-spline coefficients 10, 12, 13, 15, 22, 26, 24, 18,
14, 12.

The input data items (using the notation of Section 5) comprise
the following values in the order indicated:



    n           m

    LAMDA(j),   for j= 1,2,...,NCAP7

    C(j),       for j= 1,2,...,NCAP7-4

    x(i),       for i=1,2,...,m

The example program is written in a general form that will enable
the values and derivatives of a cubic spline having an arbitrary
number of knots to be evaluated at a set of arbitrary points. Any
number of data sets may be supplied. The only changes required to
the program relate to the dimensions of the arrays LAMDA and C.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E02 -- Curve and Surface Fitting                          E02BDF
        E02BDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02BDF computes the definite integral of a cubic spline from its
B-spline representation.
```

2. Specification

```
SUBROUTINE E02BDF (NCAP7, LAMDA, C, DEFINT, IFAIL)
INTEGER          NCAP7, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), DEFINT
```

3. Description

This routine computes the definite integral of the cubic spline s(x) between the limits x=a and x=b, where a and b are respectively the lower and upper limits of the range over which s(x) is defined. It is assumed that s(x) is represented in terms

of its B-spline coefficients c_i, for i=1,2,...,n+3 and

(augmented) ordered knot set $(\lambda)_i$, for i=1,2,...,n+7, with $(\lambda)_i = a$, for i = 1,2,3,4 and $(\lambda)_i = b$, for

i=n+4,n+5,n+6,n+7, (see E02BAF), i.e.,

$$s(x) = \sum_{i=1}^{q} c_i N_i(x).$$

Here q=n+3, n is the number of intervals of the spline and $N_i(x)$ denotes the normalised B-spline of degree 3 (order 4) defined upon the knots $(\lambda)_i, (\lambda)_{i+1}, \ldots, (\lambda)_{i+4}$.

The method employed uses the formula given in Section 3 of Cox [1].

E02BDF can be used to determine the definite integrals of cubic spline fits and interpolants produced by E02BAF.

4. References

[1] Cox M G (1975) An Algorithm for Spline Interpolation. J. Inst. Math. Appl. 15 95--108.

5. Parameters

1: NCAP7 -- INTEGER Input

On entry: n+7, where n is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range a to b) over which the spline is defined. Constraint: NCAP7 >= 8.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array Input
On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, $(\text{lambda})_j$ for

j=1,2,...,n+7. Constraint: the LAMDA(j) must be in non-decreasing order with LAMDA(NCAP7-3) > LAMDA(4) and satisfy
LAMDA(1)=LAMDA(2)=LAMDA(3)=LAMDA(4)
and

LAMDA(NCAP7-3)=LAMDA(NCAP7-2)=LAMDA(NCAP7-1)=LAMDA(NCAP7).

3: C(NCAP7) -- DOUBLE PRECISION array Input
On entry: the coefficient c_i of the B-spline $N_i(x)$, for

i=1,2,...,n+3. The remaining elements of the array are not used.

4: DEFINT -- DOUBLE PRECISION Output
On exit: the value of the definite integral of s(x) between the limits x=a and x=b, where a=$(\text{lambda})_4$ and b=$(\text{lambda})_{n+4}$.

5: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are

```
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL= 2
     At least one of the following restrictions on the knots is
     violated:
          LAMDA(NCAP7-3) > LAMDA(4),

          LAMDA(j) >= LAMDA(j-1),
     for j = 2,3,...,NCAP7, with equality in the cases
     j=2,3,4,NCAP7-2,NCAP7-1, and NCAP7.

7. Accuracy

The rounding errors are such that the computed value of the
integral is exact for a slightly perturbed set of B-spline
coefficients c  differing in a relative sense from those supplied
             i
by no more than 2.2*(n+3)*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to


n+7.

9. Example

Determine the definite integral over the interval 0<=x<=6 of a
cubic spline having 6 interior knots at the positions (lambda)=1,
3, 3, 3, 4, 4, the 8 additional knots 0, 0, 0, 0, 6, 6, 6, 6, and
the 10 B-spline coefficients 10, 12, 13, 15, 22, 26, 24, 18, 14,
12.

The input data items (using the notation of Section 5) comprise
the following values in the order indicated:



    n

    LAMDA(j) for j = 1,2,...,NCAP7
    ,

    C(j),    for j = 1,2,...,NCAP7-3

The example program is written in a general form that will enable
```

```
the definite integral of a cubic spline having an arbitrary
number of knots to be computed. Any number of data sets may be
supplied. The only changes required to the program relate to the
dimensions of the arrays LAMDA and C.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                              E02BEF
        E02BEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02BEF computes a cubic spline approximation to an arbitrary set
of data points. The knots of the spline are located
automatically, but a single parameter must be specified to
control the trade-off between closeness of fit and smoothness of
fit.

2. Specification

       SUBROUTINE E02BEF (START, M, X, Y, W, S, NEST, N, LAMDA,
      1                   C, FP, WRK, LWRK, IWRK, IFAIL)
       INTEGER          M, NEST, N, LWRK, IWRK(NEST), IFAIL
       DOUBLE PRECISION X(M), Y(M), W(M), S, LAMDA(NEST), C(NEST),
      1                 FP, WRK(LWRK)
       CHARACTER*1      START

3. Description

This routine determines a smooth cubic spline approximation s(x)
to the set of data points (x ,y ), with weights w , for
                            r  r                r
r=1,2,...,m.

The spline is given in the B-spline representation

                            n-4
                            --
                     s(x)= >   c N (x)                        (1)
                            --  i i
                            i=1
```

```
where N (x) denotes the normalised cubic B-spline defined upon
       i
the knots (lambda) ,(lambda)   ,...,(lambda)   .
                  i         i+1             i+4

The total number n of these knots and their values
(lambda) ,...,(lambda)  are chosen automatically by the routine.
        1             n
The knots (lambda) ,...,(lambda)    are the interior knots; they
                  5             n-4
divide the approximation interval [x ,x ] into n-7 sub-intervals.
                                    1  m
The coefficients c ,c ,...,c    are then determined as the
                  1  2      n-4
solution of the following constrained minimization problem:

minimize

                            n-4
                            --        2
                   (eta)= >  (delta)                          (2)
                            --        i
                            i=5

subject to the constraint

                          m
                          --          2
                 (theta)= >  (epsilon) <=S                    (3)
                          --          r
                          r=1

where: (delta)    stands for the discontinuity jump in the third
              i   order derivative of s(x) at the interior knot
            (lambda) ,
                          i

       (epsilon)  denotes the weighted residual w (y -s(x )),
                r                                 r  r    r

and    S          is a non-negative number to be specified by
                  the user.

The quantity (eta) can be seen as a measure of the (lack of)
smoothness of s(x), while closeness of fit is measured through
(theta). By means of the parameter S, 'the smoothing factor', the
user will then control the balance between these two (usually
conflicting) properties. If S is too large, the spline will be
too smooth and signal will be lost (underfit); if S is too small,
```

the spline will pick up too much noise (overfit). In the extreme cases the routine will return an interpolating spline ((theta)=0) if S is set to zero, and the weighted least-squares cubic polynomial ((eta)=0) if S is set very large. Experimenting with S values between these two extremes should result in a good compromise. (See Section 8.2 for advice on choice of S.)

The method employed is outlined in Section 8.3 and fully described in Dierckx [1], [2] and [3]. It involves an adaptive strategy for locating the knots of the cubic spline (depending on the function underlying the data and on the value of S), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline, or of its derivatives or definite integral, can subsequently be computed by calling E02BBF, E02BCF or E02BDF, as described in Section 8.4.

4. References

[1] Dierckx P (1975) An Algorithm for Smoothing, Differentiating and Integration of Experimental Data Using Spline Functions. J. Comput. Appl. Math. 1 165--184.

[2] Dierckx P (1982) A Fast Algorithm for Smoothing Data on a Rectangular Grid while using Spline Functions. SIAM J. Numer. Anal. 19 1286--1304.

[3] Dierckx P (1981) An Improved Algorithm for Curve Fitting with Spline Functions. Report TW54. Department of Computer Science, Katholieke Universiteit Leuven.

[4] Reinsch C H (1967) Smoothing by Spline Functions. Num. Math. 10 177--183.

5. Parameters

1: START -- CHARACTER*1 Input
On entry: START must be set to 'C' or 'W'.

If START = 'C' (Cold start), the routine will build up the knot set starting with no interior knots. No values need be assigned to the parameters N, LAMDA, WRK or IWRK.

If START = 'W' (Warm start), the routine will restart the knot-placing strategy using the knots found in a previous call of the routine. In this case, the parameters N, LAMDA, WRK, and IWRK must be unchanged from that previous call. This warm start can save much time in searching for a satisfactory value of S. Constraint: START = 'C' or 'W'.

```
2:  M -- INTEGER                                              Input
    On entry: m, the number of data points. Constraint: M >= 4.

3:  X(M) -- DOUBLE PRECISION array                            Input
    On entry: the values x  of the independent variable
                          r
    (abscissa) x, for r=1,2,...,m. Constraint: x <x <...<x
                                                 1  2      m

4:  Y(M) -- DOUBLE PRECISION array                            Input
    On entry: the values y  of the dependent variable
                          r
    (ordinate) y, for r=1,2,...,m.

5:  W(M) -- DOUBLE PRECISION array                            Input
    On entry: the values w  of the weights, for r=1,2,...,m.
                          r
    For advice on the choice of weights, see the Chapter
    Introduction, Section 2.1.2. Constraint: W(r) > 0, for
    r=1,2,...,m.

6:  S -- DOUBLE PRECISION                                     Input
    On entry: the smoothing factor, S.

    If S=0.0, the routine returns an interpolating spline.

    If S is smaller than machine precision, it is assumed equal
    to zero.

    For advice on the choice of S, see Section 3 and Section 8.2
    Constraint: S >= 0.0.

7:  NEST -- INTEGER                                           Input
    On entry: an over-estimate for the number, n, of knots
    required. Constraint: NEST >= 8. In most practical
    situations, NEST = M/2 is sufficient. NEST never needs to be
    larger than M + 4, the number of knots needed for
    interpolation (S = 0.0).

8:  N -- INTEGER                                      Input/Output
    On entry: if the warm start option is used, the value of N
    must be left unchanged from the previous call. On exit: the
    total number, n, of knots of the computed spline.

9:  LAMDA(NEST) -- DOUBLE PRECISION array             Input/Output
    On entry: if the warm start option is used, the values
    LAMDA(1), LAMDA(2),...,LAMDA(N) must be left unchanged from
    the previous call. On exit: the knots of the spline i.e.,
    the positions of the interior knots LAMDA(5), LAMDA(6),...
```

,LAMDA(N-4) as well as the positions of the additional knots
LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = x and
                                             1

LAMDA(N-3) = LAMDA(N-2) = LAMDA(N-1) = LAMDA(N) = x needed
                                                   m
for the B-spline representation.

10: C(NEST) -- DOUBLE PRECISION array Output
On exit: the coefficient c of the B-spline N (x) in the
                          i   i
spline approximation s(x), for i=1,2,...,n-4.

11: FP -- DOUBLE PRECISION Output
On exit: the sum of the squared weighted residuals, (theta), of the computed spline approximation. If FP = 0.0, this is an interpolating spline. FP should equal S within a relative tolerance of 0.001 unless n=8 when the spline has no interior knots and so is simply a cubic polynomial. For knots to be inserted, S must be set to a value below the value of FP produced in this case.

12: WRK(LWRK) -- DOUBLE PRECISION array Workspace
On entry: if the warm start option is used, the values WRK (1),...,WRK(n) must be left unchanged from the previous call.

13: LWRK -- INTEGER Input
On entry:
the dimension of the array WRK as declared in the (sub)program from which E02BEF is called.
Constraint: LWRK>=4*M+16*NEST+41.

14: IWRK(NEST) -- INTEGER array Workspace
On entry: if the warm start option is used, the values IWRK (1), ..., IWRK(n) must be left unchanged from the previous call.

This array is used as workspace.

15: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

```
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry START /= 'C' or 'W',

     or       M < 4,

     or       S < 0.0,

     or       S = 0.0 and NEST < M + 4,

     or       NEST < 8,

     or       LWRK<4*M+16*NEST+41.

IFAIL= 2
     The weights are not all strictly positive.

IFAIL= 3
     The values of X(r), for r=1,2,...,M, are not in strictly
     increasing order.

IFAIL= 4
     The number of knots required is greater than NEST. Try
     increasing NEST and, if necessary, supplying larger arrays
     for the parameters LAMDA, C, WRK and IWRK. However, if NEST
     is already large, say NEST > M/2, then this error exit may
     indicate that S is too small.

IFAIL= 5
     The iterative process used to compute the coefficients of
     the approximating spline has failed to converge. This error
     exit may occur if S has been set very small. If the error
     persists with increased S, consult NAG.

If IFAIL = 4 or 5, a spline approximation is returned, but it
fails to satisfy the fitting criterion (see (2) and (3) in
Section 3) - perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its
weighted sum of squared residuals FP is equal to the smoothing
factor S, up to a specified relative tolerance of 0.001 - except
that if n=8, FP may be significantly less than S: in this case
the computed spline is simply a weighted least-squares polynomial
approximation of degree 3, i.e., a spline with no interior knots.
```

8. Further Comments

8.1. Timing

The time taken for a call of E02BEF depends on the complexity of the shape of the data, the value of the smoothing factor S, and the number of data points. If E02BEF is to be called for different values of S, much time can be saved by setting START =

8.2. Choice of S

If the weights have been correctly chosen (see Section 2.1.2 of the Chapter Introduction), the standard deviation of w y would
r r
be the same for all r, equal to (sigma), say. In this case,
2
choosing the smoothing factor S in the range (sigma) (m+-\/2m), as suggested by Reinsch [4], is likely to give a good start in the search for a satisfactory value. Otherwise, experimenting with different values of S will be required from the start, taking account of the remarks in Section 3.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for S and so determine the least-squares cubic polynomial; the value returned for FP, call it FP , gives an upper bound for S.
0
Then progressively decrease the value of S to obtain closer fits - say by a factor of 10 in the beginning, i.e., S=FP /10, S=FP
0 0
/100, and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location, generally depend on the value of S and on the behaviour of the function underlying the data. However, if E02BEF is called with START = 'W', the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of S and START = 'W', a fit can finally be accepted as satisfactory, it may be worthwhile to call E02BEF once more with the selected value for S but now using START = 'C'. Often, E02BEF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Outline of Method Used

If S=0, the requisite number of knots is known in advance, i.e., n=m+4; the interior knots are located immediately as (lambda) =

i
x , for i=5,6,...,n-4. The corresponding least-squares spline
 i-2
(see E02BAF) is then an interpolating spline and therefore a solution of the problem.

If S>0, a suitable knot set is built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a spline is fitted to the data by least-squares (see E02BAF) and (theta), the weighted sum of squares of residuals, is computed. If (theta)>S, new knots are added to the knot set to reduce (theta) at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of S and on the progress made so far in reducing (theta). Sooner or later, we find that (theta)<=S and at that point the knot set is accepted. The routine then goes on to compute the (unique) spline which has this knot set and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has (theta)=S. The routine computes the spline by an iterative scheme which is ended when (theta)=S within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02BAF.

An exception occurs when the routine finds at the start that, even with no interior knots (n=8), the least-squares spline already has its weighted sum of squares of residuals <=S. In this case, since this spline (which is simply a cubic polynomial) also has an optimal value for the smoothness measure (eta), namely zero, it is returned at once as the (trivial) solution. It will usually mean that S has been chosen too large.

For further details of the algorithm and its use, see Dierckx [3]

8.4. Evaluation of Computed Spline

The value of the computed spline at a given value X may be obtained in the double precision variable S by the call:

```
      CALL E02BBF(N,LAMDA,C,X,S,IFAIL)
```

where N, LAMDA and C are the output parameters of E02BEF.

The values of the spline and its first three derivatives at a given value X may be obtained in the double precision array SDIF of dimension at least 4 by the call:

```
       CALL E02BCF(N,LAMDA,C,X,LEFT,SDIF,IFAIL)

where if LEFT = 1, left-hand derivatives are computed and if LEFT
/= 1, right-hand derivatives are calculated. The value of LEFT is
only relevant if X is an interior knot.

The value of the definite integral of the spline over the
interval X(1) to X(M) can be obtained in the double precision
variable SINT by the call:

       CALL E02BDF(N,LAMDA,C,SINT,IFAIL)

9. Example

This example program reads in a set of data values, followed by a
set of values of S. For each value of S it calls E02BEF to
compute a spline approximation, and prints the values of the
knots and the B-spline coefficients c .
                                     i

The program includes code to evaluate the computed splines, by
calls to E02BBF, at the points x  and at points mid-way between
                                r
them. These values are not printed out, however; instead the
results are illustrated by plots of the computed splines,
together with the data points (indicated by *) and the positions
of the knots (indicated by vertical lines): the effect of
decreasing S can be clearly seen. (The plots were obtained by
calling NAG Graphical Supplement routine J06FAF(*).)


         Please see figures in printed Reference Manual

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E02 -- Curve and Surface Fitting                          E02DAF
        E02DAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DAF forms a minimal, weighted least-squares bicubic spline
```

surface fit with prescribed knots to a given set of data points.

2. Specification

```
 SUBROUTINE E02DAF (M, PX, PY, X, Y, F, W, LAMDA, MU,
1                   POINT, NPOINT, DL, C, NC, WS, NWS, EPS,
2                   SIGMA, RANK, IFAIL)
 INTEGER          M, PX, PY, POINT(NPOINT), NPOINT, NC, NWS,
1                 RANK, IFAIL
 DOUBLE PRECISION X(M), Y(M), F(M), W(M), LAMDA(PX), MU(PY),
1                 DL(NC), C(NC), WS(NWS), EPS, SIGMA
```

3. Description

This routine determines a bicubic spline fit s(x,y) to the set of data points (x_r, y_r, f_r) with weights w_r, for r=1,2,...,m. The two sets of internal knots of the spline, {(lambda)} and {(mu)}, associated with the variables x and y respectively, are prescribed by the user. These knots can be thought of as dividing the data region of the (x,y) plane into panels (see diagram in Section 5). A bicubic spline consists of a separate bicubic polynomial in each panel, the polynomials joining together with continuity up to the second derivative across the panel boundaries.

s(x,y) has the property that (Sigma), the sum of squares of its weighted residuals $(rho)_r$, for r=1,2,...,m, where

$$(rho)_r = w_r (s(x_r, y_r) - f_r), \qquad (1)$$

is as small as possible for a bicubic spline with the given knot sets. The routine produces this minimized value of (Sigma) and the coefficients c_{ij} in the B-spline representation of s(x,y) - see Section 8. E02DEF and E02DFF are available to compute values of the fitted spline from the coefficients c_{ij}.

The least-squares criterion is not always sufficient to determine the bicubic spline uniquely: there may be a whole family of splines which have the same minimum sum of squares. In these cases, the routine selects from this family the spline for which the sum of squares of the coefficients c_{ij} is smallest: in other words, the minimal least-squares solution. This choice, although arbitrary, reduces the risk of unwanted fluctuations in the spline fit. The method employed involves forming a system of m

linear equations in the coefficients c_{ij} and then computing its least-squares solution, which will be the minimal least-squares solution when appropriate. The basis of the method is described in Hayes and Halliday [4]. The matrix of the equation is formed using a recurrence relation for B-splines which is numerically stable (see Cox [1] and de Boor [2] - the former contains the more elementary derivation but, unlike [2], does not cover the case of coincident knots). The least-squares solution is also obtained in a stable manner by using orthogonal transformations, viz. a variant of Givens rotation (see Gentleman [3]). This requires only one row of the matrix to be stored at a time. Advantage is taken of the stepped-band structure which the matrix possesses when the data points are suitably ordered, there being at most sixteen non-zero elements in any row because of the definition of B-splines. First the matrix is reduced to upper triangular form and then the diagonal elements of this triangle are examined in turn. When an element is encountered whose square, divided by the mean squared weight, is less than a threshold (epsilon), it is replaced by zero and the rest of the elements in its row are reduced to zero by rotations with the remaining rows. The rank of the system is taken to be the number of non-zero diagonal elements in the final triangle, and the non-zero rows of this triangle are used to compute the minimal least-squares solution. If all the diagonal elements are non-zero, the rank is equal to the number of coefficients c_{ij} and the solution obtained is the ordinary least-squares solution, which is unique in this case.

4. References

[1] Cox M G (1972) The Numerical Evaluation of B-splines. J. Inst. Math. Appl. 10 134--149.

[2] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

[3] Gentleman W M (1973) Least-squares Computations by Givens Transformations without Square Roots. J. Inst. Math. Applic. 12 329--336.

[4] Hayes J G and Halliday J (1974) The Least-squares Fitting of Cubic Spline Surfaces to General Data Sets. J. Inst. Math. Appl. 14 89--103.

5. Parameters

1: M -- INTEGER Input
On entry: the number of data points, m. Constraint: M > 1.

2: PX -- INTEGER Input

3: PY -- INTEGER Input
On entry: the total number of knots (lambda) and (mu) associated with the variables x and y, respectively.
Constraint: PX >= 8 and PY >= 8.

(They are such that PX-8 and PY-8 are the corresponding numbers of interior knots.) The running time and storage required by the routine are both minimized if the axes are labelled so that PY is the smaller of PX and PY.

4: X(M) -- DOUBLE PRECISION array Input

5: Y(M) -- DOUBLE PRECISION array Input

6: F(M) -- DOUBLE PRECISION array Input
On entry: the co-ordinates of the data point (x_r, y_r, f_r), for
r=1,2,...,m. The order of the data points is immaterial, but see the array POINT, below.

7: W(M) -- DOUBLE PRECISION array Input
On entry: the weight w_r of the rth data point. It is important to note the definition of weight implied by the equation (1) in Section 3, since it is also common usage to define weight as the square of this weight. In this routine, each w_r should be chosen inversely proportional to the (absolute) accuracy of the corresponding f_r, as expressed, for example, by the standard deviation or probable error of the f_r. When the f_r are all of the same accuracy, all the w_r may be set equal to 1.0.

8: LAMDA(PX) -- DOUBLE PRECISION array Input/Output
On entry: LAMDA(i+4) must contain the ith interior knot (lambda) associated with the variable x, for
$i+4$
i=1,2,...,PX-8. The knots must be in non-decreasing order and lie strictly within the range covered by the data values of x. A knot is a value of x at which the spline is allowed to be discontinuous in the third derivative with respect to x, though continuous up to the second derivative. This degree of continuity can be reduced, if the user requires, by the use of coincident knots, provided that no more than four knots are chosen to coincide at any point. Two, or

three, coincident knots allow loss of continuity in, respectively, the second and first derivative with respect to x at the value of x at which they coincide. Four coincident knots split the spline surface into two independent parts. For choice of knots see Section 8. On exit: the interior knots LAMDA(5) to LAMDA(PX-4) are unchanged, and the segments LAMDA(1:4) and LAMDA(PX-3:PX) contain additional (exterior) knots introduced by the routine in order to define the full set of B-splines required. The four knots in the first segment are all set equal to the lowest data value of x and the other four additional knots are all set equal to the highest value: there is experimental evidence that coincident end-knots are best for numerical accuracy. The complete array must be left undisturbed if E02DEF or E02DFF is to be used subsequently.

9: MU(PY) -- DOUBLE PRECISION array Input
On entry: MU(i+4) must contain the ith interior knot (mu) i+4 associated with the variable y, i=1,2,...,PY-8. The same remarks apply to MU as to LAMDA above, with Y replacing X, and y replacing x.

10: POINT(NPOINT) -- INTEGER array Input
On entry: indexing information usually provided by E02ZAF which enables the data points to be accessed in the order which produces the advantageous matrix structure mentioned in Section 3. This order is such that, if the (x,y) plane is thought of as being divided into rectangular panels by the two sets of knots, all data in a panel occur before data in succeeding panels, where the panels are numbered from bottom to top and then left to right with the usual arrangement of axes, as indicated in the diagram.

Please see figure in printed Reference Manual

A data point lying exactly on one or more panel sides is considered to be in the highest numbered panel adjacent to the point. E02ZAF should be called to obtain the array POINT, unless it is provided by other means.

11: NPOINT -- INTEGER Input
On entry:
the dimension of the array POINT as declared in the (sub)program from which E02DAF is called.
Constraint: NPOINT >= M + (PX-7)*(PY-7).

12: DL(NC) -- DOUBLE PRECISION array Output
On exit: DL gives the squares of the diagonal elements of the reduced triangular matrix, divided by the mean squared

```
     weight. It includes those elements, less than (epsilon),
     which are treated as zero (see Section 3).

13:  C(NC) -- DOUBLE PRECISION array                          Output
     On exit: C gives the coefficients of the fit. C((PY-4)*(i-
     1)+j) is the coefficient c    of Section 3 and Section 8 for
                               ij
     i=1,2,...,PX-4 and j=1,2,...,PY-4. These coefficients are
     used by E02DEF or E02DFF to calculate values of the fitted
     function.

14:  NC -- INTEGER                                            Input
     On entry: the value (PX-4)*(PY-4).

15:  WS(NWS) -- DOUBLE PRECISION array                    Workspace

16:  NWS -- INTEGER                                           Input
     On entry:
     the dimension of the array WS as declared in the
     (sub)program from which E02DAF is called.
     Constraint: NWS>=(2*NC+1)*(3*PY-6)-2.

17:  EPS -- DOUBLE PRECISION                                  Input
     On entry: a threshold (epsilon) for determining the
     effective rank of the system of linear equations. The rank
     is determined as the number of elements of the array DL (see
     below) which are non-zero. An element of DL is regarded as
     zero if it is less than (epsilon). Machine precision is a
     suitable value for (epsilon) in most practical applications
     which have only 2 or 3 decimals accurate in data. If some
     coefficients of the fit prove to be very large compared with
     the data ordinates, this suggests that (epsilon) should be
     increased so as to decrease the rank. The array DL will give
     a guide to appropriate values of (epsilon) to achieve this,
     as well as to the choice of (epsilon) in other cases where
     some experimentation may be needed to determine a value
     which leads to a satisfactory fit.

18:  SIGMA -- DOUBLE PRECISION                                Output
     On exit: (Sigma), the weighted sum of squares of residuals.
     This is not computed from the individual residuals but from
     the right-hand sides of the orthogonally-transformed linear
     equations. For further details see Hayes and Halliday [4]
     page 97. The two methods of computation are theoretically
     equivalent, but the results may differ because of rounding
     error.

19:  RANK -- INTEGER                                          Output
     On exit: the rank of the system as determined by the value
     of the threshold (epsilon). When RANK = NC, the least-
```

```
     squares solution is unique: in other cases the minimal
     least-squares solution is computed.

20:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     At least one set of knots is not in non-decreasing order, or
     an interior knot is outside the range of the data values.

IFAIL= 2
     More than four knots coincide at a single point, possibly
     because all data points have the same value of x (or y) or
     because an interior knot coincides with an extreme data
     value.

IFAIL= 3
     Array POINT does not indicate the data points in panel
     order. Call E02ZAF to obtain a correct array.

IFAIL= 4
     On entry M <= 1,

     or       PX < 8,

     or       PY < 8,

     or       NC /= (PX-4)*(PY-4),

     or       NWS is too small,

     or       NPOINT is too small.

IFAIL= 5
     All the weights w  are zero or rank determined as zero.
                     r

7. Accuracy

The computation of the B-splines and reduction of the observation
matrix to triangular form are both numerically stable.
```

8. Further Comments

The time taken by this routine is approximately proportional to
                                                    2
the number of data points, m, and to (3*(PY-4)+4) .

The B-spline representation of the bicubic spline is

$$s(x,y)= \sum_{ij} c_{ij} M_i(x) N_j(y)$$

summed over i=1,2,...,PX-4 and over j=1,2,...,PY-4. Here $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines,the former defined on the knots $\lambda_i ,\lambda_{i+1} ,...,\lambda_{i+4}$ and the latter on the knots $\mu_j ,\mu_{j+1} ,...,\mu_{j+4}$. For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [2] for normalised B-splines.

The choice of the interior knots, which help to determine the spline's shape, must largely be a matter of trial and error. It is usually best to start with a small number of knots and, examining the fit at each stage, add a few knots at a time at places where the fit is particularly poor. In intervals of x or y where the surface represented by the data changes rapidly, in function value or derivatives, more knots will be needed than elsewhere. In some cases guidance can be obtained by analogy with the case of coincident knots: for example, just as three coincident knots can produce a discontinuity in slope, three close knots can produce rapid change in slope. Of course, such rapid changes in behaviour must be adequately represented by the data points, as indeed must the behaviour of the surface generally, if a satisfactory fit is to be achieved. When there is no rapid change in behaviour, equally-spaced knots will often suffice.

In all cases the fit should be examined graphically before it is accepted as satisfactory.

The fit obtained is not defined outside the rectangle

$$\lambda_4 \le x \le \lambda_{PX-3} , \quad \mu_4 \le y \le \mu_{PY-3}$$

```
The reason for taking the extreme data values of x and y for
these four knots is that, as is usual in data fitting, the fit
cannot be expected to give satisfactory values outside the data
region. If, nevertheless, the user requires values over a larger
rectangle, this can be achieved by augmenting the data with two
artificial data points (a,c,0) and (b,d,0) with zero weight,
where a<=x<=b, c<=y<=d defines the enlarged rectangle. In the
case when the data are adequate to make the least-squares
solution unique (RANK = NC), this enlargement will not affect the
fit over the original rectangle, except for possibly enlarged
rounding errors, and will simply continue the bicubic polynomials
in the panels bordering the rectangle out to the new boundaries:
in other cases the fit will be affected. Even using the original
rectangle there may be regions within it, particularly at its
corners, which lie outside the data region and where, therefore,
the fit will be unreliable. For example, if there is no data
point in panel 1 of the diagram in Section 5, the least-squares
criterion leaves the spline indeterminate in this panel: the
minimal spline determined by the subroutine in this case passes
through the value zero at the point ((lambda) ,(mu) ).
                                                 4     4

9. Example

This example program reads a value for (epsilon), and a set of
data points, weights and knot positions. If there are more y
knots than x knots, it interchanges the x and y axes. It calls
E02ZAF to sort the data points into panel order, E02DAF to fit a
bicubic spline to them, and E02DEF to evaluate the spline at the
data points.

Finally it prints:

     the weighted sum of squares of residuals computed from the
     linear equations;

     the rank determined by E02DAF;

     data points, fitted values and residuals in panel order;

     the weighted sum of squares of the residuals;

     the coefficients of the spline fit.

The program is written to handle any number of data sets.

Note: the data supplied in this example is not typical of a
realistic problem: the number of data points would normally be
much larger (in which case the array dimensions and the value of
NWS in the program would have to be increased); and the value of
```

```
(epsilon) would normally be much smaller on most machines (see
                                                    -6
Section 5; the relatively large value of 10   has been chosen in
order to illustrate a minimal least-squares solution when RANK <
NC; in this example NC = 24).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E02 -- Curve and Surface Fitting                              E02DCF
        E02DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DCF computes a bicubic spline approximation to a set of data
values, given on a rectangular grid in the x-y plane. The knots
of the spline are located automatically, but a single parameter
must be specified to control the trade-off between closeness of
fit and smoothness of fit.

2. Specification

       SUBROUTINE E02DCF (START, MX, X, MY, Y, F, S, NXEST,
      1                   NYEST, NX, LAMDA, NY, MU, C, FP, WRK,
      2                   LWRK, IWRK, LIWRK, IFAIL)
       INTEGER           MX, MY, NXEST, NYEST, NX, NY, LWRK, IWRK
      1                  (LIWRK), LIWRK, IFAIL
       DOUBLE PRECISION  X(MX), Y(MY), F(MX*MY), S, LAMDA(NXEST),
      1                  MU(NYEST), C((NXEST-4)*(NYEST-4)), FP, WRK
      2                  (LWRK)
       CHARACTER*1       START

3. Description

This routine determines a smooth bicubic spline approximation
s(x,y) to the set of data points (x ,y ,f   ), for q=1,2,...,m
                                  q  r  q,r                 x
and r=1,2,...,m .
             y

The spline is given in the B-spline representation
```

$$s(x,y)= \sum_{i=1}^{n_x-4} \sum_{j=1}^{n_y-4} c_{ij} M_i(x) N_j(y), \qquad (1)$$

where $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines, the former defined on the knots $(\lambda)_i$ to $(\lambda)_{i+4}$ and the latter on the knots $(\mu)_j$ to $(\mu)_{j+4}$. For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [1] for normalised B-splines.

The total numbers n_x and n_y of these knots and their values $(\lambda)_1,\ldots,(\lambda)_{n_x}$ and $(\mu)_1,\ldots,(\mu)_{n_y}$ are chosen automatically by the routine. The knots $(\lambda)_5,\ldots,(\lambda)_{n_x-4}$ and $(\mu)_5,\ldots,(\mu)_{n_y-4}$ are the interior knots; they divide the approximation domain $[x_4,x_{n_x-3}]*[y_4,y_{n_y-3}]$ into $(n_x-7)*(n_y-7)$ subpanels $[(\lambda)_i,(\lambda)_{i+1}]*[(\mu)_j,(\mu)_{j+1}]$, for $i=4,5,\ldots,n_x-4$, $j=4,5,\ldots,n_y-4$. Then, much as in the curve case (see E02BEF), the coefficients c_{ij} are determined as the solution of the following constrained minimization problem:

minimize

$$(\eta), \qquad (2)$$

subject to the constraint

$$(\theta)= \sum_{q=1}^{m_x} \sum_{r=1}^{m_y} (\epsilon)_{q,r}^2 \leq S, \qquad (3)$$

where (eta) is a measure of the (lack of) smoothness of s(x,y). Its value depends on the discontinuity jumps in s(x,y) across the boundaries of the subpanels. It is zero only when there are no discontinuities and is positive otherwise, increasing with the size of the jumps (see Dierckx [2] for details).

(epsilon) denotes the residual $f_{q,r} - s(x_q, y_r)$,

and S is a non-negative number to be specified by the user.

By means of the parameter S, 'the smoothing factor', the user will then control the balance between smoothness and closeness of fit, as measured by the sum of squares of residuals in (3). If S is too large, the spline will be too smooth and signal will be lost (underfit); if S is too small, the spline will pick up too much noise (overfit). In the extreme cases the routine will return an interpolating spline ((theta)=0) if S is set to zero, and the least-squares bicubic polynomial ((eta)=0) if S is set very large. Experimenting with S-values between these two extremes should result in a good compromise. (See Section 8.3 for advice on choice of S.)

The method employed is outlined in Section 8.5 and fully described in Dierckx [2] and [3]. It involves an adaptive strategy for locating the knots of the bicubic spline (depending on the function underlying the data and on the value of S), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline can subsequently be computed by calling E02DEF or E02DFF as described in Section 8.6.

4. References

[1] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

[2] Dierckx P (1982) A Fast Algorithm for Smoothing Data on a Rectangular Grid while using Spline Functions. SIAM J. Numer. Anal. 19 1286--1304.

[3] Dierckx P (1981) An Improved Algorithm for Curve Fitting with Spline Functions. Report TW54. Department of Computer Science, Katholieke Universiteit Leuven.

[4] Hayes J G and Halliday J (1974) The Least-squares Fitting of Cubic Spline Surfaces to General Data Sets. J. Inst. Math. Appl. 14 89--103.

[5] Reinsch C H (1967) Smoothing by Spline Functions. Num. Math. 10 177--183.

5. Parameters

1: START -- CHARACTER*1 Input
On entry: START must be set to 'C' or 'W'.

If START = 'C' (Cold start), the routine will build up the knot set starting with no interior knots. No values need be assigned to the parameters NX, NY, LAMDA, MU, WRK or IWRK.

If START = 'W' (Warm start), the routine will restart the knot-placing strategy using the knots found in a previous call of the routine. In this case, the parameters NX, NY, LAMDA, MU, WRK and IWRK must be unchanged from that previous call. This warm start can save much time in searching for a satisfactory value of S. Constraint: START = 'C' or 'W'.

2: MX -- INTEGER Input
On entry: m_x, the number of grid points along the x axis.
Constraint: MX >= 4.

3: X(MX) -- DOUBLE PRECISION array Input
On entry: X(q) must be set to x_q, the x co-ordinate of the qth grid point along the x axis, for q=1,2,...,m_x.
Constraint: $x_1 < x_2 < \ldots < x_{m_x}$.

4: MY -- INTEGER Input
On entry: m_y, the number of grid points along the y axis.
Constraint: MY >= 4.

5: Y(MY) -- DOUBLE PRECISION array Input
On entry: Y(r) must be set to y_r, the y co-ordinate of the rth grid point along the y axis, for r=1,2,...,m_y.
Constraint: $y_1 < y_2 < \ldots < y_{m_y}$.

6: F(MX*MY) -- DOUBLE PRECISION array Input
On entry: F(m *(q-1)+r) must contain the data value f ,

y q,r
for q=1,2,...,m and r=1,2,...,m .
x y

7: S -- DOUBLE PRECISION Input
On entry: the smoothing factor, S.

If S=0.0, the routine returns an interpolating spline.

If S is smaller than machine precision, it is assumed equal to zero.

For advice on the choice of S, see Section 3 and Section 8.3
Constraint: S >= 0.0.

8: NXEST -- INTEGER Input

9: NYEST -- INTEGER Input
On entry: an upper bound for the number of knots n and n
x y
required in the x- and y-directions respectively.

In most practical situations, NXEST =m /2 and NYEST m /2 is
x y
sufficient. NXEST and NYEST never need to be larger than
m +4 and m +4 respectively, the numbers of knots needed for
x y
interpolation (S=0.0). See also Section 8.4. Constraint:
NXEST >= 8 and NYEST >= 8.

10: NX -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NX must be left unchanged from the previous call. On exit: the total number of knots, n , of the computed spline with
x
respect to the x variable.

11: LAMDA(NXEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values LAMDA(1), LAMDA(2),...,LAMDA(NX) must be left unchanged from the previous call. On exit: LAMDA contains the complete set of knots (lambda) associated with the x variable, i.e., the
i
interior knots LAMDA(5), LAMDA(6), ..., LAMDA(NX-4) as well as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = X(1) and LAMDA(NX-3) = LAMDA(NX-2) = LAMDA(NX-1) = LAMDA(NX) = X(MX) needed for the B-spline representation.

12: NY -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NY

must be left unchanged from the previous call. On exit: the total number of knots, n , of the computed spline with
y
respect to the y variable.

13: MU(NYEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values MU (1), MU(2),...,MU(NY) must be left unchanged from the previous call. On exit: MU contains the complete set of knots (mu) associated with the y variable, i.e., the
i
interior knots MU(5), MU(6),...,MU(NY-4) as well as the additional knots MU(1) = MU(2) = MU(3) = MU(4) = Y(1) and MU (NY-3) = MU(NY-2) = MU(NY-1) = MU(NY) = Y(MY) needed for the B-spline representation.

14: C((NXEST-4)*(NYEST-4)) -- DOUBLE PRECISION array Output
On exit: the coefficients of the spline approximation. C((n -4)*(i-1)+j) is the coefficient c defined in Section 3.
y ij

15: FP -- DOUBLE PRECISION Output
On exit: the sum of squared residuals, (theta), of the computed spline approximation. If FP = 0.0, this is an interpolating spline. FP should equal S within a relative tolerance of 0.001 unless NX = NY = 8, when the spline has no interior knots and so is simply a bicubic polynomial. For knots to be inserted, S must be set to a value below the value of FP produced in this case.

16: WRK(LWRK) -- DOUBLE PRECISION array Workspace
On entry: if the warm start option is used, the values WRK (1),...,WRK(4) must be left unchanged from the previous call.

This array is used as workspace.

17: LWRK -- INTEGER Input
On entry:
the dimension of the array WRK as declared in the (sub)program from which E02DCF is called.
Constraint:
LWRK>=4*(MX+MY)+11*(NXEST+NYEST)+NXEST*MY

+max(MY,NXEST)+54.

18: IWRK(LIWRK) -- INTEGER array Workspace
On entry: if the warm start option is used, the values IWRK (1), ..., IWRK(3) must be left unchanged from the previous call.

```
     This array is used as workspace.

19:  LIWRK -- INTEGER                                          Input
     On entry:
     the dimension of the array IWRK as declared in the
     (sub)program from which E02DCF is called.
     Constraint: LIWRK >= 3 + MX + MY + NXEST + NYEST.

20:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry START /= 'C' or 'W',

     or       MX < 4,

     or       MY < 4,

     or       S < 0.0,

     or       S = 0.0 and NXEST < MX + 4,

     or       S = 0.0 and NYEST < MY + 4,

     or       NXEST < 8,

     or       NYEST < 8,

     or       LWRK < 4*(MX+MY)+11*(NXEST+NYEST)+NXEST*MY+
                     +max(MY,NXEST)+54

     or       LIWRK < 3 + MX + MY + NXEST + NYEST.

IFAIL= 2
     The values of X(q), for q = 1,2,...,MX, are not in strictly
     increasing order.
```

```
IFAIL= 3
     The values of Y(r), for r = 1,2,...,MY, are not in strictly
     increasing order.

IFAIL= 4
     The number of knots required is greater than allowed by
     NXEST and NYEST. Try increasing NXEST and/or NYEST and, if
     necessary, supplying larger arrays for the parameters LAMDA,
     MU, C, WRK and IWRK. However, if NXEST and NYEST are already
     large, say NXEST > MX/2 and NYEST > MY/2, then this error
     exit may indicate that S is too small.

IFAIL= 5
     The iterative process used to compute the coefficients of
     the approximating spline has failed to converge. This error
     exit may occur if S has been set very small. If the error
     persists with increased S, consult NAG.

If IFAIL = 4 or 5, a spline approximation is returned, but it
fails to satisfy the fitting criterion (see (2) and (3) in
Section 3) -- perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its
sum of squared residuals FP is equal to the smoothing factor S,
up to a specified relative tolerance of 0.001 - except that if
n =8 and n =8, FP may be significantly less than S: in this case
 x        y
the computed spline is simply the least-squares bicubic
polynomial approximation of degree 3, i.e., a spline with no
interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02DCF depends on the complexity of
the shape of the data, the value of the smoothing factor S, and
the number of data points. If E02DCF is to be called for
different values of S, much time can be saved by setting START =

8.2. Weighting of Data Points

E02DCF does not allow individual weighting of the data values. If
these were determined to widely differing accuracies, it may be
better to use E02DDF. The computation time would be very much
longer, however.

8.3. Choice of S
```

If the standard deviation of f is the same for all q and r
q,r
(the case for which this routine is designed - see Section 8.2.)
and known to be equal, at least approximately, to (sigma), say,
then following Reinsch [5] and choosing the smoothing factor S in
2
the range (sigma) (m+-\/2m), where m=m m , is likely to give a
x y
good start in the search for a satisfactory value. If the
standard deviations vary, the sum of their squares over all the
data points could be used. Otherwise experimenting with different
values of S will be required from the start, taking account of
the remarks in Section 3.

In that case, in view of computation time and memory
requirements, it is recommended to start with a very large value
for S and so determine the least-squares bicubic polynomial; the
value returned for FP, call it FP , gives an upper bound for S.
0
Then progressively decrease the value of S to obtain closer fits
- say by a factor of 10 in the beginning, i.e., S=FP /10,
0
S=FP /100, and so on, and more carefully as the approximation
0
shows more details.

The number of knots of the spline returned, and their location,
generally depend on the value of S and on the behaviour of the
function underlying the data. However, if E02DCF is called with
START = 'W', the knots returned may also depend on the smoothing
factors of the previous calls. Therefore if, after a number of
trials with different values of S and START = 'W', a fit can
finally be accepted as satisfactory, it may be worthwhile to call
E02DCF once more with the selected value for S but now using
START = 'C'. Often, E02DCF then returns an approximation with the
same quality of fit but with fewer knots, which is therefore
better if data reduction is also important.

8.4. Choice of NXEST and NYEST

The number of knots may also depend on the upper bounds NXEST and
NYEST. Indeed, if at a certain stage in E02DCF the number of
knots in one direction (say n) has reached the value of its
x
upper bound (NXEST), then from that moment on all subsequent
knots are added in the other (y) direction. Therefore the user
has the option of limiting the number of knots the routine
locates in any direction. For example, by setting NXEST = 8 (the
lowest allowable value for NXEST), the user can indicate that he

wants an approximation which is a simple cubic polynomial in the variable x.

8.5. Outline of Method Used

If S=0, the requisite number of knots is known in advance, i.e., n =m +4 and n =m +4; the interior knots are located immediately as (lambda) = x and (mu) = y , for i=5,6,...,n -4 and j=5,6,...,n -4. The corresponding least-squares spline is then an interpolating spline and therefore a solution of the problem.

If S>0, suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares, and (theta), the sum of squares of residuals, is computed. If (theta)>S, new knots are added to one knot set or the other so as to reduce (theta) at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of S and on the progress made so far in reducing (theta). Sooner or later, we find that (theta)<=S and at that point the knot sets are accepted. The routine then goes on to compute the (unique) spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has (theta)=S. The routine computes the spline by an iterative scheme which is ended when (theta)=S within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02BAF for least-squares curve fitting.

An exception occurs when the routine finds at the start that, even with no interior knots (n =n =8), the least-squares spline already has its sum of residuals <=S. In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure (eta), namely zero, it is returned at once as the (trivial) solution. It will usually mean that S has been chosen too large.

For further details of the algorithm and its use see Dierckx [2].

8.6. Evaluation of Computed Spline

The values of the computed spline at the points (TX(r),TY(r)), for r = 1,2,...,N, may be obtained in the double precision array FF, of length at least N, by the following code:

```
      IFAIL = 0
      CALL E02DEF(N,NX,NY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DCF , WRK is a double precision workspace array of length at least NY-4, and IWRK is an integer workspace array of length at least NY-4.

To evaluate the computed spline on a KX by KY rectangular grid of points in the x-y plane, which is defined by the x co-ordinates stored in TX(q), for q=1,2,...,KX, and the y co-ordinates stored in TY(r), for r=1,2,...,KY, returning the results in the double precision array FG which is of length at least KX*KY, the following call may be used:

```
      IFAIL = 0
      CALL E02DFF(KX,KY,NX,NY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
     *            IWRK,LIWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DCF , WRK is a double precision workspace array of length at least LWRK = min(NWRK1,NWRK2), NWRK1 = KX*4+NX, NWRK2 = KY*4+NY, and IWRK is an integer workspace array of length at least LIWRK = KY + NY - 4 if NWRK1 >= NWRK2, or KX + NX - 4 otherwise. The result of the spline evaluated at grid point (q,r) is returned in element (KY*(q-1)+r) of the array FG.

9. Example

This example program reads in values of MX, MY, x_q , for q = 1,2,. ordinates $f_{q,r}$ defined at the grid points (x_q, y_r). It then calls E02DCF to compute a bicubic spline approximation for one specified value of S, and prints the values of the computed knots and B-spline coefficients. Finally it evaluates the spline at a small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E02 -- Curve and Surface Fitting E02DDF

E02DDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for

your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DDF computes a bicubic spline approximation to a set of
scattered data. The knots of the spline are located
automatically, but a single parameter must be specified to
control the trade-off between closeness of fit and smoothness of
fit.

2. Specification

```
      SUBROUTINE E02DDF (START, M, X, Y, F, W, S, NXEST, NYEST,
     1                   NX, LAMDA, NY, MU, C, FP, RANK, WRK,
     2                   LWRK, IWRK, LIWRK, IFAIL)
      INTEGER           M, NXEST, NYEST, NX, NY, RANK, LWRK, IWRK
     1                  (LIWRK), LIWRK, IFAIL
      DOUBLE PRECISION X(M), Y(M), F(M), W(M), S, LAMDA(NXEST),
     1                 MU(NYEST), C((NXEST-4)*(NYEST-4)), FP, WRK
     2                 (LWRK)
      CHARACTER*1       START
```

3. Description

This routine determines a smooth bicubic spline approximation
s(x,y) to the set of data points (x ,y ,f) with weights w , for
                                  r r r   r
r=1,2,...,m.

The approximation domain is considered to be the rectangle
[x   ,x   ]*[y   ,y   ], where x   (y   ) and x   (y   ) denote
  min max   min max   min   min   max   max
the lowest and highest data values of x (y).

The spline is given in the B-spline representation

$$s(x,y)=\sum_{i=1}^{n_x-4}\sum_{j=1}^{n_y-4} c_{ij}\,M_i(x)N_j(y), \qquad (1)$$

where M (x) and N (y) denote normalised cubic B-splines, the
       i   j
former defined on the knots (lambda) to (lambda)   and the
                                   i   i+4
latter on the knots (mu) to (mu)   . For further details, see

Hayes and Halliday [4] for bicubic splines and de Boor [1] for normalised B-splines.

The total numbers n_x and n_y of these knots and their values $(\lambda)_1,\ldots,(\lambda)_{n_x}$ and $(\mu)_1,\ldots,(\mu)_{n_y}$ are chosen automatically by the routine. The knots $(\lambda)_5,\ldots,(\lambda)_{n_x-4}$ and $(\mu)_5,\ldots,(\mu)_{n_y-4}$ are the interior knots; they divide the approximation domain $[x_{min},x_{max}]*[y_{min},y_{max}]$ into $(n_x-7)*(n_y-7)$ subpanels $[(\lambda)_i,(\lambda)_{i+1}]*[(\mu)_j,(\mu)_{j+1}]$, for $i=4,5,\ldots,n_x-4$; $j=4,5,\ldots,n_y-4$. Then, much as in the curve case (see E02BEF), the coefficients c_{ij} are determined as the solution of the following constrained minimization problem:

minimize

$$(\eta), \tag{2}$$

subject to the constraint

$$(\theta)=\sum_{r=1}^{m}(\varepsilon)_r^2 \leq S \tag{3}$$

where: (η) is a measure of the (lack of) smoothness of s(x,y). Its value depends on the discontinuity jumps in s(x,y) across the boundaries of the subpanels. It is zero only when there are no discontinuities and is positive otherwise, increasing with the size of the jumps (see Dierckx [2] for details).

$(\varepsilon)_r$ denotes the weighted residual $w_r(f_r-s(x_r,y_r))$,

and S is a non-negative number to be specified by the user.

By means of the parameter S, 'the smoothness factor', the user will then control the balance between smoothness and closeness of

fit, as measured by the sum of squares of residuals in (3). If S is too large, the spline will be too smooth and signal will be lost (underfit); if S is too small, the spline will pick up too much noise (overfit). In the extreme cases the method would return an interpolating spline ((theta)=0) if S were set to zero, and returns the least-squares bicubic polynomial ((eta)=0) if S is set very large. Experimenting with S-values between these two extremes should result in a good compromise. (See Section 8.2 for advice on choice of S.) Note however, that this routine, unlike E02BEF and E02DCF, does not allow S to be set exactly to zero: to compute an interpolant to scattered data, E01SAF or E01SEF should be used.

The method employed is outlined in Section 8.5 and fully described in Dierckx [2] and [3]. It involves an adaptive strategy for locating the knots of the bicubic spline (depending on the function underlying the data and on the value of S), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline can subsequently be computed by calling E02DEF or E02DFF as described in Section 8.6.

4. References

[1] De Boor C (1972) On Calculating with B-splines. J. Approx. Theory. 6 50--62.

[2] Dierckx P (1981) An Algorithm for Surface Fitting with Spline Functions. IMA J. Num. Anal. 1 267--283.

[3] Dierckx P (1981) An Improved Algorithm for Curve Fitting with Spline Functions. Report TW54. Department of Computer Science, Katholieke Universiteit Leuven.

[4] Hayes J G and Halliday J (1974) The Least-squares Fitting of Cubic Spline Surfaces to General Data Sets. J. Inst. Math. Appl. 14 89--103.

[5] Peters G and Wilkinson J H (1970) The Least-squares Problem and Pseudo-inverses. Comput. J. 13 309--316.

[6] Reinsch C H (1967) Smoothing by Spline Functions. Num. Math. 10 177--183.

5. Parameters

1: START -- CHARACTER*1 Input
On entry: START must be set to 'C' or 'W'.

If START = 'C' (Cold start), the routine will build up the knot set starting with no interior knots. No values need be assigned to the parameters NX, NY, LAMDA, MU or WRK.

If START = 'W' (Warm start), the routine will restart the knot-placing strategy using the knots found in a previous call of the routine. In this case, the parameters NX, NY, LAMDA, MU and WRK must be unchanged from that previous call. This warm start can save much time in searching for a satisfactory value of S. Constraint: START = 'C' or 'W'.

2: M -- INTEGER Input
On entry: m, the number of data points.

The number of data points with non-zero weight (see W below) must be at least 16.

3: X(M) -- DOUBLE PRECISION array Input

4: Y(M) -- DOUBLE PRECISION array Input

5: F(M) -- DOUBLE PRECISION array Input
On entry: X(r), Y(r), F(r) must be set to the co-ordinates of (x_r, y_r, f_r), the rth data point, for r=1,2,...,m. The order of the data points is immaterial.

6: W(M) -- DOUBLE PRECISION array Input
On entry: W(r) must be set to w_r, the rth value in the set of weights, for r=1,2,...,m. Zero weights are permitted and the corresponding points are ignored, except when determining x_{min}, x_{max}, y_{min} and y_{max} (see Section 8.4). For advice on the choice of weights, see Section 2.1.2 of the Chapter Introduction. Constraint: the number of data points with non-zero weight must be at least 16.

7: S -- DOUBLE PRECISION Input
On entry: the smoothing factor, S.

For advice on the choice of S, see Section 3 and Section 8.2 . Constraint: S > 0.0.

8: NXEST -- INTEGER Input

9: NYEST -- INTEGER Input
On entry: an upper bound for the number of knots n_x and n_y

required in the x- and y-directions respectively.

In most practical situations, NXEST = NYEST = 4+\/m/2 is sufficient. See also Section 8.3. Constraint: NXEST >= 8 and NYEST >= 8.

10: NX -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NX must be left unchanged from the previous call. On exit: the total number of knots, n_x, of the computed spline with respect to the x variable.

11: LAMDA(NXEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values LAMDA(1), LAMDA(2),...,LAMDA(NX) must be left unchanged from the previous call. On exit: LAMDA contains the complete set of knots (lambda) associated with the x variable, i.e., the interior knots λ_i LAMDA(5), LAMDA(6),...,LAMDA(NX-4) as well as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = x_{min} and LAMDA(NX-3) = LAMDA(NX-2) = LAMDA(NX-1) = LAMDA(NX) = x_{max} needed for the B-spline representation (where x_{min} and x_{max} are as described in Section 3).

12: NY -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NY must be left unchanged from the previous call. On exit: the total number of knots, n_y, of the computed spline with respect to the y variable.

13: MU(NYEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values MU(1) MU(2),...,MU(NY) must be left unchanged from the previous call. On exit: MU contains the complete set of knots (mu) associated with the y variable, i.e., the interior knots μ_i MU(5), MU(6),...,MU(NY-4) as well as the additional knots MU(1) = MU(2) = MU(3) = MU(4) = y_{min} and MU(NY-3) = MU(NY-2) = MU(NY-1) = MU(NY) = y_{max} needed for the B-spline representation (where y_{min} and y_{max} are as described in Section 3).

14: C((NXEST-4)*(NYEST-4)) -- DOUBLE PRECISION array Output
On exit: the coefficients of the spline approximation. C(
(n -4)*(i-1)+j) is the coefficient c defined in Section 3.
y ij

15: FP -- DOUBLE PRECISION Output
On exit: the weighted sum of squared residuals, (theta), of the computed spline approximation. FP should equal S within a relative tolerance of 0.001 unless NX = NY = 8, when the spline has no interior knots and so is simply a bicubic polynomial. For knots to be inserted, S must be set to a value below the value of FP produced in this case.

16: RANK -- INTEGER Output
On exit: RANK gives the rank of the system of equations used to compute the final spline (as determined by a suitable machine-dependent threshold). When RANK = (NX-4)*(NY-4), the solution is unique; otherwise the system is rank-deficient and the minimum-norm solution is computed. The latter case may be caused by too small a value of S.

17: WRK(LWRK) -- DOUBLE PRECISION array Workspace
On entry: if the warm start option is used, the value of WRK (1) must be left unchanged from the previous call.

This array is used as workspace.

18: LWRK -- INTEGER Input
On entry:
the dimension of the array WRK as declared in the (sub)program from which E02DDF is called.
Constraint: LWRK >= (7*u*v+25*w)*(w+1)+2*(u+v+4*M)+23*w+56,

where

u=NXEST-4, v=NYEST-4, and w=max(u,v).

For some problems, the routine may need to compute the minimal least-squares solution of a rank-deficient system of linear equations (see Section 3). The amount of workspace required to solve such problems will be larger than specified by the value given above, which must be increased by an amount, LWRK2 say. An upper bound for LWRK2 is given by 4*u*v*w+2*u*v+4*w, where u, v and w are as above. However, if there are enough data points, scattered uniformly over the approximation domain, and if the smoothing factor S is not too small, there is a good chance that this extra workspace is not needed. A lot of memory might therefore be saved by assuming LWRK2 = 0.

```
19:  IWRK(LIWRK) -- INTEGER array                          Workspace


20:  LIWRK -- INTEGER                                          Input
     On entry:
     the dimension of the array IWRK as declared in the
     (sub)program from which E02DDF is called.
     Constraint: LIWRK>=M+2*(NXEST-7)*(NYEST-7).

21:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry START /= 'C' or 'W',

     or       the number of data points with non-zero weight <
              16,

     or       S <= 0.0,

     or       NXEST < 8,

     or       NYEST < 8,

     or       LWRK < (7*u*v+25*w)*(w+1)+2*(u+v+4*M)+23*w+56,
              where u = NXEST - 4, v = NYEST - 4 and w=max(u,v),

     or       LIWRK <M+2*(NXEST-7)*(NYEST-7).

IFAIL= 2
     On entry either all the X(r), for r = 1,2,...,M, are equal,
     or all the Y(r), for r = 1,2,...,M, are equal.

IFAIL= 3
     The number of knots required is greater than allowed by
     NXEST and NYEST. Try increasing NXEST and/or NYEST and, if
     necessary, supplying larger arrays for the parameters LAMDA,
     MU, C, WRK and IWRK. However, if NXEST and NYEST are already
```

```
     large, say NXEST, NYEST > 4 + \/M/2, then this error exit
     may indicate that S is too small.

IFAIL= 4
     No more knots can be added because the number of B-spline
     coefficients (NX-4)*(NY-4) already exceeds the number of
     data points M. This error exit may occur if either of S or M
     is too small.

IFAIL= 5
     No more knots can be added because the additional knot would
     (quasi) coincide with an old one. This error exit may occur
     if too large a weight has been given to an inaccurate data
     point, or if S is too small.

IFAIL= 6
     The iterative process used to compute the coefficients of
     the approximating spline has failed to converge. This error
     exit may occur if S has been set very small. If the error
     persists with increased S, consult NAG.

IFAIL= 7
     LWRK is too small; the routine needs to compute the minimal
     least-squares solution of a rank-deficient system of linear
     equations, but there is not enough workspace. There is no
     approximation returned but, having saved the information
     contained in NX, LAMDA, NY, MU and WRK, and having adjusted
     the value of LWRK and the dimension of array WRK
     accordingly, the user can continue at the point the program
     was left by calling E02DDF with START = 'W'. Note that the
     requested value for LWRK is only large enough for the
     current phase of the algorithm. If the routine is restarted
     with LWRK set to the minimum value requested, a larger
     request may be made at a later stage of the computation. See
     Section 5 for the upper bound on LWRK. On soft failure, the
     minimum requested value for LWRK is returned in IWRK(1) and
     the safe value for LWRK is returned in IWRK(2).

If IFAIL = 3,4,5 or 6, a spline approximation is returned, but it
fails to satisfy the fitting criterion (see (2) and (3) in
Section 3 -- perhaps only by a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its
weighted sum of squared residuals FP is equal to the smoothing
factor S, up to a specified relative tolerance of 0.001 - except
that if n =8 and n =8, FP may be significantly less than S: in
```

```
         x        y
this case the computed spline is simply the least-squares bicubic
polynomial approximation of degree 3, i.e., a spline with no
interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02DDF depends on the complexity of
the shape of the data, the value of the smoothing factor S, and
the number of data points. If E02DDF is to be called for
different values of S, much time can be saved by setting START =
It should be noted that choosing S very small considerably
increases computation time.

8.2. Choice of S

If the weights have been correctly chosen (see Section 2.1.2 of
the Chapter Introduction), the standard deviation of w f  would
                                                      r r
be the same for all r, equal to (sigma), say. In this case,
                                                     2
choosing the smoothing factor S in the range (sigma) (m+-\/2m),
as suggested by Reinsch [6], is likely to give a good start in
the search for a satisfactory value. Otherwise, experimenting
with different values of S will be required from the start.

In that case, in view of computation time and memory
requirements, it is recommended to start with a very large value
for S and so determine the least-squares bicubic polynomial; the
value returned for FP, call it FP , gives an upper bound for S.
                                  0
Then progressively decrease the value of S to obtain closer fits
- say by a factor of 10 in the beginning, i.e., S=FP /10,
                                                    0
S=FP /100, and so on, and more carefully as the approximation
    0
shows more details.

To choose S very small is strongly discouraged. This considerably
increases computation time and memory requirements. It may also
cause rank-deficiency (as indicated by the parameter RANK) and
endanger numerical stability.

The number of knots of the spline returned, and their location,
generally depend on the value of S and on the behaviour of the
function underlying the data. However, if E02DDF is called with
START = 'W', the knots returned may also depend on the smoothing
factors of the previous calls. Therefore if, after a number of
```

trials with different values of S and START = 'W', a fit can finally be accepted as satisfactory, it may be worthwhile to call E02DDF once more with the selected value for S but now using START = 'C'. Often, E02DDF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Choice of NXEST and NYEST

The number of knots may also depend on the upper bounds NXEST and NYEST. Indeed, if at a certain stage in E02DDF the number of knots in one direction (say n_x) has reached the value of its upper bound (NXEST), then from that moment on all subsequent knots are added in the other (y) direction. This may indicate that the value of NXEST is too small. On the other hand, it gives the user the option of limiting the number of knots the routine locates in any direction. For example, by setting NXEST = 8 (the lowest allowable value for NXEST), the user can indicate that he wants an approximation which is a simple cubic polynomial in the variable x.

8.4. Restriction of the approximation domain

The fit obtained is not defined outside the rectangle $[(\lambda)_4, (\lambda)_{n_x-3}] * [(\mu)_4, (\mu)_{n_y-3}]$. The reason for taking the extreme data values of x and y for these four knots is that, as is usual in data fitting, the fit cannot be expected to give satisfactory values outside the data region. If, nevertheless, the user requires values over a larger rectangle, this can be achieved by augmenting the data with two artificial data points (a,c,0) and (b,d,0) with zero weight, where [a,b]*[c,d] denotes the enlarged rectangle.

8.5. Outline of method used

First suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares and (theta), the sum of squares of residuals, is computed. If (theta)>S, a new knot is added to one knot set or the other so as to reduce (theta) at the next stage. The new knot is located in an interval where the fit is particularly poor. Sooner or later, we find that (theta)<=S and at that point the knot sets are accepted. The routine then goes on to compute a spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has (theta)=S.

The routine computes the spline by an iterative scheme which is ended when (theta)=S within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02DAF. As there also, the minimal least-squares solution is computed wherever the linear system is found to be rank-deficient.

An exception occurs when the routine finds at the start that, even with no interior knots (N = 8), the least-squares spline already has its sum of squares of residuals <=S. In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure (eta), namely zero, it is returned at once as the (trivial) solution. It will usually mean that S has been chosen too large.

For further details of the algorithm and its use see Dierckx [2].

8.6. Evaluation of computed spline

The values of the computed spline at the points (TX(r),TY(r)), for r = 1,2,...,N, may be obtained in the double precision array FF, of length at least N, by the following code:

```
      IFAIL = 0
      CALL E02DEF(N,NX,NY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DDF , WRK is a double precision workspace array of length at least NY-4, and IWRK is an integer workspace array of length at least NY-4.

To evaluate the computed spline on a KX by KY rectangular grid of points in the x-y plane, which is defined by the x co-ordinates stored in TX(q), for q=1,2,...,KX, and the y co-ordinates stored in TY(r), for r=1,2,...,KY, returning the results in the double precision array FG which is of length at least KX*KY, the following call may be used:

```
      IFAIL = 0
      CALL E02DFF(KX,KY,NX,NY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
     *            IWRK,LIWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DDF , WRK is a double precision workspace array of length at least LWRK = min(NWRK1,NWRK2), NWRK1 = KX*4+NX, NWRK2 = KY*4+NY, and

```
IWRK is an integer workspace array of length at least LIWRK = KY
+ NY - 4 if NWRK1 >= NWRK2, or KX + NX - 4 otherwise. The result
of the spline evaluated at grid point (q,r) is returned in
element (KY*(q-1)+r) of the array FG.

9. Example

This example program reads in a value of M, followed by a set of
M data points (x ,y ,f ) and their weights w . It then calls
               r  r  r                     r
E02DDF to compute a bicubic spline approximation for one
specified value of S, and prints the values of the computed knots
and B-spline coefficients. Finally it evaluates the spline at a
small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E02 -- Curve and Surface Fitting                          E02DEF
        E02DEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02DEF calculates values of a bicubic spline from its B-spline
representation.

2. Specification

       SUBROUTINE E02DEF (M, PX, PY, X, Y, LAMDA, MU, C, FF, WRK,
      1                   IWRK, IFAIL)
       INTEGER          M, PX, PY, IWRK(PY-4), IFAIL
       DOUBLE PRECISION X(M), Y(M), LAMDA(PX), MU(PY), C((PX-4)*
      1                 (PY-4)), FF(M), WRK(PY-4)

3. Description

This routine calculates values of the bicubic spline s(x,y) at
prescribed points (x ,y ), for r=1,2,...,m, from its augmented
                   r  r
knot sets {(lambda)} and {(mu)} and from the coefficients c  ,
                                                          ij
for i=1,2,...,PX-4; j=1,2,...,PY-4, in its B-spline
```

representation

$$s(x,y) = \sum_{ij} c_{ij} M_i(x) N_j(y).$$

Here $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines, the former defined on the knots $(\lambda)_i$ to $(\lambda)_{i+4}$ and the latter on the knots $(\mu)_j$ to $(\mu)_{j+4}$.

This routine may be used to calculate values of a bicubic spline given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF. It is derived from the routine B2VRE in Anthony et al [1].

4. References

[1] Anthony G T, Cox M G and Hayes J G (1982) DASL - Data Approximation Subroutine Library. National Physical Laboratory.

[2] Cox M G (1978) The Numerical Evaluation of a Spline from its B-spline Representation. J. Inst. Math. Appl. 21 135--143.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of points at which values of the spline are required. Constraint: M >= 1.

2: PX -- INTEGER Input

3: PY -- INTEGER Input
On entry: PX and PY must specify the total number of knots associated with the variables x and y respectively. They are such that PX-8 and PY-8 are the corresponding numbers of interior knots. Constraint: PX >= 8 and PY >= 8.

4: X(M) -- DOUBLE PRECISION array Input

5: Y(M) -- DOUBLE PRECISION array Input
On entry: X and Y must contain x_r and y_r, for r=1,2,...,m, respectively. These are the co-ordinates of the points at which values of the spline are required. The order of the points is immaterial. Constraint: X and Y must satisfy

```
     LAMDA(4) <= X(r) <= LAMDA(PX-3)

     and

     MU(4) <= Y(r) <= MU(PY-3), for r=1,2,...,m.

     The spline representation is not valid outside these
     intervals.

 6:  LAMDA(PX) -- DOUBLE PRECISION array                    Input

 7:  MU(PY) -- DOUBLE PRECISION array                       Input
     On entry: LAMDA and MU must contain the complete sets of
     knots {(lambda)} and {(mu)} associated with the x and y
     variables respectively. Constraint: the knots in each set
     must be in non-decreasing order, with LAMDA(PX-3) > LAMDA(4)
     and MU(PY-3) > MU(4).

 8:  C((PX-4)*(PY-4)) -- DOUBLE PRECISION array             Input
     On entry: C((PY-4)*(i-1)+j) must contain the coefficient
     c   described in Section 3, for i=1,2,...,PX-4;
      ij
     j=1,2,...,PY-4.

 9:  FF(M) -- DOUBLE PRECISION array                       Output
     On exit: FF(r) contains the value of the spline at the
     point (x ,y ), for r=1,2,...,m.
             r  r

10:  WRK(PY-4) -- DOUBLE PRECISION array                   Workspace

11:  IWRK(PY-4) -- INTEGER array                           Workspace

12:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 1,
```

```
     or       PY < 8,

     or       PX < 8.

IFAIL= 2
     On entry the knots in array LAMDA, or those in array MU, are
     not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or
     MU(PY-3) <= MU(4).

IFAIL= 3
     On entry at least one of the prescribed points (x ,y ) lies
                                                      r  r
     outside the rectangle defined by LAMDA(4), LAMDA(PX-3) and
     MU(4), MU(PY-3).

7. Accuracy

The method used to evaluate the B-splines is numerically stable,
in the sense that each computed value of s(x ,y ) can be regarded
                                           r  r
as the value that would have been obtained in exact arithmetic
from slightly perturbed B-spline coefficients. See Cox [2] for
details.

8. Further Comments

Computation time is approximately proportional to the number of
points, m, at which the evaluation is required.

9. Example

This program reads in knot sets LAMDA(1),..., LAMDA(PX) and MU(1)
,..., MU(PY), and a set of bicubic spline coefficients c  .
                                                       ij
Following these are a value for m and the co-ordinates (x ,y ),
                                                         r  r
for r=1,2,...,m, at which the spline is to be evaluated.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting                          E02DFF
       E02DFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
```

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02DFF calculates values of a bicubic spline from its B-spline representation. The spline is evaluated at all points on a rectangular grid.

2. Specification

```
      SUBROUTINE E02DFF (MX, MY, PX, PY, X, Y, LAMDA, MU, C, FF,
     1                   WRK, LWRK, IWRK, LIWRK, IFAIL)
      INTEGER           MX, MY, PX, PY, LWRK, IWRK(LIWRK), LIWRK,
     1                  IFAIL
      DOUBLE PRECISION X(MX), Y(MY), LAMDA(PX), MU(PY), C((PX-4)*
     1                  (PY-4)), FF(MX*MY), WRK(LWRK)
```

3. Description

This routine calculates values of the bicubic spline s(x,y) on a rectangular grid of points in the x-y plane, from its augmented knot sets {(lambda)} and {(mu)} and from the coefficients c_{ij} for i=1,2,...,PX-4; j=1,2,...,PY-4, in its B-spline representation

$$s(x,y) = \sum_{ij} c_{ij} M_i(x) N_j(y).$$

Here $M_i(x)$ and $N_j(y)$ denote normalised cubic B-splines, the former defined on the knots $(\lambda)_i$ to $(\lambda)_{i+4}$ and the latter on the knots $(\mu)_j$ to $(\mu)_{j+4}$.

The points in the grid are defined by co-ordinates x_q, for q=1,2,...,m$_x$, along the x axis, and co-ordinates y_r, for r=1,2,...,m$_y$ along the y axis.

This routine may be used to calculate values of a bicubic spline given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF. It is derived from the routine B2VRE in Anthony et al [1].

4. References

[1] Anthony G T, Cox M G and Hayes J G (1982) DASL - Data Approximation Subroutine Library. National Physical Laboratory.

[2] Cox M G (1978) The Numerical Evaluation of a Spline from its B-spline Representation. J. Inst. Math. Appl. 21 135--143.

5. Parameters

1: MX -- INTEGER Input

2: MY -- INTEGER Input
On entry: MX and MY must specify m_x and m_y respectively, the number of points along the x and y axis that define the rectangular grid. Constraint: MX >= 1 and MY >= 1.

3: PX -- INTEGER Input

4: PY -- INTEGER Input
On entry: PX and PY must specify the total number of knots associated with the variables x and y respectively. They are such that PX-8 and PY-8 are the corresponding numbers of interior knots. Constraint: PX >= 8 and PY >= 8.

5: X(MX) -- DOUBLE PRECISION array Input

6: Y(MY) -- DOUBLE PRECISION array Input
On entry: X and Y must contain x_q, for q=1,2,...,m_x, and y_r, for r=1,2,...,m_y, respectively. These are the x and y co-ordinates that define the rectangular grid of points at which values of the spline are required. Constraint: X and Y must satisfy

LAMDA(4) <= X(q) < X(q+1) <= LAMDA(PX-3), for q=1,2,...,m_x -1

and

MU(4) <= Y(r) < Y(r+1) <= MU(PY-3), for r=1,2,...,m_y -1.

The spline representation is not valid outside these intervals.

7: LAMDA(PX) -- DOUBLE PRECISION array Input

```
 8:  MU(PY) -- DOUBLE PRECISION array                          Input
     On entry: LAMDA and MU must contain the complete sets of
     knots {(lambda)} and {(mu)} associated with the x and y
     variables respectively. Constraint: the knots in each set
     must be in non-decreasing order, with LAMDA(PX-3) > LAMDA(4)
     and MU(PY-3) > MU(4).

 9:  C((PX-4)*(PY-4)) -- DOUBLE PRECISION array                Input
     On entry: C((PY-4)*(i-1)+j) must contain the coefficient
     c   described in Section 3, for i=1,2,...,PX-4;
      ij
     j=1,2,...,PY-4.

10:  FF(MX*MY) -- DOUBLE PRECISION array                     Output
     On exit: FF(MY*(q-1)+r) contains the value of the spline at
     the point (x ,y ), for q=1,2,...,m ; r=1,2,...,m .
                 q  r                  x             y

11:  WRK(LWRK) -- DOUBLE PRECISION array                   Workspace

12:  LWRK -- INTEGER                                          Input
     On entry:
     the dimension of the array WRK as declared in the
     (sub)program from which E02DFF is called.
     Constraint: LWRK >= min(NWRK1,NWRK2), where NWRK1=4*MX+PX,
     NWRK2=4*MY+PY.

13:  IWRK(LIWRK) -- INTEGER array                          Workspace

14:  LIWRK -- INTEGER                                         Input
     On entry:
     the dimension of the array IWRK as declared in the
     (sub)program from which E02DFF is called.
     Constraint: LIWRK >= MY + PY - 4 if NWRK1 > NWRK2, or MX +
     PX - 4 otherwise, where NWRK1 and NWRK2 are as defined in
     the description of argument LWRK.

15:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
```

```
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry MX < 1,

     or       MY < 1,

     or       PY < 8,

     or       PX < 8.

IFAIL= 2
     On entry LWRK is too small,

     or       LIWRK is too small.

IFAIL= 3
     On entry the knots in array LAMDA, or those in array MU, are
     not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or
     MU(PY-3) <= MU(4).

IFAIL= 4
     On entry the restriction LAMDA(4) <= X(1) <... < X(MX) <=
     LAMDA(PX-3), or the restriction MU(4) <= Y(1) <... < Y(MY)
     <= MU(PY-3), is violated.

7. Accuracy

The method used to evaluate the B-splines is numerically stable,
in the sense that each computed value of s(x ,y ) can be regarded
                                           r  r
as the value that would have been obtained in exact arithmetic
from slightly perturbed B-spline coefficients. See Cox [2] for
details.

8. Further Comments

Computation time is approximately proportional to m m +4(m +m ).
                                                   x y    x  y

9. Example

This program reads in knot sets LAMDA(1),..., LAMDA(PX) and MU(1)
,..., MU(PY), and a set of bicubic spline coefficients c  .
                                                        ij
Following these are values for m  and the x co-ordinates x , for
                               x                          q
q=1,2,...,m , and values for m  and the y co-ordinates y , for
           x                 y                         r
r=1,2,...,m , defining the grid of points on which the spline is
```

```
           y
to be evaluated.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting                              E02GAF
        E02GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02GAF calculates an l  solution to an over-determined system of
                      1
linear equations.

2. Specification

       SUBROUTINE E02GAF (M, A, LA, B, NPLUS2, TOLER, X, RESID,
      1                   IRANK, ITER, IWORK, IFAIL)
       INTEGER          M, LA, NPLUS2, IRANK, ITER, IWORK(M),
      1                 IFAIL
       DOUBLE PRECISION A(LA,NPLUS2), B(M), TOLER, X(NPLUS2),
      1                 RESID

3. Description

Given a matrix A with m rows and n columns (m>=n) and a vector b
with m elements, the routine calculates an l  solution to the
                                            1
over-determined system of equations

                               Ax=b.

That is to say, it calculates a vector x, with n elements, which
minimizes the l -norm (the sum of the absolute values) of the
              1
residuals

                                    m
                                    --
                           r(x)= >  |r |,
                                    --   i
```

i=1

where the residuals r are given by
i

n
--
r =b - > a x , i=1,2,...,m.
i i -- ij j
j=1

Here a is the element in row i and column j of A, b is the ith
ij i
element of b and x the jth element of x. The matrix A need not
j
be of full rank.

Typically in applications to data fitting, data consisting of m
points with co-ordinates (t ,y) are to be approximated in the l
i i 1
-norm by a linear combination of known functions (phi) (t),
j

(alpha) (phi) (t)+(alpha) (phi) (t)+...+(alpha) (phi) (t).
1 1 2 2 n n

This is equivalent to fitting an l solution to the over-
1
determined system of equations

n
--
> (phi) (t)(alpha) =y , i=1,2,...,m.
-- j i j i
j=1

Thus if, for each value of i and j, the element a of the matrix
ij
A in the previous paragraph is set equal to the value of
(phi) (t) and b is set equal to y , the solution vector x will
j i i i
contain the required values of the (alpha) . Note that the
j
independent variable t above can, instead, be a vector of several
independent variables (this includes the case where each (phi)
i
is a function of a different variable, or set of variables).

The algorithm is a modification of the simplex method of linear
programming applied to the primal formulation of the l problem

(see Barrodale and Roberts [1] and [2]). The modification allows several neighbouring simplex vertices to be passed through in a single iteration, providing a substantial improvement in efficiency.

4. References

[1] Barrodale I and Roberts F D K (1973) An Improved Algorithm for Discrete \\ll 1 Linear Approximation. SIAM J. Numer. Anal. 10 839--848.

[2] Barrodale I and Roberts F D K (1974) Solution of an Overdetermined System of Equations in the \\ll 1 -norm. Comm. ACM. 17, 6 319--320.

5. Parameters

1: M -- INTEGER Input
On entry: the number of equations, m (the number of rows of the matrix A). Constraint: M >= n >= 1.

2: A(LA,NPLUS2) -- DOUBLE PRECISION array Input/Output
On entry: A(i,j) must contain a_{ij}, the element in the ith row and jth column of the matrix A, for i=1,2,...,m and j=1,2,...,n. The remaining elements need not be set. On exit: A contains the last simplex tableau generated by the simplex method.

3: LA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which E02GAF is called.
Constraint: LA >= M + 2.

4: B(M) -- DOUBLE PRECISION array Input/Output
On entry: b_i, the ith element of the vector b, for i=1,2,...,m. On exit: the ith residual r_i corresponding to the solution vector x, for i=1,2,...,m.

5: NPLUS2 -- INTEGER Input
On entry: n+2, where n is the number of unknowns (the number of columns of the matrix A). Constraint: 3 <= NPLUS2 <= M + 2.

```
 6:  TOLER -- DOUBLE PRECISION                                  Input
     On entry: a non-negative value. In general TOLER specifies
     a threshold below which numbers are regarded as zero. The
                                          2/3
     recommended threshold value is (epsilon)    where (epsilon)
     is the machine precision. The recommended value can be
     computed within the routine by setting TOLER to zero. If
     premature termination occurs a larger value for TOLER may
     result in a valid solution. Suggested value: 0.0.

 7:  X(NPLUS2) -- DOUBLE PRECISION array                        Output
     On exit: X(j) contains the jth element of the solution
     vector x, for j=1,2,...,n. The elements X(n+1) and X(n+2)
     are unused.

 8:  RESID -- DOUBLE PRECISION                                  Output
     On exit: the sum of the absolute values of the residuals
     for the solution vector x.

 9:  IRANK -- INTEGER                                           Output
     On exit: the computed rank of the matrix A.

10:  ITER -- INTEGER                                            Output
     On exit: the number of iterations taken by the simplex
     method.

11:  IWORK(M) -- INTEGER array                                Workspace

12:  IFAIL -- INTEGER                                      Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     An optimal solution has been obtained but this may not be
     unique.

IFAIL= 2
     The calculations have terminated prematurely due to rounding
     errors. Experiment with larger values of TOLER or try
     scaling the columns of the matrix (see Section 8).

IFAIL= 3
```

```
     On entry NPLUS2 < 3,

     or       NPLUS2 > M + 2,

     or       LA < M + 2.

7. Accuracy

Experience suggests that the computational accuracy of the
solution x is comparable with the accuracy that could be obtained
by applying Gaussian elimination with partial pivoting to the n
equations satisfied by this algorithm (i.e., those equations with
zero residuals). The accuracy therefore varies with the
conditioning of the problem, but has been found generally very
satisfactory in practice.

8. Further Comments

The effects of m and n on the time and on the number of
iterations in the Simplex Method vary from problem to problem,
but typically the number of iterations is a small multiple of n
and the total time taken by the routine is approximately
                2
proportional to mn .

It is recommended that, before the routine is entered, the
columns of the matrix A are scaled so that the largest element in
each column is of the order of unity. This should improve the
conditioning of the matrix, and also enable the parameter TOLER
to perform its correct function. The solution x obtained will
then, of course, relate to the scaled form of the matrix. Thus if
the scaling is such that, for each j=1,2,...,n, the elements of
the jth column are multiplied by the constant k , the element x
                                               j              j
of the solution vector x must be multiplied by k  if it is
                                                j
desired to recover the solution corresponding to the original
matrix A.

9. Example

Suppose we wish to approximate a set of data by a curve of the
form

         t   -t
     y=Ke +Le  +M

where K, L and M are unknown. Given values y  at 5 points t  we
                                            i             i
may form the over-determined set of equations for K, L and M
```

$$e^{x_i} K + e^{-x_i} L + M = y_i, \quad i = 1,2,\ldots,5.$$

E02GAF is used to solve these in the l_1 sense.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E02 -- Curve and Surface Fitting E02ZAF
E02ZAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02ZAF sorts two-dimensional data into rectangular panels.

2. Specification

```
      SUBROUTINE E02ZAF (PX, PY, LAMDA, MU, M, X, Y, POINT,
     1                   NPOINT, ADRES, NADRES, IFAIL)
      INTEGER          PX, PY, M, POINT(NPOINT), NPOINT, ADRES
     1                 (NADRES), NADRES, IFAIL
      DOUBLE PRECISION LAMDA(PX), MU(PY), X(M), Y(M)
```

3. Description

A set of m data points with rectangular Cartesian co-ordinates x_r, y_r are sorted into panels defined by lines parallel to the y and x axes. The intercepts of these lines on the x and y axes are given in LAMDA(i), for i=5,6,...,PX-4 and MU(j), for j=5,6,...,PY-4, respectively. The subroutine orders the data so that all points in a panel occur before data in succeeding panels, where the panels are numbered from bottom to top and then left to right, with the usual arrangement of axes, as shown in the diagram. Within a panel the points maintain their original order.

Please see figure in printed Reference Manual

A data point lying exactly on one or more panel sides is taken to be in the highest-numbered panel adjacent to the point. The subroutine does not physically rearrange the data, but provides the array POINT which contains a linked list for each panel, pointing to the data in that panel. The total number of panels is (PX-7)*(PY-7).

4. References

None.

5. Parameters

1: PX -- INTEGER Input

2: PY -- INTEGER Input

On entry: PX and PY must specify eight more than the number of intercepts on the x axis and y axis, respectively. Constraint: PX >= 8 and PY >= 8.

3: LAMDA(PX) -- DOUBLE PRECISION array Input
On entry: LAMDA(5) to LAMDA(PX-4) must contain, in non-decreasing order, the intercepts on the x axis of the sides of the panels parallel to the y axis.

4: MU(PY) -- DOUBLE PRECISION array Input
On entry: MU(5) to MU(PY-4) must contain, in non-decreasing order, the intercepts on the y axis of the sides of the panels parallel to the x axis.

5: M -- INTEGER Input
On entry: the number m of data points.

6: X(M) -- DOUBLE PRECISION array Input

7: Y(M) -- DOUBLE PRECISION array Input
On entry: the co-ordinates of the rth data point (x_r, y_r), for r=1,2,...,m.

8: POINT(NPOINT) -- INTEGER array Output
On exit: for i = 1,2,...,NADRES, POINT(m+i) = I1 is the index of the first point in panel i, POINT(I1) = I2 is the index of the second point in panel i and so on.

POINT(IN) = 0 indicates that X(IN),Y(IN) was the last point in the panel.

```
     The co-ordinates of points in panel i can be accessed in
     turn by means of the following instructions:
        IN = M + I
     10 IN = POINT(IN)
        IF (IN.EQ. 0) GOTO 20
        XI = X(IN)
        YI = Y(IN)
        .
        .
        .
        GOTO 10
     20...

 9:  NPOINT -- INTEGER                                        Input
     On entry:
     the dimension of the array POINT as declared in the
     (sub)program from which E02ZAF is called.
     Constraint: NPOINT >= M + (PX-7)*(PY-7).

10:  ADRES(NADRES) -- INTEGER array                       Workspace

11:  NADRES -- INTEGER                                       Input
     On entry: the value (PX-7)*(PY-7), the number of panels
     into which the (x,y) plane is divided.

12:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The intercepts in the array LAMDA, or in the array MU, are
     not in non-decreasing order.

IFAIL= 2
     On entry PX < 8,

     or       PY < 8,

     or       M <= 0,
```

```
      or       NADRES /= (PX-7)*(PY-7),

      or       NPOINT < M + (PX-7)*(PY-7).

7. Accuracy

Not applicable.

8. Further Comments

The time taken by this routine is approximately proportional to
m*log(NADRES).

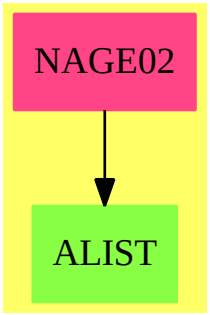
This subroutine was written to sort two dimensional data in the
manner required by routines E02DAF and E02DBF(*). The first 9
parameters of E02ZAF are the same as the parameters in E02DAF and
E02DBF(*) which have the same name.

9. Example

This example program reads in data points and the intercepts of
the panel sides on the x and y axes; it calls E02ZAF to set up
the index array POINT; and finally it prints the data points in
panel order.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.2.1 NagFittingPackage (NAGE02)



Exports:

e02adf	e02aef	e02agf	e02ahf	e02ajf
e02akf	e02baf	e02bbf	e02bcf	e02bdf
e02bef	e02daf	e02dcf	e02ddf	e02def
e02dff	e02gaf	e02zaf		

— package NAGE02 NagFittingPackage —

```
)abbrev package NAGE02 NagFittingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:59 1994
++ Description:
++ This package uses the NAG Library to find a
++ function which approximates a set of data points. Typically the
++ data contain random errors, as of experimental measurement, which
++ need to be smoothed out. To seek an approximation to the data, it
++ is first necessary to specify for the approximating function a
++ mathematical form (a polynomial, for example) which contains a
++ number of unspecified coefficients: the appropriate fitting
++ routine then derives for the coefficients the values which
++ provide the best fit of that particular form. The package deals
++ mainly with curve and surface fitting (i.e., fitting with
++ functions of one and of two variables) when a polynomial or a
++ cubic spline is used as the fitting function, since these cover
++ the most common needs. However, fitting with other functions
++ and/or more variables can be undertaken by means of general
++ linear or nonlinear routines (some of which are contained in
++ other packages) depending on whether the coefficients in the
++ function occur linearly or nonlinearly. Cases where a graph
++ rather than a set of data points is given can be treated simply
++ by first reading a suitable set of points from the graph.
++ The package also contains routines for evaluating,
++ differentiating and integrating polynomial and spline curves and
++ surfaces, once the numerical values of their coefficients have
++ been determined.

NagFittingPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    e02adf : (Integer,Integer,Integer,Matrix DoubleFloat,_
        Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
     ++ e02adf(m,kplus1,nrows,x,y,w,ifail)
     ++ computes weighted least-squares polynomial approximations
     ++ to an arbitrary set of data points.
     ++ See \downlink{Manual Page}{manpageXXe02adf}.
    e02aef : (Integer,Matrix DoubleFloat,DoubleFloat,Integer) -> Result
     ++ e02aef(nplus1,a,xcap,ifail)
```

```
 ++ evaluates a polynomial from its Chebyshev-series
 ++ representation.
 ++ See \downlink{Manual Page}{manpageXXe02aef}.
e02agf : (Integer,Integer,Integer,DoubleFloat,_
    DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Integer,Matrix Integer,Integer,Integer,Integer) -> Result
 ++ e02agf(m,kplus1,nrows,xmin,xmax,x,y,w,mf,xf,yf,lyf,ip,lwrk,liwrk,ifail)
 ++ computes constrained weighted least-squares polynomial
 ++ approximations in Chebyshev-series form to an arbitrary set of
 ++ data points. The values of the approximations and any number of
 ++ their derivatives can be specified at selected points.
 ++ See \downlink{Manual Page}{manpageXXe02agf}.
e02ahf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_
    Integer,Integer,Integer,Integer,Integer) -> Result
 ++ e02ahf(np1,xmin,xmax,a,ia1,la,iadif1,ladif,ifail)
 ++ determines the coefficients in the Chebyshev-series
 ++ representation of the derivative of a polynomial given in
 ++ Chebyshev-series form.
 ++ See \downlink{Manual Page}{manpageXXe02ahf}.
e02ajf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_
    Integer,Integer,DoubleFloat,Integer,Integer,Integer) -> Result
 ++ e02ajf(np1,xmin,xmax,a,ia1,la,qatm1,iaint1,laint,ifail)
 ++ determines the coefficients in the Chebyshev-series
 ++ representation of the indefinite integral of a polynomial given
 ++ in Chebyshev-series form.
 ++ See \downlink{Manual Page}{manpageXXe02ajf}.
e02akf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_
    Integer,Integer,DoubleFloat,Integer) -> Result
 ++ e02akf(np1,xmin,xmax,a,ia1,la,x,ifail)
 ++ evaluates a polynomial from its Chebyshev-series
 ++ representation, allowing an arbitrary index increment for
 ++ accessing the array of coefficients.
 ++ See \downlink{Manual Page}{manpageXXe02akf}.
e02baf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ e02baf(m,ncap7,x,y,w,lamda,ifail)
 ++ computes a weighted least-squares approximation to an
 ++ arbitrary set of data points by a cubic splines
 ++ prescribed by the user. Cubic spline can also be
 ++ carried out.
 ++ See \downlink{Manual Page}{manpageXXe02baf}.
e02bbf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,_
    Integer) -> Result
 ++ e02bbf(ncap7,lamda,c,x,ifail)
 ++ evaluates a cubic spline representation.
 ++ See \downlink{Manual Page}{manpageXXe02bbf}.
e02bcf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,_
    Integer,Integer) -> Result
 ++ e02bcf(ncap7,lamda,c,x,left,ifail)
```

```
 ++ evaluates a cubic spline and its first three derivatives
 ++ from its B-spline representation.
 ++ See \downlink{Manual Page}{manpageXXe02bcf}.
e02bdf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ e02bdf(ncap7,lamda,c,ifail)
 ++ computes the definite integral from its
 ++ B-spline representation.
 ++ See \downlink{Manual Page}{manpageXXe02bdf}.
e02bef : (String,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
    Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
    Matrix Integer) -> Result
 ++ e02bef(start,m,x,y,w,s,nest,lwrk,n,lamda,ifail,wrk,iwrk)
 ++ computes a cubic spline approximation to an arbitrary set
 ++ of data points. The knot are located
 ++ automatically, but a single parameter must be specified to
 ++ control the trade-off between closeness of fit and smoothness of
 ++ fit.
 ++ See \downlink{Manual Page}{manpageXXe02bef}.
e02daf : (Integer,Integer,Integer,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix Integer,Integer,Integer,Integer,_
    DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ e02daf(m,px,py,x,y,f,w,mu,point,npoint,nc,nws,eps,lamda,ifail)
 ++ forms a minimal, weighted least-squares bicubic spline
 ++ surface fit with prescribed knots to a given set of data points.
 ++ See \downlink{Manual Page}{manpageXXe02daf}.
e02dcf : (String,Integer,Matrix DoubleFloat,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Integer,Integer,Integer,Integer,Matrix DoubleFloat,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix Integer,Integer) -> Result
 ++ e02dcf(start,mx,x,my,y,f,s,nxest,nyest,lwrk,liwrk,nx,_
 ++        lamda,ny,mu,wrk,iwrk,ifail)
 ++ computes a bicubic spline approximation to a set of data
 ++ values, given on a rectangular grid in the x-y plane. The knots
 ++ of the spline are located automatically, but a single parameter
 ++ must be specified to control the trade-off between closeness of
 ++ fit and smoothness of fit.
 ++ See \downlink{Manual Page}{manpageXXe02dcf}.
e02ddf : (String,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Integer,Integer,Integer,Integer,Matrix DoubleFloat,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ e02ddf(start,m,x,y,f,w,s,nxest,nyest,lwrk,liwrk,nx,_
 ++        lamda,ny,mu,wrk,ifail)
 ++ computes a bicubic spline approximation to a set of
 ++ scattered data are located
 ++ automatically, but a single parameter must be specified to
 ++ control the trade-off between closeness of fit and smoothness of
```

```
    ++ fit.
    ++ See \downlink{Manual Page}{manpageXXe02ddf}.
  e02def : (Integer,Integer,Integer,Matrix DoubleFloat,_
      Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      Matrix DoubleFloat,Integer) -> Result
    ++ e02def(m,px,py,x,y,lamda,mu,c,ifail)
    ++ calculates values of a bicubic spline
    ++ representation.
    ++ See \downlink{Manual Page}{manpageXXe02def}.
  e02dff : (Integer,Integer,Integer,Integer,_
      Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer,_
      Integer) -> Result
    ++ e02dff(mx,my,px,py,x,y,lamda,mu,c,lwrk,liwrk,ifail)
    ++ calculates values of a bicubic spline
    ++ representation. The spline is evaluated at all points on a
    ++ rectangular grid.
    ++ See \downlink{Manual Page}{manpageXXe02dff}.
  e02gaf : (Integer,Integer,Integer,DoubleFloat,_
      Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
    ++ e02gaf(m,la,nplus2,toler,a,b,ifail)
    ++ calculates an l  solution to an over-determined system of
    ++ linear equations.
    ++ See \downlink{Manual Page}{manpageXXe02gaf}.
  e02zaf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
      Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
      Integer,Integer) -> Result
    ++ e02zaf(px,py,lamda,mu,m,x,y,npoint,nadres,ifail)
    ++ sorts two-dimensional data into rectangular panels.
    ++ See \downlink{Manual Page}{manpageXXe02zaf}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
  import AnyFunctions1(Integer)
  import AnyFunctions1(Matrix DoubleFloat)
  import AnyFunctions1(DoubleFloat)
  import AnyFunctions1(Matrix Integer)
  import AnyFunctions1(String)


  e02adf(mArg:Integer,kplus1Arg:Integer,nrowsArg:Integer,_
      xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
      wArg:Matrix DoubleFloat,_
```

```
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02adf",_
        ["m"::S,"kplus1"::S,"nrows"::S,"ifail"::S,"x"::S,"y"::S,_
        "w"::S,"a"::S,"s"::S_
        ,"work1"::S,"work2"::S]$Lisp,_
        ["a"::S,"s"::S,"work1"::S,"work2"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["w"::S,"m"::S]$Lisp,["a"::S,"nrows"::S,"kplus1"::S]$Lisp,_
        ["s"::S,"kplus1"::S]$Lisp,["work1"::S,_
        ["*"::S,3$Lisp,"m"::S]$Lisp]$Lisp_
        ,["work2"::S,["*"::S,2$Lisp,"kplus1"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"kplus1"::S,"nrows"::S_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["a"::S,"s"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,kplus1Arg::Any,nrowsArg::Any,ifailArg::Any,_
        xArg::Any,yArg::Any,wArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e02aef(nplus1Arg:Integer,aArg:Matrix DoubleFloat,xcapArg:DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02aef",_
        ["nplus1"::S,"xcap"::S,"p"::S,"ifail"::S,"a"::S]$Lisp,_
        ["p"::S]$Lisp,_
        [["double"::S,["a"::S,"nplus1"::S]$Lisp,"xcap"::S_
        ,"p"::S]$Lisp_
        ,["integer"::S,"nplus1"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["p"::S,"ifail"::S]$Lisp,_
        [([nplus1Arg::Any,xcapArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e02agf(mArg:Integer,kplus1Arg:Integer,nrowsArg:Integer,_
        xminArg:DoubleFloat,xmaxArg:DoubleFloat,xArg:Matrix DoubleFloat,_
        yArg:Matrix DoubleFloat,wArg:Matrix DoubleFloat,mfArg:Integer,_
        xfArg:Matrix DoubleFloat,yfArg:Matrix DoubleFloat,lyfArg:Integer,_
        ipArg:Matrix Integer,lwrkArg:Integer,liwrkArg:Integer,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02agf",_
        ["m"::S,"kplus1"::S,"nrows"::S,"xmin"::S,"xmax"::S_
        ,"mf"::S,"lyf"::S,"lwrk"::S,"liwrk"::S,"np1"::S_
        ,"ifail"::S,"x"::S,"y"::S,"w"::S,"xf"::S,"yf"::S_
        ,"ip"::S,"a"::S,"s"::S,"wrk"::S,"iwrk"::S_
        ]$Lisp,_
        ["a"::S,"s"::S,"np1"::S,"wrk"::S,"iwrk"::S]$Lisp,_
```

```
    [["double"::S,"xmin"::S,"xmax"::S,["x"::S,"m"::S]$Lisp_
    ,["y"::S,"m"::S]$Lisp,["w"::S,"m"::S]$Lisp,["xf"::S,"mf"::S]$Lisp,_
    ["yf"::S,"lyf"::S]$Lisp,["a"::S,"nrows"::S,"kplus1"::S]$Lisp_
    ,["s"::S,"kplus1"::S]$Lisp,["wrk"::S,"lwrk"::S]$Lisp]$Lisp_
    ,["integer"::S,"m"::S,"kplus1"::S,"nrows"::S_
    ,"mf"::S,"lyf"::S,["ip"::S,"mf"::S]$Lisp,"lwrk"::S,"liwrk"::S,_
    "np1"::S,"ifail"::S,["iwrk"::S,"liwrk"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["a"::S,"s"::S,"np1"::S,"wrk"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,kplus1Arg::Any,nrowsArg::Any,xminArg::Any,_
    xmaxArg::Any,mfArg::Any,lyfArg::Any,lwrkArg::Any,liwrkArg::Any,_
    ifailArg::Any,xArg::Any,yArg::Any,wArg::Any,xfArg::Any,_
    yfArg::Any,ipArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02ahf(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,_
    aArg:Matrix DoubleFloat,ia1Arg:Integer,laArg:Integer,_
    iadif1Arg:Integer,ladifArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02ahf",_
    ["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S_
    ,"iadif1"::S,"ladif"::S,"patm1"::S,"ifail"::S,"a"::S,"adif"::S]$Lisp,_
    ["patm1"::S,"adif"::S]$Lisp,_
    [["double"::S,"xmin"::S,"xmax"::S,["a"::S,"la"::S]$Lisp_
    ,"patm1"::S,["adif"::S,"ladif"::S]$Lisp]$Lisp_
    ,["integer"::S,"np1"::S,"ia1"::S,"la"::S,"iadif1"::S_
    ,"ladif"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["patm1"::S,"adif"::S,"ifail"::S]$Lisp,_
    [([np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,_
    iadif1Arg::Any,ladifArg::Any,ifailArg::Any,aArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02ajf(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,_
    aArg:Matrix DoubleFloat,ia1Arg:Integer,laArg:Integer,_
    qatm1Arg:DoubleFloat,iaint1Arg:Integer,laintArg:Integer,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02ajf",_
    ["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S_
    ,"qatm1"::S,"iaint1"::S,"laint"::S,"ifail"::S,"a"::S,"aint"::S]$Lisp,_
    ["aint"::S]$Lisp,_
    [["double"::S,"xmin"::S,"xmax"::S,["a"::S,"la"::S]$Lisp_
    ,"qatm1"::S,["aint"::S,"laint"::S]$Lisp]$Lisp_
    ,["integer"::S,"np1"::S,"ia1"::S,"la"::S,"iaint1"::S_
    ,"laint"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["aint"::S,"ifail"::S]$Lisp,_
```

```
        [([np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,_
        qatm1Arg::Any,iaint1Arg::Any,laintArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e02akf(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,_
        aArg:Matrix DoubleFloat,ia1Arg:Integer,laArg:Integer,_
        xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02akf",_
        ["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S_
        ,"x"::S,"result"::S,"ifail"::S,"a"::S]$Lisp,_
        ["result"::S]$Lisp,_
        [["double"::S,"xmin"::S,"xmax"::S,["a"::S,"la"::S]$Lisp_
        ,"x"::S,"result"::S]$Lisp_
        ,["integer"::S,"np1"::S,"ia1"::S,"la"::S,"ifail"::S_
        ]$Lisp_
        ]$Lisp,_
        ["result"::S,"ifail"::S]$Lisp,_
        [([np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,_
        xArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e02baf(mArg:Integer,ncap7Arg:Integer,xArg:Matrix DoubleFloat,_
        yArg:Matrix DoubleFloat,wArg:Matrix DoubleFloat,_
        lamdaArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02baf",_
        ["m"::S,"ncap7"::S,"ss"::S,"ifail"::S,"x"::S,"y"::S,"w"::S,_
        "c"::S,"lamda"::S_
        ,"work1"::S,"work2"::S]$Lisp,_
        ["c"::S,"ss"::S,"work1"::S,"work2"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["w"::S,"m"::S]$Lisp,["c"::S,"ncap7"::S]$Lisp,"ss"::S,_
        ["lamda"::S,"ncap7"::S]$Lisp,["work1"::S,"m"::S]$Lisp_
        ,["work2"::S,["*"::S,4$Lisp,"ncap7"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"ncap7"::S,"ifail"::S_
        ]$Lisp_
        ]$Lisp,_
        ["c"::S,"ss"::S,"lamda"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,ncap7Arg::Any,ifailArg::Any,xArg::Any,yArg::Any,_
        wArg::Any,lamdaArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e02bbf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
        cArg:Matrix DoubleFloat,xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02bbf",_
```

```
    ["ncap7"::S,"x"::S,"s"::S,"ifail"::S,"lamda"::S,"c"::S]$Lisp,_
    ["s"::S]$Lisp,_
    [["double"::S,["lamda"::S,"ncap7"::S]$Lisp_
    ,["c"::S,"ncap7"::S]$Lisp,"x"::S,"s"::S]$Lisp_
    ,["integer"::S,"ncap7"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["s"::S,"ifail"::S]$Lisp,_
    [([ncap7Arg::Any,xArg::Any,ifailArg::Any,lamdaArg::Any,cArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02bcf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
    cArg:Matrix DoubleFloat,_
    xArg:DoubleFloat,leftArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02bcf",_
    ["ncap7"::S,"x"::S,"left"::S,"ifail"::S,"lamda"::S,_
    "c"::S,"s"::S]$Lisp,_
    ["s"::S]$Lisp,_
    [["double"::S,["lamda"::S,"ncap7"::S]$Lisp_
    ,["c"::S,"ncap7"::S]$Lisp,"x"::S,["s"::S,4$Lisp]$Lisp]$Lisp_
    ,["integer"::S,"ncap7"::S,"left"::S,"ifail"::S_
    ]$Lisp_
    ]$Lisp,_
    ["s"::S,"ifail"::S]$Lisp,_
    [([ncap7Arg::Any,xArg::Any,leftArg::Any,ifailArg::Any,_
    lamdaArg::Any,cArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02bdf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
    cArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02bdf",_
    ["ncap7"::S,"defint"::S,"ifail"::S,"lamda"::S,"c"::S]$Lisp,_
    ["defint"::S]$Lisp,_
    [["double"::S,["lamda"::S,"ncap7"::S]$Lisp_
    ,["c"::S,"ncap7"::S]$Lisp,"defint"::S]$Lisp_
    ,["integer"::S,"ncap7"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["defint"::S,"ifail"::S]$Lisp,_
    [([ncap7Arg::Any,ifailArg::Any,lamdaArg::Any,cArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02bef(startArg:String,mArg:Integer,xArg:Matrix DoubleFloat,_
    yArg:Matrix DoubleFloat,wArg:Matrix DoubleFloat,sArg:DoubleFloat,_
    nestArg:Integer,lwrkArg:Integer,nArg:Integer,_
    lamdaArg:Matrix DoubleFloat,ifailArg:Integer,_
```

```
    wrkArg:Matrix DoubleFloat,_
    iwrkArg:Matrix Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02bef",_
    ["start"::S,"m"::S,"s"::S,"nest"::S,"lwrk"::S_
    ,"fp"::S,"n"::S,"ifail"::S,"x"::S,"y"::S,"w"::S,"c"::S,"lamda"::S_
    ,"wrk"::S,"iwrk"::S]$Lisp,_
    ["c"::S,"fp"::S]$Lisp,_
    [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
    ,["w"::S,"m"::S]$Lisp,"s"::S,["c"::S,"nest"::S]$Lisp,_
    "fp"::S,["lamda"::S,"nest"::S]$Lisp,["wrk"::S,"lwrk"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"m"::S,"nest"::S,"lwrk"::S,"n"::S_
    ,"ifail"::S,["iwrk"::S,"nest"::S]$Lisp]$Lisp_
    ,["character"::S,"start"::S]$Lisp_
    ]$Lisp,_
    ["c"::S,"fp"::S,"n"::S,"lamda"::S,"ifail"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    [([startArg::Any,mArg::Any,sArg::Any,nestArg::Any,lwrkArg::Any,_
    nArg::Any,ifailArg::Any,xArg::Any,yArg::Any,wArg::Any,_
    lamdaArg::Any,wrkArg::Any,iwrkArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02daf(mArg:Integer,pxArg:Integer,pyArg:Integer,_
    xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
    fArg:Matrix DoubleFloat,_
    wArg:Matrix DoubleFloat,muArg:Matrix DoubleFloat,_
    pointArg:Matrix Integer,_
    npointArg:Integer,ncArg:Integer,nwsArg:Integer,_
    epsArg:DoubleFloat,lamdaArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02daf",_
    ["m"::S,"px"::S,"py"::S,"npoint"::S,"nc"::S_
    ,"nws"::S,"eps"::S,"sigma"::S,"rank"::S,"ifail"::S_
    ,"x"::S,"y"::S,"f"::S,"w"::S,"mu"::S_
    ,"point"::S,"dl"::S,"c"::S,"lamda"::S,"ws"::S_
    ]$Lisp,_
    ["dl"::S,"c"::S,"sigma"::S,"rank"::S,"ws"::S]$Lisp,_
    [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
    ,["f"::S,"m"::S]$Lisp,["w"::S,"m"::S]$Lisp,_
    ["mu"::S,"py"::S]$Lisp,"eps"::S,["dl"::S,"nc"::S]$Lisp,_
    ["c"::S,"nc"::S]$Lisp_
    ,"sigma"::S,["lamda"::S,"px"::S]$Lisp,["ws"::S,"nws"::S]$Lisp]$Lisp_
    ,["integer"::S,"m"::S,"px"::S,"py"::S,["point"::S,"npoint"::S]$Lisp_
    ,"npoint"::S,"nc"::S,"nws"::S,"rank"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["dl"::S,"c"::S,"sigma"::S,"rank"::S,"lamda"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,pxArg::Any,pyArg::Any,npointArg::Any,ncArg::Any,_
    nwsArg::Any,epsArg::Any,ifailArg::Any,xArg::Any,yArg::Any,_
```

```
    fArg::Any,wArg::Any,muArg::Any,pointArg::Any,lamdaArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02dcf(startArg:String,mxArg:Integer,xArg:Matrix DoubleFloat,_
    myArg:Integer,yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
    sArg:DoubleFloat,nxestArg:Integer,nyestArg:Integer,_
    lwrkArg:Integer,liwrkArg:Integer,nxArg:Integer,_
    lamdaArg:Matrix DoubleFloat,nyArg:Integer,muArg:Matrix DoubleFloat,_
    wrkArg:Matrix DoubleFloat,iwrkArg:Matrix Integer,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02dcf",_
    ["start"::S,"mx"::S,"my"::S,"s"::S,"nxest"::S_
    ,"nyest"::S,"lwrk"::S,"liwrk"::S,"fp"::S,"nx"::S_
    ,"ny"::S,"ifail"::S,"x"::S,"y"::S,"f"::S,"c"::S,"lamda"::S_
    ,"mu"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    ["c"::S,"fp"::S]$Lisp,_
    [["double"::S,["x"::S,"mx"::S]$Lisp,["y"::S,"my"::S]$Lisp_
    ,["f"::S,["*"::S,"mx"::S,"my"::S]$Lisp]$Lisp,"s"::S,_
    ["c"::S,["*"::S,["-"::S,"nxest"::S,4$Lisp]$Lisp,_
    ["-"::S,"nyest"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ,"fp"::S,["lamda"::S,"nxest"::S]$Lisp,_
    ["mu"::S,"nyest"::S]$Lisp,["wrk"::S,"lwrk"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"mx"::S,"my"::S,"nxest"::S,"nyest"::S_
    ,"lwrk"::S,"liwrk"::S,"nx"::S,"ny"::S,["iwrk"::S,"liwrk"::S]$Lisp,_
    "ifail"::S]$Lisp_
    ,["character"::S,"start"::S]$Lisp_
    ]$Lisp,_
    ["c"::S,"fp"::S,"nx"::S,"lamda"::S,"ny"::S,"mu"::S,"wrk"::S,_
    "iwrk"::S,"ifail"::S]$Lisp,_
    [([startArg::Any,mxArg::Any,myArg::Any,sArg::Any,nxestArg::Any,_
    nyestArg::Any,lwrkArg::Any,liwrkArg::Any,nxArg::Any,nyArg::Any,_
    ifailArg::Any,xArg::Any,yArg::Any,fArg::Any,lamdaArg::Any,_
    muArg::Any,wrkArg::Any,iwrkArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02ddf(startArg:String,mArg:Integer,xArg:Matrix DoubleFloat,_
    yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
    wArg:Matrix DoubleFloat,_
    sArg:DoubleFloat,nxestArg:Integer,nyestArg:Integer,_
    lwrkArg:Integer,liwrkArg:Integer,nxArg:Integer,_
    lamdaArg:Matrix DoubleFloat,nyArg:Integer,muArg:Matrix DoubleFloat,_
    wrkArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02ddf",_
    ["start"::S,"m"::S,"s"::S,"nxest"::S,"nyest"::S_
    ,"lwrk"::S,"liwrk"::S,"fp"::S,"rank"::S,"nx"::S_
```

```
    ,"ny"::S,"ifail"::S,"x"::S,"y"::S,"f"::S,"w"::S,"c"::S_
    ,"iwrk"::S,"lamda"::S,"mu"::S,"wrk"::S]$Lisp,_
    ["c"::S,"fp"::S,"rank"::S,"iwrk"::S]$Lisp,_
    [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
    ,["f"::S,"m"::S]$Lisp,["w"::S,"m"::S]$Lisp,"s"::S,_
    ["c"::S,["*"::S,["-"::S,"nxest"::S,4$Lisp]$Lisp,_
    ["-"::S,"nyest"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ,"fp"::S,["lamda"::S,"nxest"::S]$Lisp,["mu"::S,"nyest"::S]$Lisp,_
    ["wrk"::S,"lwrk"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"m"::S,"nxest"::S,"nyest"::S_
    ,"lwrk"::S,"liwrk"::S,"rank"::S,["iwrk"::S,"liwrk"::S]$Lisp,_
    "nx"::S,"ny"::S,"ifail"::S]$Lisp_
    ,["character"::S,"start"::S]$Lisp_
    ]$Lisp,_
    ["c"::S,"fp"::S,"rank"::S,"iwrk"::S,"nx"::S,"lamda"::S,"ny"::S,_
    "mu"::S,"wrk"::S,"ifail"::S]$Lisp,_
    [([startArg::Any,mArg::Any,sArg::Any,nxestArg::Any,nyestArg::Any,_
    lwrkArg::Any,liwrkArg::Any,nxArg::Any,nyArg::Any,ifailArg::Any,_
    xArg::Any,yArg::Any,fArg::Any,wArg::Any,lamdaArg::Any,muArg::Any,_
    wrkArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02def(mArg:Integer,pxArg:Integer,pyArg:Integer,_
    xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
    lamdaArg:Matrix DoubleFloat,_
    muArg:Matrix DoubleFloat,cArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02def",_
    ["m"::S,"px"::S,"py"::S,"ifail"::S,"x"::S,"y"::S,"lamda"::S,_
     "mu"::S,"c"::S_
    ,"ff"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    ["ff"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
    ,["lamda"::S,"px"::S]$Lisp,["mu"::S,"py"::S]$Lisp,_
    ["c"::S,["*"::S,["-"::S,"px"::S,4$Lisp]$Lisp,_
    ["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ,["ff"::S,"m"::S]$Lisp,["wrk"::S,_
    ["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ,["integer"::S,"m"::S,"px"::S,"py"::S,"ifail"::S_
    ,["iwrk"::S,["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ]$Lisp,_
    ["ff"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,pxArg::Any,pyArg::Any,ifailArg::Any,xArg::Any,_
    yArg::Any,lamdaArg::Any,muArg::Any,cArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result
```

```
e02dff(mxArg:Integer,myArg:Integer,pxArg:Integer,_
    pyArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
    lamdaArg:Matrix DoubleFloat,muArg:Matrix DoubleFloat,_
    cArg:Matrix DoubleFloat,_
    lwrkArg:Integer,liwrkArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02dff",_
    ["mx"::S,"my"::S,"px"::S,"py"::S,"lwrk"::S_
    ,"liwrk"::S,"ifail"::S,"x"::S,"y"::S,"lamda"::S,"mu"::S,"c"::S_
    ,"ff"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    ["ff"::S,"wrk"::S,"iwrk"::S]$Lisp,_
    [["double"::S,["x"::S,"mx"::S]$Lisp,["y"::S,"my"::S]$Lisp_
    ,["lamda"::S,"px"::S]$Lisp,["mu"::S,"py"::S]$Lisp,_
    ["c"::S,["*"::S,["-"::S,"px"::S,4$Lisp]$Lisp,_
    ["-"::S,"py"::S,4$Lisp]$Lisp]$Lisp]$Lisp_
    ,["ff"::S,["*"::S,"mx"::S,"my"::S]$Lisp]$Lisp,_
    ["wrk"::S,"lwrk"::S]$Lisp]$Lisp_
    ,["integer"::S,"mx"::S,"my"::S,"px"::S,"py"::S_
    ,"lwrk"::S,"liwrk"::S,"ifail"::S,["iwrk"::S,"liwrk"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["ff"::S,"ifail"::S]$Lisp,_
    [([mxArg::Any,myArg::Any,pxArg::Any,pyArg::Any,lwrkArg::Any,_
    liwrkArg::Any,ifailArg::Any,xArg::Any,yArg::Any,lamdaArg::Any,_
    muArg::Any,cArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02gaf(mArg:Integer,laArg:Integer,nplus2Arg:Integer,_
    tolerArg:DoubleFloat,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e02gaf",_
    ["m"::S,"la"::S,"nplus2"::S,"toler"::S,"resid"::S,
    "irank"::S,"iter"::S,"ifail"::S,"x"::S,"a"::S,"b"::S,"iwork"::S]$Lisp,_
    ["x"::S,"resid"::S,"irank"::S,"iter"::S,"iwork"::S]$Lisp,_
    [["double"::S,"toler"::S,["x"::S,"nplus2"::S]$Lisp,
    "resid"::S,["a"::S,"la"::S,"nplus2"::S]$Lisp,_
    ["b"::S,"m"::S]$Lisp]$Lisp_
    ,["integer"::S,"m"::S,"la"::S,"nplus2"::S,"irank"::S_
    ,"iter"::S,"ifail"::S,["iwork"::S,"m"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["x"::S,"resid"::S,"irank"::S,"iter"::S,"a"::S,"b"::S,_
    "ifail"::S]$Lisp,_
    [([mArg::Any,laArg::Any,nplus2Arg::Any,tolerArg::Any,_
    ifailArg::Any,aArg::Any,bArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e02zaf(pxArg:Integer,pyArg:Integer,lamdaArg:Matrix DoubleFloat,_
    muArg:Matrix DoubleFloat,mArg:Integer,xArg:Matrix DoubleFloat,_
```

```
        yArg:Matrix DoubleFloat,npointArg:Integer,nadresArg:Integer,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e02zaf",_
        ["px"::S,"py"::S,"m"::S,"npoint"::S,"nadres"::S_
        ,"ifail"::S,"lamda"::S,"mu"::S,"x"::S,"y"::S,"point"::S_
        ,"adres"::S]$Lisp,_
        ["point"::S,"adres"::S]$Lisp,_
        [["double"::S,["lamda"::S,"px"::S]$Lisp,["mu"::S,"py"::S]$Lisp_
        ,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp]$Lisp_
        ,["integer"::S,"px"::S,"py"::S,"m"::S,"npoint"::S_
        ,"nadres"::S,["point"::S,"npoint"::S]$Lisp,"ifail"::S,_
        ["adres"::S,"nadres"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["point"::S,"ifail"::S]$Lisp,_
        [([pxArg::Any,pyArg::Any,mArg::Any,npointArg::Any,nadresArg::Any,_
        ifailArg::Any,lamdaArg::Any,muArg::Any,xArg::Any,yArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGE02.dotabb** —

```
"NAGE02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGE02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE02" -> "ALIST"
```

———-

15.3 package NAGF04 NagLinearEquationSolvingPackage

— **NagLinearEquationSolvingPackage.help** —

```
F04 -- Simultaneous Linear Equations          Introduction -- F04
                          Chapter F04
                 Simultaneous Linear Equations

1. Scope of the Chapter

This chapter, together with two routines in Chapter F07, is
concerned with the solution of the matrix equation AX=B, where B
```

may be a single vector or a matrix of multiple right-hand sides. The matrix A may be real, complex, symmetric, Hermitian positive-definite, or sparse. It may also be rectangular, in which case a least-squares solution is obtained.

2. Background to the Problems

A set of linear equations may be written in the form

Ax=b

where the known matrix A, with real or complex coefficients, is of size m by n, (m rows and n columns), the known right-hand vector b has m components (m rows and one column), and the required solution vector x has n components (n rows and one column). There may sometimes be p vectors b_i , i=1,2,...,p on the right-hand side and the equations may then be written as

AX=B

the required matrix X having as its p columns the solutions of $Ax_i = b_i$, i=1,2,...,p. Some routines deal with the latter case, but for clarity only the case p=1 is discussed here.

The most common problem, the determination of the unique solution of Ax=b, occurs when m=n and A is non-singular, that is rank(A)=n problem, discussed in Section 2.2 below, is the determination of the least-squares solution of Ax~=b, i.e., the determination of a vector x which minimizes the Euclidean length (two norm) of the residual vector r=b-Ax. The usual case has m>n and rank(A)=n, in which case x is unique.

2.1. Unique Solution of Ax=b

Most of the routines in this chapter, as well as two routines in Chapter F07, solve this particular problem. The solution is obtained by performing either an LU factorization, or a Cholesky factorization, as discussed in Section 2 of the F01 Chapter Introduction.

Two of the routines in this chapter use a process called iterative refinement to improve the initial solution in order to obtain a solution that is correct to working accuracy. It should be emphasised that if A and b are not known exactly then not all the figures in this solution may be meaningful. To be more precise, if x is the exact solution of the equations

Ax=b

and x is the solution of the perturbed equations

(A+E)x=b+e,

then, provided that (kappa)(A) ||E||/||A|| <=1,

$$\frac{\|x-x\|}{\|x\|} \le \frac{(\kappa)(A)}{1-(\kappa)(A)\frac{\|E\|}{\|A\|}}\left(\frac{\|E\|}{\|A\|}+\frac{\|e\|}{\|b\|}\right),$$

where $(\kappa)(A)=\|A\|\|A^{-1}\|$ is the condition number of A with respect to inversion. Thus, if A is ill-conditioned ((kappa)(A) is large), x may differ significantly from x. Often (kappa)(A) ||E||/||A|| <<1 in which case the above bound effectively reduces to

$$\frac{\|x-x\|}{\|x\|} \le (\kappa)(A)\left(\frac{\|E\|}{\|A\|}+\frac{\|e\|}{\|b\|}\right).$$

2.2. The Least-squares Solution of Ax~=b

The least-squares problem is to find a vector x to minimize

$$r^T r, \quad \text{where } r=b-Ax.$$

When m>=n and rank(A)=n then the solution vector x is unique. For the cases where x is not unique the routines in this chapter

obtain the minimal length solution, that is the vector x for
T
which x x is a minimum.

2.3. Calculating the Inverse of a Matrix

The routines in this chapter can also be used to calculate the inverse of a square matrix A by solving the equation

$$AX=I,$$

where I is the identity matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Purpose Routines

Many of the routines in this chapter perform the complete solution of the required equations, but some of the routines, as well as the routines in Chapter F07, assume that a prior factorization has been performed, using the appropriate factorization routine from Chapter F01 or Chapter F07. These, so-called, general purpose routines can be useful when explicit information on the factorization is required, as well as the solution of the equations, or when the solution is required for multiple right-hand sides, or for a sequence of right-hand sides.

Note that some of the routines that perform a complete solution also allow multiple right-hand sides.

3.2. Iterative Refinement

The routines that perform iterative refinement are more costly than those that do not perform iterative refinement, both in terms of time and storage, and should only be used if the problem really warrants the additional accuracy provided by these routines. The storage requirements are approximately doubled, while the additional time is not usually prohibitive since the initial factorization is used at each iteration.

3.3. Sparse Matrix Routines

The routines for sparse matrices should usually be used only when the number of non-zero elements is very small, less than 10% of the total number of elements of A. Additionally, when the matrix is symmetric positive-definite the sparse routines should generally be used only when A does not have a (variable) band structure.

There are four routines for solving sparse linear equations, two

for solving general real systems (F04AXF and F04QAF), one for solving symmetric positive-definite systems (F04MAF) and one for solving symmetric systems that may, or may not, be positive-definite (F04MBF). F04AXF and F04MAF utilise factorizations of the matrix A obtained by routines in Chapter F01, while the other two routines use iterative techniques and require a user-supplied function to compute matrix-vector products Ac and $A^T c$ for any given vector c. The routines requiring factorizations will usually be faster and the factorization can be utilised to solve for several right-hand sides, but the original matrix has to be explicitly supplied and is overwritten by the factorization, and the storage requirements will usually be substantially more than those of the iterative routines.

Routines F04MBF and F04QAF both allow the user to supply a pre-conditioner.

F04MBF can be used to solve systems of the form (A-(lambda)I)x=b, which can be useful in applications such as Rayleigh quotient iteration.

F04QAF also solves sparse least-squares problems and allows the solution of damped (regularized) least-squares problems.

3.4. Decision Trees

If at any stage the answer to a question is 'Don't know' this should be read as 'No'.

For those routines that need to be preceded by a factorization routine, the appropriate routine name is given in brackets after the name of the routine for solving the equations. Note also that you may be directed to a routine in Chapter F07.

3.4.1. Routines for unique solution of Ax=b

Please see figure in printed Reference Manual

3.4.2. Routines for Least-squares problems

Please see figure in printed Reference Manual

```
F04 -- Simultaneous Linear Equations              Contents -- F04
Chapter F04

Eigenvalues and Eigenvectors

F04ADF  Approximate solution of complex simultaneous linear
        equations with multiple right-hand sides

F04ARF  Approximate solution of real simultaneous linear
        equations, one right-hand side

F04ASF  Accurate solution of real symmetric positive-definite
        simultaneous linear equations, one right-hand side

F04ATF  Accurate solution of real simultaneous linear equations,
        one right-hand side

F04AXF  Approximate solution of real sparse simultaneous linear
        equations (coefficient matrix already factorized by
        F01BRF or F01BSF)

F04FAF  Approximate solution of real symmetric positive-definite
        tridiagonal simultaneous linear equations, one right-hand
        side

F04JGF  Least-squares (if rank = n) or minimal least-squares (if
        rank <n) solution of m real equations in n unknowns, rank
        <=n, m>=n

F04MAF  Real sparse symmetric positive-definite simultaneous
        linear equations (coefficient matrix already factorized)

F04MBF  Real sparse symmetric simultaneous linear equations

F04MCF  Approximate solution of real symmetric positive-definite
        variable-bandwidth simultaneous linear equations
        (coefficient matrix already factorized)

F04QAF  Sparse linear least-squares problem, m real equations in
        n unknowns

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations                      F04ADF
        F04ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
```

not included in the Foundation Library.

1. Purpose

F04ADF calculates the approximate solution of a set of complex linear equations with multiple right-hand sides, using an LU factorization with partial pivoting.

2. Specification

```
      SUBROUTINE F04ADF (A, IA, B, IB, N, M, C, IC, WKSPCE,
     1                   IFAIL)
      INTEGER              IA, IB, N, M, IC, IFAIL
      DOUBLE PRECISION     WKSPCE(*)
      COMPLEX(KIND(1.0D0)) A(IA,*), B(IB,*), C(IC,*)
```

3. Description

Given a set of complex linear equations AX=B, the routine first computes an LU factorization of A with partial pivoting, PA=LU, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. The columns x of the solution X are found by forward and backward substitution in Ly=Pb and Ux=y, where b is a column of the right-hand side matrix B.

4. References

[1] Wilkinson J H and Reinsch C (1971) Handbook for Automatic Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

1: A(IA,*) -- COMPLEX(KIND(1.0D)) array Input/Output
Note: the second dimension of the array A must be at least max(1,N).
On entry: the n by n matrix A. On exit: A is overwritten by the lower triangular matrix L and the off-diagonal elements of the upper triangular matrix U. The unit diagonal elements of U are not stored.

2: IA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F04ADF is called.
Constraint: IA >= max(1,N).

3: B(IB,*) -- COMPLEX(KIND(1.0D)) array Input
Note: the second dimension of the array B must be at least max(1,M).
On entry: the n by m right-hand side matrix B. See also

```
     Section 8.

 4:  IB -- INTEGER                                              Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F04ADF is called.
     Constraint: IB >= max(1,N).

 5:  N -- INTEGER                                               Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 6:  M -- INTEGER                                               Input
     On entry: m, the number of right-hand sides. Constraint: M
     >= 0.

 7:  C(IC,*) -- COMPLEX(KIND(1.0D)) array                      Output
     Note: the second dimension of the array C must be at least
     max(1,M).
     On exit: the n by m solution matrix X. See also Section 8.

 8:  IC -- INTEGER                                              Input
     On entry:
     the first dimension of the array C as declared in the
     (sub)program from which F04ADF is called.
     Constraint: IC >= max(1,N).

 9:  WKSPCE(*) -- DOUBLE PRECISION array                     Workspace
     Note: the dimension of the array WKSPCE must be at least
     max(1,N).

10:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The matrix A is singular, possibly due to rounding errors.

IFAIL= 2
     On entry N < 0,
```

```
     or       M < 0,

     or       IA < max(1,N),

     or       IB < max(1,N),

     or       IC < max(1,N).

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. For a detailed error analysis see
Wilkinson and Reinsch [1] page 106.

8. Further Comments

                                                                    3
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters B and C, in which case the solution
vectors will overwrite the right-hand sides. However this is not
standard Fortran 77, and may not work on all systems.

9. Example

To solve the set of linear equations AX=B where

                        (1  1+2i  2+10i)
                      A=(1+i  3i -5+14i)
                        (1+i  5i -8+20i)

and


                             (1)
                           B=(0).
                             (0)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F04 -- Simultaneous Linear Equations                        F04ARF
        F04ARF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ARF calculates the approximate solution of a set of real linear equations with a single right-hand side, using an LU factorization with partial pivoting.

2. Specification

```
SUBROUTINE F04ARF (A, IA, B, N, C, WKSPCE, IFAIL)
INTEGER          IA, N, IFAIL
DOUBLE PRECISION A(IA,*), B(*), C(*), WKSPCE(*)
```

3. Description

Given a set of linear equations, Ax=b, the routine first computes an LU factorization of A with partial pivoting, PA=LU, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. The approximate solution x is found by forward and backward substitution in Ly=Pb and Ux=y, where b is the right-hand side.

4. References

[1] Wilkinson J H and Reinsch C (1971) Handbook for Automatic Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

1: A(IA,*) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array A must be at least max(1,N).
On entry: the n by n matrix A. On exit: A is overwritten by the lower triangular matrix L and the off-diagonal elements of the upper triangular matrix U. The unit diagonal elements of U are not stored.

2: IA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F04ARF is called.
Constraint: IA >= max(1,N).

3: B(*) -- DOUBLE PRECISION array Input
Note: the dimension of the array B must be at least max(1,N).

```
     On entry: the right-hand side vector b.

 4:  N -- INTEGER                                              Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 5:  C(*) -- DOUBLE PRECISION array                           Output
     Note: the dimension of the array C must be at least
     max(1,N).
     On exit: the solution vector x.

 6:  WKSPCE(*) -- DOUBLE PRECISION array                   Workspace
     Note: the dimension of the array WKSPCE must be at least
     max(1,N).

 7:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The matrix A is singular, possibly due to rounding errors.

IFAIL= 2
     On entry N < 0,

     or       IA < max(1,N).

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. For a detailed error analysis see
Wilkinson and Reinsch [1] page 107.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
```

```
array supplied for parameters B and C, in which case the solution
vector will overwrite the right-hand side. However this is not
standard Fortran 77, and may not work on all systems.

9. Example

To solve the set of linear equations Ax=b where

                          ( 33  16  72)
                        A=(-24 -10 -57)
                          ( -8  -4 -17)

and


                            (-359)
                          b=( 281).
                            (  85)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F04 -- Simultaneous Linear Equations                    F04ASF
       F04ASF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F04ASF calculates the accurate solution of a set of real
symmetric positive-definite linear equations with a single right-
hand side,  Ax=b, using a Cholesky factorization and iterative
refinement.

2. Specification

      SUBROUTINE F04ASF (A, IA, B, N, C, WK1, WK2, IFAIL)
      INTEGER          IA, N, IFAIL
      DOUBLE PRECISION A(IA,*), B(*), C(*), WK1(*), WK2(*)

3. Description

Given a set of real linear equations Ax=b, where A is a symmetric
positive-definite matrix, the routine first computes a Cholesky
```

```
                          T
factorization of A as A=LL  where L is lower triangular. An
approximation to x is found by forward and backward substitution.
The residual vector r=b-Ax is then calculated using additional
                                                              T
precision and a correction d to x is found by solving LL d=r. x
is then replaced by x+d, and this iterative refinement of the
solution is repeated until machine accuracy is obtained.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,*) -- DOUBLE PRECISION array                Input/Output
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the upper triangle of the n by n positive-definite
     symmetric matrix A. The elements of the array below the
     diagonal need not be set. On exit: the elements of the array
     below the diagonal are overwritten; the upper triangle of A
     is unchanged.

 2:  IA -- INTEGER                                          Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F04ASF is called.
     Constraint: IA >= max(1,N).

 3:  B(*) -- DOUBLE PRECISION array                         Input
     Note: the dimension of the array B must be at least
     max(1,N).
     On entry: the right-hand side vector b.

 4:  N -- INTEGER                                           Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 5:  C(*) -- DOUBLE PRECISION array                        Output
     Note: the dimension of the array C must be at least
     max(1,N).
     On exit: the solution vector x.

 6:  WK1(*) -- DOUBLE PRECISION array                   Workspace
     Note: the dimension of the array WK1 must be at least
     max(1,N).

 7:  WK2(*) -- DOUBLE PRECISION array                   Workspace
     Note: the dimension of the array WK2 must be at least
```

```
     max(1,N).

 8:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The matrix A is not positive-definite, possibly due to
     rounding errors.

IFAIL= 2
     Iterative refinement fails to improve the solution, i.e.,
     the matrix A is too ill-conditioned.

IFAIL= 3
     On entry N < 0,

     or       IA < max(1,N).

7. Accuracy

The computed solutions should be correct to full machine
accuracy. For a detailed error analysis see Wilkinson and Reinsch
[1] page 39.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

The routine must not be called with the same name for parameters
B and C.

9. Example

To solve the set of linear equations Ax=b where

                          (5  7  6  5)
                          (7 10  8  7)
```

```
                              A=(6  8 10  9)
                                (5  7  9 10)
```

and

```
                                  (23)
                                  (32)
                                b=(33).
                                  (31)
```

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F04 -- Simultaneous Linear Equations F04ATF
F04ATF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ATF calculates the accurate solution of a set of real linear equations with a single right-hand side, using an LU factorization with partial pivoting, and iterative refinement.

2. Specification

```
      SUBROUTINE F04ATF (A, IA, B, N, C, AA, IAA, WKS1, WKS2,
     1                   IFAIL)
      INTEGER           IA, N, IAA, IFAIL
      DOUBLE PRECISION A(IA,*), B(*), C(*), AA(IAA,*), WKS1(*),
     1                 WKS2(*)
```

3. Description

Given a set of real linear equations, Ax=b, the routine first computes an LU factorization of A with partial pivoting, PA=LU, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. An approximation to x is found by forward and backward substitution in Ly=Pb and Ux=y. The residual vector r=b-Ax is then calculated using additional precision, and a correction d to x is found by solving LUd=r. x is replaced by x+d , and this iterative refinement of the solution is repeated until full machine accuracy is obtained.

```
4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  A(IA,*) -- DOUBLE PRECISION array                        Input
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the n by n matrix A.

 2:  IA -- INTEGER                                            Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F04ATF is called.
     Constraint: IA >= max(1,N).

 3:  B(*) -- DOUBLE PRECISION array                           Input
     Note: the dimension of the array B must be at least
     max(1,N).
     On entry: the right-hand side vector b.

 4:  N -- INTEGER                                             Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 5:  C(*) -- DOUBLE PRECISION array                          Output
     Note: the dimension of the array C must be at least
     max(1,N).
     On exit: the solution vector x.

 6:  AA(IAA,*) -- DOUBLE PRECISION array                     Output
     Note: the second dimension of the array AA must be at least
     max(1,N).
     On exit: the triangular factors L and U, except that the
     unit diagonal elements of U are not stored.

 7:  IAA -- INTEGER                                           Input
     On entry:
     the first dimension of the array AA as declared in the
     (sub)program from which F04ATF is called.
     Constraint: IAA >= max(1,N).

 8:  WKS1(*) -- DOUBLE PRECISION array                    Workspace
     Note: the dimension of the array WKS1 must be at least
     max(1,N).

 9:  WKS2(*) -- DOUBLE PRECISION array                    Workspace
     Note: the dimension of the array WKS2 must be at least
```

```
     max(1,N).

10:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The matrix A is singular, possibly due to rounding errors.

IFAIL= 2

     Iterative refinement fails to improve the solution, i.e.,
     the matrix A is too ill-conditioned.

IFAIL= 3
     On entry N < 0,

     or       IA < max(1,N),

     or       IAA < max(1,N).

7. Accuracy

The computed solutions should be correct to full machine
accuracy. For a detailed error analysis see Wilkinson and Reinsch
[1] page 107.

8. Further Comments

                                                                  3
The time taken by the routine is approximately proportional to n

The routine must not be called with the same name for parameters
B and C.

9. Example

To solve the set of linear equations Ax=b where
```

```
                                  ( 33  16  72)
                                A=(-24 -10 -57)
                                  ( -8  -4 -17)
```

and

```
                                    (-359)
                                 b=( 281).
                                    (  85)
```

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F04 -- Simultaneous Linear Equations F04AXF
F04AXF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04AXF calculates the approximate solution of a set of real sparse linear equations with a single right-hand side, Ax=b or $A^T x=b$, where A has been factorized by F01BRF or F01BSF.

2. Specification

```
      SUBROUTINE F04AXF (N, A, LICN, ICN, IKEEP, RHS, W, MTYPE,
     1                   IDISP, RESID)
      INTEGER          N, LICN, ICN(LICN), IKEEP(5*N), MTYPE,
     1                 IDISP(2)
      DOUBLE PRECISION A(LICN), RHS(N), W(N), RESID
```

3. Description

To solve a system of real linear equations Ax=b or $A^T x=b$, where A is a general sparse matrix, A must first be factorized by F01BRF or F01BSF. F04AXF then computes x by block forward or backward substitution using simple forward and backward substitution within each diagonal block.

The method is fully described in Duff [1].

```
4. References

[1]   Duff I S (1977) MA28 -- a set of Fortran subroutines for
      sparse unsymmetric linear equations. A.E.R.E. Report R.8730.
      HMSO.

5. Parameters

 1:  N -- INTEGER                                              Input
     On entry: n, the order of the matrix A.

 2:  A(LICN) -- DOUBLE PRECISION array                         Input
     On entry: the non-zero elements in the factorization of the
     matrix A, as returned by F01BRF or F01BSF.

 3:  LICN -- INTEGER                                           Input
     On entry:
     the dimension of the arrays A and ICN as declared in the
     (sub)program from which F04AXF is called.

 4:  ICN(LICN) -- INTEGER array                                Input
     On entry: the column indices of the non-zero elements of
     the factorization, as returned by F01BRF or F01BSF.

 5:  IKEEP(5*N) -- INTEGER array                               Input
     On entry: the indexing information about the factorization,
     as returned by F01BRF or F01BSF.

 6:  RHS(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: the right-hand side vector b. On exit: RHS is
     overwritten by the solution vector x.

 7:  W(N) -- DOUBLE PRECISION array                       Workspace

 8:  MTYPE -- INTEGER                                          Input
     On entry: MTYPE specifies the task to be performed:
          if MTYPE = 1, solve Ax=b,

                                T
          if MTYPE /= 1, solve A x=b.

 9:  IDISP(2) -- INTEGER array                                 Input
     On entry: the values returned in IDISP by F01BRF.

10:  RESID -- DOUBLE PRECISION                                Output
     On exit: the value of the maximum residual,
              --
     max(|b - > a  x |), over all the unsatisfied equations, in
           i  -- ij j
```

```
          j
     case F01BRF or F01BSF has been used to factorize a singular
     or rectangular matrix.

6. Error Indicators and Warnings

None.

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. Since F04AXF is always used with either
F01BRF or F01BSF, the user is recommended to set GROW = .TRUE. on
entry to these routines and to examine the value of W(1) on exit
(see the routine documents for F01BRF and F01BSF). For a detailed
error analysis see Duff [1] page 17.

If storage for the original matrix is available then the error
can be estimated by calculating the residual

                                   T
                    r=b-Ax  (or b-A x)

and calling F04AXF again to find a correction (delta) for x by
solving

                                  T
                A(delta)=r  (or A (delta)=r).

8. Further Comments

If the factorized form contains (tau) non-zeros (IDISP(2) = (tau)
) then the time taken is very approximately 2(tau) times longer
than the inner loop of full matrix code. Some advantage is taken
                                                  T
of zeros in the right-hand side when solving A x=b (MTYPE /= 1).

9. Example

To solve the set of linear equations Ax=b where

                     ( 5  0  0  0 0  0)
                     ( 0  2 -1  2 0  0)
                     ( 0  0  3  0 0  0)
                   A=(-2  0  0  1 1  0)
                     (-1  0  0 -1 2 -3)
                     (-1 -1  0  0 0  6)

and
```

```
                                  (15)
                                  (12)
                                  (18)
                               b=( 3).
                                  (-6)
                                  ( 0)
```

The non-zero elements of A and indexing information are read in by the program, as described in the document for F01BRF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F04 -- Simultaneous Linear Equations F04FAF
F04FAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04FAF calculates the approximate solution of a set of real symmetric positive-definite tridiagonal linear equations.

2. Specification

```
SUBROUTINE F04FAF (JOB, N, D, E, B, IFAIL)
INTEGER          JOB, N, IFAIL
DOUBLE PRECISION D(N), E(N), B(N)
```

3. Description

F04FAF is based upon the Linpack routine DPTSL (see Dongarra et al [1]) and solves the equations

$$Tx=b,$$

where T is a real n by n symmetric positive-definite tridiagonal matrix, using a modified symmetric Gaussian elimination algorithm to factorize T as $T=MKM^T$, where K is diagonal and M is a matrix of multipliers as described in Section 8.

When the input parameter JOB is supplied as 1, then the routine

```
assumes that a previous call to F04FAF has already factorized T;
otherwise JOB must be supplied as 0.

4. References

[1]   Dongarra J J, Moler C B, Bunch J R and Stewart G W (1979)
      LINPACK Users' Guide. SIAM, Philadelphia.

5. Parameters

 1:  JOB -- INTEGER                                          Input
     On entry: specifies the job to be performed by F04FAF as
     follows:
     JOB = 0
           The matrix T is factorized and the equations Tx=b are
           solved for x.

     JOB = 1
           The matrix T is assumed to have already been
           factorized by a previous call to F04FAF with JOB = 0;
           the equations Tx=b are solved for x.

 2:  N -- INTEGER                                            Input
     On entry: n, the order of the matrix T. Constraint: N >= 1.

 3:  D(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: if JOB = 0, D must contain the diagonal elements
     of T. If JOB = 1, D must contain the diagonal matrix K, as
     returned by a previous call of F04FAF with JOB = 0. On
     exit: if JOB = 0, D is overwritten by the diagonal matrix K
     of the factorization. If JOB = 1, D is unchanged.

 4:  E(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: if JOB = 0, E must contain the super-diagonal
     elements of T, stored in E(2) to E(n). If JOB = 1, E must
     contain the off-diagonal elements of the matrix M, as
     returned by a previous call of F04FAF with JOB = 0. E(1) is
     not used. On exit: if JOB = 0, E(2) to E(n) are overwritten
     by the off-diagonal elements of the matrix M of the
     factorization. If JOB = 1, E is unchanged.

 5:  B(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: the right-hand side vector b. On exit: B is
     overwritten by the solution vector x.

 6:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
```

```
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry N < 1,

     or       JOB /= 0 or 1.

IFAIL= 2
     The matrix T is either not positive-definite or is nearly
     singular. This failure can only occur when JOB = 0 and
     inspection of the elements of D will give an indication of
     why failure has occurred. If an element of D is close to
     zero, then T is probably nearly singular; if an element of D
     is negative but not close to zero, then T is not positive-
     definite.

IFAILOverflow
     If overflow occurs during the execution of this routine,
     then either T is very nearly singular or an element of the
     right-hand side vector b is very large. In this latter case
     the equations should be scaled so that no element of b is
     very large. Note that to preserve symmetry it is necessary
                                                        T
     to scale by a transformation of the form (PTP )b=Px, where P
     is a diagonal matrix.

IFAILUnderflow
     Any underflows that occur during the execution of this
     routine are harmless.

7. Accuracy

The computed factorization (see Section 8) will satisfy the
equation

                                T
                             MKM =T+E

where ||E|| <=2(epsilon)||T|| , p=1,F,infty,
           p                 p

(epsilon) being the machine precision. The computed solution of


the equations Tx=b, say x, will satisfy an equation of the form
```

```
                          (T+F)x=b,
```

where F can be expected to satisfy a bound of the form

```
               ||F||<=(alpha)(epsilon)||T||,
```

(alpha) being a modest constant. This implies that the relative

error in x satisfies

```
               ||x-x||
               -------<=c(T)(alpha)(epsilon),
                ||x||
```

where c(T) is the condition number of T with respect to

inversion. Thus if T is nearly singular, x can be expected to have a large relative error.

8. Further Comments

The time taken by the routine is approximately proportional to n.

The routine eliminates the off-diagonal elements of T by simultaneously performing symmetric Gaussian elimination from the top and the bottom of T. The result is that T is factorized as

```
                           T
                        T=MKM ,
```

where K is a diagonal matrix and M is a matrix of the form

```
              (1  0  0 ..  0    0 0    .. 0 0    0 )
              (m  1  0 ..  0    0 0    .. 0 0    0 )
              ( 2                                  )
              (0  m  1 ..  0    0 0    .. 0 0    0 )
              (    3                               )
              (.  .  . ..  .    . .    .. . .    . )
              (.  .  . ..  .    . .    .. . .    . )
              (0  0  0 ..  1    0 0    .. 0 0    0 )
            M=(0  0  0 ..  m    1 m    .. 0 0    0 )
              (             j+1    j+2             )
              (0  0  0 ..  0    0 ..   1 0  0    0 )
```

```
(.  .  . ..  .    . .    .. . .    . )
(.  .  . ..  .    . .    .. . .    . )
(0  0  0 ..  0    0 0    .. 1 m    0 )
(                              n-1   )
(0  0  0 ..  0    0 0    .. 0 1    m )
(                                   n)
(0  0  0 ..  0    0 0 .   . 0 0    1 )
```

j being the integer part of n/2. (For example when n=5,j=2.) The diagonal elements of K are returned in D with k_i in the ith element of D and m_i is returned in the ith element of E.

The routine fails with IFAIL = 2 if any diagonal element of K is non-positive. It should be noted that T may be nearly singular even if all the diagonal elements of K are positive, but in this case at least one element of K is almost certain to be small relative to |||T|||. If there is any doubt as to whether or not T is nearly singular, then the user should consider examining the diagonal elements of K.

9. Example

To solve the symmetric positive-definite equations

$$Tx_1 = b_1$$

and

$$Tx_2 = b_2$$

where

$$T = \begin{pmatrix} 4 & -2 & 0 & 0 & 0 \\ -2 & 10 & -6 & 0 & 0 \\ 0 & -6 & 29 & 15 & 0 \\ 0 & 0 & 15 & 25 & 8 \\ 0 & 0 & 0 & 8 & 5 \end{pmatrix}, \quad b_1 = \begin{pmatrix} 6 \\ 9 \\ 2 \\ (14) \\ (7) \end{pmatrix}, \quad b_2 = \begin{pmatrix} (10) \\ (4) \\ 9 \\ (65) \\ (23) \end{pmatrix}.$$

The equations are solved by two calls to F04FAF, the first with JOB = 0 and the second, using the factorization from the first call, with JOB = 1.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%

```
F04 -- Simultaneous Linear Equations                          F04JGF
        F04JGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F04JGF finds the solution of a linear least-squares problem, Ax=b
, where A is a real m by n (m>=n) matrix and b is an m element
vector. If the matrix of observations is not of full rank, then
the minimal least-squares solution is returned.

2. Specification

       SUBROUTINE F04JGF (M, N, A, NRA, B, TOL, SVD, SIGMA,
      1                   IRANK, WORK, LWORK, IFAIL)
       INTEGER          M, N, NRA, IRANK, LWORK, IFAIL
       DOUBLE PRECISION A(NRA,N), B(M), TOL, SIGMA, WORK(LWORK)
       LOGICAL          SVD

3. Description

The minimal least-squares solution of the problem Ax=b is the
vector x of minimum (Euclidean) length which minimizes the length
of the residual vector r=b-Ax.

The real m by n (m>=n) matrix A is factorized as

                                 (U)
                               A=Q(0)

where Q is an m by m orthogonal matrix and U is an n by n upper
triangular matrix. If U is of full rank, then the least-squares
solution is given by

                                  -1    T
                              x=(U   0)Q b.

If U is not of full rank, then the singular value decomposition
of U is obtained so that U is factorized as

                                    T
                                U=RDP ,

where R and P are n by n orthogonal matrices and D is the n by n
```

diagonal matrix

$$D=\mathrm{diag}(\sigma_1,\sigma_2,\ldots,\sigma_n),$$

with $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$, these being the singular values of A. If the singular values $\sigma_{k+1},\ldots,\sigma_n$ are negligible, but σ_k is not negligible, relative to the data errors in A, then the rank of A is taken to be k and the minimal least-squares solution is given by

$$x=P\begin{pmatrix} S^{-1} & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} R & 0 \\ 0 & I \end{pmatrix}Q^T b,$$

where $S=\mathrm{diag}(\sigma_1,\sigma_2,\ldots,\sigma_k)$.

This routine obtains the factorizations by a call to F02WDF(*).

The routine also returns the value of the standard error

$$\sigma = \begin{cases} \sqrt{\dfrac{r^T r}{m-k}}, & \text{if } m>k, \\ 0, & \text{if } m=k, \end{cases}$$

where $r^T r$ is the residual sum of squares and k the rank of A.

4. References

[1] Lawson C L and Hanson R J (1974) Solving Least-squares Problems. Prentice-Hall.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER Input
On entry: n, the number of columns of A. Constraint: 1 <= N <= M.

3: A(NRA,N) -- DOUBLE PRECISION array Input/Output

```
     On entry: the m by n matrix A. On exit: if SVD is returned
     as .FALSE., A} is overwritten by details of the QU
     factorization of A (see F02WDF(*) for further details). If
     SVD is returned as .TRUE., the first n rows of A are
     overwritten by the right-hand singular vectors, stored by
     rows; and the remaining rows of the array are used as
     workspace.

4:   NRA -- INTEGER                                          Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F04JGF is called.
     Constraint: NRA >= M.

5:   B(M) -- DOUBLE PRECISION array                  Input/Output
     On entry: the right-hand side vector b. On exit: the first
     n elements of B contain the minimal least-squares solution
     vector x. The remaining m-n elements are used for workspace.

6:   TOL -- DOUBLE PRECISION                                  Input
     On entry: a relative tolerance to be used to determine the
     rank of A. TOL should be chosen as approximately the largest
     relative error in the elements of A. For example, if the
     elements of A are correct to about 4 significant figures
                                          -4
     then TOL should be set to about 5*10  . See Section 8 for a
     description of how TOL is used to determine rank. If TOL is
     outside the range ((epsilon),1.0), where (epsilon) is the
     machine precision, then the value (epsilon) is used in place
     of TOL. For most problems this is unreasonably small.

7:   SVD -- LOGICAL                                          Output
     On exit: SVD is returned as .FALSE. if the least-squares
     solution has been obtained from the QU factorization of A.
     In this case A is of full rank. SVD is returned as .TRUE. if
     the least-squares solution has been obtained from the
     singular value decomposition of A.

8:   SIGMA -- DOUBLE PRECISION                               Output


                                                        / T
     On exit: the standard error, i.e., the value \/ r r/(m-k)
     when m>k, and the value zero when m=k. Here r is the
     residual vector b-Ax and k is the rank of A.

9:   IRANK -- INTEGER                                       Output
     On exit: k, the rank of the matrix A. It should be noted
     that it is possible for IRANK to be returned as n and SVD to
     be returned as .TRUE.. This means that the matrix U only
```

```
     just failed the test for non-singularity.

10:  WORK(LWORK) -- DOUBLE PRECISION array                Output
     On exit: if SVD is returned as .FALSE., then the first n
     elements of WORK contain information on the QU factorization
     of A (see parameter A above and F02WDF(*)), and WORK(n+1)
                                           -1
     contains the condition number ||U|| ||U  ||  of the
                                      E       E
     upper triangular matrix U.

     If SVD is returned as .TRUE., then the first n elements of
     WORK contain the singular values of A arranged in descending
     order and WORK(n+1) contains the total number of iterations
     taken by the QR algorithm. The rest of WORK is used as
     workspace.

11:  LWORK -- INTEGER                                        Input
     On entry:
     the dimension of the array WORK as declared in the
     (sub)program from which F04JGF is called.
     Constraint: LWORK >= 4*N.

12:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry N < 1,

     or       M < N,

     or       NRA < M,

     or       LWORK < 4*N.

IFAIL= 2
     The QR algorithm has failed to converge to the singular
     values in 50*N iterations. This failure can only happen when
     the singular value decomposition is employed, but even then
     it is not likely to occur.
```

```
7. Accuracy

                                    T
The computed factors Q, U, R, D and P  satisfy the relations

                   (U)        (R 0)(D) T
                 Q(0)=A+E , Q(0 I)(0)P =A+F,

where

                ||E|| <=c (epsilon)||A|| ,
                     2   1             2


                ||F|| <=c (epsilon)||A|| ,
                     2   2             2

(epsilon) being the machine precision, and c  and c  being modest
                                            1      2
functions of m and n. Note that ||A|| =(sigma) .
                                    2        1

For a fuller discussion, covering the accuracy of the solution x
see Lawson and Hanson [1], especially pp 50 and 95.

8. Further Comments

If the least-squares solution is obtained from the QU
factorization then the time taken by the routine is approximately
                2
proportional to n (3m-n). If the least-squares solution is
obtained from the singular value decomposition then the time
                                      2
taken is approximately proportional to n (3m+19n). The
approximate proportionality factor is the same in each case.

This routine is column biased and so is suitable for use in paged
environments.

Following the QU factorization of A the condition number

                                  -1
                    c(U)=||U|| ||U  ||
                              E       E

is determined and if c(U) is such that

                         c(U)*TOL>1.0

then U is regarded as singular and the singular values of A are
```

```
computed. If this test is not satisfied, U is regarded as non-
singular and the rank of A is set to n. When the singular values
are computed the rank of A, say k, is returned as the largest
integer such that

                     (sigma) >TOL*(sigma) ,
                            k            1

unless (sigma) =0 in which case k is returned as zero. That is,
              1
singular values which satisfy (sigma) <=TOL*(sigma)  are regarded
                                     i             1
as negligible because relative perturbations of order TOL can
make such singular values zero.

9. Example

To obtain a least-squares solution for r=b-Ax, where

                (0.05  0.05 0.25 -0.25)      (1)
                (0.25  0.25 0.05 -0.05)      (2)
                (0.35  0.35 1.75 -1.75)      (3)
              A=(1.75  1.75 0.35 -0.35),   B=(4)
                (0.30 -0.30 0.30  0.30)      (5)
                (0.40 -0.40 0.40  0.40)      (6)

                                        -4
and the value TOL is to be taken as 5*10  .

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F04 -- Simultaneous Linear Equations                        F04MAF
        F04MAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

To solve a sparse symmetric positive-definite system of linear
equations, Ax=b, using a pre-conditioned conjugate gradient
method, where A has been factorized by F01MAF.

2. Specification
```

```
      SUBROUTINE F04MAF (N, NZ, A, LICN, IRN, LIRN, ICN, B, ACC,
     1                   NOITS, WKEEP, WORK, IKEEP, INFORM,
     2                   IFAIL)
      INTEGER          N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN),
     1                 NOITS(2), IKEEP(2*N), INFORM(4), IFAIL
      DOUBLE PRECISION A(LICN), B(N), ACC(2), WKEEP(3*N), WORK
     1                 (3*N)
```

3. Description

F04MAF solves the n linear equations

$$Ax=b, \tag{1}$$

where A is a sparse symmetric positive-definite matrix, following the incomplete Cholesky factorization by F01MAF, given by

$$C=PLDL^{T}P^{T}, \quad WAW=C+E,$$

where P is a permutation matrix, L is a unit lower triangular matrix, D is a diagonal matrix with positive diagonal elements, E is an error matrix representing elements dropped during the factorization and diagonal elements that have been modified to ensure that C is positive-definite, and W is a diagonal matrix, chosen to make the diagonal elements of WAW unity.

Equation (1) is solved by applying a pre-conditioned conjugate gradient method to the equations

$$(WAW)(W^{-1}x)=Wb, \tag{2}$$

using C as the pre-conditioning matrix. Details of the conjugate gradient method are given in Munksgaard [1].

The iterative procedure is terminated if

$$||Wr||_2 <=(eta), \tag{3}$$

where r is the residual vector r=b-Ax, $||r||_2$ denotes the Euclidean length of r, (eta) is a user-supplied tolerance and x is the current approximation to the solution. Notice that

$$Wr=Wb-(WAW)(W^{-1}x)$$

so that Wr is the residual of the normalised equations (2).

F04MAF is based on the Harwell Library routine MA31B.

4. References

[1] Munksgaard N (1980) Solving Sparse Symmetric Sets of Linear Equations by Pre-conditioned Conjugate Gradients. ACM Trans. Math. Softw. 6 206--219.

5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix A. Constraint: N >= 1.

2: NZ -- INTEGER Input
On entry: the number of non-zero elements in the upper triangular part of the matrix A, including the number of elements on the leading diagonal. Constraint: NZ >= N.

3: A(LICN) -- DOUBLE PRECISION array Input
On entry: the first LROW elements, where LROW is the value supplied in INFORM(1), must contain details of the factorization, as returned by F01MAF.

4: LICN -- INTEGER Input
On entry: the length of the array A, as declared in the (sub)program from which F04MAF is called. It need never be larger than the value of LICN supplied to F01MAF.
Constraint: LICN >= INFORM(1).

5: IRN(LIRN) -- INTEGER array Input
On entry: the first LCOL elements, where LCOL is the value supplied in INFORM(2), must contain details of the factorization, as returned by F01MAF.

6: LIRN -- INTEGER Input
On entry: the length of the array IRN, as declared in the (sub)program from which F04MAF is called. It need never be larger than the value of LIRN supplied to F01MAF.
Constraint: LIRN >= INFORM(2).

7: ICN(LICN) -- INTEGER array Input
On entry: the first LROW elements, where LROW is the value supplied in INFORM(1), must contain details of the factorization, as returned by F01MAF.

8: B(N) -- DOUBLE PRECISION array Input/Output
On entry: the right-hand side vector b. On exit: B is overwritten by the solution vector x.

9: ACC(2) -- DOUBLE PRECISION array Input/Output
On entry: ACC(1) specifies the tolerance for convergence, (eta), in equation (3) of Section 3. If ACC(1) is outside the range [(epsilon),1], where (epsilon) is the machine precision, then the value (epsilon) is used in place of ACC (1). ACC(2) need not be set. On exit: ACC(2) contains the actual value of $||Wr||_2$ at the final point. ACC(1) is unchanged.

10: NOITS(2) -- INTEGER array Input/Output
On entry: NOITS(1) specifies the maximum permitted number of iterations. If NOITS(1) < 1, then the value 100 is used in its place. NOITS(2) need not be set. On exit: NOITS(2) contains the number of iterations taken to converge. NOITS (1) is unchanged.

11: WKEEP(3*N) -- DOUBLE PRECISION array Input
On entry: WKEEP must be unchanged from the previous call of F01MAF.

12: WORK(3*N) -- DOUBLE PRECISION array Output
On exit: WORK(1) contains a lower bound for the condition number of A. The rest of the array is used for workspace.

13: IKEEP(2*N) -- INTEGER array Input
On entry: IKEEP must be unchanged from the previous call of F01MAF.

14: INFORM(4) -- INTEGER array Input
On entry: INFORM must be unchanged from the previous call of F01MAF.

15: IFAIL -- INTEGER Input/Output
For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, IFAIL must be set to a value with the decimal expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.
a=0 specifies hard failure, otherwise soft failure;

b=0 suppresses error messages, otherwise error messages will be printed (see Section 6);

c=0 suppresses warning messages, otherwise warning messages will be printed (see Section 6).

```
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
     On entry N < 1,

     or       NZ < N,

     or       LICN < INFORM(1),

     or       LIRN < INFORM(2).

IFAIL= 2
     Convergence has not taken place within the requested NOITS
     (1) number of iterations. ACC(2) gives the value ||Wr|| ,
                                                        2
     for the final point. Either too few iterations have been
     allowed, or the requested convergence criterion cannot be
     met.

IFAIL= 3
     The matrix A is singular, or nearly singular. Singularity
     has been detected during the conjugate gradient iterations,
     so that the computations are not complete.

IFAIL= 4
     The matrix A is singular, or nearly singular. The message
     output on the current error message channel will include an
     estimate of the condition number of A. In the case of soft
     failure an approximate solution is returned such that the
     value ||Wr||  is given by ACC(2) and the estimate (a lower
                2
     bound) of the condition number is returned in WORK(1).

7. Accuracy

On successful return, or on return with IFAIL = 2 or IFAIL = 4
the computed solution will satisfy equation (3) of Section 3,
with (eta) = ACC(2).
```

8. Further Comments

The time taken by the routine will depend upon the sparsity of the factorization and the number of iterations required. The number of iterations will be affected by the nature of the factorization supplied by F01MAF. The more incomplete the factorization, the higher the number of iterations required by F04MAF.

When the solution of several systems of equations, all with the same matrix of coefficients, A, is required, then F01MAF need be called only once to factorize A. This is illustrated in the context of an eigenvalue problem in the example program for F02FJF.

9. Example

The example program illustrates the use of F01MAF in conjunction with F04MAF to solve the 16 linear equations Ax=b, where

$$
A=\begin{pmatrix}
1 & a & & a & & & & & & & & & & & & \\
a & 1 & a & & a & & & & & & & & & & & \\
 & a & 1 & a & & a & & & & & & & & & & \\
 & & a & 1 & 0 & & a & & & & & & & & & \\
a & & & 0 & 1 & a & & a & & & & & & & & \\
 & a & & & a & 1 & a & & a & & & & & & & \\
 & & a & & & a & 1 & a & & a & & & & & & \\
 & & & a & & & a & 1 & 0 & & a & & & & & \\
 & & & & a & & & 0 & 1 & a & & a & & & & \\
 & & & & & a & & & a & 1 & a & & a & & & \\
 & & & & & & a & & & a & 1 & a & & a & & \\
 & & & & & & & a & & & a & 1 & 0 & & a & \\
 & & & & & & & & a & & & 0 & 1 & a & & \\
 & & & & & & & & & a & & & a & 1 & a & \\
 & & & & & & & & & & a & & & a & 1 & a \\
 & & & & & & & & & & & a & & & a & 1
\end{pmatrix}
$$

$$
\text{where } a=-\frac{1}{4}.
$$

$$
b=\left(\frac{1}{2}\ \frac{1}{4}\ \frac{1}{4}\ \frac{1}{2}\ \frac{1}{4}\ 0\ 0\ \frac{1}{4}\ \frac{1}{4}\ 0\ 0\ \frac{1}{4}\ \frac{1}{2}\ \frac{1}{4}\ \frac{1}{4}\ \frac{1}{2}\right)^{T}
$$

The n by n matrix A arises in the solution of Laplace's equation in a unit-square, using a five-point formula with a 6 by 6 discretisation, with unity on the boundaries.

```
The drop tolerance, DROPTL, is taken as 0.1 and the density
factor, DENSW, is taken as 0.8. The value IFAIL = 111 is used so
that advisory and error messages will be printed, but soft
failure would occur if IFAIL were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution
from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of routines
F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F04 -- Simultaneous Linear Equations                        F04MBF
        F04MBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F04MBF solves a system of real sparse symmetric linear equations
using a Lanczos algorithm.

2. Specification

       SUBROUTINE F04MBF (N, B, X, APROD, MSOLVE, PRECON, SHIFT,
      1                   RTOL, ITNLIM, MSGLVL, ITN, ANORM,
      2                   ACOND, RNORM, XNORM, WORK, RWORK,
      3                   LRWORK, IWORK, LIWORK, INFORM, IFAIL)
       INTEGER           N, ITNLIM, MSGLVL, ITN, LRWORK, IWORK
      1                  (LIWORK), LIWORK, INFORM, IFAIL
       DOUBLE PRECISION  B(N), X(N), SHIFT, RTOL, ANORM, ACOND,
      1                  RNORM, XNORM, WORK(N,5), RWORK(LRWORK)
       LOGICAL           PRECON
       EXTERNAL          APROD, MSOLVE

3. Description

F04MBF solves the system of linear equations

                     (A-(lambda)I)x=b                        (3.1)

where A is an n by n sparse symmetric matrix and (lambda) is a
```

scalar, which is of course zero if the solution of the equations

$$Ax=b$$

is required. It should be noted that neither A nor (A-(lambda)I) need be positive-definite.

(lambda) is supplied as the parameter SHIFT, and allows F04MBF to be used for finding eigenvectors of A in methods such as Rayleigh quotient iteration (see for example Lewis [1]), in which case (lambda) will be an approximation to an eigenvalue of A and b an approximation to an eigenvector of A.

The routine also provides an option to allow pre-conditioning and this will often reduce the number of iterations required by F04MBF.

F04MBF is based upon algorithm SYMMLQ (see Paige and Saunders [2]) and solves the equations by an algorithm based upon the

Lanczos process. Details of the method are given in Paige and Saunders [2]. The routine does not require A explicitly, but A is specified via a user-supplied routine APROD which, given an n element vector c, must return the vector z given by

$$z=Ac.$$

The pre-conditioning option is based on the following reasoning. If A can be expressed in the form

$$A=I+B$$

where B is of rank (rho), then the Lanczos process converges (in exact arithmetic) in at most (rho) iterations. If more generally A can be expressed in the form

$$A=M+C$$

where M is symmetric positive-definite and C has rank (rho), then

$$M^{-(1/2)}\,A\,M^{-(1/2)} = I + M^{-(1/2)}\,C\,M^{-(1/2)}$$

and $M^{-(1/2)}\,A\,M^{-(1/2)}$ also has rank (rho), and the Lanczos process applied to $M^{-(1/2)}\,A\,M^{-(1/2)}$ would again converge in at most (rho) iterations. On a computer, the number of iterations may be greater than (rho), but the Lanczos process may still be expected

to converge rapidly. F04MBF does not require M AM to be formed explicitly, but implicitly solves the equations

```
 -(1/2)             -(1/2)   -(1/2)       1/2
M      (A-(lambda)I)M      y=M      b , y=M   x        (3.2)
```

with the user being required to supply a routine MSOLVE to solve the equations

```
                          Mz=c.                          (3.3)
```

For the pre-conditioning option to be effective, it is desirable that equations (3.3) can be solved efficiently. The example program in Section 9 illustrates the use of this option.

If we let r denote the residual vector

```
                     r=b-(A-(lambda)I)x
```

corresponding to an iterate x, then, when pre-conditioning has not been requested, the iterative procedure is terminated if it is estimated that

```
           ||r||<=tol.||A-(lambda)I||.||x||,              (3.4)
```

where tol is a user-supplied tolerance, ||r|| denotes the Euclidean length of the vector r and ||A|| denotes the Frobenius (Euclidean) norm of the matrix A. When pre-conditioning has been requested, the iterative procedure is terminated if it is estimated that

```
   -(1/2)            -(1/2)             -(1/2)      1/2
||M      r||<=tol.||M      (A-(lambda)I)M     ||.||M   x||. (3.5)
```

Note that

```
 -(1/2)    -(1/2)    -(1/2)              -(1/2)  1/2
M      r=(M      b)-M      (A-(lambda)I)M      (M   x)
```

```
    -(1/2)
```

so that M r is the residual vector corresponding to equation (3.2). The routine will also terminate if it is estimated that

```
         ||A-(lambda)I||.||x||>=||b||/(epsilon),          (3.6)
```

where (epsilon) is the machine precision, when pre-conditioning has not been requested; or if it is estimated that

```
   -(1/2)             -(1/2)      1/2        -(1/2)
```

```
||M      (A-(lambda)I)M      ||.||M   x||>=||M      b||/(epsilon)
                                                              (3.7)
```

when pre-conditioning has been requested. If (3.6) is satisfied then x is almost certainly an eigenvector of A corresponding to the eigenvalue (lambda). If (lambda) was set to 0 (for the solution of Ax=b), then this condition simply means that A is effectively singular.

4. References

[1] Lewis J G (1977) Algorithms for sparse matrix eigenvalue problems. Technical Report STAN-CS-77-595. Computer Science Department, Stanford University.

[2] Paige C C and Saunders M A (1975) Solution of Sparse Indefinite Systems of Linear Equations. SIAM J. Numer. Anal. 12 617--629.

5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix A. Constraint: N >= 1.

2: B(N) -- DOUBLE PRECISION array Input
On entry: the right-hand side vector b.

3: X(N) -- DOUBLE PRECISION array Output
On exit: the solution vector x.

4: APROD -- SUBROUTINE, supplied by the user.
External Procedure
APROD must return the vector y=Ax for a given vector x.

Its specification is:

```
      SUBROUTINE APROD (IFLAG, N, X, Y, RWORK, LRWORK,
     1                  IWORK,LIWORK)
      INTEGER          IFLAG, N, LRWORK, LIWORK, IWORK
     1                 (LIWORK)
      DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)
```

1: IFLAG -- INTEGER Input/Output
On entry: IFLAG is always non-negative. On exit: IFLAG may be used as a flag to indicate a failure in the computation of Ax. If IFLAG is negative on exit from APROD, F04MBF will exit immediately with IFAIL set to IFLAG.

```
 2:  N -- INTEGER                                          Input
     On entry: n, the order of the matrix A.

 3:  X(N) -- DOUBLE PRECISION array                        Input
     On entry: the vector x for which Ax is required.

 4:  Y(N) -- DOUBLE PRECISION array                       Output
     On exit: the vector y=Ax.

 5:  RWORK(LRWORK) -- DOUBLE PRECISION array  User Workspace

 6:  LRWORK -- INTEGER                                     Input

 7:  IWORK(LIWORK) -- INTEGER array           User Workspace

 8:  LIWORK -- INTEGER                                     Input
     APROD is called from F04MBF with the parameters RWORK,
     LRWORK, IWORK and LIWORK as supplied to F04MBF. The
     user is free to use the arrays RWORK and IWORK to
     supply information to APROD and MSOLVE as an
     alternative to using COMMON.
APROD must be declared as EXTERNAL in the (sub)program
from which F04MBF is called. Parameters denoted as
Input must not be changed by this procedure.

5:  MSOLVE -- SUBROUTINE, supplied by the user.
                                              External Procedure
MSOLVE is only referenced when PRECON is supplied as .TRUE..
When PRECON is supplied as .FALSE., then F04MBF may be
called with APROD as the actual argument for MSOLVE. When
PRECON is supplied as .TRUE., then MSOLVE must return the
solution y of the equations My=x for a given vector x, where
M must be symmetric positive-definite.

Its specification is:

      SUBROUTINE MSOLVE (IFLAG, N, X, Y, RWORK,
     1                   LRWORK, IWORK,LIWORK)
      INTEGER           IFLAG, N, LRWORK, LIWORK, IWORK
     1                  (LIWORK)
      DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)

 1:  IFLAG -- INTEGER                             Input/Output
     On entry: IFLAG is always non-negative. On exit: IFLAG
     may be used as a flag to indicate a failure in the
     solution of My=x.

     If IFLAG is negative on exit from MSOLVE, F04MBF will
     exit immediately with IFAIL set to IFLAG.
```

```
     2:  N -- INTEGER                                        Input
         On entry: n, the order of the matrix M.

     3:  X(N) -- DOUBLE PRECISION array                      Input
         On entry: the vector x for which the equations My=x are
         to be solved.

     4:  Y(N) -- DOUBLE PRECISION array                     Output
         On exit: the solution to the equations My=x.

     5:  RWORK(LRWORK) -- DOUBLE PRECISION array  User Workspace

     6:  LRWORK -- INTEGER                                   Input

     7:  IWORK(LIWORK) -- INTEGER array           User Workspace

     8:  LIWORK -- INTEGER                                   Input
         MSOLVE is called from F04MBF with the parameters RWORK,
         LRWORK, IWORK and LIWORK as supplied to F04MBF. The
         user is free to use the arrays RWORK and IWORK to
         supply information to APROD and MSOLVE as an
         alternative to using COMMON.
    MSOLVE must be declared as EXTERNAL in the (sub)program
    from which F04MBF is called. Parameters denoted as
    Input must not be changed by this procedure.

 6:  PRECON -- LOGICAL                                       Input
     On entry: PRECON specifies whether or not pre-conditioning
     is required. If PRECON = .TRUE., then pre-conditioning will
     be invoked and MSOLVE will be referenced by F04MBF; if
     PRECON = .FALSE., then MSOLVE is not referenced.

 7:  SHIFT -- DOUBLE PRECISION                               Input
     On entry: the value of (lambda). If the equations Ax=b are
     to be solved, then SHIFT must be supplied as zero.

 8:  RTOL -- DOUBLE PRECISION                                Input
     On entry: the tolerance for convergence, tol, of equation
     (3.4). RTOL should not normally be less than (epsilon),
     where (epsilon) is the machine precision.

 9:  ITNLIM -- INTEGER                                       Input
     On entry: an upper limit on the number of iterations. If
     ITNLIM <= 0, then the value N is used in place of ITNLIM.

10:  MSGLVL -- INTEGER                                       Input
     On entry: the level of printing from F04MBF. If MSGLVL <= 0,
     then no printing occurs, but otherwise messages will be
     output on the advisory message channel (see X04ABF). A
     description of the printed output is given in Section 5.1
```

```
     below. The level of printing is determined as follows:
     MSGLVL <= 0
           No printing.

     MSGLVL = 1
           A brief summary is printed just prior to return from
           F04MBF.

     MSGLVL >= 2
           A summary line is printed periodically to monitor the
           progress of F04MBF, together with a brief summary just
           prior to return from F04MBF.

11:  ITN -- INTEGER                                          Output
     On exit: the number of iterations performed.

12:  ANORM -- DOUBLE PRECISION                               Output
     On exit: an estimate of ||A-(lambda)I|| when PRECON =
                    -(1/2)              -(1/2)
     .FALSE., and ||M      (A-(lambda)I)M      || when PRECON =
     .TRUE..

13:  ACOND -- DOUBLE PRECISION                               Output
     On exit: an estimate of the condition number of (A-
     (lambda)I) when PRECON = .FALSE., and of
      -(1/2)              -(1/2)
     M      (A-(lambda)I)M       when PRECON = .TRUE.. This will
     usually be a substantial under-estimate.

14:  RNORM -- DOUBLE PRECISION                               Output
     On exit: ||r||, where r=b-(A-(lambda)I)x and x is the
     solution returned in X.

15:  XNORM -- DOUBLE PRECISION                               Output
     On exit: ||x||, where x is the solution returned in X.

16:  WORK(5*N) -- DOUBLE PRECISION array                   Workspace

17:  RWORK(LRWORK) -- DOUBLE PRECISION array       User Workspace
     RWORK is not used by F04MBF, but is passed directly to
     routines APROD and MSOLVE and may be used to pass
     information to these routines.

18:  LRWORK -- INTEGER                                        Input
     On entry: the length of the array RWORK as declared in the
     (sub)program from which F04MBF is called. Constraint: LRWORK
     >= 1.

19:  IWORK(LIWORK) -- INTEGER array                User Workspace
     IWORK is not used by F04MBF, but is passed directly to
```

routines APROD and MSOLVE and may be used to pass information to these routines.

20: LIWORK -- INTEGER Input
On entry: the length of the array IWORK as declared in the (sub)program from which F04MBF is called. Constraint: LIWORK >= 1.

21: INFORM -- INTEGER Output
On exit: the reason for termination of F04MBF as follows:
INFORM = 0
The right-hand side vector b=0 so that the exact solution is x=0. No iterations are performed in this case.

INFORM = 1
The termination criterion of equation (3.4) has been satisfied with tol as the value supplied in RTOL.

INFORM = 2
The termination criterion of equation (3.4) has been satisfied with tol equal to (epsilon), where (epsilon) is the machine precision. The value supplied in RTOL must have been less than (epsilon) and was too small for the machine.

INFORM = 3
The termination criterion of equation (3.5) has been satisfied so that X is almost certainly an eigenvector of A corresponding to the eigenvalue SHIFT.
The values INFORM = 4 and INFORM = 5 correspond to failure with IFAIL = 3 or IFAIL = 2 respectively (see Section 6) and when IFAIL is negative, INFORM will be set to the same negative value.

22: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

5.1. Description of the Printed Output

When MSGLVL > 0, then F04MBF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF).

The following notation is used in the output.

Output	Meaning
RBAR	$M^{-(1/2)}(b-(A-(\text{lambda})I)x)=r$
ABAR	$M^{-(1/2)}(A-(\text{lambda})I)M^{-(1/2)}=A$
Y	$M^{1/2}x$
R	b-(A-(lambda)I)x
NORM(A)	||A||

Of course, when pre-conditioning has not been requested then the first three reduce to (b-(A-(lambda)I)x), (A-(lambda)I) and x respectively. When MSGLVL >= 2 then some initial information is printed and the following notation is used.

Output	Meaning
BETA1	$(b^T M^{-1} b)^{1/2} == (\text{beta})_1$
ALFA1	$(1/(\text{beta})_1)(M^{-1/2} b)(M^{-1/2} AM^{-1/2})(M^{-1/2} b) == (\text{alpha})_1$

and a summary line is printed periodically giving the following information:

Output	Meaning
ITN	Iteration number, k.
X1(LQ)	The first element of the vector x_k^L, where x_k^L is the current iterate. See Paige and Saunders [2] for details.
X1(CG)	The first element of the vector x_k^C, where x_k^C is the vector that would be obtained by conjugate gradients. See Paige and Saunders [2] for details.

```
   NORM(RBAR) ||r||, where r is as defined above and x is either
                L     C
              x  or x  depending upon which is the best current
               k     k
              approximation to the solution. (See LQ/CG below).

   NORM(T)    The value ||T ||, where T  is the tridiagonal
                               k          k
              matrix of the Lanczos process. This increases


              monotonically and is a lower bound on ||A||.

   COND(L)    A monotonically increasing lower bound on the
                                                    -1
              condition number of A, ||A||||(A)  ||.

                              L
   LQ/CG      L is printed if x  is the best current
                              k
              approximation to the solution and C is printed
              otherwise.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL< 0
     A negative value of IFAIL indicates an exit from F04MBF
     because the user has set IFLAG negative in APROD or MSOLVE.
     The value of IFAIL will be the same as the user's setting of
     IFLAG.

IFAIL= 1
     On entry N < 1,

     or       LRWORK < 1,

     or       LIWORK < 1.

IFAIL= 2
     The pre-conditioning matrix M does not appear to be
     positive-definite. The user should check that MSOLVE is
     working correctly.

IFAIL= 3
     The limit on the number of iterations has been reached. If
```

```
     IFAIL = 1 on entry then the latest approximation to the
     solution is returned in X and the values ANORM, ACOND, RNORM
     and XNORM are also returned.

The value of INFORM contains additional information about the
termination of the routine and users must examine INFORM to judge
whether the routine has performed successfully for the problem in
hand. In particular INFORM = 3 denotes that the matrix A-
(lambda)I is effectively singular: if the purpose of calling
F04MBF is to solve a system of equations Ax=b, then this
condition must be regarded as a failure, but if the purpose is to
compute an eigenvector, this result would be very satisfactory.

7. Accuracy

The computed solution x will satisfy the equation

                        r=b-(A-(lambda)I)x

where the value ||r|| is as returned in the parameter RNORM.

8. Further Comments

The time taken by the routine is likely to be principally
determined by the time taken in APROD and, when pre-conditioning
has been requested, in MSOLVE. Each of these routines is called
once every iteration.

The time taken by the remaining operations in F04MBF is
approximately proportional to n.

9. Example

To solve the 10 equations Ax=b given by

                  (2 1 0 0 0 0 0 0 0 3)      (6)
                  (1 2 1 0 0 0 0 0 0 0)      (4)
                  (0 1 2 1 0 0 0 0 0 0)      (4)
                  (0 0 1 2 1 0 0 0 0 0)      (4)
                  (0 0 0 1 2 1 0 0 0 0)      (4)
                A=(0 0 0 0 1 2 1 0 0 0),   b=(4).
                  (0 0 0 0 0 1 2 1 0 0)      (4)
                  (0 0 0 0 0 0 1 2 1 0)      (4)
                  (0 0 0 0 0 0 0 1 2 1)      (4)
                  (3 0 0 0 0 0 0 0 1 2)      (6)

The tridiagonal part of A is positive-definite and such
tridiagonal equations can be solved efficiently by F04FAF. The
form of A suggests that this tridiagonal part is a good candidate
for the pre-conditioning matrix M and so we illustrate the use of
```

F04MBF by pre-conditioning with the 10 by 10 matrix

$$M=\begin{pmatrix} 2 & 1 & 0 & \ldots & 0 \\ 1 & 2 & 1 & \ldots & 0 \\ 0 & 1 & 2 & \ldots & 0 \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & . \\ 0 & 0 & 0 & \ldots & 2 \end{pmatrix}.$$

Since A-M has only 2 non-zero elements and is obviously of rank 2, we can expect F04MBF to converge very quickly in this example. Of course, in practical problems we shall not usually be able to make such a good choice of M.

The example sets the tolerance RTOL = 10^{-5}.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F04 -- Simultaneous Linear Equations F04MCF
F04MCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04MCF computes the approximate solution of a system of real linear equations with multiple right-hand sides, AX=B, where A is a symmetric positive-definite variable-bandwidth matrix, which has previously been factorized by F01MCF. Related systems may also be solved.

2. Specification

```
      SUBROUTINE F04MCF (N, AL, LAL, D, NROW, IR, B, NRB,
     1                   ISELCT, X, NRX, IFAIL)
      INTEGER           N, LAL, NROW(N), IR, NRB, ISELCT, NRX,
     1                  IFAIL
      DOUBLE PRECISION AL(LAL), D(N), B(NRB,IR), X(NRX,IR)
```

3. Description

```
The normal use of this routine is the solution of the systems
AX=B, following a call of F01MCF to determine the Cholesky
                    T
factorization A=LDL  of the symmetric positive-definite variable-
bandwidth matrix A.

However, the routine may be used to solve any one of the
following systems of linear algebraic equations:

        T
(1)  LDL X = B (usual system),

(2)  LDX   = B (lower triangular system),

       T
(3)  DL X  = B (upper triangular system),

       T
(4)  LL X  = B

(5)  LX    = B (unit lower triangular system),

      T
(6)  L X   = B (unit upper triangular system).

L denotes a unit lower triangular variable-bandwidth matrix of
order n, D a diagonal matrix of order n, and B a set of right-
hand sides.

The matrix L is represented by the elements lying within its
envelope i.e., between the first non-zero of each row and the
diagonal (see Section 9 for an example). The width NROW(i) of the
ith row is the number of elements between the first non-zero
element and the element on the diagonal inclusive.

4. References

[1]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  N -- INTEGER                                          Input
     On entry: n, the order of the matrix L. Constraint: N >= 1.

 2:  AL(LAL) -- DOUBLE PRECISION array                     Input
     On entry: the elements within the envelope of the lower
     triangular matrix L, taken in row by row order, as returned
     by F01MCF. The unit diagonal elements of L must be stored
```

```
    explicitly.

3:  LAL -- INTEGER                                          Input
    On entry:
    the dimension of the array AL as declared in the
    (sub)program from which F04MCF is called.
    Constraint: LAL >= NROW(1) + NROW(2) +... + NROW(n).

4:  D(N) -- DOUBLE PRECISION array                          Input
    On entry: the diagonal elements of the diagonal matrix D. D
    is not referenced if ISELCT >= 4.

5:  NROW(N) -- INTEGER array                                Input
    On entry: NROW(i) must contain the width of row i of L,
    i.e.,the number of elements between the first (leftmost)
    non-zero element and the element on the diagonal, inclusive.
    Constraint: 1 <= NROW(i)<=i.

6:  IR -- INTEGER                                           Input
    On entry: r, the number of right-hand sides. Constraint: IR
    >= 1.

7:  B(NRB,IR) -- DOUBLE PRECISION array                     Input
    On entry: the n by r right-hand side matrix B. See also
    Section 8.

8:  NRB -- INTEGER                                          Input
    On entry:
    the first dimension of the array B as declared in the
    (sub)program from which F04MCF is called.
    Constraint: NRB >= N.

9:  ISELCT -- INTEGER                                       Input
    On entry: ISELCT must specify the type of system to be
    solved, as follows:

                          T
    ISELCT = 1:  solve LDL X = B,

    ISELCT = 2:  solve LDX = B,

                         T
    ISELCT = 3:  solve DL X = B,

                         T
    ISELCT = 4:  solve LL X = B,

    ISELCT = 5:  solve LX = B,

                        T
```

```
     ISELCT = 6:  solve L X = B.

10:  X(NRX,IR) -- DOUBLE PRECISION array                    Output
     On exit: the n by r solution matrix X. See also Section 8.

11:  NRX -- INTEGER                                          Input
     On entry:
     the first dimension of the array X as declared in the
     (sub)program from which F04MCF is called.
     Constraint: NRX >= N.

12:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry N < 1,

     or       for some i, NROW(i)<1 or NROW(i) > i,

     or       LAL < NROW(1) + NROW(2) +... + NROW(N).

IFAIL= 2

     On entry IR < 1,

     or       NRB < N,

     or       NRX < N.

IFAIL= 3
     On entry ISELCT < 1,

     or       ISELCT > 6.

IFAIL= 4
     The diagonal matrix D is singular, i.e., at least one of the
     elements of D is zero. This can only occur if ISELCT <= 3.

IFAIL= 5
     At least one of the diagonal elements of L is not equal to
     unity.
```

```
7. Accuracy

The usual backward error analysis of the solution of triangular
system applies: each computed solution vector is exact for
slightly perturbed matrices L and D, as appropriate (cf.
Wilkinson and Reinsch [1] pp 25--27, 54--55).

8. Further Comments

The time taken by the routine is approximately proportional to
pr, where

                p=NROW(1)+NROW(2)+...+NROW(n).

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for the parameters B and X, in which case the
solution matrix will overwrite the right-hand side matrix.
However this is not standard Fortran 77 and may not work in all
implementations.

9. Example

To solve the system of equations AX=B, where

                      (1  2  0  0  5  0)
                      (2  5  3  0 14  0)
                      (0  3 13  0 18  0)
                    A=(0  0  0 16  8 24)
                      (5 14 18  8 55 17)
                      (0  0  0 24 17 77)

and

                           ( 6 -10)
                           (15 -21)
                           (11  -3)
                         B=( 0  24)
                           (51 -39)
                           (46  67)

Here A is symmetric and positive-definite and must first be
factorized by F01MCF.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
F04 -- Simultaneous Linear Equations                        F04QAF
        F04QAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F04QAF solves sparse unsymmetric equations, sparse linear least-
squares problems and sparse damped linear least-squares problems,
using a Lanczos algorithm.

2. Specification

       SUBROUTINE F04QAF (M, N, B, X, SE, APROD, DAMP, ATOL,
      1                   BTOL, CONLIM, ITNLIM, MSGLVL, ITN,
      2                   ANORM, ACOND, RNORM, ARNORM, XNORM,
      3                   WORK, RWORK, LRWORK, IWORK, LIWORK,
      4                   INFORM, IFAIL)
       INTEGER           M, N, ITNLIM, MSGLVL, ITN, LRWORK, IWORK
      1                  (LIWORK), LIWORK, INFORM, IFAIL
       DOUBLE PRECISION B(M), X(N), SE(N), DAMP, ATOL, BTOL,
      1                  CONLIM, ANORM, ACOND, RNORM, ARNORM,
      2                  XNORM, WORK(N,2), RWORK(LRWORK)
       EXTERNAL          APROD

3. Description

F04QAF can be used to solve a system of linear equations

                              Ax=b                                  (3.1)

where A is an n by n sparse unsymmetric matrix, or can be used to
solve linear least-squares problems, so that F04QAF minimizes the
value (rho) given by

                    (rho)=||r||,  r=b-Ax                          (3.2)

where A is an m by n sparse matrix and ||r|| denotes the
                                      2  T
Euclidean length of r so that ||r|| =r r. A damping parameter,
(lambda), may be included in the least-squares problem in which
case F04QAF minimizes the value (rho) given by

                    2      2         2     2
               (rho) =||r|| +(lambda) ||x||                       (3.3)
```

(lambda) is supplied as the parameter DAMP and should of course be zero if the solution to problems (3.1) or (3.2) is required. Minimizing (rho) in (3.3) is often called ridge regression.

F04QAF is based upon algorithm LSQR (see Paige and Saunders [1] and [2]) and solves the problems by an algorithm based upon the Lanczos process. Details of the method are given in [1]. The routine does not require A explicitly, but A is specified via a user-supplied routine APROD which must perform the operations (T y+Ax) and (x+A y) for a given n element vector x and m element vector y. A parameter to APROD specifies which of the two operations is required on a given entry.

The routine also returns estimates of the standard errors of the sample regression coefficients (x , for i=1,2,...,n) given by the i diagonal elements of the estimated variance-covariance matrix V. When problem (3.2) is being solved and A is of full rank, then V is given by

$$V=s^2 (A^T A)^{-1}, \quad s^2=(rho)^2/(m-n), \quad m>n$$

and when problem (3.3) is being solved then V is given by

$$V=s^2 (A^T A+(lambda)^2 I)^{-1}, \quad s^2=(rho)^2/m, \quad (lambda)/=0.$$

Let A denote the matrix

$$A=A, \quad (lambda)=0 \ ; \ A=\begin{pmatrix} A \\ (lambda)I \end{pmatrix}, \quad (lambda)/=0, \tag{3.4}$$

let r denote the residual vector

$$r=r^{(b)}, \quad (lambda)=0 \ ; \ r=(0)-Ax, \quad (lambda)/=0 \tag{3.5}$$

corresponding to an iterate x, so that (rho)=||r|| is the function being minimized, and let ||A|| denote the Frobenius (Euclidean) norm of A. Then the routine accepts x as a solution if it is estimated that one of the following two conditions is satisfied:

```
(rho)<=tol ||A||.||x||+tol ||b||                (3.6)
          1               2

       T
   ||A r||<=tol ||A||(rho)                      (3.7)
                1
```

where tol and tol are user-supplied tolerances which estimate
         1   2
the relative errors in A and b respectively. Condition (3.6) is appropriate for compatible problems where, in theory, we expect the residual to be zero and will be satisfied by an acceptable solution x to a compatible problem. Condition (3.7) is appropriate for incompatible systems where we do not expect the residual to be zero and is based upon the observation that, in theory,

$$A^T r = 0$$

when x is a solution to the least-squares problem, and so (3.7) will be satisfied by an acceptable solution x to a linear least-squares problem.

The routine also includes a test to prevent convergence to solutions, x, with unacceptably large elements. This can happen if A is nearly singular or is nearly rank deficient. If we let

the singular values of A be

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$$

then the condition number of A is defined as

$$\mathrm{cond}(A) = \sigma_1 / \sigma_k$$

where σ_k is the smallest non-zero singular value of A and

hence k is the rank of A. When k<n, then A is rank deficient, the least-squares solution is not unique and F04QAF will normally

converge to the minimal length solution. In practice A will not have exactly zero singular values, but may instead have small singular values that we wish to regard as zero.

The routine provides for this possibility by terminating if

$$\mathrm{cond}(A) \geq c_{\mathrm{lim}} \tag{3.8}$$

where c_{lim} is a user-supplied limit on the condition number of A. For problem (3.1) termination with this condition indicates that A is nearly singular and for problem (3.2) indicates that A is nearly rank deficient and so has near linear dependencies in its columns. In this case inspection of ||r||, $||A^T r||$ and ||x||, which are all returned by the routine, will indicate whether or not an acceptable solution has been found. Condition (3.8), perhaps in conjunction with (lambda)/=0, can be used to try and ' regularise' least-squares solutions. A full discussion of the stopping criteria is given in Section 6 of reference Paige and Saunders [1].

Introduction of a non-zero damping parameter (lambda) tends to reduce the size of the computed solution and to make its components less sensitive to changes in the data, and F04QAF is applicable when a value of (lambda) is known a priori. To have an

effect, (lambda) should normally be at least \/(epsilon)||A|| where (epsilon) is the machine precision. For further discussion see Paige and Saunders [2] and the references given there.

Whenever possible the matrix A should be scaled so that the relative errors in the elements of A are all of comparable size. Such a scaling helps to prevent the least-squares problem from being unnecessarily sensitive to data errors and will normally reduce the number of iterations required. At the very least, in the absence of better information, the columns of A should be scaled to have roughly equal column length.

```
4. References

[1]   Paige C C and Saunders M A (1982) LSQR: An Algorithm for
      Sparse Linear Equations and Sparse Least-squares. ACM Trans.
      Math. Softw. 8 43--71.

[2]   Paige C C and Saunders M A (1982) ALGORITHM 583 LSQR: Sparse
      Linear Equations and Least-squares Problems. ACM Trans.
      Math. Softw. 8 195--209.

5. Parameters

 1:  M -- INTEGER                                              Input
     On entry: m, the number of rows of the matrix A.
     Constraint: M >= 1.

 2:  N -- INTEGER                                              Input
     On entry: n, the number of columns of the matrix A.
     Constraint: N >= 1.

 3:  B(M) -- DOUBLE PRECISION array                    Input/Output
     On entry: the right-hand side vector b. On exit: the array
     is overwritten.

 4:  X(N) -- DOUBLE PRECISION array                           Output
     On exit: the solution vector x.

 5:  SE(N) -- DOUBLE PRECISION array                          Output
     On exit: the estimates of the standard errors of the
     components of x. Thus SE(i) contains an estimate of the
     element v    of the estimated variance-covariance matrix V.
              ii
     The estimates returned in SE will be the lower bounds on the
     actual estimated standard errors, but will usually have at
     least one correct figure.

 6:  APROD -- SUBROUTINE, supplied by the user.
                                              External Procedure
                                                           T
     APROD must perform the operations y:=y+Ax and x:=x+A y for
     given vectors x and y.

     Its specification is:

            SUBROUTINE APROD (MODE, M, N, X, Y, RWORK,
           1                  LRWORK, IWORK, LIWORK)
            INTEGER          MODE, M, N, LRWORK, LIWORK,
           1                 IWORK(LIWORK)
            DOUBLE PRECISION X(N), Y(M), RWORK(LRWORK)
```

```
 1:  MODE -- INTEGER                                Input/Output
     On entry: MODE specifies which operation is to be
     performed:
          If MODE = 1, then APROD must compute y+Ax.

                                                      T
          If MODE = 2, then APROD must compute x+A y.
     On exit: MODE may be used as a flag to indicate a
                                                    T
     failure in the computation of y+Ax or x+A y. If MODE is
     negative on exit from APROD, F04QAF will exit
     immediately with IFAIL set to MODE.

 2:  M -- INTEGER                                          Input
     On entry: m, the number of rows of A.

 3:  N -- INTEGER                                          Input
     On entry: n, the number of columns of A.

 4:  X(N) -- DOUBLE PRECISION array              Input/Output
     On entry: the vector x. On exit: if MODE = 1, X must be
     unchanged;

                                      T
     If MODE = 2, X must contain x+A y.

 5:  Y(M) -- DOUBLE PRECISION array              Input/Output
     On entry: the vector y. On exit: if MODE = 1, Y must
     contain y+Ax;

     If MODE = 2, Y must be unchanged.

 6:  RWORK(LRWORK) -- DOUBLE PRECISION array  User Workspace

 7:  LRWORK -- INTEGER                                     Input

 8:  IWORK(LIWORK) -- INTEGER array           User Workspace

 9:  LIWORK -- INTEGER                                     Input
     APROD is called from F04QAF with the parameters RWORK,
     LRWORK, IWORK and LIWORK as supplied to F04QAF. The
     user is free to use the arrays RWORK and IWORK to
     supply information to APROD as an alternative to using
     COMMON.
APROD must be declared as EXTERNAL in the (sub)program
from which F04QAF is called. Parameters denoted as
Input must not be changed by this procedure.

7:  DAMP -- DOUBLE PRECISION                               Input
```

On entry: the value (lambda). If either problem (3.1) or problem (3.2) is to be solved, then DAMP must be supplied as zero.

8: ATOL -- DOUBLE PRECISION Input
On entry: the tolerance, tol_1, of the convergence criteria (3.6) and (3.7); it should be an estimate of the largest relative error in the elements of A. For example, if the elements of A are correct to about 4 significant figures, then ATOL should be set to about $5*10^{-4}$. If ATOL is supplied as less than (epsilon), where (epsilon) is the machine precision, then the value (epsilon) is used in place of ATOL.

9: BTOL -- DOUBLE PRECISION Input
On entry: the tolerance, tol_2, of the convergence criterion (3.6); it should be an estimate of the largest relative error in the elements of B. For example, if the elements of B are correct to about 4 significant figures, then BTOL should be set to about $5*10^{-4}$. If BTOL is supplied as less than (epsilon), where (epsilon) is the machine precision, then the value (epsilon) is used in place of BTOL.

10: CONLIM -- DOUBLE PRECISION Input
On entry: the value c_{lim} of equation (3.8); it should be an upper limit on the condition number of A. CONLIM should not normally be chosen much larger than 1.0/ATOL. If CONLIM is supplied as zero then the value 1.0/(epsilon), where (epsilon) is the machine precision, is used in place of CONLIM.

11: ITNLIM -- INTEGER Input
On entry: an upper limit on the number of iterations. If ITNLIM <= 0, then the value N is used in place of ITNLIM, but for ill-conditioned problems a higher value of ITNLIM is likely to be necessary.

12: MSGLVL -- INTEGER Input
On entry: the level of printing from F04QAF. If MSGLVL <= 0, then no printing occurs, but otherwise messages will be output on the advisory message channel (see X04ABF). A description of the printed output is given in Section 5.2 below. The level of printing is determined as follows:

MSGLVL <= 0
No printing.

MSGLVL = 1
A brief summary is printed just prior to return from F04QAF.

MSGLVL >= 2
A summary line is printed periodically to monitor the progress of F04QAF, together with a brief summary just prior to return from F04QAF.

13: ITN -- INTEGER Output
On exit: the number of iterations performed.

14: ANORM -- DOUBLE PRECISION Output

On exit: an estimate of ||A|| for the matrix A of equation (3.4).

15: ACOND -- DOUBLE PRECISION Output

On exit: an estimate of cond(A) which is a lower bound.

16: RNORM -- DOUBLE PRECISION Output

On exit: an estimate of ||r|| for the residual, r, of equation (3.5) corresponding to the solution x returned in

X. Note that ||r|| is the function being minimized.

17: ARNORM -- DOUBLE PRECISION Output
T
On exit: an estimate of the ||A r|| corresponding to the solution x returned in X.

18: XNORM -- DOUBLE PRECISION Output
On exit: an estimate of ||x|| for the solution x returned in X.

19: WORK(2*N) -- DOUBLE PRECISION array Workspace

20: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace
RWORK is not used by F04QAF, but is passed directly to routine APROD and may be used to pass information to that routine.

```
21:  LRWORK -- INTEGER                                          Input
     On entry: the length of the array RWORK as declared in the
     (sub)program from which F04QAF is called. Constraint: LRWORK
     >= 1.

22:  IWORK(LIWORK) -- INTEGER array                  User Workspace
     IWORK is not used by F04QAF, but is passed directly to
     routine APROD and may be used to pass information to that
     routine.

23:  LIWORK -- INTEGER                                          Input
     On entry: the length of the array IWORK as declared in the
     (sub)program from which F04QAF is called. Constraint: LIWORK
     >= 1.

24:  INFORM -- INTEGER                                         Output
     On exit: the reason for termination of F04QAF as follows:
     INFORM = 0
           The exact solution is x=0. No iterations are performed
           in this case.

     INFORM = 1
           The termination criterion of equation (3.6) has been
           satisfied with tol  and tol  as the values supplied in
                             1        2
           ATOL and BTOL respectively.

     INFORM = 2
           The termination criterion of equation (3.7) has been
           satisfied with tol  as the value supplied in ATOL.
                             1

     INFORM = 3
           The termination criterion of equation (3.6) has been
           satisfied with tol  and/or tol  as the value (epsilon)
                             1           2
           , where (epsilon) is the machine precision. One or
           both of the values supplied in ATOL and BTOL must have
           been less than (epsilon) and was too small for this
           machine.

     INFORM = 4
           The termination criterion of equation (3.7) has been
           satisfied with tol  as the value (epsilon), where
                             1
           (epsilon) is the machine precision. The value supplied
           in ATOL must have been less than (epsilon) and was too
           small for this machine.
     The values INFORM = 5, INFORM = 6 and INFORM = 7 correspond
```

to failure with IFAIL = 2, IFAIL = 3 and IFAIL = 4 respectively (see Section 6) and when IFAIL is negative INFORM will be set to the same negative value.

25: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

5.1. Description of the printed output

When MSGLVL > 0, then F04QAF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF).

When MSGLVL >= 2 then a summary line is printed periodically giving the following information:

Output	Meaning
ITN	Iteration number, k.
X(1)	The first element of the current iterate x_k.
FUNCTION	The current value of the function, (rho), being minimized.
COMPAT	An estimate of $\|\|r_k\|\|/\|\|b\|\|$, where r_k is the residual corresponding to x_k. This value should converge to zero (in theory) if and only if the problem is compatible. COMPAT decreases monotonically.
INCOMPAT	An estimate of $\|\|A^T r_k\|\|/(\|\|A\|\|\|\|r_k\|\|)$ which should converge to zero if and only if at the solution (rho) is non-zero. INCOMPAT is not usually monotonic.

```
   NRM(ABAR)    A monotonically increasing estimate of ||A||.

   COND(ABAR)   A monotonically increasing estimate of the


                condition number cond(A).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL< 0
     A negative value of IFAIL indicates an exit from F04QAF
     because the user has set MODE negative in APROD. The value
     of IFAIL will be the same as the user's setting of MODE.

IFAIL= 1
     On entry M < 1,

     or       N < 1,

     or       LRWORK < 1,

     or       LIWORK < 1.

IFAIL= 2
     The condition of equation (3.8) has been satisfied for the
     value of c    supplied in CONLIM. If this failure is
               lim
     unexpected the user should check that APROD is working
     correctly. Although conditions (3.6) or (3.7) have not been
     satisfied, the values returned in RNORM, ARNORM and XNORM
     may nevertheless indicate that an acceptable solution has
     been reached.

IFAIL= 3
     The conditions of equation (3.8) has been satisified for the
     value c   =1.0/(epsilon), where (epsilon) is the machine
            lim


     precision. The matrix A is nearly singular or rank deficient
     and the problem is too ill-conditioned for this machine. If
     this failure is unexpected, the user should check that APROD
     is working correctly.

IFAIL= 4
     The limit on the number of iterations has been reached. The
     number of iterations required by F04QAF and the condition of
```

the matrix A can depend strongly on the scaling of the problem. Poor scaling of the rows and columns of A should be avoided whenever possible.

7. Accuracy

When the problem is compatible, the computed solution x will satisfy the equation

$$r=b-Ax,$$

where an estimate of ||r|| is returned in the parameter RNORM. When the problem is incompatible, the computed solution x will satisfy the equation

$$A^T r=e,$$

where an estimate of ||e|| is returned in the parameter ARNORM. See also Section 6.2 of Paige and Saunders [1].

8. Further Comments

The time taken by the routine is likely to be principally determined by the time taken in APROD, which is called twice on each iteration, once with MODE = 1 and once with MODE = 2. The time taken per iteration by the remaining operations in F04QAF is approximately proportional to max(m,n).

The Lanczos process will usually converge more quickly if A is pre-conditioned by a non-singular matrix M that approximates A in some sense and is also chosen so that equations of the form My=c can efficiently be solved for y. Some discussion of pre-conditioning in the context of symmetric matrices is given in Section 3 of the document for F04MBF. In the context of F04QAF, problem (3.1) is equivalent to

$$(AM^{-1})y=b, \quad Mx=y$$

and problem (3.2) is equivalent to minimizing

$$(rho)=||r||, \quad r=b-(AM^{-1})y, \quad Mx=y.$$

Note that the normal matrix $(AM^{-1})^T (AM^{-1})=M^{-T}(A^T A)M^{-1}$ so that the pre-conditioning AM^{-1} is equivalent to the pre-conditioning

```
 -T  T   -1                        T
M  (A A)M   of the normal matrix A A.

Pre-conditioning can be incorporated into F04QAF simply by coding
                                  -1         -T T
the routine APROD to compute y+AM  x and x+M  A y in place of
           T
y+Ax and x+A y respectively, and then solving the equations Mx=y
                                  -1
for x on return from F04QAF. y+AM  x should be computed by
                                                      -T T
solving Mz=x for z and then computing y+Az, and x+M  A y should
                        T   T
be computed by solving M z=A y for z and then forming x+z.

9. Example

To solve the linear least-squares problem

                 minimize (rho)=||r||,  r=b-Ax

where A is the 13 by 12 matrix and b is the 13 element vector
given by

             ( 1  0  0 -1  0  0  0  0  0  0  0  0)
             ( 0  1  0  0 -1  0  0  0  0  0  0  0)
             ( 0  0  1 -1  0  0  0  0  0  0  0  0)
             (-1  0 -1  4 -1  0  0 -1  0  0  0  0)
             ( 0 -1  0 -1  4 -1  0  0 -1  0  0  0)
             ( 0  0  0  0 -1  1  0  0  0  0  0  0)
           A=( 0  0  0  0  0  0  1 -1  0  0  0  0),
             ( 0  0  0 -1  0  0 -1  4 -1  0 -1  0)
             ( 0  0  0  0 -1  0  0 -1  4 -1  0 -1)
             ( 0  0  0  0  0  0  0  0 -1  1  0  0)
             ( 0  0  0  0  0  0  0 -1  0  0  1  0)
             ( 0  0  0  0  0  0  0  0 -1  0  0  1)
             ( 1  1  1  0  0  1  1  0  0  1  1  1)

                              ( 0  )
                              ( 0  )
                              ( 0  )
                              ( 1  )
                              ( 1  )
                             2( 0  )
                        b=-h ( 0  )
                              ( 1  )
                              ( 1  )
                              ( 0  )
                              ( 0  )
                              ( 0  )
```

```
                                  (  -3)
                                  (-h  )
```

with h=0.1.

Such a problem can arise by considering the Neumann problem on a rectangle

```
                    (delta)u
                    --------=0
                    (delta)n

     (delta)u            2           (delta)u      /
     --------=0   (nabla) u=g(x,y)   --------=0    |u=1
     (delta)n                        (delta)n      /
                                                   c

                    (delta)u
                    --------=0
                    (delta)n
```

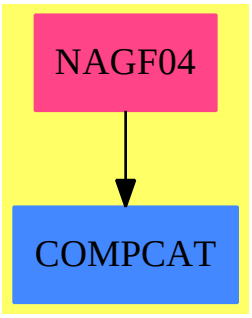
where C is the boundary of the rectangle, and discretising as illustrated below with the square mesh

Please see figure in printed Reference Manual

The 12 by 12 symmetric part of A represents the difference equations and the final row comes from the normalising condition. The example program has g(x,y)=1 at all the internal mesh points, but apart from this is written in a general manner so that the number of rows (NROWS) and columns (NCOLS) in the grid can readily be altered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

15.3.1 NagLinearEquationSolvingPackage (NAGF04)



Exports:

f04adf	f04arf	f04asf	f04atf	f04axf
f04faf	f04jgf	f04maf	f04mbf	f04mcf
f04qaf				

— package NAGF04 NagLinearEquationSolvingPackage —

```
)abbrev package NAGF04 NagLinearEquationSolvingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:31 1994
++ Description:
++ This package uses the NAG Library to solve the matrix equation \br
++ \tab{5}\axiom{AX=B}, where \axiom{B}\br
++ may be a single vector or a matrix of multiple right-hand sides.
++ The matrix \axiom{A} may be real, complex, symmetric, Hermitian positive-
++ definite, or sparse. It may also be rectangular, in which case a
++ least-squares solution is obtained.

NagLinearEquationSolvingPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    f04adf : (Integer,Matrix Complex DoubleFloat,Integer,Integer,_
        Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
     ++ f04adf(ia,b,ib,n,m,ic,a,ifail)
     ++ calculates the approximate solution of a set of complex
     ++ linear equations with multiple right-hand sides, using an LU
     ++ factorization with partial pivoting.
     ++ See \downlink{Manual Page}{manpageXXf04adf}.
    f04arf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
        Integer) -> Result
     ++ f04arf(ia,b,n,a,ifail)
     ++ calculates the approximate solution of a set of real
```

```
 ++ linear equations with a single right-hand side, using an LU
 ++ factorization with partial pivoting.
 ++ See \downlink{Manual Page}{manpageXXf04arf}.
f04asf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
    Integer) -> Result
 ++ f04asf(ia,b,n,a,ifail)
 ++ calculates the accurate solution of a set of real
 ++ symmetric positive-definite linear equations with a single right-
 ++ hand side,  Ax=b, using a Cholesky factorization and iterative
 ++ refinement.
 ++ See \downlink{Manual Page}{manpageXXf04asf}.
f04atf : (Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,_
    Integer,Integer) -> Result
 ++ f04atf(a,ia,b,n,iaa,ifail)
 ++ calculates the accurate solution of a set of real linear
 ++ equations with a single right-hand side, using an LU
 ++ factorization with partial pivoting, and iterative refinement.
 ++ See \downlink{Manual Page}{manpageXXf04atf}.
f04axf : (Integer,Matrix DoubleFloat,Integer,Matrix Integer,_
    Matrix Integer,Integer,Matrix Integer,Matrix DoubleFloat) -> Result
 ++ f04axf(n,a,licn,icn,ikeep,mtype,idisp,rhs)
 ++ calculates the approximate solution of a set of real
 ++ sparse linear equations with a single right-hand side, Ax=b or
 ++  T
 ++ A x=b, where A has been factorized by F01BRF or F01BSF.
 ++ See \downlink{Manual Page}{manpageXXf04axf}.
f04faf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Integer) -> Result
 ++ f04faf(job,n,d,e,b,ifail)
 ++ calculates the approximate solution of a set of real
 ++ symmetric positive-definite tridiagonal linear equations.
 ++ See \downlink{Manual Page}{manpageXXf04faf}.
f04jgf : (Integer,Integer,Integer,DoubleFloat,_
    Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ f04jgf(m,n,nra,tol,lwork,a,b,ifail)
 ++ finds the solution of a linear least-squares problem, Ax=b
 ++ , where A is a real m by n (m>=n) matrix and b is an m element
 ++ vector. If the matrix of observations is not of full rank, then
 ++ the minimal least-squares solution is returned.
 ++ See \downlink{Manual Page}{manpageXXf04jgf}.
f04maf : (Integer,Integer,Matrix DoubleFloat,Integer,_
    Matrix Integer,Integer,Matrix Integer,Matrix DoubleFloat,_
    Matrix Integer,Matrix Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix Integer,Integer) -> Result
 ++ f04maf(n,nz,avals,licn,irn,lirn,icn,wkeep,ikeep,
 ++        inform,b,acc,noits,ifail)
 ++ e a sparse symmetric positive-definite system of linear
 ++ equations, Ax=b, using a pre-conditioned conjugate gradient
 ++ method, where A has been factorized by F01MAF.
 ++ See \downlink{Manual Page}{manpageXXf04maf}.
```

```
  f04mbf : (Integer,Matrix DoubleFloat,Boolean,DoubleFloat,_
      Integer,Integer,Integer,Integer,DoubleFloat,Integer,_
      Union(fn:FileName,fp:Asp28(APROD)),_
      Union(fn:FileName,fp:Asp34(MSOLVE))) -> Result
   ++ f04mbf(n,b,precon,shift,itnlim,msglvl,lrwork,_
   ++        liwork,rtol,ifail,aprod,msolve)
   ++ solves a system of real sparse symmetric linear equations
   ++ using a Lanczos algorithm.
   ++ See \downlink{Manual Page}{manpageXXf04mbf}.
  f04mcf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
      Matrix Integer,Integer,Matrix DoubleFloat,Integer,Integer,_
      Integer,Integer) -> Result
   ++ f04mcf(n,al,lal,d,nrow,ir,b,nrb,iselct,nrx,ifail)
   ++ computes the approximate solution of a system of real
   ++ linear equations with multiple right-hand sides,  AX=B,  where A
   ++ is a symmetric positive-definite variable-bandwidth matrix, which
   ++ has previously been factorized by F01MCF. Related systems may
   ++ also be solved.
   ++ See \downlink{Manual Page}{manpageXXf04mcf}.
  f04qaf : (Integer,Integer,DoubleFloat,DoubleFloat,_
      DoubleFloat,DoubleFloat,Integer,Integer,Integer,Integer,_
      Matrix DoubleFloat,Integer,Union(fn:FileName,_
      fp:Asp30(APROD))) -> Result
   ++ f04qaf(m,n,damp,atol,btol,conlim,itnlim,msglvl,
   ++        lrwork,liwork,b,ifail,aprod)
   ++ solves sparse unsymmetric equations, sparse linear least-
   ++ squares problems and sparse damped linear least-squares problems,
   ++ using a Lanczos algorithm.
   ++ See \downlink{Manual Page}{manpageXXf04qaf}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
  import FortranPackage
  import AnyFunctions1(Integer)
  import AnyFunctions1(DoubleFloat)
  import AnyFunctions1(Boolean)
  import AnyFunctions1(Matrix Complex DoubleFloat)
  import AnyFunctions1(Matrix DoubleFloat)
  import AnyFunctions1(Matrix Integer)


  f04adf(iaArg:Integer,bArg:Matrix Complex DoubleFloat,ibArg:Integer,_
      nArg:Integer,mArg:Integer,icArg:Integer,_
```

```
        aArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f04adf",_
        ["ia"::S,"ib"::S,"n"::S,"m"::S,"ic"::S_
        ,"ifail"::S,"b"::S,"c"::S,"a"::S,"wkspce"::S]$Lisp,_
        ["c"::S,"wkspce"::S]$Lisp,_
        [["double"::S,["wkspce"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"ia"::S,"ib"::S,"n"::S,"m"::S_
        ,"ic"::S,"ifail"::S]$Lisp_
        ,["double complex"::S,["b"::S,"ib"::S,"m"::S]$Lisp,_
        ["c"::S,"ic"::S,"m"::S]$Lisp,["a"::S,"ia"::S,"n"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["c"::S,"a"::S,"ifail"::S]$Lisp,_
        [([iaArg::Any,ibArg::Any,nArg::Any,mArg::Any,icArg::Any,_
        ifailArg::Any,bArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f04arf(iaArg:Integer,bArg:Matrix DoubleFloat,nArg:Integer,_
        aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f04arf",_
        ["ia"::S,"n"::S,"ifail"::S,"b"::S,"c"::S,"a"::S,"wkspce"::S]$Lisp,_
        ["c"::S,"wkspce"::S]$Lisp,_
        [["double"::S,["b"::S,"n"::S]$Lisp,["c"::S,"n"::S]$Lisp_
        ,["a"::S,"ia"::S,"n"::S]$Lisp,["wkspce"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"ia"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["c"::S,"a"::S,"ifail"::S]$Lisp,_
        [([iaArg::Any,nArg::Any,ifailArg::Any,bArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f04asf(iaArg:Integer,bArg:Matrix DoubleFloat,nArg:Integer,_
        aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f04asf",_
        ["ia"::S,"n"::S,"ifail"::S,"b"::S,"c"::S,"a"::S,"wk1"::S,"wk2"::S_
        ]$Lisp,_
        ["c"::S,"wk1"::S,"wk2"::S]$Lisp,_
        [["double"::S,["b"::S,"n"::S]$Lisp,["c"::S,"n"::S]$Lisp,_
        ["a"::S,"ia"::S,"n"::S]$Lisp,["wk1"::S,"n"::S]$Lisp,_
        ["wk2"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"ia"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["c"::S,"a"::S,"ifail"::S]$Lisp,_
        [([iaArg::Any,nArg::Any,ifailArg::Any,bArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

```
f04atf(aArg:Matrix DoubleFloat,iaArg:Integer,bArg:Matrix DoubleFloat,_
    nArg:Integer,iaaArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f04atf",_
    ["ia"::S,"n"::S,"iaa"::S,"ifail"::S,"a"::S,"b"::S,"c"::S,_
    "aa"::S,"wks1"::S,"wks2"::S]$Lisp,_
    ["c"::S,"aa"::S,"wks1"::S,"wks2"::S]$Lisp,_
    [["double"::S,["a"::S,"ia"::S,"n"::S]$Lisp_
    ,["b"::S,"n"::S]$Lisp,["c"::S,"n"::S]$Lisp,_
    ["aa"::S,"iaa"::S,"n"::S]$Lisp,["wks1"::S,"n"::S]$Lisp,_
    ["wks2"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"ia"::S,"n"::S,"iaa"::S,"ifail"::S]$Lisp]$Lisp,_
    ["c"::S,"aa"::S,"ifail"::S]$Lisp,_
    [([iaArg::Any,nArg::Any,iaaArg::Any,ifailArg::Any,_
    aArg::Any,bArg::Any])@List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f04axf(nArg:Integer,aArg:Matrix DoubleFloat,licnArg:Integer,_
    icnArg:Matrix Integer,ikeepArg:Matrix Integer,mtypeArg:Integer,_
    idispArg:Matrix Integer,rhsArg:Matrix DoubleFloat): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f04axf",_
    ["n"::S,"licn"::S,"mtype"::S,"resid"::S,"a"::S,"icn"::S,_
    "ikeep"::S,"idisp"::S,"rhs"::S,"w"::S]$Lisp,_
    ["resid"::S,"w"::S]$Lisp,_
    [["double"::S,["a"::S,"licn"::S]$Lisp,"resid"::S_
    ,["rhs"::S,"n"::S]$Lisp,["w"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"n"::S,"licn"::S,["icn"::S,"licn"::S]$Lisp_
    ,["ikeep"::S,["*"::S,"n"::S,5$Lisp]$Lisp]$Lisp,_
    "mtype"::S,["idisp"::S,2$Lisp]$Lisp]$Lisp_
    ]$Lisp,_
    ["resid"::S,"rhs"::S]$Lisp,_
    [([nArg::Any,licnArg::Any,mtypeArg::Any,aArg::Any,icnArg::Any,_
    ikeepArg::Any,idispArg::Any,rhsArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f04faf(jobArg:Integer,nArg:Integer,dArg:Matrix DoubleFloat,_
    eArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f04faf",_
    ["job"::S,"n"::S,"ifail"::S,"d"::S,"e"::S,"b"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,["d"::S,"n"::S]$Lisp,["e"::S,"n"::S]$Lisp_
    ,["b"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"job"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["d"::S,"e"::S,"b"::S,"ifail"::S]$Lisp,_
    [([jobArg::Any,nArg::Any,ifailArg::Any,dArg::Any,eArg::Any,bArg::Any])_
```

```
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f04jgf(mArg:Integer,nArg:Integer,nraArg:Integer,_
    tolArg:DoubleFloat,lworkArg:Integer,aArg:Matrix DoubleFloat,_
    bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f04jgf",_
    ["m"::S,"n"::S,"nra"::S,"tol"::S,"lwork"::S_
    ,"svd"::S,"sigma"::S,"irank"::S,"ifail"::S,"work"::S,_
    "a"::S,"b"::S]$Lisp,_
    ["svd"::S,"sigma"::S,"irank"::S,"work"::S]$Lisp,_
    [["double"::S,"tol"::S,"sigma"::S,["work"::S,"lwork"::S]$Lisp_
    ,["a"::S,"nra"::S,"n"::S]$Lisp,["b"::S,"m"::S]$Lisp]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"nra"::S,"lwork"::S_
    ,"irank"::S,"ifail"::S]$Lisp_
    ,["logical"::S,"svd"::S]$Lisp_
    ]$Lisp,_
    ["svd"::S,"sigma"::S,"irank"::S,"work"::S,"a"::S,_
    "b"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,nArg::Any,nraArg::Any,tolArg::Any,lworkArg::Any,_
    ifailArg::Any,aArg::Any,bArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f04maf(nArg:Integer,nzArg:Integer,avalsArg:Matrix DoubleFloat,_
    licnArg:Integer,irnArg:Matrix Integer,lirnArg:Integer,_
    icnArg:Matrix Integer,wkeepArg:Matrix DoubleFloat,_
    ikeepArg:Matrix Integer,_
    informArg:Matrix Integer,bArg:Matrix DoubleFloat,_
    accArg:Matrix DoubleFloat,_
    noitsArg:Matrix Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,"f04maf",_
    ["n"::S,"nz"::S,"licn"::S,"lirn"::S,"ifail"::S_
    ,"avals"::S,"irn"::S,"icn"::S,"wkeep"::S,"ikeep"::S_
    ,"inform"::S,"work"::S,"b"::S,"acc"::S,"noits"::S]$Lisp,_
    ["work"::S]$Lisp,_
    [["double"::S,["avals"::S,"licn"::S]$Lisp,_
    ["wkeep"::S,["*"::S,3$Lisp,"n"::S]$Lisp]$Lisp_
    ,["work"::S,["*"::S,3$Lisp,"n"::S]$Lisp]$Lisp,_
    ["b"::S,"n"::S]$Lisp,["acc"::S,2$Lisp]$Lisp_
    ]$Lisp_
    ,["integer"::S,"n"::S,"nz"::S,"licn"::S,["irn"::S,"lirn"::S]$Lisp_
    ,"lirn"::S,["icn"::S,"licn"::S]$Lisp,["ikeep"::S,_
    ["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,["inform"::S,4$Lisp]$Lisp_
    ,["noits"::S,2$Lisp]$Lisp,"ifail"::S]$Lisp]$Lisp,_
    ["work"::S,"b"::S,"acc"::S,"noits"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,nzArg::Any,licnArg::Any,lirnArg::Any,_
    ifailArg::Any,avalsArg::Any,irnArg::Any,icnArg::Any,wkeepArg::Any,_
    ikeepArg::Any,informArg::Any,bArg::Any,accArg::Any,noitsArg::Any ])_
```

```
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f04mbf(nArg:Integer,bArg:Matrix DoubleFloat,preconArg:Boolean,_
        shiftArg:DoubleFloat,itnlimArg:Integer,msglvlArg:Integer,_
        lrworkArg:Integer,liworkArg:Integer,rtolArg:DoubleFloat,_
        ifailArg:Integer,aprodArg:Union(fn:FileName,fp:Asp28(APROD)),_
        msolveArg:Union(fn:FileName,fp:Asp34(MSOLVE))): Result ==
-- if both asps are AXIOM generated we do not need lrwork liwork
--   and will set to 1.
-- else believe the user but check that they are >0.
        if (aprodArg case fp) and (msolveArg case fp)
          then
            lrworkArg:=1
            liworkArg:=1
          else
            lrworkArg:=max(1,lrworkArg)
            liworkArg:=max(1,liworkArg)
        pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
        if aprodArg case fn
          then outputAsFortran(aprodArg.fn)
          else outputAsFortran(aprodArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(msolveFilename := aspFilename "msolve")$FOP
        if msolveArg case fn
          then outputAsFortran(msolveArg.fn)
          else outputAsFortran(msolveArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([aprodFilename,msolveFilename]$Lisp,_
        "f04mbf",_
        ["n"::S,"precon"::S,"shift"::S,"itnlim"::S,"msglvl"::S_
        ,"lrwork"::S,"liwork"::S,"itn"::S,"anorm"::S,"acond"::S_
        ,"rnorm"::S,"xnorm"::S,"inform"::S,"rtol"::S,"ifail"::S_
        ,"aprod"::S,"msolve"::S,"b"::S,"x"::S,"work"::S,"rwork"::S,"iwork"::S_
        ]$Lisp,["x"::S,"itn"::S,"anorm"::S,"acond"::S,"rnorm"::S,"xnorm"::S,_
        "inform"::S,"work"::S,"rwork"::S,"iwork"::S,"aprod"::S,_
        "msolve"::S]$Lisp,[["double"::S,["b"::S,"n"::S]$Lisp,"shift"::S_
        ,["x"::S,"n"::S]$Lisp,"anorm"::S,"acond"::S,"rnorm"::S,"xnorm"::S,_
         "rtol"::S,["work"::S,"n"::S,5$Lisp]$Lisp,_
        ["rwork"::S,"lrwork"::S]$Lisp_
        ,"aprod"::S,"msolve"::S]$Lisp_
        ,["integer"::S,"n"::S,"itnlim"::S,"msglvl"::S_
        ,"lrwork"::S,"liwork"::S,"itn"::S,"inform"::S,"ifail"::S,_
        ["iwork"::S,"liwork"::S]$Lisp]$Lisp_
        ,["logical"::S,"precon"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"itn"::S,"anorm"::S,"acond"::S,"rnorm"::S,"xnorm"::S,_
        "inform"::S,"rtol"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,preconArg::Any,shiftArg::Any,itnlimArg::Any,_
        msglvlArg::Any,lrworkArg::Any,liworkArg::Any,rtolArg::Any,_
```

```
        ifailArg::Any,bArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f04mcf(nArg:Integer,alArg:Matrix DoubleFloat,lalArg:Integer,_
        dArg:Matrix DoubleFloat,nrowArg:Matrix Integer,irArg:Integer,_
        bArg:Matrix DoubleFloat,nrbArg:Integer,iselctArg:Integer,_
        nrxArg:Integer,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f04mcf",_
        ["n"::S,"lal"::S,"ir"::S,"nrb"::S,"iselct"::S_
        ,"nrx"::S,"ifail"::S,"al"::S,"d"::S,"nrow"::S,"b"::S,"x"::S_
        ]$Lisp,_
        ["x"::S]$Lisp,_
        [["double"::S,["al"::S,"lal"::S]$Lisp,["d"::S,"n"::S]$Lisp_
        ,["b"::S,"nrb"::S,"ir"::S]$Lisp,["x"::S,"nrx"::S,"ir"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"lal"::S,["nrow"::S,"n"::S]$Lisp_
        ,"ir"::S,"nrb"::S,"iselct"::S,"nrx"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,lalArg::Any,irArg::Any,nrbArg::Any,iselctArg::Any,_
        nrxArg::Any,ifailArg::Any,alArg::Any,dArg::Any,nrowArg::Any,_
        bArg::Any ])@List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f04qaf(mArg:Integer,nArg:Integer,dampArg:DoubleFloat,_
        atolArg:DoubleFloat,btolArg:DoubleFloat,conlimArg:DoubleFloat,_
        itnlimArg:Integer,msglvlArg:Integer,lrworkArg:Integer,_
        liworkArg:Integer,bArg:Matrix DoubleFloat,ifailArg:Integer,_
        aprodArg:Union(fn:FileName,fp:Asp30(APROD))): Result ==
        pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
        if aprodArg case fn
          then outputAsFortran(aprodArg.fn)
          else outputAsFortran(aprodArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([aprodFilename]$Lisp,_
        "f04qaf",_
        ["m"::S,"n"::S,"damp"::S,"atol"::S,"btol"::S_
        ,"conlim"::S,"itnlim"::S,"msglvl"::S,"lrwork"::S,"liwork"::S_
        ,"itn"::S,"anorm"::S,"acond"::S,"rnorm"::S,"arnorm"::S_
        ,"xnorm"::S,"inform"::S,"ifail"::S,"aprod"::S,"x"::S,"se"::S,_
        "b"::S,"work"::S,"rwork"::S_
        ,"iwork"::S]$Lisp,_
        ["x"::S,"se"::S,"itn"::S,"anorm"::S,"acond"::S,"rnorm"::S,_
        "arnorm"::S,"xnorm"::S,"inform"::S,"work"::S,"rwork"::S,_
        "iwork"::S,"aprod"::S]$Lisp,_
        [["double"::S,"damp"::S,"atol"::S,"btol"::S_
        ,"conlim"::S,["x"::S,"n"::S]$Lisp,["se"::S,"n"::S]$Lisp,_
        "anorm"::S,"acond"::S,"rnorm"::S,"arnorm"::S,"xnorm"::S,_
        ["b"::S,"m"::S]$Lisp_
```

```
,["work"::S,"n"::S,2$Lisp]$Lisp,["rwork"::S,"lrwork"::S]$Lisp,_
"aprod"::S]$Lisp_
,["integer"::S,"m"::S,"n"::S,"itnlim"::S,"msglvl"::S_
,"lrwork"::S,"liwork"::S,"itn"::S,"inform"::S,"ifail"::S,_
["iwork"::S,"liwork"::S]$Lisp]$Lisp]$Lisp,_
["x"::S,"se"::S,"itn"::S,"anorm"::S,"acond"::S,"rnorm"::S,_
"arnorm"::S,"xnorm"::S,"inform"::S,"b"::S,"ifail"::S]$Lisp,_
[([mArg::Any,nArg::Any,dampArg::Any,atolArg::Any,btolArg::Any,_
conlimArg::Any,itnlimArg::Any,msglvlArg::Any,lrworkArg::Any,_
liworkArg::Any,ifailArg::Any,bArg::Any ])_
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

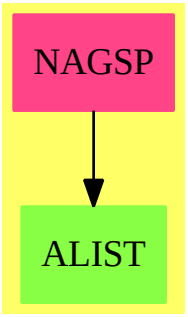
— **NAGF04.dotabb** —

```
"NAGF04" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF04"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NAGF04" -> "COMPCAT"
```

———-

15.4 package NAGSP NAGLinkSupportPackage

15.4.1 NAGLinkSupportPackage (NAGSP)



Exports:

aspFilename	checkPrecision	dimensionsOf	fortranCompilerName
fortranLinkerArgs	restorePrecision		

— **package NAGSP NAGLinkSupportPackage** —

```
)abbrev package NAGSP NAGLinkSupportPackage
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created:  March 1993
++ Date Last Updated: March 4 1994
++                    October 6 1994
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Support functions for the NAG Library Link functions

NAGLinkSupportPackage() : exports == implementation where

  exports ==> with
    fortranCompilerName : () -> String
      ++ fortranCompilerName() returns the name of the currently selected
      ++  Fortran compiler
    fortranLinkerArgs   : () -> String
      ++ fortranLinkerArgs() returns the current linker arguments
    aspFilename         : String -> String
      ++ aspFilename("f") returns a String consisting of "f" suffixed with
      ++  an extension identifying the current AXIOM session.
    dimensionsOf        : (Symbol, Matrix DoubleFloat) -> SExpression
      ++ dimensionsOf(s,m) \undocumented{}
    dimensionsOf        : (Symbol, Matrix Integer) -> SExpression
      ++ dimensionsOf(s,m) \undocumented{}
    checkPrecision      : () -> Boolean
      ++ checkPrecision() \undocumented{}
    restorePrecision    : () -> Void
      ++ restorePrecision() \undocumented{}

  implementation ==> add
    makeAs:                   (Symbol,Symbol) -> Symbol
    changeVariables:          (Expression Integer,Symbol) -> Expression Integer
    changeVariablesF:         (Expression Float,Symbol) -> Expression Float

    import String
    import Symbol

    checkPrecision():Boolean ==
      (_$fortranPrecision$Lisp = "single"::Symbol) and _
      (_$nagEnforceDouble$Lisp)  =>
        systemCommand("set fortran precision double")$MoreSystemCommands
        if _$nagMessages$Lisp  then
          print("*** Warning: Resetting fortran precision to double")_
            $PrintPackage
```

```
      true
    false

  restorePrecision():Void ==
    systemCommand("set fortran precision single")$MoreSystemCommands
    if _$nagMessages$Lisp  then
      print("** Warning: Restoring fortran precision to single")$PrintPackage
    void()$Void

  uniqueId : String := ""
  counter : Integer := 0
  getUniqueId():String ==
    if uniqueId = "" then
      uniqueId := concat(getEnv("HOST")$Lisp,getEnv("SPADNUM")$Lisp)
    concat(uniqueId,string (counter:=counter+1))

  fortranCompilerName() == string _$fortranCompilerName$Lisp
  fortranLinkerArgs() == string _$fortranLibraries$Lisp

  aspFilename(f:String):String == concat ["/tmp/",f,getUniqueId(),".f"]

  dimensionsOf(u:Symbol,m:Matrix DoubleFloat):SExpression ==
    [u,nrows m,ncols m]$Lisp
  dimensionsOf(u:Symbol,m:Matrix Integer):SExpression ==
    [u,nrows m,ncols m]$Lisp
```

———-

— **NAGSP.dotabb** —

```
"NAGSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGSP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGSP" -> "ALIST"
```

———-

15.5 package NAGD01 NagIntegrationPackage

— **NagIntegrationPackage.help** —

```
D01(3NAG)         Foundation Library (12/10/92)         D01(3NAG)

     D01 -- Quadrature                              Introduction -- D01
                                Chapter D01
```

Quadrature

1. Scope of the Chapter

This chapter provides routines for the numerical evaluation of definite integrals in one or more dimensions and for evaluating weights and abscissae of integration rules.

2. Background to the Problems

The routines in this chapter are designed to estimate:

(a) the value of a one-dimensional definite integral of the form:

$$\int_a^b f(x)dx \qquad (1)$$

where f(x) is defined by the user, either at a set of points $(x_i, f(x_i))$, for i=1,2,...,n where $a=x_1<x_2<\ldots<x_n=b$, or in the form of a function; and the limits of integration a,b may be finite or infinite.

Some methods are specially designed for integrands of the form

$$f(x)=w(x)g(x) \qquad (2)$$

which contain a factor w(x), called the weight-function, of a specific form. These methods take full account of any peculiar behaviour attributable to the w(x) factor.

(b) the value of a multi-dimensional definite integral of the form:

$$\int_{R_n} f(x_1, x_2, \ldots, x_n)dx_n \ldots dx_2 dx_1 \qquad (3)$$

where $f(x_1, x_2, \ldots, x_n)$ is a function defined by the user and R_n is some region of n-dimensional space.

The simplest form of R_n is the n-rectangle defined by:

$$a_i <= x_i <= b_i, \quad i=1,2,\ldots,n \qquad (4)$$

where a_i and b_i are constants. When a_i and b_i are functions

```
      of x  (j<i), the region can easily be transformed to the
          j
      rectangular form (see Davis and Rabinowitz [1] page 266).
      Some of the methods described incorporate the
      transformation procedure.

   2.1. One-dimensional Integrals

   To estimate the value of a one-dimensional integral, a quadrature
   rule uses an approximation in the form of a weighted sum of
   integrand values, i.e.,

                        b         N
                        /         --
                        |f(x)dx~= >  w f(x ).                    (5)
                        /         --  i   i
                        a         i=1

   The points x  within the interval [a,b] are known as the
               i
   abscissae, and the w  are known as the weights.
                       i

   More generally, if the integrand has the form (2), the
   corresponding formula is

                      b             N
                      /             --
                      |w(x)g(x)dx~= >  w g(x ).                  (6)
                      /             --  i   i
                      a             i=1

   If the integrand is known only at a fixed set of points, these
   points must be used as the abscissae, and the weighted sum is
   calculated using finite-difference methods. However, if the
   functional form of the integrand is known, so that its value at
   any abscissa is easily obtained, then a wide variety of
   quadrature rules are available, each characterised by its choice
   of abscissae and the corresponding weights.

   The appropriate rule to use will depend on the interval [a,b] -
   whether finite or otherwise - and on the form of any w(x) factor
   in the integrand. A suitable value of N depends on the general
   behaviour of f(x); or of g(x), if there is a w(x) factor present.

   Among possible rules, we mention particularly the Gaussian
   formulae, which employ a distribution of abscissae which is
   optimal for f(x) or g(x) of polynomial form.

   The choice of basic rules constitutes one of the principles on
```

```
which methods for one-dimensional integrals may be classified.
The other major basis of classification is the implementation
strategy, of which some types are now presented.

(a)   Single rule evaluation procedures

      A fixed number of abscissae, N, is used. This number and
      the particular rule chosen uniquely determine the weights
      and abscissae. No estimate is made of the accuracy of the
      result.

(b)   Automatic procedures

      The number of abscissae, N, within [a,b] is gradually
      increased until consistency is achieved to within a level
      of accuracy (absolute or relative) requested by the user.
      There are essentially two ways of doing this; hybrid forms
      of these two methods are also possible:
      (i)   whole interval procedures (non-adaptive)

            A series of rules using increasing values of N are
            successively applied over the whole interval [a,b].
            It is clearly more economical if abscissae already
            used for a lower value of N can be used again as part
            of a higher-order formula. This principle is known as
             optimal extension. There is no overlap between the
            abscissae used in Gaussian formulae of different
            orders. However, the Kronrod formulae are designed to
            give an optimal (2N+1)-point formula by adding (N+1)
            points to an N-point Gauss formula. Further
            extensions have been developed by Patterson.

      (ii)  adaptive procedures

            The interval [a,b] is repeatedly divided into a
            number of sub-intervals, and integration rules are
            applied separately to each sub-interval. Typically,
            the subdivision process will be carried further in
            the neighbourhood of a sharp peak in the integrand,
            than where the curve is smooth. Thus, the
            distribution of abscissae is adapted to the shape of
            the integrand.

            Subdivision raises the problem of what constitutes an
            acceptable accuracy in each sub-interval. The usual
            global acceptability criterion  demands that the sum
            of the absolute values of the error estimates in the
            sub-intervals should meet the conditions required of
            the error over the whole interval. Automatic
            extrapolation over several levels of subdivision may
```

eliminate the effects of some types of singularities.

An ideal general-purpose method would be an automatic method which could be used for a wide variety of integrands, was efficient (i.e., required the use of as few abscissae as possible), and was reliable (i.e., always gave results within the requested accuracy). Complete reliability is unobtainable, and generally higher reliability is obtained at the expense of efficiency, and vice versa. It must therefore be emphasised that the automatic routines in this chapter cannot be assumed to be 100% reliable. In general, however, the reliability is very high.

2.2. Multi-dimensional Integrals

A distinction must be made between cases of moderately low dimensionality (say, up to 4 or 5 dimensions), and those of higher dimensionality. Where the number of dimensions is limited, a one-dimensional method may be applied to each dimension, according to some suitable strategy, and high accuracy may be obtainable (using product rules). However, the number of integrand evaluations rises very rapidly with the number of dimensions, so that the accuracy obtainable with an acceptable amount of computational labour is limited; for example a product 9 of 3-point rules in 20 dimensions would require more than 10 integrand evaluations. Special techniques such as the Monte Carlo methods can be used to deal with high dimensions.

(a) Products of one-dimensional rules

Using a two-dimensional integral as an example, we have

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y)\,dy\,dx \simeq \sum_{i=1}^{N} w_i \left[\int_{a_2}^{b_2} f(x_i,y)\,dy\right] \tag{7}$$

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y)\,dy\,dx \simeq \sum_{i=1}^{N}\sum_{j=1}^{N} w_i v_j f(x_i,y_j) \tag{8}$$

where (w_i, x_i) and (v_j, y_j) are the weights and abscissae of the rules used in the respective dimensions.

A different one-dimensional rule may be used for each dimension, as appropriate to the range and any weight function present, and a different strategy may be used, as appropriate to the integrand behaviour as a function of each independent variable.

For a rule-evaluation strategy in all dimensions, the formula (8) is applied in a straightforward manner. For automatic strategies (i.e., attempting to attain a requested accuracy), there is a problem in deciding what accuracy must be requested in the inner integral(s). Reference to formula (7) shows that the presence of a limited but random error in the y-integration for different values of x_i can produce a 'jagged' function of x, which may be difficult to integrate to the desired accuracy and for this reason products of automatic one-dimensional routines should be used with caution (see also Lyness [3]).

(b) Monte Carlo methods

These are based on estimating the mean value of the integrand sampled at points chosen from an appropriate statistical distribution function. Usually a variance reducing procedure is incorporated to combat the fundamentally slow rate of convergence of the rudimentary form of the technique. These methods can be effective by comparison with alternative methods when the integrand contains singularities or is erratic in some way, but they are of quite limited accuracy.

(c) Number theoretic methods

These are based on the work of Korobov and Conroy and operate by exploiting implicitly the properties of the Fourier expansion of the integrand. Special rules, constructed from so-called optimal coefficients, give a particularly uniform distribution of the points throughout n-dimensional space and from their number theoretic properties minimize the error on a prescribed class of integrals. The method can be combined with the Monte Carlo procedure.

(d) Sag-Szekeres method

By transformation this method seeks to induce properties into the integrand which make it accurately integrable by the trapezoidal rule. The transformation also allows effective control over the number of integrand evaluations.

(e) Automatic adaptive procedures

An automatic adaptive strategy in several dimensions normally involves division of the region into subregions, concentrating the divisions in those parts of the region where the integrand is worst behaved. It is difficult to arrange with any generality for variable limits in the inner integral(s). For this reason, some methods use a region where all the limits are constants; this is called a hyper-rectangle. Integrals over regions defined by variable or infinite limits may be handled by transformation to a hyper-rectangle. Integrals over regions so irregular that such a transformation is not feasible may be handled by surrounding the region by an appropriate hyper-rectangle and defining the integrand to be zero outside the desired region. Such a technique should always be followed by a Monte Carlo method for integration.

The method used locally in each subregion produced by the adaptive subdivision process is usually one of three types: Monte Carlo, number theoretic or deterministic. Deterministic methods are usually the most rapidly convergent but are often expensive to use for high dimensionality and not as robust as the other techniques.

2.3. References

Comprehensive reference:

[1] Davis P J and Rabinowitz P (1975) Methods of Numerical Integration. Academic Press.

Special topics:

[2] Gladwell I (1986) Vectorisation of one dimensional quadrature codes. Techincal Report. TR7/86 NAG.

[3] Lyness J N (1983) When not to use an automatic quadrature routine. SIAM Review. 25 63--87.

[4] Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D (1983) QUADPACK, A Subroutine Package for Automatic Integration. Springer-Verlag.

[5] Sobol I M (1974) The Monte Carlo Method. The University of Chicago Press.

[6] Stroud A H (1971) Approximate Calculation of Multiple Integrals. Prentice-Hall.

3. Recommendations on Choice and Use of Routines

The following three sub-sections consider in turn routines for: one-dimensional integrals over a finite interval, and over a semi-infinite or an infinite interval; and multi-dimensional integrals. Within each sub-section, routines are classified by the type of method, which ranges from simple rule evaluation to automatic adaptive algorithms. The recommendations apply particularly when the primary objective is simply to compute the value of one or more integrals, and in these cases the automatic adaptive routines are generally the most convenient and reliable, although also the most expensive in computing time.

Note however that in some circumstances it may be counter-productive to use an automatic routine. If the results of the quadrature are to be used in turn as input to a further computation (e.g. an 'outer' quadrature or an optimization problem), then this further computation may be adversely affected by the 'jagged performance profile' of an automatic routine; a simple rule-evaluation routine may provide much better overall performance. For further guidance, the article Lyness [3] is recommended.

3.1. One-dimensional Integrals over a Finite Interval

(a) Integrand defined as a set of points

If f(x) is defined numerically at four or more points, then the Gill-Miller finite difference method (D01GAF) should be used. The interval of integration is taken to coincide with the range of x-values of the points supplied. It is in the nature of this problem that any routine may be unreliable. In order to check results independently and so as to provide an alternative technique the user may fit the integrand by Chebyshev series using E02ADF and then use routines E02AJF and E02AKF to evaluate its integral (which need not be restricted to the range of the integration points, as is the case for D01GAF). A further alternative is to fit a cubic spline to the data using E02BAF and then to evaluate its integral using E02BDF.

(b) Integrand defined as a function

If the functional form of f(x) is known, then one of the following approaches should be taken. They are arranged in the order from most specific to most general, hence the first applicable procedure in the list will be the most efficient. However, if the user does not wish to make any assumptions about the integrand, the most reliable routine to use will be D01AJF, although this will in general be

```
less efficient for simple integrals.
(i)   Rule-evaluation routines

      If f(x) is known to be sufficiently well-behaved
      (more precisely, can be closely approximated by a
      polynomial of moderate degree), a Gaussian routine
      with a suitable number of abscissae may be used.

      D01BBF may be used if it is required to examine the
      weights and abscissae. In this case, the user should
      write the code for the evaluation of quadrature
      summation (6).

(ii)  Automatic adaptive routines

      Firstly, several routines are available for
      integrands of the form w(x)g(x) where g(x) is a '
      smooth' function (i.e., has no singularities, sharp
      peaks or violent oscillations in the interval of
      integration) and w(x) is a weight function of one of
      the following forms:

                  (alpha)     (beta)          k          l
      if w(x)=(b-x)       (x-a)      (log(b-x)) (log(x-a))

      where k,l=0 or 1, (alpha),(beta)>-1: use D01APF;

      if w(x)=1/(x-c): use D01AQF (this integral is called
      the Hilbert transform of g);

      if w(x)=cos((omega)x) or sin((omega)x): use D01ANF
      (this routine can also handle certain types of
      singularities in g(x)).

      Secondly, there are some routines for general f(x).
      If f(x) is known to be free of singularities, though
      it may be oscillatory, D01AKF may be used.

      The most powerful of the finite interval integration
      routine is D01AJF (which can cope with singularities
      of several types). It may be used if none of the more
      specific situations described above applies. D01AJF
      is very reliable, particularly where the integrand
      has singularities other than at an end-point, or has
      discontinuities or cusps, and is therefore
      recommended where the integrand is known to be badly-
      behaved, or where its nature is completely unknown.

      Most of the routines in this chapter require the user
      to supply a function or subroutine to evaluate the
```

integrand at a single point.

If f(x) has singularities of certain types, discontinuities or sharp peaks occurring at known points, the integral should be evaluated separately over each of the subranges or D01ALF may be used.

3.2. One-dimensional Integrals over a Semi-infinite or Infinite Interval

(a) Integrand defined as a set of points

If f(x) is defined numerically at four or more points, and the portion of the integral lying outside the range of the points supplied may be neglected, then the Gill-Miller finite difference method, D01GAF, should be used.

(b) Integrand defined as a function

(i) Rule evaluation routines

If f(x) behaves approximately like a polynomial in x, apart from a weight function of the form

$e^{-(\beta)x}$ $(\beta)>0$ (semi-infinite interval, lower limit finite);

or $e^{-(\beta)x}$ $(\beta)<0$ (semi-infinite interval, upper limit finite);

or $e^{-(\beta)(x-(\alpha))^2}$ $(\beta)>0$ (infinite interval);

or if f(x) behaves approximately like a $(x+B)^{-1}$ polynomial (semi-infinite range); then the Gaussian routines may be used.

D01BBF may be used if it is required to examine the weights and abscissae. In this case, the user should write the code for the evaluation of quadrature summation (6).

(ii) Automatic adaptive routines

D01AMF may be used, except for integrands which decay slowly towards an infinite end-point, and oscillate in sign over the entire range. For this class, it may be possible to calculate the integral by integrating

between the zeros and invoking some extrapolation process.

D01ASF may be used for integrals involving weight functions of the form cos((omega)x) and sin((omega)x) over a semi-infinite interval (lower limit finite).

The following alternative procedures are mentioned for completeness, though their use will rarely be necessary:

(1) If the integrand decays rapidly towards an infinite end-point, a finite cut-off may be chosen, and the finite range methods applied.

(2) If the only irregularities occur in the finite part (apart from a singularity at the finite limit, with which D01AMF can cope), the range may be divided, with D01AMF used on the infinite part.

(3) A transformation to finite range may be employed, e.g.

$$x = \frac{1-t}{t} \quad \text{or} \quad x = -\log t$$

will transform (0,infty) to (1,0) while for infinite ranges we have

$$\int_{-\infty}^{+\infty} f(x)\,dx = \int_{0}^{\infty} [f(x)+f(-x)]\,dx.$$

If the integrand behaves badly on (-infty,0) and well on (0,infty) or vice versa it is better to compute it as

$$\int_{-\infty}^{+\infty} f(x)\,dx = \int_{-\infty}^{0} f(x)\,dx + \int_{0}^{\infty} f(x)\,dx.$$

This saves computing unnecessary function values in the semi-infinite range where the function is well behaved.

3.3. Multi-dimensional Integrals

A number of techniques are available in this area and the choice depends to a large extent on the dimension and the required accuracy. It can be advantageous to use more than one technique as a confirmation of accuracy particularly for high dimensional

integrations. Many of the routines incorporate the transformation procedure REGION which allows general product regions to be easily dealt with in terms of conversion to the standard n-cube region.

(a) Products of one-dimensional rules (suitable for up to about 5 dimensions)

If $f(x_1, x_2, \ldots, x_n)$ is known to be a sufficiently well-behaved function of each variable x_i, apart possibly from weight functions of the types provided, a product of Gaussian rules may be used. These are provided by D01BBF. In this case, the user should write the code for the evaluation of quadrature summation (6). Rules for finite, semi-infinite and infinite ranges are included.

The one-dimensional routines may also be used recursively. For example, the two-dimensional integral

$$I = \int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x,y)\,dy\,dx$$

may be expressed as

$$I = \int_{a_1}^{b_1} F(x)\,dx,$$

where

$$F(x) = \int_{a_2}^{b_2} f(x,y)\,dy.$$

The user segment to evaluate F(x) will call the integration routine for the y-integration, which will call another user segment for f(x,y) as a function of y (x being effectively a constant). Note that, as Fortran is not a recursive language, a different library integration routine must be used for each dimension. Apart from this restriction, the full range of one-dimensional routines are available, for

finite/infinite intervals, constant/variable limits, rule evaluation/automatic strategies etc.

(b) Automatic routines (D01GBF and D01FCF)

Both routines are for integrals of the form

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} f(x_1,x_2,\ldots,x_n)\,dx_n\,dx_{n-1}\ldots dx_1 .$$

D01GBF is an adaptive Monte Carlo routine. This routine is usually slow and not recommended for high accuracy work. It is a robust routine that can often be used for low accuracy results with highly irregular integrands or when n is large.

D01FCF is an adaptive deterministic routine. Convergence is fast for well-behaved integrands. Highly accurate results can often be obtained for n between 2 and 5, using significantly fewer integrand evaluations than would be required by D01GBF. The routine will usually work when the integrand is mildly singular and for n<=10 should be used before D01GBF. If it is known in advance that the integrand is highly irregular, it is best to compare results from at least two different routines.

There are many problems for which one or both of the routines will require large amounts of computing time to obtain even moderately accurate results. The amount of computing time is controlled by the number of integrand evaluations allowed by the user, and users should set this parameter carefully, with reference to the time available and the accuracy desired.

3.4. Decision Trees

(i) One-dimensional integrals over a finite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual

(ii) One-dimensional integrals over a semi-infinite or infinite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual

D01 -- Quadrature Contents -- D01
Chapter D01

Quadrature

D01AJF 1-D quadrature, adaptive, finite interval, strategy due to Piessens and de Doncker, allowing for badly-behaved integrands

D01AKF 1-D quadrature, adaptive, finite interval, method suitable for oscillating functions

D01ALF 1-D quadrature, adaptive, finite interval, allowing for singularities at user-specified break-points

D01AMF 1-D quadrature, adaptive, infinite or semi-infinite interval

D01ANF 1-D quadrature, adaptive, finite interval, weight function cos((omega)x) or sin((omega)x)

D01APF 1-D quadrature, adaptive, finite interval, weight function with end-point singularities of algebraico-logarithmic type

D01AQF 1-D quadrature, adaptive, finite interval, weight function 1/(x-c), Cauchy principal value (Hilbert transform)

D01ASF 1-D quadrature, adaptive, semi-infinite interval, weight function cos((omega)x) or sin((omega)x)

D01BBF Weights and abscissae for Gaussian quadrature rules

D01FCF Multi-dimensional adaptive quadrature over hyper-rectangle

D01GAF 1-D quadrature, integration of function defined by data values, Gill-Miller method

D01GBF Multi-dimensional quadrature over hyper-rectangle, Monte Carlo method

%%%

```
D01AJF(3NAG)      Foundation Library (12/10/92)      D01AJF(3NAG)

     D01 -- Quadrature                                          D01AJF
            D01AJF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     D01AJF is a general-purpose integrator which calculates an
     approximation to the integral of a function f(x) over a finite
     interval [a,b]:

                                  b
                                  /
                               I= |f(x)dx.
                                  /
                                  a

     2. Specification

           SUBROUTINE D01AJF (F, A, B, EPSABS, EPSREL, RESULT,
          1                   ABSERR, W, LW, IW, LIW, IFAIL)
           INTEGER          LW, IW(LIW), LIW, IFAIL
           DOUBLE PRECISION F, A, B, EPSABS, EPSREL, RESULT, ABSERR, W
          1                 (LW)
           EXTERNAL         F

     3. Description

     D01AJF is based upon the QUADPACK routine QAGS (Piessens et al
     [3]). It is an adaptive routine, using the Gauss 10-point and
     Kronrod 21-point rules. The algorithm, described by de Doncker
     [1], incorporates a global acceptance criterion (as defined by
     Malcolm and Simpson [2]) together with the (epsilon)-algorithm
     (Wynn [4]) to perform extrapolation. The local error estimation
     is described by Piessens et al [3].

     The routine is suitable as a general purpose integrator, and can
     be used when the integrand has singularities, especially when
     these are of algebraic or logarithmic type.

     D01AJF requires the user to supply a function to evaluate the
     integrand at a single point.

     The routine D01ATF(*) uses an identical algorithm but requires
```

```
the user to supply a subroutine to evaluate the integrand at an
array of points. Therefore D01ATF(*) will be more efficient if
the evaluation can be performed in vector mode on a vector-
processing machine.

4. References

[1]   De Doncker E (1978) An Adaptive Extrapolation Algorithm for
      Automatic Integration. Signum Newsletter. 13 (2) 12--18.

[2]   Malcolm M A and Simpson R B (1976) Local Versus Global
      Strategies for Adaptive Quadrature. ACM Trans. Math. Softw.
      1 129--146.

[3]   Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D
      (1983) QUADPACK, A Subroutine Package for Automatic
      Integration. Springer-Verlag.

[4]   Wynn P (1956) On a Device for Computing the e (S )
                                                     m  n
      Transformation. Math. Tables Aids Comput. 10 91--96.

5. Parameters

 1:  F -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                                 External Procedure
     F must return the value of the integrand f at a given point.

     Its specification is:

           DOUBLE PRECISION FUNCTION F (X)
           DOUBLE PRECISION X

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the point at which the integrand f must be
          evaluated.
     F must be declared as EXTERNAL in the (sub)program from
     which D01AJF is called. Parameters denoted as Input
     must not be changed by this procedure.

 2:  A -- DOUBLE PRECISION                                  Input
     On entry: the lower limit of integration, a.

 3:  B -- DOUBLE PRECISION                                  Input
     On entry: the upper limit of integration, b. It is not
     necessary that a<b.

 4:  EPSABS -- DOUBLE PRECISION                             Input
     On entry: the absolute accuracy required. If EPSABS is
     negative, the absolute value is used. See Section 7.
```

```
 5:  EPSREL -- DOUBLE PRECISION                                  Input
     On entry: the relative accuracy required. If EPSREL is
     negative, the absolute value is used. See Section 7.

 6:  RESULT -- DOUBLE PRECISION                                  Output
     On exit: the approximation to the integral I.

 7:  ABSERR -- DOUBLE PRECISION                                  Output
     On exit: an estimate of the modulus of the absolute error,
     which should be an upper bound for |I-RESULT|.

 8:  W(LW) -- DOUBLE PRECISION array                             Output
     On exit: details of the computation, as described in
     Section 8.

 9:  LW -- INTEGER                                               Input
     On entry:
     the dimension of the array W as declared in the (sub)program
     from which D01AJF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which the interval
     of integration may be divided by the routine. The number of
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: a value
     in the range 800 to 2000 is adequate for most problems.
     Constraint: LW >= 4.

10:  IW(LIW) -- INTEGER array                                    Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

11:  LIW -- INTEGER                                              Input
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01AJF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW. Suggested
     value: LIW = LW/4. Constraint: LIW >= 1.

12:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
```

```
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requirements
     being achieved. Look at the integrand in order to determine
     the integration difficulties. If the position of a local
     difficulty within the interval can be determined (e.g. a
     singularity of the integrand or its derivative, a peak, a
     discontinuity, etc) you will probably gain from splitting up
     the interval at this point and calling the integrator on the
     subranges. If necessary, another integrator, which is
     designed for handling the type of difficulty involved, must
     be used. Alternatively, consider relaxing the accuracy
     requirements specified by EPSABS and EPSREL, or increasing
     the amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. The error may be under-estimated. Consider
     requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
     subdivision around one (or more) points of the interval.

IFAIL= 4
     The requested tolerance cannot be achieved, because the
     extrapolation does not increase the accuracy satisfactorily;
     the returned result is the best which can be obtained. The
     same advice applies as in the case of IFAIL = 1.

IFAIL= 5
     The integral is probably divergent, or slowly convergent.
     Please note that divergence can occur with any non-zero
     value of IFAIL.

IFAIL= 6
     On entry LW < 4,

     or       LIW < 1.
```

```
7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerance. Moreover it returns the quantity ABSERR which,
in normal circumstances, satisfies

                   |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01AJF along with the integral
contributions and error estimates over the sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
the value of the integral over the sub-interval [a ,b ] in the
                                                  i  i
partition of [a,b] and e  be the corresponding absolute error
                       i
estimate.

       b
        i                     n
       /                      --
Then,  | f(x)dx~=r  and RESULT= >  r , unless D01AJF terminates
       /          i            --  i
       a                      i=1
        i
while testing for divergence of the integral (see Piessens et al
[3], Section 3.4.3). In this case, RESULT (and ABSERR) are taken
to be the values returned from the extrapolation process. The
value of n is returned in IW(1), and the values a , b , e  and r
                                                 i   i   i      i
are stored consecutively in the array W, that is:

     a =W(i),
```

```
      i

     b =W(n+i),
      i

     e =W(2n+i) and
      i

     r =W(3n+i).
      i
```

9. Example

To compute

```
                  2(pi)
                  /         x sin(30x)
                  |     ----------------dx.
                  /
                  0        /(  (  x  )2)
                          / (1-(-----) )
                         \/  (  (2(pi)) )
```

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

D01AKF(3NAG) Foundation Library (12/10/92) D01AKF(3NAG)

D01 -- Quadrature D01AKF
D01AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01AKF is an adaptive integrator, especially suited to oscillating, non-singular integrands, which calculates an approximation to the integral of a function f(x) over a finite interval [a,b]:

```
                              b
                              /
                           I= |f(x)dx.
                              /
                              a
```

```
2. Specification

       SUBROUTINE D01AKF (F, A, B, EPSABS, EPSREL, RESULT,
      1                   ABSERR, W, LW, IW, LIW, IFAIL)
       INTEGER          LW, IW(LIW), LIW, IFAIL
       DOUBLE PRECISION F, A, B, EPSABS, EPSREL, RESULT, ABSERR, W
      1                 (LW)
       EXTERNAL         F

3. Description

D01AKF is based upon the QUADPACK routine QAG (Piessens et al [3]
). It is an adaptive routine, using the Gauss 30-point and
Kronrod 61-point rules. A 'global' acceptance criterion (as
defined by Malcolm and Simpson [1]) is used. The local error
estimation is described in Piessens et al [3].

Because this routine is based on integration rules of high order,
it is especially suitable for non-singular oscillating
integrands.

D01AKF requires the user to supply a function to evaluate the
integrand at a single point.

The routine D01AUF(*) uses an identical algorithm but requires
the user to supply a subroutine to evaluate the integrand at an
array of points. Therefore D01AUF(*) will be more efficient if
the evaluation can be performed in vector mode on a vector-
processing machine.

D01AUF(*) also has an additional parameter KEY which allows the
user to select from six different Gauss-Kronrod rules.

4. References

[1]   Malcolm M A and Simpson R B (1976) Local Versus Global
      Strategies for Adaptive Quadrature. ACM Trans. Math. Softw.
      1 129--146.

[2]   Piessens R (1973) An Algorithm for Automatic Integration.
      Angewandte Informatik. 15 399--401.

[3]   Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D
      (1983) QUADPACK, A Subroutine Package for Automatic
      Integration. Springer-Verlag.

5. Parameters

 1:  F -- DOUBLE PRECISION FUNCTION, supplied by the user.
```

```
                                                    External Procedure
F must return the value of the integrand f at a given point.

Its specification is:

       DOUBLE PRECISION FUNCTION F (X)
       DOUBLE PRECISION X

 1:  X -- DOUBLE PRECISION                                  Input
     On entry: the point at which the integrand f must be
     evaluated.
F must be declared as EXTERNAL in the (sub)program from
which D01AKF is called. Parameters denoted as Input
must not be changed by this procedure.

2:  A -- DOUBLE PRECISION                                  Input
    On entry: the lower limit of integration, a.

3:  B -- DOUBLE PRECISION                                  Input
    On entry: the upper limit of integration, b. It is not
    necessary that a<b.

4:  EPSABS -- DOUBLE PRECISION                             Input
    On entry: the absolute accuracy required. If EPSABS is
    negative, the absolute value is used. See Section 7.

5:  EPSREL -- DOUBLE PRECISION                             Input
    On entry: the relative accuracy required. If EPSREL is
    negative, the absolute value is used. See Section 7.

6:  RESULT -- DOUBLE PRECISION                            Output
    On exit: the approximation to the integral I.

7:  ABSERR -- DOUBLE PRECISION                            Output
    On exit: an estimate of the modulus of the absolute error,
    which should be an upper bound |I-RESULT|.

8:  W(LW) -- DOUBLE PRECISION array                       Output
    On exit: details of the computation, as described in
    Section 8.

9:  LW -- INTEGER                                          Input
    On entry: the dimension of W, as declared in the (sub)
    program from which D01AKF is called. The value of LW
    (together with that of LIW below) imposes a bound on the
    number of sub-intervals into which the interval of
    integration may be divided by the routine. The number of
    sub-intervals cannot exceed LW/4. The more difficult the
    integrand, the larger LW should be. Suggested value: a value
    in the range 800 to 2000 is adequate for most problems.
```

```
     Constraint: LW >= 4. See IW below.

10:  IW(LIW) -- INTEGER array                                  Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

11:  LIW -- INTEGER                                             Input
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01AKF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW. Suggested
     value: LIW = LW/4. Constraint: LIW >= 1.

12:  IFAIL -- INTEGER                                     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requirements
     being achieved. Look at the integrand in order to determine
     the integration difficulties. Probably another integrator
     which is designed for handling the type of difficulty
     involved must be used. Alternatively, consider relaxing the
     accuracy requirements specified by EPSABS and EPSREL, or
     increasing the amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. Consider  requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
```

```
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
     On entry LW < 4,

     or       LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                  tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which,
in normal circumstances satisfies

                     |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01AKF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
the value of the integral over the sub-interval [a ,b ] in the
                                                 i  i
partition of [a,b] and e  be the corresponding absolute error
                       i
                 b
                  i                      n
                  /                      --
estimate. Then,   | f(x)dx~=r  and RESULT= >  r . The value of n
                  /          i           --  i
                  a                      i=1
                   i
is returned in IW(1), and the values a , b , e  and r  are stored
                                      i   i   i      i
```

```
consecutively in the array W, that is:

     a =W(i),
      i

     b =W(n+i),
      i

     e =W(2n+i) and
      i

     r =W(3n+i).
      i

9. Example

To compute

                      2(pi)
                      /
                      |    x sin(30x) cosx dx.
                      /
                      0

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
D01ALF(3NAG)      Foundation Library (12/10/92)      D01ALF(3NAG)

     D01 -- Quadrature                                          D01ALF
            D01ALF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     D01ALF is a general purpose integrator which calculates an
     approximation to the integral of a function f(x) over a finite
     interval [a,b]:

                                  b
                                  /
                              I= |f(x)dx
                                  /
```

a

where the integrand may have local singular behaviour at a finite number of points within the integration interval.

2. Specification

```
      SUBROUTINE D01ALF (F, A, B, NPTS, POINTS, EPSABS, EPSREL,
     1                   RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
      INTEGER           NPTS, LW, IW(LIW), LIW, IFAIL
      DOUBLE PRECISION F, A, B, POINTS(*), EPSABS, EPSREL,
     1                 RESULT, ABSERR, W(LW)
      EXTERNAL          F
```

3. Description

D01ALF is based upon the QUADPACK routine QAGP (Piessens et al [3]). It is very similar to D01AJF, but allows the user to supply difficult. It is an adaptive routine, using the Gauss 10-point and Kronrod 21-point rules. The algorithm described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The user-supplied 'break-points'always occur as the end-points of some sub-interval during the adaptive process. The local error estimation is described by Piessens et al [3].

4. References

[1] De Doncker E (1978) An Adaptive Extrapolation Algorithm for Automatic Integration. Signum Newsletter. 13 (2) 12--18.

[2] Malcolm M A and Simpson R B (1976) Local Versus Global Strategies for Adaptive Quadrature. ACM Trans. Math. Softw. 1 129--146.

[3] Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D (1983) QUADPACK, A Subroutine Package for Automatic Integration. Springer-Verlag.

[4] Wynn P (1956) On a Device for Computing the $e_m(S_n)$ Transformation. Math. Tables Aids Comput. 10 91--96.

5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure
F must return the value of the integrand f at a given point.

```
Its specification is:

       DOUBLE PRECISION FUNCTION F (X)
       DOUBLE PRECISION X

 1:  X -- DOUBLE PRECISION                                  Input
     On entry: the point at which the integrand f must be
     evaluated.
F must be declared as EXTERNAL in the (sub)program from
which D01ALF is called. Parameters denoted as Input
must not be changed by this procedure.

 2:  A -- DOUBLE PRECISION                                  Input
     On entry: the lower limit of integration, a.

 3:  B -- DOUBLE PRECISION                                  Input
     On entry: the upper limit of integration, b. It is not
     necessary that a<b.

 4:  NPTS -- INTEGER                                        Input
     On entry: the number of user-supplied break-points within
     the integration interval. Constraint: NPTS >= 0.

 5:  POINTS(NPTS) -- DOUBLE PRECISION array                 Input
     On entry: the user-specified break-points. Constraint: the
     break-points must all lie within the interval of integration
     (but may be supplied in any order).

 6:  EPSABS -- DOUBLE PRECISION                             Input
     On entry: the absolute accuracy required. If EPSABS is
     negative, the absolute value is used. See Section 7.

 7:  EPSREL -- DOUBLE PRECISION                             Input
     On entry: the relative accuracy required. If EPSREL is
     negative, the absolute value is used. See Section 7.

 8:  RESULT -- DOUBLE PRECISION                             Input
     On entry: the approximation to the integral I.

 9:  ABSERR -- DOUBLE PRECISION                            Output
     On exit: an estimate of the modulus of the absolute error,
     which should be an upper bound for |I-RESULT|.

10:  W(LW) -- DOUBLE PRECISION array                       Output
     On exit: details of the computation, as described in
     Section 8.

11:  LW -- INTEGER                                         Input
     On entry:
     the dimension of the array W as declared in the (sub)program
```

```
     from which D01ALF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which the interval
     of integration may be divided by the routine. The number of
     sub-intervals cannot exceed (LW-2*NPTS-4)/4. The more
     difficult the integrand, the larger LW should be. Suggested
     value: a value in the range 800 to 2000 is adequate for most
     problems. Constraint: LW>=2*NPTS+8.

12:  IW(LIW) -- INTEGER array                                Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

13:  LIW -- INTEGER                                           Input
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01ALF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed (LIW-NPTS-2)/2.
     Suggested value: LIW = LW/2. Constraint: LIW >= NPTS + 4.

14:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached, without the accuracy
     requirements being achieved. Look at the integrand in order
     to determine the integration difficulties. If the position
     of a local difficulty within the interval can be determined
     (e.g. a singularity of the integrand or its derivative, a
     peak, a discontinuity, etc) it should be supplied to the
     routine as an element of the vector POINTS. If necessary,
```

```
     another integrator should be used, which is designed for
     handling the type of difficulty involved. Alternatively,
     consider relaxing the accuracy requirements specified by
     EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. The error may be under-estimated. Consider
     requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
     The requested tolerance cannot be achieved, because the
     extrapolation does not increase the accuracy satisfactorily;
     the result returned is the best which can be obtained.  The
     same advice applies as in the case IFAIL = 1.

IFAIL= 5
     The integral is probably divergent, or slowly convergent.
     Please note that divergence can also occur with any other
     non-zero value of IFAIL.

IFAIL= 6
     The input is invalid: break-points are specified outside the
     integration range, NPTS > LIMIT or NPTS < 0. RESULT and
     ABSERR are set to zero.

IFAIL= 7
     On entry LW<2*NPTS+8,

     or       LIW<NPTS+4.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:
                        |I-RESULT|<=tol,

where

                 tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which,
in normal circumstances, satisfies
```

```
                    |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and on the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01ALF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
the value of the integral over the sub-interval [a ,b ] in the
                                                 i  i
partition of [a,b] and e  be the corresponding absolute error
                       i
                 b
                  i                      n
                 /                       --
estimate. Then,  | f(x)dx~=r  and RESULT= >  r  unless D01ALF
                 /          i            --  i
                 a                       i=1
                  i
terminates while testing for divergence of the integral (see
Piessens et al [3] Section 3.4.3). In this case, RESULT (and
ABSERR) are taken to be the values returned from the
extrapolation process. The value of n is returned in IW(1), and
the values a , b , e  and r  are stored consecutively in the
            i   i   i      i
array W, that is:

     a =W(i),
      i

     b =W(n+i),
      i

     e =W(2n+i) and
      i

     r =W(3n+i).
      i

9. Example

To compute
```

```
                                  /     1
                                  | ---------dx.
                                  /
                                  0 \/|x-1/7|
```

A break-point is specified at x=1/7, at which point the integrand is infinite. (For definiteness the function FST returns the value 0.0 at this point.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

D01AMF(3NAG) Foundation Library (12/10/92) D01AMF(3NAG)

D01 -- Quadrature D01AMF
D01AMF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01AMF calculates an approximation to the integral of a function f(x) over an infinite or semi-infinite interval [a,b]:

```
                                  b
                                  /
                               I= |f(x)dx
                                  /
                                  a
```

2. Specification

```
       SUBROUTINE D01AMF (F, BOUND, INF, EPSABS, EPSREL, RESULT,
      1                   ABSERR, W, LW, IW, LIW, IFAIL)
       INTEGER           INF, LW, IW(LIW), LIW, IFAIL
       DOUBLE PRECISION F, BOUND, EPSABS, EPSREL, RESULT, ABSERR,
      1                 W(LW)
       EXTERNAL          F
```

3. Description

D01AMF is based on the QUADPACK routine QAGI (Piessens et al [3]) [0,1] using one of the identities:

```
                a            1
                /            / (   1-t) 1
                |     f(x)dx= |f(a- ---) --dt
                /            / (    t )  2
                -infty       0          t

                 infty       1
                 /           / (   1-t) 1
                 |    f(x)dx= |f(a+ ---) --dt
                 /           / (    t )  2
                 a           0          t
```

```
infty         infty               1
/             /                   /[ ( 1-t)  ( -1+t)] 1
|     f(x)dx= |    (f(x)+f(-x))dx= |[f( ---)+f( ----)] --dt
/             /                   /[ (  t )  (  t  )]  2
-infty        0                   0                  t
```

where a represents a finite integration limit. An adaptive procedure, based on the Gauss seven-point and Kronrod 15-point rules, is then employed on the transformed integral. The algorithm, described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The local error estimation is described by Piessens et al [3].

4. References

[1] De Doncker E (1978) An Adaptive Extrapolation Algorithm for Automatic Integration. Signum Newsletter. 13 (2) 12--18.

[2] Malcolm M A and Simpson R B (1976) Local Versus Global Strategies for Adaptive Quadrature. ACM Trans. Math. Softw. 1 129--146.

[3] Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D (1983) QUADPACK, A Subroutine Package for Automatic Integration. Springer-Verlag.

[4] Wynn P (1956) On a Device for Computing the e (S) Transformation. Math. Tables Aids Comput. 10 91--96.
                                                  m n

5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure

On entry: the point at which the integrand f must be evaluated.

```
Its specification is:

       DOUBLE PRECISION FUNCTION F (X)
       DOUBLE PRECISION X

 1:  X -- DOUBLE PRECISION                                  Input
     On entry: the point at which the integrand f must be
     evaluated.
F must be declared as EXTERNAL in the (sub)program from
which D01AMF is called. Parameters denoted as Input
must not be changed by this procedure.

2:  BOUND -- DOUBLE PRECISION                               Input
    On entry: the finite limit of the integration range (if
    present). BOUND is not used if the interval is doubly
    infinite.

3:  INF -- INTEGER                                          Input
    On entry: indicates the kind of integration range:
         if INF =1, the range is [BOUND, +infty)

         if INF =-1, the range is (-infty, BOUND]

         if INF =+2, the range is (-infty, +infty).
     Constraint: INF =-1, 1 or 2.

4:  EPSABS -- DOUBLE PRECISION                              Input
    On entry: the absolute accuracy required. If EPSABS is
    negative, the absolute value is used. See Section 7.

5:  EPSREL -- DOUBLE PRECISION                              Input
    On entry: the relative accuracy required. If EPSREL is
    negative, the absolute value is used. See Section 7.

6:  RESULT -- DOUBLE PRECISION                             Output
    On exit: the approximation to the integral I.

7:  ABSERR -- DOUBLE PRECISION                             Output
    On exit: an estimate of the modulus of the absolute error,
    which should be an upper bound for |I-RESULT|.

8:  W(LW) -- DOUBLE PRECISION array                        Output
    On exit: details of the computation, as described in
    Section 8.

9:  LW -- INTEGER                                           Input
    On entry:
    the dimension of the array W as declared in the (sub)program
    from which D01AMF is called.
```

```
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which the interval
     of integration may be divided by the routine. The number of
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: a value
     in the range 800 to 2000 is adequate for most problems.
     Constraint: LW >= 4.

10:  IW(LIW) -- INTEGER array                                Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

11:  LIW -- INTEGER                                           Input
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01AMF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW. Suggested
     value: LIW = LW/4. Constraint: LIW >= 1.

12:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.
     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the requested accuracy
     requirements being achieved. Look at the integrand in order
     to determine the integration difficulties. If the position
     of a local difficulty within the interval can be determined
     (e.g. a singularity of the integrand or its derivative, a
     peak, a discontinuity, etc) you will probably gain from
     splitting up the interval at this point and calling D01AMF
     on the infinite subrange and an appropriate integrator on
     the finite subrange. Alternatively, consider relaxing the
```

```
     accuracy requirements specified by EPSABS and EPSREL, or
     increasing the amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. The error may be underestimated. Consider
     requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
     The requested tolerance cannot be achieved, because the
     extrapolation does not increase the accuracy satisfactorily;
     the returned result is the best which can be obtained.  The
     same advice applies as in the case of IFAIL = 1.

IFAIL= 5
     The integral is probably divergent, or slowly convergent. It
     must be noted that divergence can also occur with any other
     non-zero value of IFAIL.

IFAIL= 6
     On entry LW < 4,

     or       LIW < 1,

     or       INF /= -1, 1 or 2.
     Please note that divergence can occur with any non-zero
     value of IFAIL.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                 tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR, which,
in normal circumstances, satisfies

                    |I-RESULT|<=ABSERR<=tol.
```

8. Further Comments

The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AMF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r denote the approximation to the value of the integral over the sub-interval $[a_i, b_i]$ in the partition of [a,b] and e_i be the corresponding absolute error estimate. Then, $\int_{a_i}^{b_i} f(x)dx \tilde{=} r_i$ and RESULT= $\sum_{i=1}^{n} r_i$ unless D01AMF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of n is returned in IW(1), and the values a_i , b_i , e_i and r_i are stored consecutively in the array W, that is:

a_i =W(i),b_i =W(n+i),e_i =W(2n+i) and r_i =W(3n+i).

Note: that this information applies to the integral transformed to (0,1) as described in Section 3, not to the original integral.

9. Example

To compute

$$\int_0^{\infty} \frac{1}{(x+1)\sqrt{x}}\,dx.$$

The exact answer is (pi).

The example program is not reproduced here. The source code for

```
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     D01ANF(3NAG)      Foundation Library (12/10/92)      D01ANF(3NAG)

          D01 -- Quadrature                                          D01ANF
                  D01ANF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          D01ANF calculates an approximation to the sine or the cosine
          transform of a function g over [a,b]:

             b                                  b
             /                                  /
          I= |g(x)||,sin((omega)x)||,dx  or  I= |g(x)||,cos((omega)x)||,dx
             /                                  /
             a                                  a

          (for a user-specified value of (omega)).

          2. Specification

                SUBROUTINE D01ANF (G, A, B, OMEGA, KEY, EPSABS, EPSREL,
               1                   RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
                INTEGER          KEY, LW, IW(LIW), LIW, IFAIL
                DOUBLE PRECISION G, A, B, OMEGA, EPSABS, EPSREL, RESULT,
               1                 ABSERR, W(LW)
                EXTERNAL         G

          3. Description

          D01ANF is based upon the QUADPACK routine QFOUR (Piessens et al
          [3]). It is an adaptive routine, designed to integrate a function
          of the form g(x)w(x), where w(x) is either sin((omega)x) or
          cos((omega)x). If a sub-interval has length

                                          -l
                                   L=|b-a|2

          then the integration over this sub-interval is performed by means
          of a modified Clenshaw-Curtis procedure (Piessens and Branders
          [2]) if L(omega)>4 and l<=20. In this case a Chebyshev-series
```

approximation of degree 24 is used to approximate g(x), while an error estimate is computed from this approximation together with that obtained using Chebyshev-series of degree 12. If the above conditions do not hold then Gauss 7-point and Kronrod 15-point rules are used. The algorithm, described in [3], incorporates a global acceptance criterion (as defined in Malcolm and Simpson [1]) together with the (epsilon)-algorithm Wynn [4] to perform extrapolation. The local error estimation is described in [3].

4. References

[1] Malcolm M A and Simpson R B (1976) Local Versus Global Strategies for Adaptive Quadrature. ACM Trans. Math. Softw. 1 129--146.

[2] Piessens R and Branders M (1975) Algorithm 002. Computation of Oscillating Integrals. J. Comput. Appl. Math. 1 153--164.

[3] Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D (1983) QUADPACK, A Subroutine Package for Automatic Integration. Springer-Verlag.

[4] Wynn P (1956) On a Device for Computing the e (S) Transformation. Math. Tables Aids Comput. 10 91--96.
m n

5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure
G must return the value of the function g at a given point.

Its specification is:

```
DOUBLE PRECISION FUNCTION G (X)
DOUBLE PRECISION X
```

1: X -- DOUBLE PRECISION Input
On entry: the point at which the function g must be evaluated.

G must be declared as EXTERNAL in the (sub)program from which D01ANF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input
On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION Input
On entry: the upper limit of integration, b. It is not necessary that a<b.

```
 4:  OMEGA -- DOUBLE PRECISION                                  Input
     On entry: the parameter (omega) in the weight function of
     the transform.

 5:  KEY -- INTEGER                                             Input
     On entry: indicates which integral is to be computed:
          if KEY = 1, w(x)=cos((omega)x);

          if KEY = 2, w(x)=sin((omega)x).
      Constraint: KEY = 1 or 2.

 6:  EPSABS -- DOUBLE PRECISION                                 Input
     On entry: the absolute accuracy required. If EPSABS is
     negative, the absolute value is used. See Section 7.

 7:  EPSREL -- DOUBLE PRECISION                                 Input
     On entry: the relative accuracy required. If EPSREL is
     negative, the absolute value is used. See Section 7.

 8:  RESULT -- DOUBLE PRECISION                                 Output
     On exit: the approximation to the integral I.

 9:  ABSERR -- DOUBLE PRECISION                                 Output
     On exit: an estimate of the modulus of the absolute error,
     which should be an upper bound for |I-RESULT|.

10:  W(LW) -- DOUBLE PRECISION array                            Output
     On exit: details of the computation, as described in
     Section 8.

11:  LW -- INTEGER                                              Input
     On entry:
     the dimension of the array W as declared in the (sub)program
     from which D01ANF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which the interval
     of integration may be divided by the routine. The number of
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: a value
     in the range 800 to 2000 is adequate for most problems.
     Constraint: LW >= 4.

12:  IW(LIW) -- INTEGER array                                   Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

13:  LIW -- INTEGER                                             Input
     On entry:
     the dimension of the array IW as declared in the
```

```
     (sub)program from which D01ANF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW/2. Suggested
     value: LIW = LW/2. Constraint: LIW >= 2.

14:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requested
     being achieved. Look at the integrand in order to determine
     the integration difficulties. If the position of a local
     difficulty within the interval can be determined (e.g. a
     singularity of the integrand or its derivative, a peak, a
     discontinuity, etc) you will probably gain from splitting up
     the interval at this point and calling the integrator on the
     subranges. If necessary, another integrator, which is
     designed for handling the type of difficulty involved, must
     be used. Alternatively consider relaxing the accuracy
     requirements specified by EPSABS and EPSREL, or increasing
     amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. The error may be underestimated. Consider
     requesting less accuracy.

IFAIL= 3
     Extremely bad local behaviour of g(x) causes a very strong
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.
```

```
IFAIL= 4
     The requested tolerance cannot be achieved because the
     extrapolation does not increase the accuracy satisfactorily;
     the returned result is the best which can be obtained.  The
     same advice applies as in the case of IFAIL = 1.

IFAIL= 5
     The integral is probably divergent, or slowly convergent. It
     must be noted that divergence can occur with any non-zero
     value of IFAIL.

IFAIL= 6
     On entry KEY < 1,

     or       KEY > 2.

IFAIL= 7
     On entry LW < 4,

     or       LIW < 2.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                 tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
tolerances. Moreover it returns the quantity ABSERR, which, in
normal circumstances, satisfies

                    |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and on the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01ANF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
```

```
the value of the integral over the sub-interval [a ,b ] in the
                                                   i  i
partition of [a,b] and e  be the corresponding absolute error
                        i
                b
                 i                          n
                /                           --
estimate. Then, | g(x)w(x)dx~=r  and RESULT= >  r  unless D01ANF
                /             i             --  i
                a                           i=1
                 i
terminates while testing for divergence of the integral (see
Piessens et al [3] Section 3.4.3). In this case, RESULT (and
ABSERR) are taken to be the values returned from the
extrapolation process. The value of n is returned in IW(1), and
the values a , b , e  and r  are stored consecutively in the
            i   i   i      i
array W, that is:

     a =W(i),
      i

     b =W(n+i),
      i

     e =W(2n+i) and
      i

     r =W(3n+i).
      i

9. Example

To compute

                       1
                       /
                       |ln(x)sin(10(pi)x)dx.
                       /
                       0

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

D01APF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01APF is an adaptive integrator which calculates an approximation to the integral of a function g(x)w(x) over a finite interval [a,b]:

$$I = \int_a^b g(x)w(x)\,dx$$

where the weight function w has end-point singularities of algebraico-logarithmic type.

2. Specification

```
 SUBROUTINE D01APF (G, A, B, ALFA, BETA, KEY, EPSABS,
1                   EPSREL, RESULT, ABSERR, W, LW, IW, LIW,
2                   IFAIL)
 INTEGER          KEY, LW, IW(LIW), LIW, IFAIL
 DOUBLE PRECISION G, A, B, ALFA, BETA, EPSABS, EPSREL,
1                 RESULT, ABSERR, W(LW)
 EXTERNAL         G
```

3. Description

D01APF is based upon the QUADPACK routine QAWSE (Piessens et al [3]) and integrates a function of the form g(x)w(x), where the weight function w(x) may have algebraico-logarithmic singularities at the end-points a and/or b. The strategy is a modification of that in D01AKF. We start by bisecting the original interval and applying modified Clenshaw-Curtis integration of orders 12 and 24 to both halves. Clenshaw-Curtis integration is then used on all sub-intervals which have a or b as one of their end-points (Piessens et al [2]). On the other sub-intervals Gauss-Kronrod (7-15 point) integration is carried out.

A 'global' acceptance criterion (as defined by Malcolm and Simpson [1]) is used. The local error estimation control is described by Piessens et al [3].

```
4. References

[1]   Malcolm M A and Simpson R B (1976) Local Versus Global
      Strategies for Adaptive Quadrature. ACM Trans. Math. Softw.
      1 129--146.

[2]   Piessens R, Mertens I and Branders M (1974) Integration of
      Functions having End-point Singularities. Angewandte
      Informatik. 16 65--68.

[3]   Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D
      (1983) QUADPACK, A Subroutine Package for Automatic
      Integration. Springer-Verlag.

5. Parameters

 1:  G -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                                 External Procedure
     G must return the value of the function g at a given point
     X.

     Its specification is:

            DOUBLE PRECISION FUNCTION G (X)
            DOUBLE PRECISION X

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the point at which the function g must be
          evaluated.
     G must be declared as EXTERNAL in the (sub)program from
     which D01APF is called. Parameters denoted as Input
     must not be changed by this procedure.

 2:  A -- DOUBLE PRECISION                                  Input
     On entry: the lower limit of integration, a.

 3:  B -- DOUBLE PRECISION                                  Input
     On entry: the upper limit of integration, b. Constraint: B
     > A.

 4:  ALFA -- DOUBLE PRECISION                               Input
     On entry: the parameter (alpha) in the weight function.
     Constraint: ALFA>-1.

 5:  BETA -- DOUBLE PRECISION                               Input
     On entry: the parameter (beta) in the weight function.
     Constraint: BETA>-1.

 6:  KEY -- INTEGER                                         Input
     On entry: indicates which weight function is to be used:
```

```
                          (alpha)     (beta)
      if KEY = 1, w(x)=(x-a)       (b-x)

                          (alpha)     (beta)
      if KEY = 2, w(x)=(x-a)       (b-x)      ln(x-a)

                          (alpha)     (beta)
      if KEY = 3, w(x)=(x-a)       (b-x)      ln(b-x)

                          (alpha)     (beta)
      if KEY = 4, w(x)=(x-a)       (b-x)     ln(x-a)ln(b-x)

    Constraint: KEY = 1, 2, 3 or 4

 7: EPSABS -- DOUBLE PRECISION                                Input
    On entry: the absolute accuracy required. If EPSABS is
    negative, the absolute value is used. See Section 7.

 8: EPSREL -- DOUBLE PRECISION                                Input
    On entry: the relative accuracy required. If EPSREL is
    negative, the absolute value is used. See Section 7.

 9: RESULT -- DOUBLE PRECISION                               Output
    On exit: the approximation to the integral I.

10: ABSERR -- DOUBLE PRECISION                               Output
    On exit: an estimate of the modulus of the absolute error,
    which should be an upper bound for |I-RESULT|.

11: W(LW) -- DOUBLE PRECISION array                          Output
    On exit: details of the computation, as described in
    Section 8.

12: LW -- INTEGER                                            Input
    On entry:
    the dimension of the array W as declared in the (sub)program
    from which D01APF is called.
    The value of LW (together with that of LIW below) imposes a
    bound on the number of sub-intervals into which the interval
    of integration may be divided by the routine. The number of
    sub-intervals cannot exceed LW/4. The more difficult the
    integrand, the larger LW should be. Suggested value: LW =
    800 to 2000 is adequate for most problems. Constraint: LW >=
    8.

13: IW(LIW) -- INTEGER array                                 Output
    On exit: IW(1) contains the actual number of sub-intervals
    used. The rest of the array is used as workspace.

14: LIW -- INTEGER                                           Input
```

```
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01APF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW. Suggested
     value: LIW = LW/4. Constraint: LIW >= 2.

15:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requirements
     being achieved. Look at the integrand in order to determine
     the integration difficulties. If the position of a
     discontinuity or a singularity of algebraico-logarithmic
     type within the interval can be determined, the interval
     must be split up at this point and the integrator called on
     the subranges. If necessary, another integrator, which is
     designed for handling the difficulty involved, must be used.
     Alternatively consider relaxing the accuracy requirements
     specified by EPSABS and EPSREL, or increasing the amount of
     workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. Consider  requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.
```

```
IFAIL= 4
     On entry B <= A,

     or       ALFA <= -1,

     or       BETA <= -1,

     or       KEY < 1,

     or       KEY > 4.

IFAIL= 5
     On entry LW < 8,

     or       LIW < 2.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                 tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances.

Moreover it returns the quantity ABSERR which, in normal
circumstances, satisfies:

                    |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and on the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01APF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
the value of the integral over the sub-interval [a ,b ] in the
                                                  i  i
partition of [a,b] and e  be the corresponding absolute error
```

```
                                  i
                          b
                           i                          n
                          /                           --
estimate. Then,           | f(x)w(x)dx~=r  and RESULT= >  r . The value of
                          /              i            --  i
                          a                           i=1
                           i
n is returned in IW(1), and the values a , b , e  and r  are
                                                i   i   i      i
stored consecutively in the array W, that is:

     a =W(i),
      i

     b =W(n+i),
      i

     e =W(2n+i),
      i

     r =W(3n+i).
      i

9. Example

To compute:

                          1
                          /
                          |ln(x)cos(10(pi)x)dx  and
                          /
                          0

                               1
                               / sin(10x)
                               | --------dx.
                               /
                               0 \/x(1-x)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

D01AQF calculates an approximation to the Hilbert transform of a
function g(x) over [a,b]:

                              b
                              / g(x)
                           I= | ----dx
                              / x-c
                              a

for user-specified values of a, b and c.

2. Specification

       SUBROUTINE D01AQF (G, A, B, C, EPSABS, EPSREL, RESULT,
      1                   ABSERR, W, LW, IW, LIW, IFAIL)
       INTEGER          LW, IW(LIW), LIW, IFAIL
       DOUBLE PRECISION G, A, B, C, EPSABS, EPSREL, RESULT,
      1                 ABSERR, W(LW)
       EXTERNAL         G

3. Description

D01AQF is based upon the QUADPACK routine QAWC (Piessens et al
[3]) and integrates a function of the form g(x)w(x), where the
weight function

                                    1
                             w(x)= ---
                                   x-c

is that of the Hilbert transform. (If a<c<b the integral has to
be interpreted in the sense of a Cauchy principal value.) It is
an adaptive routine which employs a 'global' acceptance criterion
(as defined by Malcolm and Simpson [1]). Special care is taken to
ensure that c is never the end-point of a sub-interval (Piessens
et al[2]). On each sub-interval (c ,c ) modified Clenshaw-Curtis
                                 1  2
integration of orders 12 and 24 is performed if c -d<=c<=c +d
                                                 1        2
where d=(c -c )/20. Otherwise the Gauss 7-point and Kronrod 15-
          2  1
point rules are used. The local error estimation is described by
```

```
Piessens et al [3].

4. References

[1]   Malcolm M A and Simpson R B (1976) Local Versus Global
      Strategies for Adaptive Quadrature. ACM Trans. Math. Softw.
      1 129--146.

[2]   Piessens R, Van Roy-Branders M and Mertens I (1976) The
      Automatic Evaluation of Cauchy Principal Value Integrals.
      Angewandte Informatik. 18 31--35.

[3]   Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D
      (1983) QUADPACK, A Subroutine Package for Automatic
      Integration. Springer-Verlag.

5. Parameters

 1:  G -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                                 External Procedure
     G must return the value of the function g at a given point.

     Its specification is:

            DOUBLE PRECISION FUNCTION G (X)
            DOUBLE PRECISION X

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the point at which the function g must be
          evaluated.
     G must be declared as EXTERNAL in the (sub)program from
     which D01AQF is called. Parameters denoted as Input
     must not be changed by this procedure.

 2:  A -- DOUBLE PRECISION                                  Input
     On entry: the lower limit of integration, a.

 3:  B -- DOUBLE PRECISION                                  Input
     On entry: the upper limit of integration, b. It is not
     necessary that a<b.

 4:  C -- DOUBLE PRECISION                                  Input
     On entry: the parameter c in the weight function.
     Constraint: C must not equal A or B.

 5:  EPSABS -- DOUBLE PRECISION                             Input
     On entry: the absolute accuracy required. If EPSABS is
     negative, the absolute value is used. See Section 7.

 6:  EPSREL -- DOUBLE PRECISION                             Input
```

```
     On entry: the relative accuracy required. If EPSREL is
     negative, the absolute value is used. See Section 7.

 7:  RESULT -- DOUBLE PRECISION                              Output
     On exit: the approximation to the integral I.

 8:  ABSERR -- DOUBLE PRECISION                              Output
     On exit: an estimate of the modulus of the absolute error,
     which should be an upper bound for |I-RESULT|.

 9:  W(LW) -- DOUBLE PRECISION array                         Output
     On exit: details of the computation, as described in
     Section 8.

10:  LW -- INTEGER                                           Input
     On entry:
     the dimension of the array W as declared in the (sub)program
     from which D01AQF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which the interval
     of integration may be divided by the routine. The number of
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: LW =
     800 to 2000 is adequate for most problems. Constraint: LW >=
     4.

11:  IW(LIW) -- INTEGER array                                Output
     On exit: IW(1) contains the actual number of sub-intervals
     used. The rest of the array is used as workspace.

12:  LIW -- INTEGER                                          Input
     On entry:
     the dimension of the array IW as declared in the
     (sub)program from which D01AQF is called.
     The number of sub-intervals into which the interval of
     integration may be divided cannot exceed LIW. Suggested
     value: LIW = LW/4. Constraint: LIW >= 1.

13:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.
```

```
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requirements
     being achieved. Look at the integrand in order to determine
     the integration difficulties. Another integrator which is
     designed for handling the type of difficulty involved, must
     be used. Alternatively consider relaxing the accuracy
     requirements specified by EPSABS and EPSREL, or increasing
     the workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. Consider  requesting less accuracy.

IFAIL= 3
     Extremely bad local behaviour of g(x) causes a very strong
     subdivision around one (or more) points of the interval. The
     same advice applies as in the case of IFAIL = 1.

IFAIL= 4
     On entry C = A or C = B.

IFAIL= 5
     On entry LW < 4,

     or       LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                        |I-RESULT|<=tol,

where

                 tol=max{|EPSABS|,|EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which,
in normal circumstances satisfies:
```

```
                    |I-RESULT|<=ABSERR<=tol.

8. Further Comments

The time taken by the routine depends on the integrand and on the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01AQF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r  denote the approximation to
                                   i
the value of the integral over the sub-interval [a ,b ] in the
                                                 i  i
partition of [a,b] and e  be the corresponding absolute error
                        i
                b
                 i                          n
                /                           --
estimate. Then, | g(x)w(x)dx~=r  and RESULT= >  r . The value of
                /              i            --  i
                a                           i=1
                 i
n is returned in IW(1), and the values a , b , e  and r  are
                                        i   i   i      i
stored consecutively in the array W, that is:

     a =W(i),
      i

     b =W(n+i),
      i

     e =W(2n+i) and
      i

     r =W(3n+i).
      i

9. Example

To compute the Cauchy principal value of

                    1
                    /         dx
                    |  ----------------.
                    /    2     2     1
                    -1 (x +0.01 )(x- -)
```

```
                                                    2

         The example program is not reproduced here. The source code for
         all example programs is distributed with the NAG Foundation
         Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    D01ASF(3NAG)      Foundation Library (12/10/92)      D01ASF(3NAG)

         D01 -- Quadrature                                          D01ASF
                 D01ASF -- NAG Foundation Library Routine Document

         Note: Before using this routine, please read the Users' Note for
         your implementation to check implementation-dependent details.
         The symbol (*) after a NAG routine name denotes a routine that is
         not included in the Foundation Library.

         1. Purpose

         D01ASF calculates an approximation to the sine or the cosine
         transform of a function g over [a,infty):


            infty
            /
         I= |    g(x)||,sin((omega)x)dx  or  I=
            /
            a
          infty
          /
          |    g(x)||,cos((omega)x)dx
          /
          a

         (for a user-specified value of (omega)).

         2. Specification

                SUBROUTINE D01ASF (G, A, OMEGA, KEY, EPSABS, RESULT,
               1                   ABSERR, LIMLST, LST, ERLST, RSLST,
               2                   IERLST, W, LW, IW, LIW, IFAIL)
                INTEGER           KEY, LIMLST, LST, IERLST(LIMLST), LW, IW
               1                  (LIW), LIW, IFAIL
                DOUBLE PRECISION G, A, OMEGA, EPSABS, RESULT, ABSERR, ERLST
               1                  (LIMLST), RSLST(LIMLST), W(LW)
                EXTERNAL          G

         3. Description
```

D01ASF is based upon the QUADPACK routine QAWFE (Piessens et al [2]). It is an adaptive routine, designed to integrate a function of the form g(x)w(x) over a semi-infinite interval, where w(x) is either sin((omega)x) or cos((omega)x). Over successive intervals

$$C_k = [a+(k-1)c, a+kc], \quad k=1,2,\ldots,LST$$

integration is performed by the same algorithm as is used by D01ANF. The intervals C_k are of constant length

$$c=\{2[|(omega)|]+1\}(pi)/|(omega)|, \quad (omega)/=0$$

where [|(omega)|] represents the largest integer less than or equal to |(omega)|. Since c equals an odd number of half periods, the integral contributions over succeeding intervals will alternate in sign when the function g is positive and monotonically decreasing over [a,infty). The algorithm, described by [2], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [1]) together with the (epsilon)-algorithm (Wynn [3]) to perform extrapolation. The local error estimation is described by Piessens et al [2].

If (omega)=0 and KEY = 1, the routine uses the same algorithm as D01AMF (with EPSREL = 0.0).

In contrast to the other routines in Chapter D01, D01ASF works only with a user-specified absolute error tolerance (EPSABS). Over the interval C_k it attempts to satisfy the absolute accuracy requirement

$$EPSA_k = U_k * EPSABS$$

where $U_k = (1-p)p^{k-1}$, for k=1,2,... and p=0.9.

However, when difficulties occur during the integration over the kth sub-interval C_k such that the error flag IERLST(k) is non-zero, the accuracy requirement over subsequent intervals is relaxed. See Piessens et al [2] for more details.

4. References

[1] Malcolm M A and Simpson R B (1976) Local Versus Global Strategies for Adaptive Quadrature. ACM Trans. Math. Softw. 1 129--146.

[2] Piessens R, De Doncker-Kapenga E, Uberhuber C and Kahaner D (1983) QUADPACK, A Subroutine Package for Automatic Integration. Springer-Verlag.

[3] Wynn P (1956) On a Device for Computing the e (S) Transformation. Math. Tables Aids Comput. 10 91--96.
m n

5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure
G must return the value of the function g at a given point.

Its specification is:

```
DOUBLE PRECISION FUNCTION G (X)
DOUBLE PRECISION X
```

1: X -- DOUBLE PRECISION Input
On entry: the point at which the function g must be evaluated.

G must be declared as EXTERNAL in the (sub)program from which D01ASF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input
On entry: the lower limit of integration, a.

3: OMEGA -- DOUBLE PRECISION Input
On entry: the parameter (omega) in the weight function of the transform.

4: KEY -- INTEGER Input
On entry: indicates which integral is to be computed:
if KEY = 1, w(x)=cos((omega)x);

if KEY = 2, w(x)=sin((omega)x).
Constraint: KEY = 1 or 2.

5: EPSABS -- DOUBLE PRECISION Input
On entry: the absolute accuracy requirement. If EPSABS is negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION Output
On exit: the approximation to the integral I.

7: ABSERR -- DOUBLE PRECISION Output
On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

```
 8:  LIMLST -- INTEGER                                          Input
     On entry: an upper bound on the number of intervals C
                                                             k
     needed for the integration. Suggested value: LIMLST = 50 is
     adequate for most problems. Constraint: LIMLST >= 3.

 9:  LST -- INTEGER                                             Output
     On exit: the number of intervals C  actually used for the
                                      k
     integration.

10:  ERLST(LIMLST) -- DOUBLE PRECISION array                    Output
     On exit: ERLST(k) contains the error estimate corresponding
     to the integral contribution over the interval C , for
                                                     k
     k=1,2,...,LST.

11:  RSLST(LIMLST) -- DOUBLE PRECISION array                    Output
     On exit: RSLST(k) contains the integral contribution over
     the interval C  for k=1,2,...,LST.
                  k

12:  IERLST(LIMLST) -- INTEGER array                            Output
     On exit: IERLST(k) contains the error flag corresponding to
     RSLST(k), for k=1,2,...,LST. See Section 6.

13:  W(LW) -- DOUBLE PRECISION array                         Workspace

14:  LW -- INTEGER                                              Input
     On entry:
     the dimension of the array W as declared in the (sub)program
     from which D01ASF is called.
     The value of LW (together with that of LIW below) imposes a
     bound on the number of sub-intervals into which each
     interval C  may be divided by the routine. The number of
               k
     sub-intervals cannot exceed LW/4. The more difficult the
     integrand, the larger LW should be. Suggested value: a value
     in the range 800 to 2000 is adequate for most problems.
     Constraint: LW >= 4.

15:  IW(LIW) -- INTEGER array                                   Output
     On exit: IW(1) contains the maximum number of sub-intervals
     actually used for integrating over any of the intervals C .
                                                              k
     The rest of the array is used as workspace.

16:  LIW -- INTEGER                                             Input
     On entry:
```

```
     the dimension of the array IW as declared in the
     (sub)program from which D01ASF is called.
     The number of sub-intervals into which each interval C  may
                                                           k
     be divided cannot exceed LIW/2. Suggested value: LIW = LW/2.
     Constraint: LIW >= 2.

17:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     The maximum number of subdivisions allowed with the given
     workspace has been reached without the accuracy requirements
     being achieved. Look at the integrand in order to determine
     the integration difficulties. If the position of a local
     difficulty within the interval can be determined (e.g. a
     singularity of the integrand or its derivative, a peak, a
     discontinuity, etc) you will probably gain from splitting up
     the interval at this point and calling D01ASF on the
     infinite subrange and an appropriate integrator on the
     finite subrange. Alternatively, consider relaxing the
     accuracy requirements specified by EPSABS or increasing the
     amount of workspace.

IFAIL= 2
     Round-off error prevents the requested tolerance from being
     achieved. The error may be underestimated. Consider
     requesting less accuracy.

IFAIL= 3
     Extremely bad local integrand behaviour causes a very strong
     subdivision around one (or more) points of the interval.
     The same advice applies as in the case of IFAIL = 1.
```

```
IFAIL= 4
     The requested tolerance cannot be achieved, because the
     extrapolation does not increase the accuracy satisfactorily;
     the returned result is the best which can be obtained.  The
     same advice applies as in the case of IFAIL = 1.

IFAIL= 5
     The integral is probably divergent, or slowly convergent.
     Please note that divergence can occur with any non-zero
     value of IFAIL.

IFAIL= 6
     On entry KEY < 1,

     or       KEY > 2,

     or       LIMLST < 3.

IFAIL= 7
     Bad integration behaviour occurs within one or more of the
     intervals C . Location and type of the difficulty involved
                k
     can be determined from the vector IERLST (see below).

IFAIL= 8
     Maximum number of intervals C  (= LIMLST) allowed has been
                                  k
     achieved. Increase the value of LIMLST to allow more cycles.

IFAIL= 9
     The extrapolation table constructed for convergence
     acceleration of the series formed by the integral
     contribution over the intervals C , does not converge to the
                                      k
     required accuracy.

IFAIL= 10
     On entry LW < 4,

     or       LIW < 2.

In the cases IFAIL = 7, 8 or 9, additional information about the
cause of the error can be obtained from the array IERLST, as
follows:

IERLST(k)=1
      The maximum number of subdivisions = min(LW/4,LIW/2) has
      been achieved on the kth interval.
```

```
IERLST(k)=2
      Occurrence of round-off error is detected and prevents the
      tolerance imposed on the kth interval from being achieved.

IERLST(k)=3
      Extremely bad integrand behaviour occurs at some points of
      the kth interval.

IERLST(k)=4
      The integration procedure over the kth interval does not
      converge (to within the required accuracy) due to round-off
      in the extrapolation procedure invoked on this interval. It
      is assumed that the result on this interval is the best
      which can be obtained.

IERLST(k)=5
      The integral over the kth interval is probably divergent or
      slowly convergent. It must be noted that divergence can
      occur with any other value of IERLST(k).

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

                     |I-RESULT|<=|EPSABS|,

where EPSABS is the user-specified absolute error tolerance.
Moreover, it returns the quantity ABSERR, which, in normal
circumstances, satisfies

                 |I-RESULT|<=ABSERR<=|EPSABS|.

8. Further Comments

None.

9. Example

To compute

                   infty
                   /      1
                   |     ---cos((pi)x/2)dx.
                   /
                   0     \/x

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
D01BBF(3NAG)                D01BBF                  D01BBF(3NAG)

D01 -- Quadrature                                           D01BBF
        D01BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine
has been enhanced for use with AXIOM and is slightly different to
that offered in the standard version of the Foundation Library.

1. Purpose

D01BBF returns the weights and abscissae appropriate to a
Gaussian quadrature formula with a specified number of abscissae.
The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-
Laguerre and Gauss-Hermite.

2. Specification

       SUBROUTINE D01BBF (A, B, ITYPE, N, WEIGHT, ABSCIS, GTYPE,
      1                   IFAIL)
       INTEGER          ITYPE, N, GTYPE, IFAIL
       DOUBLE PRECISION A, B, WEIGHT(N), ABSCIS(N)

3. Description

This routine returns the weights and abscissae for use in the
Gaussian quadrature of a function f(x). The quadrature takes the
form

                              n
                              --
                          S= >  w f(x )
                              --  i   i
                              i=1

where w  are the weights and x  are the abscissae (see Davis and
       i                      i
Rabinowitz [1], Froberg [2], Ralston [3] or Stroud and Secrest
[4]).

Weights and abscissae are available for Gauss-Legendre, Gauss-
Rational, Gauss-Laguerre and Gauss-Hermite quadrature, and for a
```

selection of values of n (see Section 5).

(a) Gauss-Legendre Quadrature:

$$\tilde{S} = \int_a^b f(x)\,dx$$

where a and b are finite and it will be exact for any function of the form

$$f(x) = \sum_{i=0}^{2n-1} c_i x^i$$

(b) Gauss-Rational quadrature:

$$\tilde{S} = \int_a^{\infty} f(x)\,dx \quad (a+b>0) \quad \text{or} \quad \tilde{S} = \int_{-\infty}^{a} f(x)\,dx \quad (a+b<0)$$

and will be exact for any function of the form

$$f(x) = \sum_{i=2}^{2n+1} \frac{c_i}{(x+b)^i} = \frac{\sum_{i=0}^{2n-1} c_i (x+b)^{2n+1-i}}{(x+b)^{2n+1}}$$

(c) Gauss-Laguerre quadrature, adjusted weights option:

$$\tilde{S} = \int_a^{\infty} f(x)\,dx \quad (b>0<) \quad \text{or} \quad \tilde{S} = \int_{-\infty}^{a} f(x)\,dx \quad (b<0)$$

and will be exact for any function of the form

$$f(x) = e^{-bx} \sum_{i=0}^{2n-1} c_i x^i$$

(d) Gauss-Hermite quadrature, adjusted weights option:

$$\tilde{S} = \int_{-\infty}^{+\infty} f(x)\,dx$$

```
                          /
                          -infty
and will be exact for any function of the form

                             2 2n-1
                   -b(x-a)  --     i
              f(x)=e        >   c x   (b>0)
                            --   i
                            i=0
```

(e) Gauss-Laguerre quadrature, normal weights option:

```
    infty                                a
    /     -bx                            /      -bx
S~= |    e   f(x)dx  (b>0)   or   S~= |     e   f(x)dx
    /                                    /
    a                                    -infty


 (b<0)


and will be exact for any function of the form
                              2n-1
                              --     i
                       f(x)= >   c x
                              --   i
                              i=0
```

(f) Gauss-Hermite quadrature, normal weights option:

```
                         +infty        2
                         /      -b(x-a)
                    S~= |     e        f(x)dx
                         /
                         -infty
and will be exact for any function of the form:
                              2n-1
                              --     i
                       f(x)= >   c x
                              --   i
                              i=0
```

Note: that the Gauss-Legendre abscissae, with a=-1, b=+1, are the zeros of the Legendre polynomials; the Gauss-Laguerre abscissae, with a=0, b=1, are the zeros of the Laguerre polynomials; and the Gauss-Hermite abscissae, with a=0, b=1, are the zeros of the Hermite polynomials.

4. References

[1] Davis P J and Rabinowitz P (1967) Numerical Integration. Blaisdell Publishing Company. 33--52.

[2] Froberg C E (1965) Introduction to Numerical Analysis. Addison-Wesley. 181--187.

[3] Ralston A (1965) A First Course in Numerical Analysis. McGraw-Hill. 87--90.

[4] Stroud A H and Secrest D (1966) Gaussian Quadrature Formulas. Prentice-Hall.

5. Parameters

1: A -- DOUBLE PRECISION Input

2: B -- DOUBLE PRECISION Input
On entry: the quantities a and b as described in the appropriate subsection of Section 3.

3: ITYPE -- INTEGER Input
On entry: indicates the type of weights for Gauss-Laguerre or Gauss-Hermite quadrature (see Section 3):

if ITYPE = 1, adjusted weights will be returned;

if ITYPE = 0, normal weights will be returned.

Constraint: ITYPE = 0 or 1.

For Gauss-Legendre or Gauss-Rational quadrature, this parameter is not used.

4: N -- INTEGER Input
On entry: the number of weights and abscissae to be returned, n. Constraint: N = 1,2,3,4,5,6,8,10,12,14,16,20,24,32,48 or 64.

5: WEIGHT(N) -- DOUBLE PRECISION array Output
On exit: the N weights. For Gauss-Laguerre and Gauss-Hermite quadrature, these will be the adjusted weights if ITYPE = 1, and the normal weights if ITYPE = 0.

6: ABSCIS(N) -- DOUBLE PRECISION array Output
On exit: the N abscissae.

7: GTYPE -- INTEGER Input
On entry: The value of GTYPE indicates which quadrature formula are to be used:

```
          GTYPE = 0 for Gauss-Legendre weights and abscissae;

          GTYPE = 1 for Gauss-Rational weights and abscissae;

          GTYPE = 2 for Gauss-Laguerre weights and abscissae;

          GTYPE = 3 for Gauss-Hermite weights and abscissae.

 8:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     The N-point rule is not among those stored. If the soft fail
     option is used, the weights and abscissae returned will be
     those for the largest valid value of N less than the
     requested value, and the excess elements of WEIGHT and
     ABSCIS (i.e., up to the requested N) will be filled with
     zeros.

IFAIL= 2
     The value of A and/or B is invalid.
          Gauss-Rational: A + B = 0

          Gauss-Laguerre: B = 0

          Gauss-Hermite: B <= 0
     If the soft fail option is used the weights and abscissae
     are returned as zero.

IFAIL= 3
     Laguerre and Hermite normal weights only: underflow is
     occurring in evaluating one or more of the normal weights.
     If the soft fail option is used, the underflowing weights
     are returned as zero. A smaller value of N must be used; or
     adjusted weights should be used (ITYPE = 1). In the latter
     case, take care that underflow does not occur when
     evaluating the integrand appropriate for adjusted weights.

IFAIL=4
    GTYPE < 0 or GTYPE > 3
```

```
     7. Accuracy

     The weights and abscissae are stored for standard values of A and
     B to full machine accuracy.

     8. Further Comments

     Timing is negligible.

     9. Example

     This example program returns the abscissae and (adjusted) weights
     for the six-point Gauss-Laguerre formula.

     The example program is not reproduced here. The source code for
     all example programs is distributed with the NAG Foundation
     Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     D01FCF(3NAG)      Foundation Library (12/10/92)      D01FCF(3NAG)

          D01 -- Quadrature                                        D01FCF
                  D01FCF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          D01FCF attempts to evaluate a multi-dimensional integral (up to
          15 dimensions), with constant and finite limits, to a specified
          relative accuracy, using an adaptive subdivision strategy.

          2. Specification

                 SUBROUTINE D01FCF (NDIM, A, B, MINPTS, MAXPTS, FUNCTN,
                1                   EPS, ACC, LENWRK, WRKSTR, FINVAL, IFAIL)
                 INTEGER          NDIM, MINPTS, MAXPTS, LENWRK, IFAIL
                 DOUBLE PRECISION A(NDIM), B(NDIM), FUNCTN, EPS, ACC, WRKSTR
                1                 (LENWRK), FINVAL
                 EXTERNAL         FUNCTN

          3. Description

          The routine returns an estimate of a multi-dimensional integral
          over a hyper-rectangle (i.e., with constant limits), and also an
          estimate of the relative error. The user sets the relative
```

```
accuracy required, supplies the integrand as a function
subprogram (FUNCTN), and also sets the minimum and maximum
acceptable number of calls to FUNCTN (in MINPTS and MAXPTS).

The routine operates by repeated subdivision of the hyper-
rectangular region into smaller hyper-rectangles. In each
subregion, the integral is estimated using a seventh-degree rule,
and an error estimate is obtained by comparison with a fifth-
degree rule which uses a subset of the same points. The fourth
differences of the integrand along each co-ordinate axis are
evaluated, and the subregion is marked for possible future
subdivision in half along that co-ordinate axis which has the
largest absolute fourth difference.

If the estimated errors, totalled over the subregions, exceed the
requested relative error (or if fewer than MINPTS calls to FUNCTN
have been made), further subdivision is necessary, and is
performed on the subregion with the largest estimated error, that
subregion being halved along the appropriate co-ordinate axis.

The routine will fail if the requested relative error level has
not been attained by the time MAXPTS calls to FUNCTN have been
made; or, if the amount LENWRK of working storage is
insufficient. A formula for the recommended value of LENWRK is
given in Section 5. If a smaller value is used, and is exhausted
in the course of execution, the routine switches to a less
efficient mode of operation; only if this mode also breaks down
is insufficient storage reported.

D01FCF is based on the HALF subroutine developed by van Dooren
and de Ridder [1]. It uses a different basic rule, described by
Genz and Malik [2].

4. References

[1]   Van Dooren P and De Ridder L (1976) An Adaptive Algorithm
      for Numerical Integration over an N-dimensional Cube. J.
      Comput. Appl. Math. 2 207--217.

[2]   Genz A C and Malik A A (1980) An Adaptive Algorithm for
      Numerical Integration over an N-dimensional Rectangular
      Region. J. Comput. Appl. Math. 6 295--302.

5. Parameters

 1:  NDIM -- INTEGER                                        Input
     On entry: the number of dimensions of the integral, n.
     Constraint: 2 <= NDIM <= 15.

 2:  A(NDIM) -- DOUBLE PRECISION array                      Input
```

```
     On entry: the lower limits of integration, a , for
                                                 i
     i=1,2,...,n.

 3:  B(NDIM) -- DOUBLE PRECISION array                        Input
     On entry: the upper limits of integration, b , for
                                                 i
     i=1,2,...,n.

 4:  MINPTS -- INTEGER                                 Input/Output
     On entry: MINPTS must be set to the minimum number of
     integrand evaluations to be allowed. On exit: MINPTS
     contains the actual number of integrand evaluations used by
     D01FCF.

 5:  MAXPTS -- INTEGER                                        Input
     On entry: the maximum number of integrand evaluations to be
     allowed.
     Constraints:
              MAXPTS >= MINPTS

              MAXPTS >= (alpha),

                            NDIM       2
              where (alpha)=2    +2*NDIM +2*NDIM+1.

 6:  FUNCTN -- DOUBLE PRECISION FUNCTION, supplied by the
user.
                                              External Procedure
     FUNCTN must return the value of the integrand f at a given
     point.

     Its specification is:

            DOUBLE PRECISION FUNCTION FUNCTN (NDIM,Z)
            INTEGER          NDIM
            DOUBLE PRECISION Z(NDIM)

      1:  NDIM -- INTEGER                                     Input
          On entry: the number of dimensions of the integral, n.

      2:  Z(NDIM) -- DOUBLE PRECISION array                   Input
          On entry: the co-ordinates of the point at which the
          integrand must be evaluated.
     FUNCTN must be declared as EXTERNAL in the (sub)program
     from which D01FCF is called. Parameters denoted as
     Input must not be changed by this procedure.

 7:  EPS -- DOUBLE PRECISION                                  Input
     On entry: the relative error acceptable to the user. When
```

```
     the solution is zero or very small relative accuracy may not
     be achievable but the user may still set EPS to a reasonable
     value and check for the error exit IFAIL = 2. Constraint:
     EPS > 0.0.

 8:  ACC -- DOUBLE PRECISION                                  Output
     On exit: the estimated relative error in FINVAL.

 9:  LENWRK -- INTEGER                                         Input
     On entry:
     the dimension of the array WRKSTR as declared in the
     (sub)program from which D01FCF is called.
     Suggested value: for maximum efficiency, LENWRK >=
     (NDIM+2)*(1+MAXPTS/(alpha)) (see parameter MAXPTS for
     (alpha)).

     If LENWRK is less than this, the routine will usually run
     less efficiently and may fail. Constraint: LENWRK>=2*NDIM+4.

10:  WRKSTR(LENWRK) -- DOUBLE PRECISION array           Workspace

11:  FINVAL -- DOUBLE PRECISION                              Output
     On exit: the best estimate obtained for the integral.

12:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.
     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
     On entry NDIM < 2,

     or       NDIM > 15,

     or       MAXPTS is too small,

     or       LENWRK<2*NDIM+4,
```

```
     or       EPS <= 0.0.

IFAIL= 2
     MAXPTS was too small to obtain the required relative
     accuracy EPS. On soft failure, FINVAL and ACC contain
     estimates of the integral and the relative error, but ACC
     will be greater than EPS.

IFAIL= 3
     LENWRK was too small. On soft failure, FINVAL and ACC
     contain estimates of the integral and the relative error,
     but ACC will be greater than EPS.

7. Accuracy

A relative error estimate is output through the parameter ACC.

8. Further Comments

Execution time will usually be dominated by the time taken to
evaluate the integrand FUNCTN, and hence the maximum time that
could be taken will be proportional to MAXPTS.

9. Example

This example program estimates the integral

                        2
       1  1  1  1  4z z exp(2z z )
       /  /  /  /    1 3      1 3
       |  |  |  |  ---------------dz dz dz dz =0.575364.
       /  /  /  /             2     4  3  2  1
       0  0  0  0    (1+z +z )
                        2  4

The accuracy requested is one part in 10,000.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
D01GAF(3NAG)      Foundation Library (12/10/92)      D01GAF(3NAG)

D01 -- Quadrature                                          D01GAF
       D01GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
```

your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01GAF integrates a function which is specified numerically at four or more points, over the whole of its specified range, using third-order finite-difference formulae with error estimates, according to a method due to Gill and Miller.

2. Specification

```
SUBROUTINE D01GAF (X, Y, N, ANS, ER, IFAIL)
INTEGER          N, IFAIL
DOUBLE PRECISION X(N), Y(N), ANS, ER
```

3. Description

This routine evaluates the definite integral

$$I = \int_{x_1}^{x_n} y(x)\,dx,$$

where the function y is specified at the n-points $x_1, x_2, \ldots, x_n$, which should be all distinct, and in either ascending or descending order. The integral between successive points is calculated by a four-point finite-difference formula centred on the interval concerned, except in the case of the first and last intervals, where four-point forward and backward difference formulae respectively are employed. If n is less than 4, the routine fails. An approximation to the truncation error is integrated and added to the result. It is also returned separately to give an estimate of the uncertainty in the result. The method is due to Gill and Miller.

4. References

[1] Gill P E and Miller G F (1972) An Algorithm for the Integration of Unequally Spaced Data. Comput. J. 15 80--83.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input

```
     On entry: the values of the independent variable, i.e., the
     x ,x ,...,x . Constraint: either X(1) < X(2) <... < X(N) or
      1  2      n
     X(1) > X(2) >... > X(N).

 2:  Y(N) -- DOUBLE PRECISION array                          Input
     On entry: the values of the dependent variable y  at the
                                                     i
     points x , for i=1,2,...,n.
             i

 3:  N -- INTEGER                                            Input
     On entry: the number of points, n. Constraint: N >= 4.

 4:  ANS -- DOUBLE PRECISION                                 Output
     On exit: the estimate of the integral.

 5:  ER -- DOUBLE PRECISION                                  Output
     On exit: an estimate of the uncertainty in ANS.

 6:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     Indicates that fewer than four-points have been supplied to
     the routine.

IFAIL= 2
     Values of X are neither strictly increasing nor strictly
     decreasing.

IFAIL= 3
     Two points have the same X-value.

No error is reported arising from the relative magnitudes of ANS
and ER on return, due to the difficulty when the true answer is
zero.

7. Accuracy

No accuracy level is specified by the user before calling the
```

```
routine but on return ABS(ER) is an approximation to, but not
necessarily a bound for, |I-ANS|. If on exit IFAIL > 0, both ANS
and ER are returned as zero.

8. Further Comments

The time taken by the routine depends on the number of points
supplied, n.

In their paper, Gill and Miller [1] do not add the quantity ER to
ANS before return. However, extensive tests have shown that a
dramatic reduction in the error often results from such addition.
In other cases, it does not make an improvement, but these tend
to be cases of low accuracy in which the modified answer is not
significantly inferior to the unmodified one. The user has the
option of recovering the Gill-Miller answer by subtracting ER
from ANS on return from the routine.

9. Example

The example program evaluates the integral

                          1
                          /  4
                          | ----dx=(pi)
                          /    2
                          0 1+x

reading in the function values at 21 unequally-spaced points.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D01GBF(3NAG)      Foundation Library (12/10/92)      D01GBF(3NAG)

     D01 -- Quadrature                                          D01GBF
            D01GBF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     D01GBF returns an approximation to the integral of a function
     over a hyper-rectangular region, using a Monte Carlo method. An
```

approximate relative error estimate is also returned. This routine is suitable for low accuracy work.

2. Specification

```
      SUBROUTINE D01GBF (NDIM, A, B, MINCLS, MAXCLS, FUNCTN,
     1                   EPS, ACC, LENWRK, WRKSTR, FINEST, IFAIL)
      INTEGER          NDIM, MINCLS, MAXCLS, LENWRK, IFAIL
      DOUBLE PRECISION A(NDIM), B(NDIM), FUNCTN, EPS, ACC, WRKSTR
     1                 (LENWRK), FINEST
      EXTERNAL         FUNCTN
```

3. Description

D01GBF uses an adaptive Monte Carlo method based on the algorithm described by Lautrup [1]. It is implemented for integrals of the form:

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} f(x_1,x_2,\ldots,x_n)\,dx_n\ldots dx_2\,dx_1 .$$

Upon entry, unless LENWRK has been set to the minimum value 10*NDIM, the routine subdivides the integration region into a number of equal volume subregions. Inside each subregion the integral and the variance are estimated by means of pseudo-random sampling. All contributions are added together to produce an estimate for the whole integral and total variance. The variance along each co-ordinate axis is determined and the routine uses this information to increase the density and change the widths of the sub-intervals along each axis, so as to reduce the total variance. The total number of subregions is then increased by a factor of two and the program recycles for another iteration. The program stops when a desired accuracy has been reached or too many integral evaluations are needed for the next cycle.

4. References

[1] Lautrup B (1971) An Adaptive Multi-dimensional Integration Procedure. Proc. 2nd Coll. on Advanced Methods in Theoretical Physics, Marseille.

5. Parameters

1: NDIM -- INTEGER Input
On entry: the number of dimensions of the integral, n.

```
    Constraint: NDIM >= 1.

2:  A(NDIM) -- DOUBLE PRECISION array                        Input
    On entry: the lower limits of integration, a , for
                                                 i
    i=1,2,...,n.

3:  B(NDIM) -- DOUBLE PRECISION array                        Input
    On entry: the upper limits of integration, b , for
                                                 i
    i=1,2,...,n.

4:  MINCLS -- INTEGER                                  Input/Output
    On entry: MINCLS must be set:

    either to the minimum number of integrand evaluations to be
    allowed, in which case MINCLS >= 0;

    or to a negative value. In this case the routine assumes
    that a previous call had been made with the same parameters
    NDIM, A and B and with either the same integrand (in which
    case D01GBF continues calculation) or a similar integrand
    (in which case D01GBF begins the calculation with the
    subdivision used in the last iteration of the previous call)
    . See also WRKSTR. On exit: MINCLS contains the number of
    integrand evaluations actually used by D01GBF.

5:  MAXCLS -- INTEGER                                        Input
    On entry: the maximum number of integrand evaluations to be
    allowed. In the continuation case this is the number of new
    integrand evaluations to be allowed. These counts do not
    include zero integrand values.
    Constraints:
             MAXCLS > MINCLS,

             MAXCLS >= 4*(NDIM+1).

6:  FUNCTN -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                              External Procedure
    FUNCTN must return the value of the integrand f at a given
    point.

    Its specification is:

           DOUBLE PRECISION FUNCTION FUNCTN (NDIM, X)
           INTEGER          NDIM
           DOUBLE PRECISION X(NDIM)

     1:  NDIM -- INTEGER                                     Input
         On entry: the number of dimensions of the integral, n.
```

```
 2:  X(NDIM) -- DOUBLE PRECISION array                 Input
     On entry: the co-ordinates of the point at which the
     integrand must be evaluated.
FUNCTN must be declared as EXTERNAL in the (sub)program
from which D01GBF is called. Parameters denoted as
Input must not be changed by this procedure.

 7:  EPS -- DOUBLE PRECISION                           Input
On entry: the relative accuracy required. Constraint: EPS
>= 0.0.

 8:  ACC -- DOUBLE PRECISION                          Output
On exit: the estimated relative accuracy of FINEST.

 9:  LENWRK -- INTEGER                                 Input
On entry:
the dimension of the array WRKSTR as declared in the
(sub)program from which D01GBF is called.
For maximum efficiency, LENWRK should be about
                                  1/NDIM
              3*NDIM*(MAXCLS/4)      +7*NDIM.
If LENWRK is given the value 10*NDIM then the subroutine
uses only one iteration of a crude Monte Carlo method with
MAXCLS sample points. Constraint: LENWRK >= 10*NDIM.

10:  WRKSTR(LENWRK) -- DOUBLE PRECISION array        Input/Output
On entry: if MINCLS<0, WRKSTR must be unchanged from the
previous call of D01GBF - except that for a new integrand
WRKSTR(LENWRK) must be set to 0.0. See also MINCLS. On
exit: WRKSTR contains information about the current sub-
interval structure which could be used in later calls of
D01GBF. In particular, WRKSTR(j) gives the number of sub-
intervals used along the jth co-ordinate axis.

11:  FINEST -- DOUBLE PRECISION                      Output
On exit: the best estimate obtained for the integral.

12:  IFAIL -- INTEGER                                Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
```

```
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
     On entry NDIM < 1,

     or       MINCLS >= MAXCLS,

     or       LENWRK < 10*NDIM,

     or       MAXCLS < 4*(NDIM+1),

     or       EPS < 0.0.

IFAIL= 2
     MAXCLS was too small for D01GBF to obtain the required
     relative accuracy EPS. In this case D01GBF returns a value
     of FINEST with estimated relative error ACC, but ACC will be
     greater than EPS. This error exit may be taken before MAXCLS
     non-zero integrand evaluations have actually occurred, if
     the routine calculates that the current estimates could not
     be improved before MAXCLS was exceeded.

7. Accuracy

A relative error estimate is output through the parameter ACC.
The confidence factor is set so that the actual error should be
less than ACC 90% of the time. If a user desires a higher
confidence level then a smaller value of EPS should be used.

8. Further Comments

The running time for D01GBF will usually be dominated by the time
used to evaluate the integrand FUNCTN, so the maximum time that
could be used is approximately proportional to MAXCLS.

For some integrands, particularly those that are poorly behaved
in a small part of the integration region, D01GBF may terminate
with a value of ACC which is significantly smaller than the
actual relative error. This should be suspected if the returned
value of MINCLS is small relative to the expected difficulty of
the integral. Where this occurs, D01GBF should be called again,
but with a higher entry value of MINCLS (e.g. twice the returned
value) and the results compared with those from the previous
call.
```

```
The exact values of FINEST and ACC on return will depend (within
statistical limits) on the sequence of random numbers generated
within D01GBF by calls to G05CAF. Separate runs will produce
identical answers unless the part of the program executed prior
to calling D01GBF also calls (directly or indirectly) routines
from Chapter G05, and the series of such calls differs between
runs. If desired, the user may ensure the identity or difference
between runs of the results returned by D01GBF, by calling G05CBF
or G05CCF respectively, immediately before calling D01GBF.

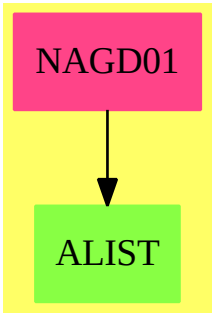
9. Example

This example program calculates the integral

       1  1  1  1  4x x exp(2x x )
       /  /  /  /    1 3      1 3
       |  |  |  |  ---------------dx dx dx dx =0.575364.
       /  /  /  /             2     1  2  3  4
       0  0  0  0    (1+x +x )
                         2  4

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.5.1 NagIntegrationPackage (NAGD01)



Exports:

d01ajf	d01akf	d01alf	d01amf	d01anf
d01apf	d01aqf	d01asf	d01bbf	d01fcf
d01gaf	d01gbf			

— package NAGD01 NagIntegrationPackage —

```
)abbrev package NAGD01 NagIntegrationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:37 1994
++ Description:
++ This package uses the NAG Library to calculate the numerical value of
++ definite integrals in one or more dimensions and to evaluate
++ weights and abscissae of integration rules.

NagIntegrationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    d01ajf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
        Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
     ++ d01ajf(a,b,epsabs,epsrel,lw,liw,ifail,f)
     ++ is a general-purpose integrator which calculates an
     ++ approximation to the integral of a function f(x) over a finite
     ++ interval [a,b]:
     ++ See \downlink{Manual Page}{manpageXXd01ajf}.
    d01akf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
        Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
     ++ d01akf(a,b,epsabs,epsrel,lw,liw,ifail,f)
     ++ is an adaptive integrator, especially suited to
     ++ oscillating, non-singular integrands, which calculates an
     ++ approximation to the integral of a function f(x) over a finite
     ++ interval [a,b]:
     ++ See \downlink{Manual Page}{manpageXXd01akf}.
    d01alf : (DoubleFloat,DoubleFloat,Integer,Matrix DoubleFloat,_
        DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
        Union(fn:FileName,fp:Asp1(F))) -> Result
     ++ d01alf(a,b,npts,points,epsabs,epsrel,lw,liw,ifail,f)
     ++ is a general purpose integrator which calculates an
     ++ approximation to the integral of a function f(x) over a finite
     ++ interval [a,b]:
     ++ See \downlink{Manual Page}{manpageXXd01alf}.
    d01amf : (DoubleFloat,Integer,DoubleFloat,DoubleFloat,_
        Integer,Integer,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
     ++ d01amf(bound,inf,epsabs,epsrel,lw,liw,ifail,f)
     ++ calculates an approximation to the integral of a function
     ++ f(x) over an infinite or semi-infinite interval [a,b]:
     ++ See \downlink{Manual Page}{manpageXXd01amf}.
    d01anf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,_
        DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
        Union(fn:FileName,fp:Asp1(G))) -> Result
     ++ d01anf(a,b,omega,key,epsabs,epsrel,lw,liw,ifail,g)
```

```
 ++ calculates an approximation to the sine or the cosine
 ++ transform of a function g over [a,b]:
 ++ See \downlink{Manual Page}{manpageXXd01anf}.
d01apf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
    Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp1(G))) -> Result
 ++ d01apf(a,b,alfa,beta,key,epsabs,epsrel,lw,liw,ifail,g)
 ++ is an adaptive integrator which calculates an
 ++ approximation to the integral of a function g(x)w(x) over a
 ++ finite interval [a,b]:
 ++ See \downlink{Manual Page}{manpageXXd01apf}.
d01aqf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
    DoubleFloat,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp1(G))) -> Result
 ++ d01aqf(a,b,c,epsabs,epsrel,lw,liw,ifail,g)
 ++ calculates an approximation to the Hilbert transform of a
 ++ function g(x) over [a,b]:
 ++ See \downlink{Manual Page}{manpageXXd01aqf}.
d01asf : (DoubleFloat,DoubleFloat,Integer,DoubleFloat,_
    Integer,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp1(G))) -> Result
 ++ d01asf(a,omega,key,epsabs,limlst,lw,liw,ifail,g)
 ++ calculates an approximation to the sine or the cosine
 ++ transform of a function g over [a,infty):
 ++ See \downlink{Manual Page}{manpageXXd01asf}.
d01bbf : (DoubleFloat,DoubleFloat,Integer,Integer,_
    Integer,Integer) -> Result
 ++ d01bbf(a,b,itype,n,gtype,ifail)
 ++ returns the weight appropriate to a
 ++ Gaussian quadrature.
 ++ The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-
 ++ Laguerre and Gauss-Hermite.
 ++ See \downlink{Manual Page}{manpageXXd01bbf}.
d01fcf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    DoubleFloat,Integer,Integer,Integer,_
    Union(fn:FileName,fp:Asp4(FUNCTN))) -> Result
 ++ d01fcf(ndim,a,b,maxpts,eps,lenwrk,minpts,ifail,functn)
 ++ attempts to evaluate a multi-dimensional integral (up to
 ++ 15 dimensions), with constant and finite limits, to a specified
 ++ relative accuracy, using an adaptive subdivision strategy.
 ++ See \downlink{Manual Page}{manpageXXd01fcf}.
d01gaf : (Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer) -> Result
 ++ d01gaf(x,y,n,ifail)
 ++ integrates a function which is specified numerically at
 ++ four or more points, over the whole of its specified range, using
 ++ third-order finite-difference formulae with error estimates,
 ++ according to a method due to Gill and Miller.
 ++ See \downlink{Manual Page}{manpageXXd01gaf}.
d01gbf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    DoubleFloat,Integer,Integer,Matrix DoubleFloat,Integer,_
```

```
      Union(fn:FileName,fp:Asp4(FUNCTN))) -> Result
    ++ d01gbf(ndim,a,b,maxcls,eps,lenwrk,mincls,wrkstr,ifail,functn)
    ++ returns an approximation to the integral of a function
    ++ over a hyper-rectangular region, using a Monte Carlo method. An
    ++ approximate relative error estimate is also returned. This
    ++ routine is suitable for low accuracy work.
    ++ See \downlink{Manual Page}{manpageXXd01gbf}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Any
    import Record
    import Integer
    import Matrix DoubleFloat
    import Boolean
    import NAGLinkSupportPackage
    import FortranPackage
    import Union(fn:FileName,fp:Asp1(F))
    import AnyFunctions1(DoubleFloat)
    import AnyFunctions1(Integer)
    import AnyFunctions1(Matrix DoubleFloat)


    d01ajf(aArg:DoubleFloat,bArg:DoubleFloat,epsabsArg:DoubleFloat,_
        epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
        ifailArg:Integer,fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
        pushFortranOutputStack(fFilename := aspFilename "f")$FOP
        if fArg case fn
          then outputAsFortran(fArg.fn)
          else outputAsFortran(fArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([fFilename]$Lisp,_
        "d01ajf",_
        ["a"::S,"b"::S,"epsabs"::S,"epsrel"::S,"lw"::S_
        ,"liw"::S,"result"::S,"abserr"::S,"ifail"::S,"f"::S_
        ,"w"::S,"iw"::S]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"f"::S]$Lisp,_
        [["double"::S,"a"::S,"b"::S,"epsabs"::S,"epsrel"::S_
        ,"result"::S,"abserr"::S,["w"::S,"lw"::S]$Lisp,"f"::S]$Lisp_
        ,["integer"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
        [([aArg::Any,bArg::Any,epsabsArg::Any,epsrelArg::Any,lwArg::Any,_
        liwArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    d01akf(aArg:DoubleFloat,bArg:DoubleFloat,epsabsArg:DoubleFloat,_
```

```
    epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
    ifailArg:Integer,fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
    pushFortranOutputStack(fFilename := aspFilename "f")$FOP
    if fArg case fn
      then outputAsFortran(fArg.fn)
      else outputAsFortran(fArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([fFilename]$Lisp,_
    "d01akf",_
    ["a"::S,"b"::S,"epsabs"::S,"epsrel"::S,"lw"::S_
    ,"liw"::S,"result"::S,"abserr"::S,"ifail"::S,"f"::S_
    ,"w"::S,"iw"::S]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"f"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,"epsabs"::S,"epsrel"::S_
    ,"result"::S,"abserr"::S,["w"::S,"lw"::S]$Lisp,"f"::S]$Lisp_
    ,["integer"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,epsabsArg::Any,epsrelArg::Any,lwArg::Any,_
    liwArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01alf(aArg:DoubleFloat,bArg:DoubleFloat,nptsArg:Integer,_
    pointsArg:Matrix DoubleFloat,epsabsArg:DoubleFloat,_
    epsrelArg:DoubleFloat,_
    lwArg:Integer,liwArg:Integer,ifailArg:Integer,_
    fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
    pushFortranOutputStack(fFilename := aspFilename "f")$FOP
    if fArg case fn
      then outputAsFortran(fArg.fn)
      else outputAsFortran(fArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([fFilename]$Lisp,_
    "d01alf",_
    ["a"::S,"b"::S,"npts"::S,"epsabs"::S,"epsrel"::S_
    ,"lw"::S,"liw"::S,"result"::S,"abserr"::S,"ifail"::S_
    ,"f"::S,"points"::S,"w"::S,"iw"::S]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"f"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,["points"::S,"*"::S]$Lisp_
    ,"epsabs"::S,"epsrel"::S,"result"::S,"abserr"::S,_
    ["w"::S,"lw"::S]$Lisp,"f"::S]$Lisp_
    ,["integer"::S,"npts"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,nptsArg::Any,epsabsArg::Any,epsrelArg::Any,_
    lwArg::Any,liwArg::Any,ifailArg::Any,pointsArg::Any ])_
    @List Any]$Lisp)$Lisp)_
```

```
        pretend List (Record(key:Symbol,entry:Any))]$Result

    d01amf(boundArg:DoubleFloat,infArg:Integer,epsabsArg:DoubleFloat,_
        epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
        ifailArg:Integer,fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
        pushFortranOutputStack(fFilename := aspFilename "f")$FOP
        if fArg case fn
          then outputAsFortran(fArg.fn)
          else outputAsFortran(fArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([fFilename]$Lisp,_
        "d01amf",_
        ["bound"::S,"inf"::S,"epsabs"::S,"epsrel"::S,"lw"::S_
        ,"liw"::S,"result"::S,"abserr"::S,"ifail"::S,"f"::S_
        ,"w"::S,"iw"::S]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"f"::S]$Lisp,_
        [["double"::S,"bound"::S,"epsabs"::S,"epsrel"::S_
        ,"result"::S,"abserr"::S,["w"::S,"lw"::S]$Lisp,"f"::S]$Lisp_
        ,["integer"::S,"inf"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
        [([boundArg::Any,infArg::Any,epsabsArg::Any,epsrelArg::Any,_
        lwArg::Any,liwArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    d01anf(aArg:DoubleFloat,bArg:DoubleFloat,omegaArg:DoubleFloat,_
        keyArg:Integer,epsabsArg:DoubleFloat,epsrelArg:DoubleFloat,_
        lwArg:Integer,liwArg:Integer,ifailArg:Integer,_
        gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
        pushFortranOutputStack(gFilename := aspFilename "g")$FOP
        if gArg case fn
          then outputAsFortran(gArg.fn)
          else outputAsFortran(gArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([gFilename]$Lisp,_
        "d01anf",_
        ["a"::S,"b"::S,"omega"::S,"key"::S,"epsabs"::S_
        ,"epsrel"::S,"lw"::S,"liw"::S,"result"::S,"abserr"::S_
        ,"ifail"::S,"g"::S,"w"::S,"iw"::S]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"g"::S]$Lisp,_
        [["double"::S,"a"::S,"b"::S,"omega"::S,"epsabs"::S_
        ,"epsrel"::S,"result"::S,"abserr"::S,["w"::S,"lw"::S]$Lisp,_
        "g"::S]$Lisp_
        ,["integer"::S,"key"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
        [([aArg::Any,bArg::Any,omegaArg::Any,keyArg::Any,epsabsArg::Any,_
```

```
    epsrelArg::Any,lwArg::Any,liwArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01apf(aArg:DoubleFloat,bArg:DoubleFloat,alfaArg:DoubleFloat,_
    betaArg:DoubleFloat,keyArg:Integer,epsabsArg:DoubleFloat,_
    epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
    ifailArg:Integer,gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
    pushFortranOutputStack(gFilename := aspFilename "g")$FOP
    if gArg case fn
      then outputAsFortran(gArg.fn)
      else outputAsFortran(gArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([gFilename]$Lisp,_
    "d01apf",_
    ["a"::S,"b"::S,"alfa"::S,"beta"::S,"key"::S_
    ,"epsabs"::S,"epsrel"::S,"lw"::S,"liw"::S,"result"::S_
    ,"abserr"::S,"ifail"::S,"g"::S,"w"::S,"iw"::S]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"g"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,"alfa"::S,"beta"::S_
    ,"epsabs"::S,"epsrel"::S,"result"::S,"abserr"::S,_
    ["w"::S,"lw"::S]$Lisp,"g"::S]$Lisp_
    ,["integer"::S,"key"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,alfaArg::Any,betaArg::Any,keyArg::Any,_
    epsabsArg::Any,epsrelArg::Any,lwArg::Any,liwArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01aqf(aArg:DoubleFloat,bArg:DoubleFloat,cArg:DoubleFloat,_
    epsabsArg:DoubleFloat,epsrelArg:DoubleFloat,lwArg:Integer,_
    liwArg:Integer,ifailArg:Integer,_
    gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
    pushFortranOutputStack(gFilename := aspFilename "g")$FOP
    if gArg case fn
      then outputAsFortran(gArg.fn)
      else outputAsFortran(gArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([gFilename]$Lisp,_
    "d01aqf",_
    ["a"::S,"b"::S,"c"::S,"epsabs"::S,"epsrel"::S_
    ,"lw"::S,"liw"::S,"result"::S,"abserr"::S,"ifail"::S_
    ,"g"::S,"w"::S,"iw"::S]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"g"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,"c"::S,"epsabs"::S_
    ,"epsrel"::S,"result"::S,"abserr"::S,_
    ["w"::S,"lw"::S]$Lisp,"g"::S]$Lisp_
    ,["integer"::S,"lw"::S,"liw"::S,["iw"::S,"liw"::S]$Lisp_
```

```
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"w"::S,"iw"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,cArg::Any,epsabsArg::Any,epsrelArg::Any,_
    lwArg::Any,liwArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

  d01asf(aArg:DoubleFloat,omegaArg:DoubleFloat,keyArg:Integer,_
    epsabsArg:DoubleFloat,limlstArg:Integer,lwArg:Integer,_
    liwArg:Integer,ifailArg:Integer,_
    gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
    pushFortranOutputStack(gFilename := aspFilename "g")$FOP
    if gArg case fn
      then outputAsFortran(gArg.fn)
      else outputAsFortran(gArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([gFilename]$Lisp,_
    "d01asf",_
    ["a"::S,"omega"::S,"key"::S,"epsabs"::S,"limlst"::S_
    ,"lw"::S,"liw"::S,"result"::S,"abserr"::S,"lst"::S_
    ,"ifail"::S,"g"::S,"erlst"::S,"rslst"::S,"ierlst"::S,"iw"::S,"w"::S_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"lst"::S,"erlst"::S,"rslst"::S,_
    "ierlst"::S,"iw"::S,"w"::S,"g"::S]$Lisp,_
    [["double"::S,"a"::S,"omega"::S,"epsabs"::S_
    ,"result"::S,"abserr"::S,["erlst"::S,"limlst"::S]$Lisp,_
    ["rslst"::S,"limlst"::S]$Lisp,["w"::S,"lw"::S]$Lisp,"g"::S]$Lisp_
    ,["integer"::S,"key"::S,"limlst"::S,"lw"::S_
    ,"liw"::S,"lst"::S,["ierlst"::S,"limlst"::S]$Lisp,_
    ["iw"::S,"liw"::S]$Lisp,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"abserr"::S,"lst"::S,"erlst"::S,"rslst"::S,_
    "ierlst"::S,"iw"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,omegaArg::Any,keyArg::Any,epsabsArg::Any,_
    limlstArg::Any,lwArg::Any,liwArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

  d01bbf(aArg:DoubleFloat,bArg:DoubleFloat,itypeArg:Integer,_
    nArg:Integer,gtypeArg:Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "d01bbf",_
    ["a"::S,"b"::S,"itype"::S,"n"::S,"gtype"::S_
    ,"ifail"::S,"weight"::S,"abscis"::S]$Lisp,_
    ["weight"::S,"abscis"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,["weight"::S,"n"::S]$Lisp_
    ,["abscis"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"itype"::S,"n"::S,"gtype"::S_
    ,"ifail"::S]$Lisp_
```

```
    ]$Lisp,_
    ["weight"::S,"abscis"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,itypeArg::Any,nArg::Any,_
    gtypeArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01fcf(ndimArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
    maxptsArg:Integer,epsArg:DoubleFloat,lenwrkArg:Integer,_
    minptsArg:Integer,ifailArg:Integer,_
    functnArg:Union(fn:FileName,fp:Asp4(FUNCTN))): Result ==
    pushFortranOutputStack(functnFilename := aspFilename "functn")$FOP
    if functnArg case fn
      then outputAsFortran(functnArg.fn)
      else outputAsFortran(functnArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([functnFilename]$Lisp,_
    "d01fcf",_
    ["ndim"::S,"maxpts"::S,"eps"::S,"lenwrk"::S,"acc"::S_
    ,"finval"::S,"minpts"::S,"ifail"::S,"functn"::S,"a"::S,_
    "b"::S,"wrkstr"::S]$Lisp,_
    ["acc"::S,"finval"::S,"wrkstr"::S,"functn"::S]$Lisp,_
    [["double"::S,["a"::S,"ndim"::S]$Lisp,["b"::S,"ndim"::S]$Lisp_
    ,"eps"::S,"acc"::S,"finval"::S,["wrkstr"::S,"lenwrk"::S]$Lisp,_
    "functn"::S]$Lisp_
    ,["integer"::S,"ndim"::S,"maxpts"::S,"lenwrk"::S_
    ,"minpts"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["acc"::S,"finval"::S,"minpts"::S,"ifail"::S]$Lisp,_
    [([ndimArg::Any,maxptsArg::Any,epsArg::Any,lenwrkArg::Any,_
    minptsArg::Any,ifailArg::Any,aArg::Any,bArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01gaf(xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,nArg:Integer,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "d01gaf",_
    ["n"::S,"ans"::S,"er"::S,"ifail"::S,"x"::S,"y"::S]$Lisp,_
    ["ans"::S,"er"::S]$Lisp,_
    [["double"::S,["x"::S,"n"::S]$Lisp,["y"::S,"n"::S]$Lisp_
    ,"ans"::S,"er"::S]$Lisp_
    ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["ans"::S,"er"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,ifailArg::Any,xArg::Any,yArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d01gbf(ndimArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
```

```
        maxclsArg:Integer,epsArg:DoubleFloat,lenwrkArg:Integer,_
        minclsArg:Integer,wrkstrArg:Matrix DoubleFloat,ifailArg:Integer,_
        functnArg:Union(fn:FileName,fp:Asp4(FUNCTN))): Result ==
        pushFortranOutputStack(functnFilename := aspFilename "functn")$FOP
        if functnArg case fn
          then outputAsFortran(functnArg.fn)
          else outputAsFortran(functnArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([functnFilename]$Lisp,_
        "d01gbf",_
        ["ndim"::S,"maxcls"::S,"eps"::S,"lenwrk"::S,"acc"::S_
        ,"finest"::S,"mincls"::S,"ifail"::S,"functn"::S,"a"::S,_
        "b"::S,"wrkstr"::S]$Lisp,_
        ["acc"::S,"finest"::S,"functn"::S]$Lisp,_
        [["double"::S,["a"::S,"ndim"::S]$Lisp,["b"::S,"ndim"::S]$Lisp_
        ,"eps"::S,"acc"::S,"finest"::S,["wrkstr"::S,"lenwrk"::S]$Lisp,_
        "functn"::S]$Lisp_
        ,["integer"::S,"ndim"::S,"maxcls"::S,"lenwrk"::S_
        ,"mincls"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["acc"::S,"finest"::S,"mincls"::S,"wrkstr"::S,"ifail"::S]$Lisp,_
        [([ndimArg::Any,maxclsArg::Any,epsArg::Any,lenwrkArg::Any,_
        minclsArg::Any,ifailArg::Any,aArg::Any,bArg::Any,wrkstrArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGD01.dotabb** —

```
"NAGD01" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGD01"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGD01" -> "ALIST"
```

———-

15.6 package NAGE01 NagInterpolationPackage

— **NagInterpolationPackage.help** —

```
E01 -- Interpolation                          Introduction -- E01
                            Chapter E01
                           Interpolation
```

1. Scope of the Chapter

This chapter is concerned with the interpolation of a function of one or two variables. When provided with the value of the function (and possibly one or more of its lowest-order derivatives) at each of a number of values of the variable(s), the routines provide either an interpolating function or an interpolated value. For some of the interpolating functions, there are supporting routines to evaluate, differentiate or integrate them.

2. Background to the Problems

In motivation and in some of its numerical processes, this chapter has much in common with Chapter E02 (Curve and Surface Fitting). For this reason, we shall adopt the same terminology and refer to dependent variable and independent variable(s) instead of function and variable(s). Where there is only one independent variable, we shall denote it by x and the dependent variable by y. Thus, in the basic problem considered in this chapter, we are given a set of distinct values $x_1, x_2, \ldots, x_m$ of x and a corresponding set of values $y_1, y_2, \ldots, y_m$ of y, and we shall describe the problem as being one of interpolating the data points (x_r, y_r), rather than interpolating a function. In modern usage, however, interpolation can have either of two rather different meanings, both relevant to routines in this chapter. They are

(a) the determination of a function of x which takes the value y_r at $x=x_r$, for r=1,2,...,m (an interpolating function or interpolant),

(b) the determination of the value (interpolated value or interpolate) of an interpolating function at any given value, $\hat{x}$, say x, of x within the range of the x_r (so as to estimate the value at $\hat{x}$ of the function underlying the data).

The latter is the older meaning, associated particularly with the use of mathematical tables. The term 'function underlying the data', like the other terminology described above, is used so as to cover situations additional to those in which the data points have been computed from a known function, as with a mathematical

table. In some contexts, the function may be unknown, perhaps representing the dependency of one physical variable on another, say temperature upon time.

Whether the underlying function is known or unknown, the object of interpolation will usually be to approximate it to acceptable accuracy by a function which is easy to evaluate anywhere in some range of interest. Piecewise polynomials such as cubic splines (see Section 2.2 of the E02 Chapter Introduction for definitions of terms in this case), being easy to evaluate and also capable of approximating a wide variety of functions, are the types of function mostly used in this chapter as interpolating functions.

Piecewise polynomials also, to a large extent, avoid the well-known problem of large unwanted fluctuations which can arise when interpolating a data set with a simple polynomial. Fluctuations can still arise but much less frequently and much less severely than with simple polynomials. Unwanted fluctuations are avoided altogether by a routine using piecewise cubic polynomials having only first derivative continuity. It is designed especially for monotonic data, but for other data still provides an interpolant which increases, or decreases, over the same intervals as the data.

The concept of interpolation can be generalised in a number of ways. For example, we may be required to estimate the value of $\hat{}$ the underlying function at a value x outside the range of the data. This is the process of extrapolation. In general, it is a good deal less accurate than interpolation and is to be avoided whenever possible.

Interpolation can also be extended to the case of two independent variables. We shall denote these by x and y, and the dependent variable by f. Methods used depend markedly on whether or not the data values of f are given at the intersections of a rectangular mesh in the (x,y)-plane. If they are, bicubic splines (see Section 2.3.2 of the E02 Chapter Introduction) are very suitable and usually very effective for the problem. For other cases, perhaps where the f values are quite arbitrarily scattered in the (x,y)-plane, polynomials and splines are not at all appropriate and special forms of interpolating function have to be employed. Many such forms have been devised and two of the most successful are in routines in this chapter. They both have continuity in first, but not higher, derivatives.

2.1. References

[1] Froberg C E (1970) Introduction to Numerical Analysis. Addison-Wesley (2nd Edition).

[2] Dahlquist G and Bjork A (1974) Numerical Methods. Prentice-Hall.

3. Recommendations on Choice and Use of Routines

3.1. General

Before undertaking interpolation, in other than the simplest cases, the user should seriously consider the alternative of using a routine from Chapter E02 to approximate the data by a polynomial or spline containing significantly fewer coefficients than the corresponding interpolating function. This approach is much less liable to produce unwanted fluctuations and so can often provide a better approximation to the function underlying the data.

When interpolation is employed to approximate either an underlying function or its values, the user will need to be satisfied that the accuracy of approximation achieved is adequate. There may be a means for doing this which is particular to the application, or the routine used may itself provide a means. In other cases, one possibility is to repeat the interpolation using one or more extra data points, if they are available, or otherwise one or more fewer, and to compare the results. Other possibilities, if it is an interpolating function which is determined, are to examine the function graphically, if that gives sufficient accuracy, or to observe the behaviour of the differences in a finite-difference table, formed from evaluations of the interpolating function at equally-spaced values of x over the range of interest. The spacing should be small enough to cause the typical size of the differences to decrease as the order of difference increases.

3.2. One Independent Variable

E01BAF computes an interpolating cubic spline, using a particular choice for the set of knots which has proved generally satisfactory in practice. If the user wishes to choose a different set, a cubic spline routine from Chapter E02, namely E02BAF, may be used in its interpolating mode, setting NCAP7 = M+4 and all elements of the parameter W to unity. These routines provide the interpolating function in B-spline form (see Section 2.2.2 in the E02 Chapter Introduction). Routines for evaluating, differentiating and integrating this form are discussed in Section 3.7 of the E02 Chapter Introduction.

The cubic spline does not always avoid unwanted fluctuations, especially when the data show a steep slope close to a region of small slope, or when the data inadequately represent the

underlying curve. In such cases, E01BEF can be very useful. It derives a piecewise cubic polynomial (with first derivative continuity) which, between any adjacent pair of data points, either increases all the way, or decreases all the way (or stays constant). It is especially suited to data which are monotonic over their whole range.

In this routine, the interpolating function is represented simply by its value and first derivative at the data points. Supporting routines compute its value and first derivative elsewhere, as well as its definite integral over an arbitary interval.

3.3. Two Independent Variables

3.3.1. Data on a rectangular mesh

Given the value f_{qr} of the dependent variable f at the point (x_q, y_r) in the plane of the independent variables x and y, for each q=1,2,...,m and r=1,2,...,n (so that the points (x_q, y_r) lie at the m*n intersections of a rectangular mesh), E01DAF computes an interpolating bicubic spline, using a particular choice for each of the spline's knot-set. This choice, the same as in E01BAF , has proved generally satisfactory in practice. If, instead, the user wishes to specify his own knots, a routine from Chapter E02, namely E02DAF, may be adapted (it is more cumbersome for the purpose, however, and much slower for larger problems). Using m and n in the above sense, the parameter M must be set to m*n, PX and PY must be set to m+4 and n+4 respectively and all elements of W should be set to unity. The recommended value for EPS is zero.

3.3.2. Arbitrary data

As remarked at the end of Section 2, special types of interpolating are required for this problem, which can often be difficult to solve satisfactorily. Two of the most successful are employed in E01SAF and E01SEF, the two routines which (with their respective evaluation routines E01SBF and E01SFF) are provided for the problem. Definitions can be found in the routine documents. Both interpolants have first derivative continuity and are 'local', in that their value at any point depends only on data in the immediate neighbourhood of the point. This latter feature is necessary for large sets of data to avoid prohibitive computing time.

The relative merits of the two methods vary with the data and it is not possible to predict which will be the better in any

particular case.

3.4. Index

Derivative, of interpolant from E01BEF E01BGF

Evaluation, of interpolant
from E01BEF E01BFF
from E01SAF E01SBF
from E01SEF E01SFF
Extrapolation, one variable E01BEF
Integration (definite) of interpolant from E01BEF E01BHF
Interpolated values, one variable, from interpolant from E01BEF E01BFF
E01BGF
Interpolated values, two variables,
from interpolant from E01SAF E01SBF
from interpolant from E01SEF E01SFF
Interpolating function, one variable,
cubic spline E01BAF
other piecewise polynomial E01BEF
Interpolating function, two variables
bicubic spline E01DAF
other piecewise polynomial E01SAF
modified Shepard method E01SEF

Chapter E01

Interpolation

E01BAF Interpolating functions, cubic spline interpolant, one variable

E01BEF Interpolating functions, monotonicity-preserving, piecewise cubic Hermite, one variable

E01BFF Interpolated values, interpolant computed by E01BEF, function only, one variable,

E01BGF Interpolated values, interpolant computed by E01BEF, function and 1st derivative, one variable

E01BHF Interpolated values, interpolant computed by E01BEF, definite integral, one variable

E01DAF Interpolating functions, fitting bicubic spline, data on rectangular grid

```
E01SAF  Interpolating functions, method of Renka and Cline, two
        variables

E01SBF  Interpolated values, evaluate interpolant computed by
        E01SAF, two variables

E01SEF  Interpolating functions, modified Shepard's method, two
        variables

E01SFF  Interpolated values, evaluate interpolant computed by
        E01SEF, two variables
```

%%

```
E01 -- Interpolation                                          E01BAF
        E01BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01BAF determines a cubic spline interpolant to a given set of
data.

2. Specification

       SUBROUTINE E01BAF (M, X, Y, LAMDA, C, LCK, WRK, LWRK,
      1                   IFAIL)
       INTEGER          M, LCK, LWRK, IFAIL
       DOUBLE PRECISION X(M), Y(M), LAMDA(LCK), C(LCK), WRK(LWRK)

3. Description

This routine determines a cubic spline s(x), defined in the range
x <=x<=x , which interpolates (passes exactly through) the set of
 1      m
data points (x ,y ), for i=1,2,...,m, where m>=4 and x <x <...<x
              i  i                                  1  2      m
end conditions are not imposed. The spline interpolant chosen has
m-4 interior knots (lambda) ,(lambda) ,...,(lambda) , which are
                           5         6             m
set to the values of x ,x ,...,x    respectively. This spline is
                      3  4      m-2
represented in its B-spline form (see Cox [1]):

                              m
                              --
```

$$s(x) = \sum_{i=1} c_i N_i(x),$$

where $N_i(x)$ denotes the normalised B-Spline of degree 3, defined upon the knots $(\lambda)_i, (\lambda)_{i+1}, \ldots, (\lambda)_{i+4}$, and c_i denotes its coefficient, whose value is to be determined by the routine.

The use of B-splines requires eight additional knots $(\lambda)_1$, $(\lambda)_2$, $(\lambda)_3$, $(\lambda)_4$, $(\lambda)_{m+1}$, $(\lambda)_{m+2}$, $(\lambda)_{m+3}$ and $(\lambda)_{m+4}$ to be specified; the routine sets the first four of these to x_1 and the last four to x_m.

The algorithm for determining the coefficients is as described in [1] except that QR factorization is used instead of LU decomposition. The implementation of the algorithm involves setting up appropriate information for the related routine E02BAF followed by a call of that routine. (For further details of E02BAF, see the routine document.)

Values of the spline interpolant, or of its derivatives or definite integral, can subsequently be computed as detailed in Section 8.

4. References

[1] Cox M G (1975) An Algorithm for Spline Interpolation. J. Inst. Math. Appl. 15 95--108.

[2] Cox M G (1977) A Survey of Numerical Methods for Data and Function Approximation. The State of the Art in Numerical Analysis. (ed D A H Jacobs) Academic Press. 627--668.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of data points. Constraint: M >= 4.

2: X(M) -- DOUBLE PRECISION array Input
On entry: X(i) must be set to x_i, the ith data value of the independent variable x, for i=1,2,...,m. Constraint: X(i) < X(i+1), for i=1,2,...,M-1.

```
 3:  Y(M) -- DOUBLE PRECISION array                          Input
     On entry: Y(i) must be set to y , the ith data value of the
                                    i
     dependent variable y, for i=1,2,...,m.

 4:  LAMDA(LCK) -- DOUBLE PRECISION array                   Output
     On exit: the value of (lambda) , the ith knot, for
                                 i
     i=1,2,...,m+4.

 5:  C(LCK) -- DOUBLE PRECISION array                       Output
     On exit: the coefficient c  of the B-spline N (x), for
                              i                   i
     i=1,2,...,m. The remaining elements of the array are not
     used.

 6:  LCK -- INTEGER                                          Input
     On entry:
     the dimension of the arrays LAMDA and C as declared in the
     (sub)program from which E01BAF is called.
     Constraint: LCK >= M + 4.

 7:  WRK(LWRK) -- DOUBLE PRECISION array                Workspace

 8:  LWRK -- INTEGER                                         Input
     On entry:
     the dimension of the array WRK as declared in the
     (sub)program from which E01BAF is called.
     Constraint: LWRK>=6*M+16.

 9:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry M < 4,

     or       LCK<M+4,

     or       LWRK<6*M+16.
```

```
IFAIL= 2
     The X-values fail to satisfy the condition

     X(1) < X(2) < X(3) <... < X(M).

7. Accuracy

The rounding errors incurred are such that the computed spline is
an exact interpolant for a slightly perturbed set of ordinates
y +(delta)y . The ratio of the root-mean-square value of the
 i         i
(delta)y  to that of the y  is no greater than a small multiple
        i                 i
of the relative machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to m.

All the x  are used as knot positions except x  and x    . This
         i                                    2      m-1
choice of knots (see Cox [2]) means that s(x) is composed of m-3
cubic arcs as follows. If m=4, there is just a single arc space
spanning the whole interval x  to x . If m>=5, the first and last
                             1     4
arcs span the intervals x  to x  and x    to x  respectively.
                        1     3      m-2     m
Additionally if m>=6, the ith arc, for i=2,3,...,m-4 spans the
interval x    to x   .
          i+1     i+2

After the call

       CALL E01BAF (M, X, Y, LAMDA, C, LCK, WRK, LWRK, IFAIL)

the following operations may be carried out on the interpolant
s(x).

The value of s(x) at x = XVAL can be provided in the real
variable SVAL by the call

       CALL E02BBF (M+4, LAMDA, C, XVAL, SVAL, IFAIL)

The values of s(x) and its first three derivatives at x = XVAL
can be provided in the real array SDIF of dimension 4, by the
call

       CALL E02BCF (M+4, LAMDA, C, XVAL, LEFT, SDIF, IFAIL)

Here LEFT must specify whether the left- or right-hand value of
```

```
the third derivative is required (see E02BCF for details).

The value of the integral of s(x) over the range x  to x  can be
                                                   1     m
provided in the real variable SINT by

       CALL E02BDF (M+4, LAMDA, C, SINT, IFAIL)

9. Example

The example program sets up data from 7 values of the exponential
function in the interval 0 to 1. E01BAF is then called to compute
a spline interpolant to these data.

The spline is evaluated by E02BBF, at the data points and at
points halfway between each adjacent pair of data points, and the
                                  x
spline values and the values of e  are printed out.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E01 -- Interpolation                                        E01BEF
        E01BEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01BEF computes a monotonicity-preserving piecewise cubic Hermite
interpolant to a set of data points.

2. Specification

       SUBROUTINE E01BEF (N, X, F, D, IFAIL)
       INTEGER          N, IFAIL
       DOUBLE PRECISION X(N), F(N), D(N)

3. Description

This routine estimates first derivatives at the set of data
points (x ,f ), for r=1,2,...,n, which determine a piecewise
         r  r
cubic Hermite interpolant to the data, that preserves
```

monotonicity over ranges where the data points are monotonic. If the data points are only piecewise monotonic, the interpolant will have an extremum at each point where monotonicity switches direction. The estimates of the derivatives are computed by a formula due to Brodlie, which is described in Fritsch and Butland [1], with suitable changes at the boundary points.

The routine is derived from routine PCHIM in Fritsch [2].

Values of the computed interpolant, and of its first derivative and definite integral, can subsequently be computed by calling E01BFF, E01BGF and E01BHF, as described in Section 8

4. References

[1] Fritsch F N and Butland J (1984) A Method for Constructing Local Monotone Piecewise Cubic Interpolants. SIAM J. Sci. Statist. Comput. 5 300--304.

[2] Fritsch F N (1982) PCHIP Final Specifications. Report UCID-30194. Lawrence Livermore National Laboratory.

5. Parameters

1: N -- INTEGER Input
On entry: n, the number of data points. Constraint: N >= 2.

2: X(N) -- DOUBLE PRECISION array Input
On entry: X(r) must be set to x_r, the rth value of the independent variable (abscissa), for r=1,2,...,n.
Constraint: X(r) < X(r+1).

3: F(N) -- DOUBLE PRECISION array Input
On entry: F(r) must be set to f_r, the rth value of the dependent variable (ordinate), for r=1,2,...,n.

4: D(N) -- DOUBLE PRECISION array Output
On exit: estimates of derivatives at the data points. D(r) contains the derivative at X(r).

5: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

```
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 2.

IFAIL= 2
     The values of X(r), for r=1,2,...,N, are not in strictly
     increasing order.

7. Accuracy

The computational errors in the array D should be negligible in
most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to n.

The values of the computed interpolant at the points PX(i), for
i=1,2,...,M, may be obtained in the real array PF, of length at
least M, by the call:


        CALL E01BFF(N,X,F,D,M,PX,PF,IFAIL)

where N, X and F are the input parameters to E01BEF and D is the
output parameter from E01BEF.

The values of the computed interpolant at the points PX(i), for i
= 1,2,...,M, together with its first derivatives, may be obtained
in the real arrays PF and PD, both of length at least M, by the
call:

        CALL E01BGF(N,X,F,D,M,PX,PF,PD,IFAIL)

where N, X, F and D are as described above.

The value of the definite integral of the interpolant over the
interval A to B can be obtained in the real variable PINT by the
call:

        CALL E01BHF(N,X,F,D,A,B,PINT,IFAIL)

where N, X, F and D are as described above.
```

```
9. Example

This example program reads in a set of data points, calls E01BEF
to compute a piecewise monotonic interpolant, and then calls
E01BFF to evaluate the interpolant at equally spaced points.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E01 -- Interpolation                                          E01BFF
        E01BFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01BFF evaluates a piecewise cubic Hermite interpolant at a set
of points.

2. Specification

       SUBROUTINE E01BFF (N, X, F, D, M, PX, PF, IFAIL)
       INTEGER          N, M, IFAIL
       DOUBLE PRECISION X(N), F(N), D(N), PX(M), PF(M)

3. Description

This routine evaluates a piecewise cubic Hermite interpolant, as
computed by E01BEF, at the points PX(i), for i=1,2,...,m. If any
point lies outside the interval from X(1) to X(N), a value is
extrapolated from the nearest extreme cubic, and a warning is
returned.

The routine is derived from routine PCHFE in Fritsch [1].

4. References

[1]   Fritsch F N (1982) PCHIP Final Specifications. Report UCID-
      30194. Lawrence Livermore National Laboratory.

5. Parameters

 1:  N -- INTEGER                                             Input
```

```
 2:  X(N) -- DOUBLE PRECISION array                          Input

 3:  F(N) -- DOUBLE PRECISION array                          Input

 4:  D(N) -- DOUBLE PRECISION array                          Input
     On entry: N, X, F and D must be unchanged from the previous
     call of E01BEF.

 5:  M -- INTEGER                                            Input
     On entry: m, the number of points at which the interpolant
     is to be evaluated. Constraint: M >= 1.

 6:  PX(M) -- DOUBLE PRECISION array                         Input
     On entry: the m values of x at which the interpolant is to
     be evaluated.

 7:  PF(M) -- DOUBLE PRECISION array                        Output
     On exit: PF(i) contains the value of the interpolant
     evaluated at the point PX(i), for i=1,2,...,m.

 8:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 2.

IFAIL= 2
     The values of X(r), for r = 1,2,...,N, are not in strictly
     increasing order.

IFAIL= 3
     On entry M < 1.

IFAIL= 4
     At least one of the points PX(i), for i = 1,2,...,M, lies
     outside the interval [X(1),X(N)], and extrapolation was
     performed at all such points. Values computed at such points
     may be very unreliable.
```

```
7. Accuracy

The computational errors in the array PF should be negligible in
most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to
the number of evaluation points, m. The evaluation will be most
efficient if the elements of PX are in non-decreasing order (or,
more generally, if they are grouped in increasing order of the
intervals [X(r-1),X(r)]). A single call of E01BFF with m>1 is
more efficient than several calls with m=1.

9. Example

This example program reads in values of N, X, F and D, and then
calls E01BFF to evaluate the interpolant at equally spaced
points.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E01 -- Interpolation                                        E01BGF
        E01BGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01BGF evaluates a piecewise cubic Hermite interpolant and its
first derivative at a set of points.

2. Specification

      SUBROUTINE E01BGF (N, X, F, D, M, PX, PF, PD, IFAIL)
      INTEGER          N, M, IFAIL
      DOUBLE PRECISION X(N), F(N), D(N), PX(M), PF(M), PD(M)

3. Description

This routine evaluates a piecewise cubic Hermite interpolant, as
computed by E01BEF, at the points PX(i), for i=1,2,...,m. The
```

```
first derivatives at the points are also computed. If any point
lies outside the interval from X(1) to X(N), values of the
interpolant and its derivative are extrapolated from the nearest
extreme cubic, and a warning is returned.

If values of the interpolant only, and not of its derivative, are
required, E01BFF should be used.

The routine is derived from routine PCHFD in Fritsch [1].

4. References

[1]   Fritsch F N (1982) PCHIP Final Specifications. Report UCID-
      30194. Lawrence Livermore National Laboratory.

5. Parameters

 1:  N -- INTEGER                                              Input

 2:  X(N) -- DOUBLE PRECISION array                            Input

 3:  F(N) -- DOUBLE PRECISION array                            Input

 4:  D(N) -- DOUBLE PRECISION array                            Input
     On entry: N, X, F and D must be unchanged from the previous
     call of E01BEF.

 5:  M -- INTEGER                                              Input
     On entry: m, the number of points at which the interpolant
     is to be evaluated. Constraint: M >= 1.

 6:  PX(M) -- DOUBLE PRECISION array                           Input
     On entry: the m values of x at which the interpolant is to
     be evaluated.

 7:  PF(M) -- DOUBLE PRECISION array                          Output
     On exit: PF(i) contains the value of the interpolant
     evaluated at the point PX(i), for i=1,2,...,m.

 8:  PD(M) -- DOUBLE PRECISION array                          Output
     On exit: PD(i) contains the first derivative of the
     interpolant evaluated at the point PX(i), for i=1,2,...,m.

 9:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).
```

```
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 2.

IFAIL= 2
     The values of X(r), for r = 1,2,...,N, are not in strictly
     increasing order.

IFAIL= 3
     On entry M < 1.

IFAIL= 4
     At least one of the points PX(i), for i = 1,2,...,M, lies
     outside the interval [X(1),X(N)], and extrapolation was
     performed at all such points. Values computed at these
     points may be very unreliable.

7. Accuracy

The computational errors in the arrays PF and PD should be
negligible in most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to
the number of evaluation points, m. The evaluation will be most
efficient if the elements of PX are in non-decreasing order (or,
more generally, if they are grouped in increasing order of the
intervals [X(r-1),X(r)]). A single call of E01BGF with m>1 is
more efficient than several calls with m=1.

9. Example

This example program reads in values of N, X, F and D, and calls
E01BGF to compute the values of the interpolant and its
derivative at equally spaced points.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E01 -- Interpolation                                          E01BHF
        E01BHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01BHF evaluates the definite integral of a piecewise cubic
Hermite interpolant over the interval [a,b].

2. Specification

       SUBROUTINE E01BHF (N, X, F, D, A, B, PINT, IFAIL)
       INTEGER          N, IFAIL
       DOUBLE PRECISION X(N), F(N), D(N), A, B, PINT

3. Description

This routine evaluates the definite integral of a piecewise cubic
Hermite interpolant, as computed by E01BEF, over the interval
[a,b].

If either a or b lies outside the interval from X(1) to X(N)
computation of the integral involves extrapolation and a warning
is returned.

The routine is derived from routine PCHIA in Fritsch [1].

4. References

[1]   Fritsch F N (1982) PCHIP Final Specifications. Report UCID-
      30194. Lawrence Livermore National Laboratory .

5. Parameters

 1:  N -- INTEGER                                             Input

 2:  X(N) -- DOUBLE PRECISION array                           Input

 3:  F(N) -- DOUBLE PRECISION array                           Input

 4:  D(N) -- DOUBLE PRECISION array                           Input
     On entry: N, X, F and D must be unchanged from the previous
     call of E01BEF.

 5:  A -- DOUBLE PRECISION                                    Input
```

```
 6:  B -- DOUBLE PRECISION                                    Input
     On entry: the interval [a,b] over which integration is to
     be performed.

 7:  PINT -- DOUBLE PRECISION                                 Output
     On exit: the value of the definite integral of the
     interpolant over the interval [a,b].

 8:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 2.

IFAIL= 2
     The values of X(r), for r = 1,2,...,N, are not in strictly
     increasing order.

IFAIL= 3
     On entry at least one of A or B lies outside the interval [X
     (1),X(N)], and extrapolation was performed to compute the
     integral. The value returned is therefore unreliable.

7. Accuracy

The computational error in the value returned for PINT should be
negligible in most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to
the number of data points included within the interval [a,b].

9. Example

This example program reads in values of N, X, F and D. It then
reads in pairs of values for A and B, and evaluates the definite
integral of the interpolant over the interval [A,B] until end-of-
```

```
file is reached.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E01 -- Interpolation                                          E01DAF
        E01DAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01DAF computes a bicubic spline interpolating surface through a
set of data values, given on a rectangular grid in the x-y plane.

2. Specification

       SUBROUTINE E01DAF (MX, MY, X, Y, F, PX, PY, LAMDA, MU, C,
      1                   WRK, IFAIL)
       INTEGER          MX, MY, PX, PY, IFAIL
       DOUBLE PRECISION X(MX), Y(MY), F(MX*MY), LAMDA(MX+4), MU(MX
      1                 +4), C(MX*MY), WRK((MX+6)*(MY+6))

3. Description

This routine determines a bicubic spline interpolant to the set
of data points (x ,y ,f   ), for q=1,2,...,m ; r=1,2,...,m . The
                q  r  q,r                  x            y
spline is given in the B-spline representation

                          m   m
                           x   y
                          --  --
                s(x,y)= >   >  c  M (x)N (y),
                          --  --  ij i    j
                          i=1 j=1

such that

                        s(x ,y )=f   ,
                           q  r   q,r

where M (x) and N (y) denote normalised cubic B-splines, the
       i         j
```

former defined on the knots (lambda) to (lambda) and the
latter on the knots (mu) to (mu) , and the c are the spline
coefficients. These knots, as well as the coefficients, are
determined by the routine, which is derived from the routine
B2IRE in Anthony et al[1]. The method used is described in
Section 8.2.

For further information on splines, see Hayes and Halliday [4]
for bicubic splines and de Boor [3] for normalised B-splines.

Values of the computed spline can subsequently be obtained by
calling E02DEF or E02DFF as described in Section 8.3.

4. References

[1] Anthony G T, Cox M G and Hayes J G (1982) DASL - Data
Approximation Subroutine Library. National Physical
Laboratory.

[2] Cox M G (1975) An Algorithm for Spline Interpolation. J.
Inst. Math. Appl. 15 95--108.

[3] De Boor C (1972) On Calculating with B-splines. J. Approx.
Theory. 6 50--62.

[4] Hayes J G and Halliday J (1974) The Least-squares Fitting of
Cubic Spline Surfaces to General Data Sets. J. Inst. Math.
Appl. 14 89--103.

5. Parameters

1: MX -- INTEGER Input

2: MY -- INTEGER Input
On entry: MX and MY must specify m and m respectively,
the number of points along the x and y axis that define the
rectangular grid. Constraint: MX >= 4 and MY >= 4.

3: X(MX) -- DOUBLE PRECISION array Input

4: Y(MY) -- DOUBLE PRECISION array Input
On entry: X(q) and Y(r) must contain x , for q=1,2,...,m ,
and y , for r=1,2,...,m , respectively. Constraints:
X(q) < X(q+1), for q=1,2,...,m -1,

```
          Y(r) < Y(r+1), for r=1,2,...,m -1.
                                        y

 5:  F(MX*MY) -- DOUBLE PRECISION array                        Input
     On entry: F(m *(q-1)+r) must contain f    , for q=1,2,...,m ;
                  y                         q,r                 x
     r=1,2,...,m .
               y

 6:  PX -- INTEGER                                             Output

 7:  PY -- INTEGER                                             Output
     On exit: PX and PY contain m +4 and m +4, the total number
                                 x        y
     of knots of the computed spline with respect to the x and y
     variables, respectively.

 8:  LAMDA(MX+4) -- DOUBLE PRECISION array                     Output

 9:  MU(MY+4) -- DOUBLE PRECISION array                        Output
     On exit: LAMDA contains the complete set of knots (lambda)
                                                                  i
     associated with the x variable, i.e., the interior knots
     LAMDA(5), LAMDA(6), ..., LAMDA(PX-4), as well as the
     additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) =
     X(1) and LAMDA(PX-3) = LAMDA(PX-2) = LAMDA(PX-1) = LAMDA(PX)
     = X(MX) needed for the B-spline representation. MU contains
     the corresponding complete set of knots (mu)  associated
                                                 i
     with the y variable.

10:  C(MX*MY) -- DOUBLE PRECISION array                        Output
     On exit: the coefficients of the spline interpolant. C(
     m *(i-1)+j) contains the coefficient c   described in
      y                                    ij
     Section 3.

11:  WRK((MX+6)*(MY+6)) -- DOUBLE PRECISION array       Workspace

12:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings
```

```
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry MX < 4,

     or       MY < 4.

IFAIL= 2
     On entry either the values in the X array or the values in
     the Y array are not in increasing order.

IFAIL= 3
     A system of linear equations defining the B-spline
     coefficients was singular; the problem is too ill-
     conditioned to permit solution.

7. Accuracy

The main sources of rounding errors are in steps (2), (3), (6)
and (7) of the algorithm described in Section 8.2. It can be
shown (Cox [2]) that the matrix A  formed in step (2) has
                                 x
elements differing relatively from their true values by at most a
small multiple of 3(epsilon), where (epsilon) is the machine
precision. A  is 'totally positive', and a linear system with
           x
such a coefficient matrix can be solved quite safely by
elimination without pivoting. Similar comments apply to steps (6)
and (7). Thus the complete process is numerically stable.

8. Further Comments

8.1. Timing

The time taken by this routine is approximately proportional to
m m .
 x y

8.2. Outline of method used

The process of computing the spline consists of the following
steps:

(1)   choice of the interior x-knots (lambda) , (lambda) ,...,
                                              5          6
      (lambda)   as (lambda) =x   , for i=5,6,...,m ,
              m             i  i-2                x
```

x

(2) formation of the system

$$A_x E = F,$$

where A_x is a band matrix of order m_x and bandwidth 4, containing in its qth row the values at x_q of the B-splines in x, F is the m_x by m_y rectangular matrix of values $f_{q,r}$, and E denotes an m_x by m_y rectangular matrix of intermediate coefficients,

(3) use of Gaussian elimination to reduce this system to band triangular form,

(4) solution of this triangular system for E,

(5) choice of the interior y knots $(\mu)_5, (\mu)_6, \ldots, (\mu)_{m_y}$ as $(\mu)_i = y_{i-2}$, for $i=5,6,\ldots,m_y$,

(6) formation of the system

$$A_y C^T = E^T,$$

where A_y is the counterpart of A_x for the y variable, and C denotes the m_x by m_y rectangular matrix of values of c_{ij},

(7) use of Gaussian elimination to reduce this system to band triangular form,

(8) solution of this triangular system for C^T and hence C.

For computational convenience, steps (2) and (3), and likewise steps (6) and (7), are combined so that the formation of A_x and A_y and the reductions to triangular form are carried out one row at a time.

8.3. Evaluation of Computed Spline

The values of the computed spline at the points (TX(r),TY(r)), for r = 1,2,...,N, may be obtained in the double precision array FF, of length at least N, by the following call:

```
      IFAIL = 0
      CALL E02DEF(N,PX,PY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where PX, PY, LAMDA, MU and C are the output parameters of E01DAF , WRK is a double precision workspace array of length at least PY-4, and IWRK is an integer workspace array of length at least PY-4.

To evaluate the computed spline on an NX by NY rectangular grid of points in the x-y plane, which is defined by the x co-ordinates stored in TX(q), for q = 1,2,...,NX, and the y co-ordinates stored in TY(r), for r = 1,2,...,NY, returning the results in the double precision array FG which is of length at least NX*NY, the following call may be used:

```
       IFAIL = 0
       CALL E02DFF(NX,NY,PX,PY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
      *            IWRK,LIWRK,IFAIL)
```

where PX, PY, LAMDA, MU and C are the output parameters of E01DAF , WRK is a double precision workspace array of length at least LWRK = min(NWRK1,NWRK2), NWRK1 = NX*4+PX, NWRK2 = NY*4+PY, and IWRK is an integer workspace array of length at least LIWRK = NY + PY - 4 if NWRK1 > NWRK2, or NX + PX - 4 otherwise. The result of the spline evaluated at grid point (q,r) is returned in element (NY*(q-1)+r) of the array FG.

9. Example

This program reads in values of m , x for q=1,2,...,m , m and
x q x y
y for r=1,2,...,m , followed by values of the ordinates f
r y q,r
defined at the grid points (x ,y). It then calls E01DAF to
q r
compute a bicubic spline interpolant of the data values, and prints the values of the knots and B-spline coefficients. Finally it evaluates the spline at a small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%

E01 -- Interpolation E01SAF
E01SAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01SAF generates a two-dimensional surface interpolating a set of scattered data points, using the method of Renka and Cline.

2. Specification

```
SUBROUTINE E01SAF (M, X, Y, F, TRIANG, GRADS, IFAIL)
INTEGER          M, TRIANG(7*M), IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), GRADS(2,M)
```

3. Description

This routine constructs an interpolating surface F(x,y) through a set of m scattered data points (x_r, y_r, f_r), for r=1,2,...,m, using a method due to Renka and Cline. In the (x,y) plane, the data points must be distinct. The constructed surface is continuous and has continuous first derivatives.

The method involves firstly creating a triangulation with all the (x,y) data points as nodes, the triangulation being as nearly equiangular as possible (see Cline and Renka [1]). Then gradients in the x- and y-directions are estimated at node r, for r=1,2,...,m, as the partial derivatives of a quadratic function of x and y which interpolates the data value f_r, and which fits the data values at nearby nodes (those within a certain distance chosen by the algorithm) in a weighted least-squares sense. The weights are chosen such that closer nodes have more influence than more distant nodes on derivative estimates at node r. The computed partial derivatives, with the f_r values, at the three nodes of each triangle define a piecewise polynomial surface of a certain form which is the interpolant on that triangle. See Renka and Cline [4] for more detailed information on the algorithm, a development of that by Lawson [2]. The code is derived from Renka [3].

The interpolant F(x,y) can subsequently be evaluated at any point

(x,y) inside or outside the domain of the data by a call to
E01SBF. Points outside the domain are evaluated by extrapolation.

4. References

[1] Cline A K and Renka R L (1984) A Storage-efficient Method for Construction of a Thiessen Triangulation. Rocky Mountain J. Math. 14 119--139.

[2] Lawson C L (1977) Software for C^1 Surface Interpolation. Mathematical Software III. (ed J R Rice) Academic Press. 161--194.

[3] Renka R L (1984) Algorithm 624: Triangulation and Interpolation of Arbitrarily Distributed Points in the Plane. ACM Trans. Math. Softw. 10 440--442.

[4] Renka R L and Cline A K (1984) A Triangle-based C^1 Interpolation Method. Rocky Mountain J. Math. 14 223--237.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of data points. Constraint: M >= 3.

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input
On entry: the co-ordinates of the rth data point, for r=1,2,...,m. The data points are accepted in any order, but see Section 8. Constraint: The (x,y) nodes must not all be collinear, and each node must be unique.

5: TRIANG(7*M) -- INTEGER array Output
On exit: a data structure defining the computed triangulation, in a form suitable for passing to E01SBF.

6: GRADS(2,M) -- DOUBLE PRECISION array Output
On exit: the estimated partial derivatives at the nodes, in a form suitable for passing to E01SBF. The derivatives at node r with respect to x and y are contained in GRADS(1,r) and GRADS(2,r) respectively, for r=1,2,...,m.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential

```
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 3.

IFAIL= 2
     On entry all the (X,Y) pairs are collinear.

IFAIL= 3
     On entry (X(i),Y(i)) = (X(j),Y(j)) for some i/=j.

7. Accuracy

On successful exit, the computational errors should be negligible
in most situations but the user should always check the computed
surface for acceptability, by drawing contours for instance. The
surface always interpolates the input data exactly.

8. Further Comments

The time taken for a call of E01SAF is approximately proportional
to the number of data points, m. The routine is more efficient
if, before entry, the values in X, Y, F are arranged so that the
X array is in ascending order.

9. Example

This program reads in a set of 30 data points and calls E01SAF to
construct an interpolating surface. It then calls E01SBF to
evaluate the interpolant at a sample of points on a rectangular
grid.

Note that this example is not typical of a realistic problem: the
number of data points would normally be larger, and the
interpolant would need to be evaluated on a finer grid to obtain
an accurate plot, say.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

E01 -- Interpolation E01SBF
E01SBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01SBF evaluates at a given point the two-dimensional interpolant function computed by E01SAF.

2. Specification

```
      SUBROUTINE E01SBF (M, X, Y, F, TRIANG, GRADS, PX, PY, PF,
     1                   IFAIL)
      INTEGER          M, TRIANG(7*M), IFAIL
      DOUBLE PRECISION X(M), Y(M), F(M), GRADS(2,M), PX, PY, PF
```

3. Description

This routine takes as input the parameters defining the interpolant F(x,y) of a set of scattered data points (x_r, y_r, f_r), for r=1,2,...,m, as computed by E01SAF, and evaluates the interpolant at the point (px,py).

If (px,py) is equal to (x_r, y_r) for some value of r, the returned value will be equal to f_r.

If (px,py) is not equal to (x_r, y_r) for any r, the derivatives in GRADS will be used to compute the interpolant. A triangle is sought which contains the point (px,py), and the vertices of the triangle along with the partial derivatives and f_r values at the vertices are used to compute the value F(px,py). If the point (px,py) lies outside the triangulation defined by the input parameters, the returned value is obtained by extrapolation. In this case, the interpolating function F is extended linearly beyond the triangulation boundary. The method is described in more detail in Renka and Cline [2] and the code is derived from Renka [1].

E01SBF must only be called after a call to E01SAF.

4. References

[1] Renka R L (1984) Algorithm 624: Triangulation and Interpolation of Arbitrarily Distributed Points in the Plane. ACM Trans. Math. Softw. 10 440--442.

[2] Renka R L and Cline A K (1984) A Triangle-based C^1 Interpolation Method. Rocky Mountain J. Math. 14 223--237.

5. Parameters

1: M -- INTEGER Input

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input

5: TRIANG(7*M) -- INTEGER array Input

6: GRADS(2,M) -- DOUBLE PRECISION array Input
On entry: M, X, Y, F, TRIANG and GRADS must be unchanged from the previous call of E01SAF.

7: PX -- DOUBLE PRECISION Input

8: PY -- DOUBLE PRECISION Input
On entry: the point (px,py) at which the interpolant is to be evaluated.

9: PF -- DOUBLE PRECISION Output
On exit: the value of the interpolant evaluated at the point (px,py).

10: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

```
If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 3.

IFAIL= 2
     On entry the triangulation information held in the array
     TRIANG does not specify a valid triangulation of the data
     points. TRIANG may have been corrupted since the call to
     E01SAF.

IFAIL= 3
     The evaluation point (PX,PY) lies outside the nodal
     triangulation, and the value returned in PF is computed by
     extrapolation.

7. Accuracy

Computational errors should be negligible in most practical
situations.

8. Further Comments

The time taken for a call of E01SBF is approximately proportional
to the number of data points, m.

The results returned by this routine are particularly suitable
for applications such as graph plotting, producing a smooth
surface from a number of scattered points.

9. Example

See the example for E01SAF.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
E01 -- Interpolation                                          E01SEF
        E01SEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose
```

E01SEF generates a two-dimensional surface interpolating a set of scattered data points, using a modified Shepard method.

2. Specification

```
      SUBROUTINE E01SEF (M, X, Y, F, RNW, RNQ, NW, NQ, FNODES,
     1                   MINNQ, WRK, IFAIL)
      INTEGER          M, NW, NQ, MINNQ, IFAIL
      DOUBLE PRECISION X(M), Y(M), F(M), RNW, RNQ, FNODES(5*M),
     1                 WRK(6*M)
```

3. Description

This routine constructs an interpolating surface F(x,y) through a set of m scattered data points (x_r, y_r, f_r), for r=1,2,...,m, using a modification of Shepard's method. The surface is continuous and has continuous first derivatives.

The basic Shepard method, described in [2], interpolates the input data with the weighted mean

$$F(x,y) = \frac{\sum_{r=1}^{m} w_r(x,y) f_r}{\sum_{r=1}^{m} w_r(x,y)},$$

$$\text{where } w_r(x,y) = \frac{1}{d_r^2} \quad \text{and} \quad d_r = (x-x_r)^2 + (y-y_r)^2 .$$

The basic method is global in that the interpolated value at any point depends on all the data, but this routine uses a modification due to Franke and Nielson described in [1], whereby the method becomes local by adjusting each $w_r(x,y)$ to be zero outside a circle with centre (x_r, y_r) and some radius R_w. Also, to improve the performance of the basic method, each f_r above is

replaced by a function f (x,y), which is a quadratic fitted by
                        r
weighted least-squares to data local to (x ,y) and forced to
                                          r r
interpolate (x ,y ,f). In this context, a point (x,y) is defined
             r r r
to be local to another point if it lies within some distance R
                                                              q
of it. Computation of these quadratics constitutes the main work done by this routine. If there are less than 5 other points within distance R from (x ,y), the quadratic is replaced by a
                q   r r
linear function. In cases of rank-deficiency, the minimum norm solution is computed.

The user may specify values for R and R , but it is usually
                                 w   q
easier to choose instead two integers N and N , from which the
                                       w   q
routine will compute R and R . These integers can be thought of
                      w   q
as the average numbers of data points lying within distances R
                                                              w
and R respectively from each node. Default values are provided,
     q
and advice on alternatives is given in Section 8.2.

The interpolant F(x,y) generated by this routine can subsequently be evaluated for any point (x,y) in the domain of the data by a call to E01SFF.

4. References

[1] Franke R and Nielson G (1980) Smooth Interpolation of Large Sets of Scattered Data. Internat. J. Num. Methods Engrg. 15 1691--1704.

[2] Shepard D (1968) A Two-dimensional Interpolation Function for Irregularly Spaced Data. Proc. 23rd Nat. Conf. ACM. Brandon/Systems Press Inc., Princeton. 517--523.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of data points. Constraint: M >= 3.

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input
On entry: the co-ordinates of the rth data point, for r=1,2,...,m. The order of the data points is immaterial. Constraint: each of the (X(r),Y(r)) pairs must be unique.

5: RNW -- DOUBLE PRECISION Input/Output

6: RNQ -- DOUBLE PRECISION Input/Output
On entry: suitable values for the radii R_w and R_q, described in Section 3. Constraint: RNQ <= 0 or 0 < RNW <= RNQ. On exit: if RNQ is set less than or equal to zero on entry, then default values for both of them will be computed from the parameters NW and NQ, and RNW and RNQ will contain these values on exit.

7: NW -- INTEGER Input

8: NQ -- INTEGER Input
On entry: if RNQ > 0.0 and RNW > 0.0 then NW and NQ are not referenced by the routine. Otherwise, NW and NQ must specify suitable values for the integers N_w and N_q described in Section 3.

If NQ is less than or equal to zero on entry, then default values for both of them, namely NW = 9 and NQ = 18, will be used. Constraint: NQ <= 0 or 0 < NW <= NQ.

9: FNODES(5*M) -- DOUBLE PRECISION array Output
On exit: the coefficients of the constructed quadratic nodal functions. These are in a form suitable for passing to E01SFF.

10: MINNQ -- INTEGER Output
On exit: the minimum number of data points that lie within radius RNQ of any node, and thus define a nodal function. If MINNQ is very small (say, less than 5), then the interpolant may be unsatisfactory in regions where the data points are sparse.

11: WRK(6*M) -- DOUBLE PRECISION array Workspace

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

```
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 3.

IFAIL= 2
     On entry RNQ > 0 and either RNW > RNQ or RNW <= 0.

IFAIL= 3
     On entry NQ > 0 and either NW > NQ or NW <= 0.

IFAIL= 4
     On entry (X(i),Y(i)) is equal to (X(j),Y(j)) for some i/=j.

7. Accuracy

On successful exit, the computational errors should be negligible
in most situations but the user should always check the computed
surface for acceptability, by drawing contours for instance. The
surface always interpolates the input data exactly.

8. Further Comments

8.1. Timing

The time taken for a call of E01SEF is approximately proportional
to the number of data points, m, provided that N  is of the same
                                                q
order as its default value (18). However if N  is increased so
                                            q
that the method becomes more global, the time taken becomes
                                 2
approximately proportional to m .
                 m N} {{\rm w}}$ and ${\rm N} {{\rm q}}$
8.2. Choice of ${\
Note first that the radii R  and R , described in Section 3, are
                          w      q


                    / N            / N
             D   /   w      D   /   q
computed as  -  /   -- and  -  /   -- respectively, where D is
             2\/    m       2/    m
the maximum distance between any pair of data points.
```

```
Default values N =9 and N =18 work quite well when the data
                w        q
points are fairly uniformly distributed. However, for data having
some regions with relatively few points or for small data sets
(m<25), a larger value of N  may be needed. This is to ensure a
                           w
reasonable number of data points within a distance R  of each
                                                    w
node, and to avoid some regions in the data area being left
outside all the discs of radius R  on which the weights w (x,y)
                                 w                        r
are non-zero. Maintaining N  approximately equal to 2N  is
                           q                          w
usually an advantage.

Note however that increasing N  and N  does not improve the
                             w      q
quality of the interpolant in all cases. It does increase the
computational cost and makes the method less local.

9. Example

This program reads in a set of 30 data points and calls E01SEF to
construct an interpolating surface. It then calls E01SFF to
evaluate the interpolant at a sample of points on a rectangular
grid.

Note that this example is not typical of a realistic problem: the
number of data points would normally be larger, and the
interpolant would need to be evaluated on a finer grid to obtain
an accurate plot, say.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E01 -- Interpolation                                        E01SFF
        E01SFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E01SFF evaluates at a given point the two-dimensional
interpolating function computed by E01SEF.
```

2. Specification

```
      SUBROUTINE E01SFF (M, X, Y, F, RNW, FNODES, PX, PY, PF,
     1                   IFAIL)
      INTEGER          M, IFAIL
      DOUBLE PRECISION X(M), Y(M), F(M), RNW, FNODES(5*M), PX,
     1                 PY, PF
```

3. Description

This routine takes as input the interpolant F(x,y) of a set of scattered data points (x_r, y_r, f_r), for r=1,2,...,m, as computed by E01SEF, and evaluates the interpolant at the point (px,py).

If (px,py) is equal to (x_r, y_r) for some value of r, the returned value will be equal to f_r.

If (px,py) is not equal to (x_r, y_r) for any r, all points that are within distance RNW of (px,py), along with the corresponding nodal functions given by FNODES, will be used to compute a value of the interpolant.

E01SFF must only be called after a call to E01SEF.

4. References

[1] Franke R and Nielson G (1980) Smooth Interpolation of Large Sets of Scattered Data. Internat. J. Num. Methods Engrg. 15 1691--1704.

[2] Shepard D (1968) A Two-dimensional Interpolation Function for Irregularly Spaced Data. Proc. 23rd Nat. Conf. ACM. Brandon/Systems Press Inc., Princeton. 517--523.

5. Parameters

1: M -- INTEGER Input

2: X(M) -- DOUBLE PRECISION array Input

3: Y(M) -- DOUBLE PRECISION array Input

4: F(M) -- DOUBLE PRECISION array Input

5: RNW -- DOUBLE PRECISION Input

```
 6:  FNODES(5*M) -- DOUBLE PRECISION array                  Input
     On entry: M, X, Y, F, RNW and FNODES must be unchanged from
     the previous call of E01SEF.

 7:  PX -- DOUBLE PRECISION                                 Input

 8:  PY -- DOUBLE PRECISION                                 Input
     On entry: the point (px,py) at which the interpolant is to
     be evaluated.

 9:  PF -- DOUBLE PRECISION                                Output
     On exit: the value of the interpolant evaluated at the
     point (px,py).

10:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 3.

IFAIL= 2
     The interpolant cannot be evaluated because the evaluation
     point (PX,PY) lies outside the support region of the data
     supplied in X, Y and F. This error exit will occur if
     (PX,PY) lies at a distance greater than or equal to RNW from
     every point given by arrays X and Y.

     The value 0.0 is returned in PF. This value will not provide
     continuity with values obtained at other points (PX,PY),
     i.e., values obtained when IFAIL = 0 on exit.

7. Accuracy

Computational errors should be negligible in most practical
situations.

8. Further Comments
```

```
The time taken for a call of E01SFF is approximately proportional
to the number of data points, m.

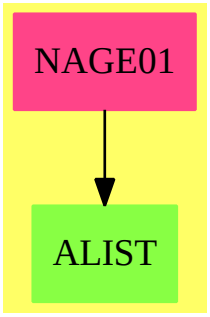
The results returned by this routine are particularly suitable
for applications such as graph plotting, producing a smooth
surface from a number of scattered points.

9. Example

See the example for E01SEF.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.6.1 NagInterpolationPackage (NAGE01)



Exports:

e01baf	e01bef	e01bff	e01bgf	e01bhf
e01daf	e01saf	e01sbf	e01sef	e01sff

— package NAGE01 NagInterpolationPackage —

```
)abbrev package NAGE01 NagInterpolationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:53 1994
++ Description:
++ This package uses the NAG Library to calculate the interpolation of a
++ function of one or two variables. When provided with the value of the
++ function (and possibly one or more of its lowest-order
```

```
++ derivatives) at each of a number of values of the variable(s),
++ the routines provide either an interpolating function or an
++ interpolated value. For some of the interpolating functions,
++ there are supporting routines to evaluate, differentiate or
++ integrate them.

NagInterpolationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    e01baf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
        Integer,Integer) -> Result
     ++ e01baf(m,x,y,lck,lwrk,ifail)
     ++ determines a cubic spline to a given set of
     ++ data.
     ++ See \downlink{Manual Page}{manpageXXe01baf}.
    e01bef : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
     ++ e01bef(n,x,f,ifail)
     ++ computes a monotonicity-preserving piecewise cubic Hermite
     ++ interpolant to a set of data points.
     ++ See \downlink{Manual Page}{manpageXXe01bef}.
    e01bff: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer,Matrix DoubleFloat,Integer) -> Result
     ++ e01bff(n,x,f,d,m,px,ifail)
     ++ evaluates a piecewise cubic Hermite interpolant at a set
     ++ of points.
     ++ See \downlink{Manual Page}{manpageXXe01bff}.
    e01bgf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer,Matrix DoubleFloat,Integer) -> Result
     ++ e01bgf(n,x,f,d,m,px,ifail)
     ++ evaluates a piecewise cubic Hermite interpolant and its
     ++ first derivative at a set of points.
     ++ See \downlink{Manual Page}{manpageXXe01bgf}.
    e01bhf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        DoubleFloat,DoubleFloat,Integer) -> Result
     ++ e01bhf(n,x,f,d,a,b,ifail)
     ++ evaluates the definite integral of a piecewise cubic
     ++ Hermite interpolant over the interval [a,b].
     ++ See \downlink{Manual Page}{manpageXXe01bhf}.
    e01daf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
     ++ e01daf(mx,my,x,y,f,ifail)
     ++ computes a bicubic spline interpolating surface through a
     ++ set of data values, given on a rectangular grid in the x-y plane.
     ++ See \downlink{Manual Page}{manpageXXe01daf}.
    e01saf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer) -> Result
     ++ e01saf(m,x,y,f,ifail)
     ++ generates a two-dimensional surface interpolating a set of
```

```
    ++ scattered data points, using the method of Renka and Cline.
    ++ See \downlink{Manual Page}{manpageXXe01saf}.
  e01sbf: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      Matrix Integer,Matrix DoubleFloat,DoubleFloat,DoubleFloat,_
      Integer) -> Result
    ++ e01sbf(m,x,y,f,triang,grads,px,py,ifail)
    ++ evaluates at a given point the two-dimensional interpolant
    ++ function computed by E01SAF.
    ++ See \downlink{Manual Page}{manpageXXe01sbf}.
  e01sef: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      Integer,Integer,DoubleFloat,DoubleFloat,Integer) -> Result
    ++ e01sef(m,x,y,f,nw,nq,rnw,rnq,ifail)
    ++ generates a two-dimensional surface interpolating a set of
    ++ scattered data points, using a modified Shepard method.
    ++ See \downlink{Manual Page}{manpageXXe01sef}.
  e01sff: (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      DoubleFloat,Matrix DoubleFloat,DoubleFloat,DoubleFloat,_
      Integer) -> Result
    ++ e01sff(m,x,y,f,rnw,fnodes,px,py,ifail)
    ++ evaluates at a given point the two-dimensional
    ++ interpolating function computed by E01SEF.
    ++ See \downlink{Manual Page}{manpageXXe01sff}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
  import AnyFunctions1(Integer)
  import AnyFunctions1(Matrix DoubleFloat)
  import AnyFunctions1(Matrix Integer)
  import AnyFunctions1(DoubleFloat)


  e01baf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
      lckArg:Integer,lwrkArg:Integer,ifailArg:Integer): Result ==
      [(invokeNagman(NIL$Lisp,_
      "e01baf",_
      ["m"::S,"lck"::S,"lwrk"::S,"ifail"::S,"x"::S,"y"::S,_
      "lamda"::S,"c"::S,"wrk"::S_
      ]$Lisp,_
      ["lamda"::S,"c"::S,"wrk"::S]$Lisp,_
      [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
      ,["lamda"::S,"lck"::S]$Lisp,["c"::S,"lck"::S]$Lisp,_
      ["wrk"::S,"lwrk"::S]$Lisp]$Lisp_
      ,["integer"::S,"m"::S,"lck"::S,"lwrk"::S,"ifail"::S_
```

```
        ]$Lisp_
        ]$Lisp,_
        ["lamda"::S,"c"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,lckArg::Any,lwrkArg::Any,ifailArg::Any,_
        xArg::Any,yArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01bef(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01bef",_
        ["n"::S,"ifail"::S,"x"::S,"f"::S,"d"::S]$Lisp,_
        ["d"::S]$Lisp,_
        [["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp_
        ,["d"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["d"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,ifailArg::Any,xArg::Any,fArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01bff(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
        dArg:Matrix DoubleFloat,mArg:Integer,pxArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01bff",_
        ["n"::S,"m"::S,"ifail"::S,"x"::S,"f"::S,"d"::S,"px"::S,"pf"::S_
        ]$Lisp,_
        ["pf"::S]$Lisp,_
        [["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp_
        ,["d"::S,"n"::S]$Lisp,["px"::S,"m"::S]$Lisp,_
        ["pf"::S,"m"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"m"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["pf"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,mArg::Any,ifailArg::Any,xArg::Any,fArg::Any,_
        dArg::Any,pxArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01bgf(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
        dArg:Matrix DoubleFloat,mArg:Integer,pxArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01bgf",_
        ["n"::S,"m"::S,"ifail"::S,"x"::S,"f"::S,"d"::S,"px"::S,"pf"::S_
        ,"pd"::S]$Lisp,_
        ["pf"::S,"pd"::S]$Lisp,_
```

```
    [["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp_
    ,["d"::S,"n"::S]$Lisp,["px"::S,"m"::S]$Lisp,_
    ["pf"::S,"m"::S]$Lisp,["pd"::S,"m"::S]$Lisp]$Lisp_
    ,["integer"::S,"n"::S,"m"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["pf"::S,"pd"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,mArg::Any,ifailArg::Any,xArg::Any,_
    fArg::Any,dArg::Any,pxArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e01bhf(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
    dArg:Matrix DoubleFloat,aArg:DoubleFloat,bArg:DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e01bhf",_
    ["n"::S,"a"::S,"b"::S,"pint"::S,"ifail"::S_
    ,"x"::S,"f"::S,"d"::S]$Lisp,_
    ["pint"::S]$Lisp,_
    [["double"::S,["x"::S,"n"::S]$Lisp,["f"::S,"n"::S]$Lisp_
    ,["d"::S,"n"::S]$Lisp,"a"::S,"b"::S,"pint"::S]$Lisp_
    ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["pint"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,aArg::Any,bArg::Any,ifailArg::Any,xArg::Any,_
    fArg::Any,dArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e01daf(mxArg:Integer,myArg:Integer,xArg:Matrix DoubleFloat,_
    yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e01daf",_
    ["mx"::S,"my"::S,"px"::S,"py"::S,"ifail"::S_
    ,"x"::S,"y"::S,"f"::S,"lamda"::S,"mu"::S_
    ,"c"::S,"wrk"::S]$Lisp,_
    ["px"::S,"py"::S,"lamda"::S,"mu"::S,"c"::S,"wrk"::S]$Lisp,_
    [["double"::S,["x"::S,"mx"::S]$Lisp,["y"::S,"my"::S]$Lisp_
    ,["f"::S,["*"::S,"mx"::S,"my"::S]$Lisp]$Lisp,_
    ["lamda"::S,["+"::S,"mx"::S,4$Lisp]$Lisp]$Lisp,_
    ["mu"::S,["+"::S,"mx"::S,4$Lisp]$Lisp]$Lisp_
    ,["c"::S,["*"::S,"mx"::S,"my"::S]$Lisp]$Lisp,_
    ["wrk"::S,["*"::S,["+"::S,"mx"::S,6$Lisp]$Lisp,_
    ["+"::S,"my"::S,6$Lisp]$Lisp]$Lisp]$Lisp_
    ]$Lisp_
    ,["integer"::S,"mx"::S,"my"::S,"px"::S,"py"::S_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["px"::S,"py"::S,"lamda"::S,"mu"::S,"c"::S,"ifail"::S]$Lisp,_
```

```
        [([mxArg::Any,myArg::Any,ifailArg::Any,xArg::Any,_
        yArg::Any,fArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01saf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        fArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01saf",_
        ["m"::S,"ifail"::S,"x"::S,"y"::S,"f"::S,"triang"::S,"grads"::S_
        ]$Lisp,_
        ["triang"::S,"grads"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["f"::S,"m"::S]$Lisp,["grads"::S,2$Lisp,"m"::S]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,["triang"::S,["*"::S,7$Lisp,"m"::S]$Lisp]$Lisp_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["triang"::S,"grads"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,ifailArg::Any,xArg::Any,yArg::Any,fArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01sbf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        fArg:Matrix DoubleFloat,triangArg:Matrix Integer,_
        gradsArg:Matrix DoubleFloat,_
        pxArg:DoubleFloat,pyArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01sbf",_
        ["m"::S,"px"::S,"py"::S,"pf"::S,"ifail"::S_
        ,"x"::S,"y"::S,"f"::S,"triang"::S,"grads"::S_
        ]$Lisp,_
        ["pf"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["f"::S,"m"::S]$Lisp,["grads"::S,2$Lisp,"m"::S]$Lisp,_
        "px"::S,"py"::S,"pf"::S]$Lisp_
        ,["integer"::S,"m"::S,["triang"::S,["*"::S,7$Lisp,"m"::S]$Lisp]$Lisp_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["pf"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,pxArg::Any,pyArg::Any,ifailArg::Any,xArg::Any,_
        yArg::Any,fArg::Any,triangArg::Any,gradsArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01sef(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        fArg:Matrix DoubleFloat,nwArg:Integer,nqArg:Integer,_
        rnwArg:DoubleFloat,rnqArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01sef",_
        ["m"::S,"nw"::S,"nq"::S,"minnq"::S,"rnw"::S_
```

```
        ,"rnq"::S,"ifail"::S,"x"::S,"y"::S,"f"::S,"fnodes"::S,"wrk"::S_
        ]$Lisp,_
        ["fnodes"::S,"minnq"::S,"wrk"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["f"::S,"m"::S]$Lisp,["fnodes"::S,["*"::S,5$Lisp,"m"::S]$Lisp]$Lisp,_
        "rnw"::S,"rnq"::S,["wrk"::S,["*"::S,6$Lisp,"m"::S]$Lisp]$Lisp_
        ]$Lisp_
        ,["integer"::S,"m"::S,"nw"::S,"nq"::S,"minnq"::S_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["fnodes"::S,"minnq"::S,"rnw"::S,"rnq"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nwArg::Any,nqArg::Any,rnwArg::Any,rnqArg::Any,_
        ifailArg::Any,xArg::Any,yArg::Any,fArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e01sff(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        fArg:Matrix DoubleFloat,rnwArg:DoubleFloat,_
        fnodesArg:Matrix DoubleFloat,_
        pxArg:DoubleFloat,pyArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e01sff",_
        ["m"::S,"rnw"::S,"px"::S,"py"::S,"pf"::S_
        ,"ifail"::S,"x"::S,"y"::S,"f"::S,"fnodes"::S]$Lisp,_
        ["pf"::S]$Lisp,_
        [["double"::S,["x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_
        ,["f"::S,"m"::S]$Lisp,"rnw"::S,["fnodes"::S,_
        ["*"::S,5$Lisp,"m"::S]$Lisp]$Lisp,"px"::S,"py"::S,"pf"::S]$Lisp_
        ,["integer"::S,"m"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["pf"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,rnwArg::Any,pxArg::Any,pyArg::Any,_
        ifailArg::Any,xArg::Any,yArg::Any,fArg::Any,fnodesArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGE01.dotabb** —

```
"NAGE01" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGE01"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE01" -> "ALIST"
```

———-

15.7 package NAGF07 NagLapack

— **NagLapack.help** —

```
F07 -- Linear Equations (LAPACK)              Introduction -- F07
                          Chapter F07
                 Linear Equations (LAPACK)

1. Scope of the Chapter

This chapter provides four routines concerned with matrix
factorization, and the solution of systems of linear equations
following the matrix factorizations.

2. Background to the Problems

Background material, together with pointers to the routines in
this chapter, are to be found in the F01 and F04 Chapter
Introductions.

3. Recommendations on Choice and Use of Routines

The routines in this chapter are derived from the LAPACK project
and may also be called using the LAPACK name, which is given in
brackets following the F07 name in the following descriptions.

Routine F07ADF (DGETRF) performs an LU factorization of a real m
by n matrix A. Following the use of this routine, F07AEF (DGETRS)
may be used to solve a system of n non-singular linear equations,
with one or more right-hand sides.

Routine F07FDF (DPOTRF) performs the Cholesky factorization of a
real symmetric positive-definite matrix A. Following the use of
this routine, F07FEF (DPOTRS) may be used to solve a system of
symmetric positive-definite linear equations, with one or more
right-hand sides.


F07 -- Linear Equations (LAPACK)                  Contents -- F07
Chapter F07

Linear Equations (LAPACK)

F07ADF  (DGETRF) LU factorization of real m by n matrix

F07AEF  (DGETRS) Solution of real system of linear equations,
        multiple right-hand sides, matrix already factorized by
        F07ADF
```

```
          F07FDF  (DPOTRF) Cholesky factorization of real symmetric
                  positive-definite matrix

          F07FEF  (DPOTRS) Solution of real symmetric positive-definite
                  system of linear equations, multiple right-hand sides,
                  matrix already factorized by F07FDF

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          F07 -- Linear Equations (LAPACK)                          F07ADF
                  F07ADF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          F07ADF (DGETRF) computes the LU factorization of a real m by n
          matrix.

          2. Specification

                 SUBROUTINE F07ADF (M, N, A, LDA, IPIV, INFO)
                 ENTRY            M, N, A, LDA, IPIV, INFO
                 INTEGER          M, N, LDA, IPIV(*), INFO
                 DOUBLE PRECISION A(LDA,*)

          The ENTRY statement enables the routine to be called by its
          LAPACK name.

          3. Description

          This routine forms the LU factorization of a real m by n matrix A
          as A=PLU, where P is a permutation matrix, L is lower triangular
          with unit diagonal elements (lower trapezoidal if m>n) and U is
          upper triangular (upper trapezoidal if m<n). Usually A is square
          (m=n), and both L and U are triangular. The routine uses partial
          pivoting, with row interchanges.

          4. References

          [1]   Golub G H and Van Loan C F (1989) Matrix Computations (2nd
                Edition). Johns Hopkins University Press, Baltimore,
                Maryland.

          5. Parameters
```

```
 1:  M -- INTEGER                                              Input
     On entry: m, the number of rows of the matrix A.
     Constraint: M >= 0.

 2:  N -- INTEGER                                              Input
     On entry: n, the number of columns of the matrix A.
     Constraint: N >= 0.

 3:  A(LDA,*) -- DOUBLE PRECISION array                  Input/Output
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the m by n matrix A. On exit: A is overwritten by
     the factors L and U; the unit diagonal elements of L are not
     stored.

 4:  LDA -- INTEGER                                            Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F07ADF is called.
     Constraint: LDA >= max(1,M).

 5:  IPIV(*) -- INTEGER array                                 Output
     Note: the dimension of the array IPIV must be at least
     max(1,min(M,N)).
     On exit: the pivot indices. Row i of the matrix A was
     interchanged with row IPIV(i) for i=1,2,...,min(m,n).

 6:  INFO -- INTEGER                                          Output
     On exit: INFO = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

INFO < 0
     If INFO = -i, the ith parameter has an illegal value. An
     explanatory message is output, and execution of the program
     is terminated.

INFO > 0
     If INFO = i, u   is exactly zero. The factorization has been
                   ii
     completed but the factor U is exactly singular, and division
     by zero will occur if it is subsequently used to solve a
                                                      -1
     system of linear equations or to compute A   .

7. Accuracy

The computed factors L and U are the exact factors of a perturbed
matrix A+E, where
```

```
                |E|<=c(min(m,n))(epsilon)P|L||U|,

c(n) is a modest linear function of n, and (epsilon) is the
machine precision.

8. Further Comments

The total number of floating-point operations is approximately
 2 3                          1 2                  1 2
 -n  if m=n (the usual case), -n (3m-n) if m>n and -m (3n-m) if
 3                            3                    3
m<n.

A call to this routine with m=n may be followed by calls to the
routines:

                                        T
     F07AEF (DGETRS) to solve AX=B or A X=B;

     F07AGF (DGECON)(*) to estimate the condition number of A;

     F07AJF (DGETRI)(*) to compute the inverse of A.

The complex analogue of this routine is F07ARF (ZGETRF)(*).

9. Example

To compute the LU factorization of the matrix A, where

                   ( 1.80  2.88  2.05 -0.89)
                   ( 5.25 -2.95 -0.95 -3.80)
                 A=( 1.58 -2.69 -2.90 -1.04).
                   (-1.11 -0.66 -0.59  0.80)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F07 -- Linear Equations (LAPACK)                          F07AEF
       F07AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose
```

```
F07AEF (DGETRS) solves a real system of linear equations with
                                    T
multiple right-hand sides, AX=B or A X=B, where A has been
factorized by F07ADF (DGETRF).

2. Specification

       SUBROUTINE F07AEF (TRANS, N, NRHS, A, LDA, IPIV, B, LDB,
      1                   INFO)
       ENTRY             TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO
       INTEGER           N, NRHS, LDA, IPIV(*), LDB, INFO
       DOUBLE PRECISION A(LDA,*), B(LDB,*)
       CHARACTER*1       TRANS

The ENTRY statement enables the routine to be called by its
LAPACK name.

3. Description

                                                          T
To solve a real system of linear equations AX=B or A X=B, this
routine must be preceded by a call to F07ADF (DGETRF)which
computes the LU factorization of A as A=PLU. The solution is
computed by forward and backward substitution.

If TRANS = 'N', the solution is computed by solving PLY=B and
then UX=Y.

                                                                T
If TRANS = 'T' or 'C', the solution is computed by solving U Y=B
          T T
and then L P X=Y.

4. References

[1]   Golub G H and Van Loan C F (1989) Matrix Computations (2nd
      Edition). Johns Hopkins University Press, Baltimore,
      Maryland.

5. Parameters

 1:  TRANS -- CHARACTER*1                                    Input
     On entry: indicates the form of the equations as follows:
          if TRANS = 'N', then AX=B is solved for X;

                                      T
          if TRANS = 'T' or 'C', then A X=B is solved for X.
      Constraint: TRANS = 'N', 'T' or 'C'.
```

```
 2:  N -- INTEGER                                              Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 3:  NRHS -- INTEGER                                           Input
     On entry: r, the number of right-hand sides. Constraint:
     NRHS >= 0.

 4:  A(LDA,*) -- DOUBLE PRECISION array                        Input
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the LU factorization of A, as returned by F07ADF
     (DGETRF).

 5:  LDA -- INTEGER                                            Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F07AEF is called.
     Constraint: LDA >= max(1,N).

 6:  IPIV(*) -- INTEGER array                                  Input
     Note: the dimension of the array IPIV must be at least
     max(1,N).
     On entry: the pivot indices, as returned by F07ADF (DGETRF).

 7:  B(LDB,*) -- DOUBLE PRECISION array              Input/Output
     Note: the second dimension of the array B must be at least
     max(1,NRHS).
     On entry: the n by r right-hand side matrix B. On exit: the
     n by r solution matrix X.

 8:  LDB -- INTEGER                                            Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F07AEF is called.
     Constraint: LDB >= max(1,N).

 9:  INFO -- INTEGER                                          Output
     On exit: INFO = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

INFO < 0
     If INFO = -i, the ith parameter has an illegal value. An
     explanatory message is output, and execution of the program
     is terminated.

7. Accuracy

For each right-hand side vector b, the computed solution x is the
```

```
exact solution of a perturbed system of equations (A+E)x=b, where

                  |E|<=c(n)(epsilon)P|L||U|,

c(n) is a modest linear function of n, and (epsilon) is the
machine precision.

   ^
If x is the true solution, then the computed solution x satisfies
a forward error bound of the form

                   ^
              ||x-x||
                     infty
              ------------ <= c(n)cond(A,x)(epsilon)
               ||x||
                     infty

                     -1
where cond(A,x)=|||A  ||A||x|||     /||x||      <=
                                infty      infty
            -1
cond(A)=|||A  ||A|||     <=(kappa)     (A). Note that cond(A,x)
                     infty         infty
                                           T
can be much smaller than cond(A), and cond(A ) can be much larger
(or smaller) than cond(A).

Forward and backward error bounds can be computed by calling
F07AHF (DGERFS)(*), and an estimate for (kappa)     (A) can be
                                                 infty
obtained by calling F07AGF (DGECON)(*) with NORM ='I'.

8. Further Comments

The total number of floating-point operations is approximately
  2
2n r.

This routine may be followed by a call to F07AHF (DGERFS)(*) to
refine the solution and return an error estimate.

The complex analogue of this routine is F07ASF (ZGETRS)(*).

9. Example

To solve the system of equations AX=B, where

                    ( 1.80  2.88  2.05 -0.89)
                    ( 5.25 -2.95 -0.95 -3.80)
```

```
                        A=( 1.58 -2.69 -2.90 -1.04)
                          (-1.11 -0.66 -0.59  0.80)
```

and

```
                              ( 9.52  18.47)
                              (24.35   2.25)
                            B=( 0.77 -13.28).
                              (-6.22  -6.21)
```

Here A is unsymmetric and must first be factorized by F07ADF (DGETRF)).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F07 -- Linear Equations (LAPACK) F07FDF
F07FDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F07FDF (DPOTRF) computes the Cholesky factorization of a real symmetric positive-definite matrix.

2. Specification

```
SUBROUTINE F07FDF (UPLO, N, A, LDA, INFO)
ENTRY            UPLO, N, A, LDA, INFO
INTEGER          N, LDA, INFO
DOUBLE PRECISION A(LDA,*)
CHARACTER*1      UPLO
```

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

This routine forms the Cholesky factorization of a real symmetric positive-definite matrix A either as $A=U^{T}U$ if UPLO = 'U' or $A=LL^{T}$ if UPLO = 'L', where U is an upper triangular matrix and L is

```
lower triangular.

4. References

[1]   Demmel J W (1989) On Floating-point Errors in Cholesky.
      LAPACK Working Note No. 14. University of Tennessee,
      Knoxville.

[2]   Golub G H and Van Loan C F (1989) Matrix Computations (2nd
      Edition). Johns Hopkins University Press, Baltimore,
      Maryland.

5. Parameters

 1:  UPLO -- CHARACTER*1                                        Input
     On entry: indicates whether the upper or lower triangular
     part of A is stored and how A is factorized, as follows:
          if UPLO = 'U', then the upper triangular part of A is
                                          T
          stored and A is factorized as U U, where U is upper
          triangular;

          if UPLO = 'L', then the lower triangular part of A is
                                           T
          stored and A is factorized as LL , where L is lower
          triangular.
      Constraint: UPLO = 'U' or 'L'.

 2:  N -- INTEGER                                               Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 3:  A(LDA,*) -- DOUBLE PRECISION array              Input/Output
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the n by n symmetric positive-definite matrix A.
     If UPLO = 'U', the upper triangle of A must be stored and
     the elements of the array below the diagonal are not
     referenced; if UPLO = 'L', the lower triangle of A must be
     stored and the elements of the array above the diagonal are
     not referenced. On exit: the upper or lower triangle of A is
     overwritten by the Cholesky factor U or L as specified by
     UPLO.

 4:  LDA -- INTEGER                                             Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F07FDF is called.
     Constraint: LDA >= max(1,N).

 5:  INFO -- INTEGER                                           Output
```

```
     On exit: INFO = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

INFO < 0
     If INFO = -i, the ith parameter has an illegal value. An
     explanatory message is output, and execution of the program
     is terminated.

INFO > 0
     If INFO = i, the leading minor of order i is not positive-
     definite and the factorization could not be completed. Hence
     A itself is not positive-definite. This may indicate an
     error in forming the matrix A. To factorize a symmetric
     matrix which is not positive-definite, call F07MDF
     (DSYTRF)(*) instead.

7. Accuracy

If UPLO = 'U', the computed factor U is the exact factor of a
perturbed matrix A+E, where

                                    T
                  |E|<=c(n)(epsilon)|U ||U|,

c(n) is a modest linear function of n, and (epsilon) is the
machine precision. If UPLO = 'L', a similar statement holds for
the computed factor L. It follows that


|e  |<=c(n)(epsilon)  /a  a  .
  ij                \/  ii jj

8. Further Comments

The total number of floating-point operations is approximately
 1 3
 -n .
 3

A call to this routine may be followed by calls to the routines:

     F07FEF (DPOTRS) to solve AX=B;

     F07FGF (DPOCON)(*) to estimate the condition number of A;

     F07FJF (DPOTRI)(*) to compute the inverse of A.

The complex analogue of this routine is F07FRF (ZPOTRF)(*).
```

```
          9. Example

          To compute the Cholesky factorization of the matrix A, where

                              ( 4.16 -3.12  0.56 -0.10)
                              (-3.12  5.03 -0.83  1.18)
                            A=( 0.56 -0.83  0.76  0.34).
                              (-0.10  1.18  0.34  1.18)

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          F07 -- Linear Equations (LAPACK)                          F07FEF
                 F07FEF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          F07FEF (DPOTRS) solves a real symmetric positive-definite system
          of linear equations with multiple right-hand sides, AX=B, where A
          has been factorized by F07FDF (DPOTRF).

          2. Specification

                 SUBROUTINE F07FEF (UPLO, N, NRHS, A, LDA, B, LDB, INFO)
                 ENTRY            UPLO, N, NRHS, A, LDA, B, LDB, INFO
                 INTEGER          N, NRHS, LDA, LDB, INFO
                 DOUBLE PRECISION A(LDA,*), B(LDB,*)
                 CHARACTER*1      UPLO

          The ENTRY statement enables the routine to be called by its
          LAPACK name.

          3. Description

          To solve a real symmetric positive-definite system of linear
          equations AX=B, this routine must be preceded by a call to
          F07FDF (DPOTRF) which computes the Cholesky factorization of A.
          The solution X is computed by forward and backward substitution.

                            T
          If UPLO = 'U', A=U U, where U is upper triangular; the solution X
```

```
                          T
is computed by solving U Y=B and then UX=Y.

                     T
If UPLO = 'L', A=LL , where L is lower triangular; the solution X
                                        T
is computed by solving LY=B and then L X=Y.

4. References

[1]   Golub G H and Van Loan C F (1989) Matrix Computations (2nd
      Edition). Johns Hopkins University Press, Baltimore,
      Maryland.

5. Parameters

 1:  UPLO -- CHARACTER*1                                        Input
     On entry: indicates whether the upper or lower triangular
     part of A is stored and how A is factorized, as follows:
                                T
          if UPLO = 'U', then A=U U where U is upper triangular;

                                 T
          if UPLO = 'L', then A=LL  where L is lower triangular.
      Constraint: UPLO = 'U' or 'L'.

 2:  N -- INTEGER                                               Input
     On entry: n, the order of the matrix A. Constraint: N >= 0.

 3:  NRHS -- INTEGER                                            Input
     On entry: r, the number of right-hand sides. Constraint:
     NRHS >= 0.

 4:  A(LDA,*) -- DOUBLE PRECISION array                         Input
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the Cholesky factor of A, as returned by F07FDF
     (DPOTRF).

 5:  LDA -- INTEGER                                             Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F07FEF is called.
     Constraint: LDA >=max(1,N).

 6:  B(LDB,*) -- DOUBLE PRECISION array              Input/Output
     Note: the second dimension of the array B must be at least
     max(1,NRHS).
     On entry: the n by r right-hand side matrix B.
```

```
 7:  LDB -- INTEGER                                              Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F07FEF is called.
     Constraint: LDB >=max(1,N).

 8:  INFO -- INTEGER                                            Output
     On exit: INFO = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

INFO < 0
     If INFO = -i, the ith parameter has an illegal value. An
     explanatory message is output, and execution of the program
     is terminated.

7. Accuracy

For each right-hand side vector b, the computed solution x is the
exact solution of a perturbed system of equations (A+E)x=b, where

                          T
     |E|<=c(n)(epsilon)|U ||U| if UPLO = 'U',

                            T
     |E|<=c(n)(epsilon)|L||L | if UPLO = 'L',

c(n) is a modest linear function of n, and (epsilon) is the
machine precision.

   ^
If x is the true solution, then the computed solution x satisfies
a forward bound of the form

                   ^
              ||x-x||
                    infty
              ------------<=c(n)cond(A,x)(epsilon)
               ||x||
                    infty

                   -1
where cond(A,x)=|||A  ||A||x|||     /||x||      <=
                               infty      infty
           -1
cond(A)=|||A  ||A|||     <=(kappa)     (A). Note that cond(A,x)
                   infty         infty
can be much smaller than cond(A).
```

Forward and backward error bounds can be computed by calling F07FHF (DPORFS)(*), and an estimate for (kappa) (A) (infty =(kappa) (A)) can be obtained by calling F07FGF (DPOCON)(*). 1

8. Further Comments

The total number of floating-point operations is approximately $2n^2 r$.

This routine may be followed by a call to F07FHF (DPORFS)(*) to refine the solution and return an error estimate.

The complex analogue of this routine is F07FSF (ZPOTRS)(*).

9. Example

To compute the Cholesky factorization of the matrix A, where

$$A = \begin{pmatrix} 4.16 & -3.12 & 0.56 & -0.10 \\ -3.12 & 5.03 & -0.83 & 1.18 \\ 0.56 & -0.83 & 0.76 & 0.34 \\ -0.10 & 1.18 & 0.34 & 1.18 \end{pmatrix}.$$

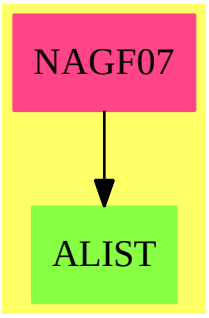
and

$$B = \begin{pmatrix} 8.70 & 8.30 \\ -13.35 & 2.13 \\ 1.89 & 1.61 \\ -4.14 & 5.00 \end{pmatrix}.$$

Here A is symmetric positive-definite and must first be factorized by F07FDF (DPOTRF).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

15.7.1 NagLapack (NAGF07)



Exports:
f07adf f07aef f07fdf f07fef

— package NAGF07 NagLapack —

```
)abbrev package NAGF07 NagLapack
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:42 1994
++ Description:
++ This package uses the NAG Library to compute matrix
++ factorizations, and to solve systems of linear equations
++ following the matrix factorizations.

NagLapack(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    f07adf : (Integer,Integer,Integer,Matrix DoubleFloat) -> Result
     ++ f07adf(m,n,lda,a)
     ++ (DGETRF) computes the LU factorization of a real m by n
     ++ matrix.
     ++ See \downlink{Manual Page}{manpageXXf07adf}.
    f07aef : (String,Integer,Integer,Matrix DoubleFloat,_
        Integer,Matrix Integer,Integer,Matrix DoubleFloat) -> Result
     ++ f07aef(trans,n,nrhs,a,lda,ipiv,ldb,b)
     ++ (DGETRS) solves a real system of linear equations with
     ++                                      T
     ++ multiple right-hand sides, AX=B or A X=B, where A has been
     ++ factorized by F07ADF (DGETRF).
     ++ See \downlink{Manual Page}{manpageXXf07aef}.
    f07fdf : (String,Integer,Integer,Matrix DoubleFloat) -> Result
     ++ f07fdf(uplo,n,lda,a)
     ++ (DPOTRF) computes the Cholesky factorization of a real
```

```
    ++ symmetric positive-definite matrix.
    ++ See \downlink{Manual Page}{manpageXXf07fdf}.
   f07fef : (String,Integer,Integer,Matrix DoubleFloat,_
       Integer,Integer,Matrix DoubleFloat) -> Result
    ++ f07fef(uplo,n,nrhs,a,lda,ldb,b)
    ++ (DPOTRS) solves a real symmetric positive-definite system
    ++ of linear equations with multiple right-hand sides, AX=B, where A
    ++ has been factorized by F07FDF (DPOTRF).
    ++ See \downlink{Manual Page}{manpageXXf07fef}.
 Implementation ==> add

   import Lisp
   import DoubleFloat
   import Any
   import Record
   import Integer
   import Matrix DoubleFloat
   import Boolean
   import NAGLinkSupportPackage
   import AnyFunctions1(Integer)
   import AnyFunctions1(Matrix DoubleFloat)
   import AnyFunctions1(String)
   import AnyFunctions1(Matrix Integer)


   f07adf(mArg:Integer,nArg:Integer,ldaArg:Integer,_
       aArg:Matrix DoubleFloat): Result ==
       [(invokeNagman(NIL$Lisp,_
       "f07adf",_
       ["m"::S,"n"::S,"lda"::S,"info"::S,"ipiv"::S,"a"::S]$Lisp,_
       ["ipiv"::S,"info"::S]$Lisp,_
       [["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp_
       ]$Lisp_
       ,["integer"::S,"m"::S,"n"::S,"lda"::S,["ipiv"::S,"m"::S]$Lisp_
       ,"info"::S]$Lisp_
       ]$Lisp,_
       ["ipiv"::S,"info"::S,"a"::S]$Lisp,_
       [([mArg::Any,nArg::Any,ldaArg::Any,aArg::Any ])_
       @List Any]$Lisp)$Lisp)_
       pretend List (Record(key:Symbol,entry:Any))]$Result

   f07aef(transArg:String,nArg:Integer,nrhsArg:Integer,_
       aArg:Matrix DoubleFloat,ldaArg:Integer,ipivArg:Matrix Integer,_
       ldbArg:Integer,bArg:Matrix DoubleFloat): Result ==
       [(invokeNagman(NIL$Lisp,_
       "f07aef",_
       ["trans"::S,"n"::S,"nrhs"::S,"lda"::S,"ldb"::S_
       ,"info"::S,"a"::S,"ipiv"::S,"b"::S]$Lisp,_
       ["info"::S]$Lisp,_
       [["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp_
```

```
      ,["b"::S,"ldb"::S,"nrhs"::S]$Lisp]$Lisp_
      ,["integer"::S,"n"::S,"nrhs"::S,"lda"::S,["ipiv"::S,"n"::S]$Lisp_
      ,"ldb"::S,"info"::S]$Lisp_
      ,["character"::S,"trans"::S]$Lisp_
      ]$Lisp,_
      ["info"::S,"b"::S]$Lisp,_
      [([transArg::Any,nArg::Any,nrhsArg::Any,ldaArg::Any,_
      ldbArg::Any,aArg::Any,ipivArg::Any,bArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result

  f07fdf(uploArg:String,nArg:Integer,ldaArg:Integer,_
      aArg:Matrix DoubleFloat): Result ==
      [(invokeNagman(NIL$Lisp,_
      "f07fdf",_
      ["uplo"::S,"n"::S,"lda"::S,"info"::S,"a"::S]$Lisp,_
      ["info"::S]$Lisp,_
      [["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp_
      ]$Lisp_
      ,["integer"::S,"n"::S,"lda"::S,"info"::S]$Lisp_
      ,["character"::S,"uplo"::S]$Lisp_
      ]$Lisp,_
      ["info"::S,"a"::S]$Lisp,_
      [([uploArg::Any,nArg::Any,ldaArg::Any,aArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result

  f07fef(uploArg:String,nArg:Integer,nrhsArg:Integer,_
      aArg:Matrix DoubleFloat,ldaArg:Integer,ldbArg:Integer,_
      bArg:Matrix DoubleFloat): Result ==
      [(invokeNagman(NIL$Lisp,_
      "f07fef",_
      ["uplo"::S,"n"::S,"nrhs"::S,"lda"::S,"ldb"::S_
      ,"info"::S,"a"::S,"b"::S]$Lisp,_
      ["info"::S]$Lisp,_
      [["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp_
      ,["b"::S,"ldb"::S,"nrhs"::S]$Lisp]$Lisp_
      ,["integer"::S,"n"::S,"nrhs"::S,"lda"::S,"ldb"::S_
      ,"info"::S]$Lisp_
      ,["character"::S,"uplo"::S]$Lisp_
      ]$Lisp,_
      ["info"::S,"b"::S]$Lisp,_
      [([uploArg::Any,nArg::Any,nrhsArg::Any,ldaArg::Any,_
      ldbArg::Any,aArg::Any,bArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGF07.dotabb** —

```
"NAGF07" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF07"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGF07" -> "ALIST"
```

———-

15.8 package NAGF01 NagMatrixOperationsPackage

— **NagMatrixOperationsPackage.help** —

```
F01 -- Matrix Factorization                    Introduction -- F01
                          Chapter F01
                     Matrix Factorization

1. Scope of the Chapter

This chapter provides facilities for matrix factorizations and
associated transformations.

2. Background to the Problems

An n by n matrix may be factorized as

                                T
                           A=PLUQ ,

where L and U are respectively lower and uper triangular
matrices, and P and Q are permutation matrices. This is called an
LU factorization. For general dense matrices it is usual to
choose Q=I and to then choose P to ensure that the factorization
is numerically stable. For sparse matrices, judicious choice of P
and Q ensures numerical stability as well as maintaining as much
sparsity as possible in the factors L and U. The LU factorization
is normally used in connection with the solution of the linear
equations

                                Ax=b,

whose solution, x, may then be obtained by solving in succession
the simpler equations

                                T
```

$$Ly=P\,b,\quad Uz=y,\quad x=Qz$$

the first by forward substitution and the second by backward substitution. Routines to perform this solution are to be found in Chapter F04.

When A is symmetric positive-definite then we can choose $U=L^T$ and Q=P, to give the Cholesky factorization. This factorization is numerically stable without permutations, but in the sparse case the permutations can again be used to try to maintain sparsity. The Cholesky factorization is sometimes expressed as

$$A=PLDL^TP^T,$$

where D is a diagonal matrix with positive diagonal elements and L is unit lower triangular.

The LU factorization can also be performed on rectangular matrices, but in this case it is more usual to perform a QR factorization. When A is an m by n (m>=n) matrix this is given by

$$A=Q\begin{pmatrix}R\\0\end{pmatrix},$$

where R is an n by n upper triangular matrix and Q is an orthogonal (unitary in the complex case) matrix.

3. Recommendations on Choice and Use of Routines

Routine F07ADF performs the LU factorization of a real m by n dense matrix.

The LU factorization of a sparse matrix is performed by routine F01BRF. Following the use of F01BRF, matrices with the same sparsity pattern may be factorized by routine F01BSF.

The Cholesky factorization of a real symmetric positive-definite dense matrix is performed by routine F07FDF.

Routine F01MCF performs the Cholesky factorization of a real symmetric positive-definite variable band (skyline) matrix, and a general sparse symmetric positive-definite matrix may be factorized using routine F01MAF.

The QR factorization of an m by n (m>=n) matrix is performed by routine F01QCF in the real case, and F01RCF in the complex case. Following the use of F01QCF, operations with Q may be performed

```
using routine F01QDF and some, or all, of the columns of Q may be
formed using routine F01QEF. Routines F01RDF and F01REF perform
the same tasks following the use of F01RCF.


F01 -- Matrix Factorizations                        Contents -- F01
Chapter F01

Matrix Factorizations

F01BRF  LU factorization of real sparse matrix

F01BSF  LU factorization of real sparse matrix with known
        sparsity pattern

          T
F01MAF  LL  factorization of real sparse symmetric positive-
        definite matrix

           T
F01MCF  LDL  factorization of real symmetric positive-definite
        variable-bandwidth matrix

F01QCF  QR factorization of real m by n matrix (m>=n)

                                                              T
F01QDF  Operations with orthogonal matrices, compute QB or Q B
        after factorization by F01QCF

F01QEF  Operations with orthogonal matrices, form columns of Q
        after factorization by F01QCF

F01RCF  QR factorization of complex m by n matrix (m>=n)

                                                            H
F01RDF  Operations with unitary matrices, compute QB or Q B after
        factorization by F01RCF

F01REF  Operations with unitary matrices, form columns of Q after
        factorization by F01RCF
```

%%%

```
F01 -- Matrix Factorizations                              F01BRF
        F01BRF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.
```

1. Purpose

F01BRF factorizes a real sparse matrix. The routine either forms the LU factorization of a permutation of the entire matrix, or, optionally, first permutes the matrix to block lower triangular form and then only factorizes the diagonal blocks.

2. Specification

```
      SUBROUTINE F01BRF (N, NZ, A, LICN, IRN, LIRN, ICN, PIVOT,
     1                   IKEEP, IW, W, LBLOCK, GROW, ABORT,
     2                   IDISP, IFAIL)
      INTEGER           N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN),
     1                  IKEEP(5*N), IW(8*N), IDISP(10), IFAIL
      DOUBLE PRECISION  A(LICN), PIVOT, W(N)
      LOGICAL           LBLOCK, GROW, ABORT(4)
```

3. Description

Given a real sparse matrix A, this routine may be used to obtain the LU factorization of a permutation of A,

$$PAQ=LU$$

where P and Q are permutation matrices, L is unit lower triangular and U is upper triangular. The routine uses a sparse variant of Gaussian elimination, and the pivotal strategy is designed to compromise between maintaining sparsity and controlling loss of accuracy through round-off.

Optionally the routine first permutes the matrix into block lower triangular form and then only factorizes the diagonal blocks. For some matrices this gives a considerable saving in storage and execution time.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve
T
sparse systems of linear equations $Ax=b$ or $A^T x=b$. If several matrices of the same sparsity pattern are to be factorized, F01BSF should be used for the second and subsequent matrices.

The method is fully described by Duff [1].

4. References

[1] Duff I S (1977) MA28 -- a set of Fortran subroutines for

```
      sparse unsymmetric linear equations. A.E.R.E. Report R.8730.
      HMSO.

5. Parameters

 1:  N -- INTEGER                                              Input
     On entry: n, the order of the matrix A. Constraint: N > 0.

 2:  NZ -- INTEGER                                             Input
     On entry: the number of non-zero elements in the matrix A.
     Constraint: NZ > 0.

 3:  A(LICN) -- DOUBLE PRECISION array               Input/Output
     On entry: A(i), for i = 1,2,...,NZ must contain the non-
     zero elements of the sparse matrix A. They can be in any
     order since the routine will reorder them. On exit: the non-
     zero elements in the LU factorization. The array must not be
     changed by the user between a call of this routine and a
     call of F04AXF.

 4:  LICN -- INTEGER                                           Input
     On entry:
     the dimension of the arrays A and ICN as declared in the
     (sub)program from which F01BRF is called.
     Since the factorization is returned in A and ICN, LICN
     should be large enough to accommodate this and should
     ordinarily be 2 to 4 times as large as NZ. Constraint: LICN
     >= NZ.

 5:  IRN(LIRN) -- INTEGER array                      Input/Output
     On entry: IRN(i), for i = 1,2,...,NZ must contain the row
     index of the non-zero element stored in A(i). On exit: the
     array is overwritten and is not needed for subsequent calls
     of F01BSF or F04AXF.

 6:  LIRN -- INTEGER                                           Input
     On entry:
     the dimension of the array IRN as declared in the
     (sub)program from which F01BRF is called.
     It need not be as large as LICN; normally it will not need
     to be very much greater than NZ. Constraint: LIRN >= NZ.

 7:  ICN(LICN) -- INTEGER array                      Input/Output
     On entry: ICN(i), for i = 1,2,...,NZ must contain the
     column index of the non-zero element stored in A(i). On
     exit: the column indices of the non-zero elements in the
     factorization. The array must not be changed by the user
     between a call of this routine and subsequent calls of
     F01BSF or F04AXF.
```

```
 8:  PIVOT -- DOUBLE PRECISION                                  Input
     On entry: PIVOT should have a value in the range 0.0 <=
     PIVOT <= 0.9999 and is used to control the choice of pivots.
     If PIVOT < 0.0, the value 0.0 is assumed, and if PIVOT > 0.
     9999, the value 0.9999 is assumed. When searching a row for
     a pivot, any element is excluded which is less than PIVOT
     times the largest of those elements in the row available as
     pivots. Thus decreasing PIVOT biases the algorithm to
     maintaining sparsity at the expense of stability. Suggested
     value: PIVOT = 0.1 has been found to work well on test
     examples.

 9:  IKEEP(5*N) -- INTEGER array                                Output
     On exit: indexing information about the factorization. The
     array must not be changed by the user between a call of this
     routine and calls of F01BSF or F04AXF.

10:  IW(8*N) -- INTEGER array                                  Workspace

11:  W(N) -- DOUBLE PRECISION array                            Output
     On exit: if GROW = .TRUE., W(1) contains an estimate (an
     upper bound) of the increase in size of elements encountered
     during the factorization (see GROW); the rest of the array
     is used as workspace.

     If GROW = .FALSE., the array is not used.

12:  LBLOCK -- LOGICAL                                         Input
     On entry: if LBLOCK = .TRUE., the matrix is pre-ordered
     into block lower triangular form before the LU factorization
     is performed; otherwise the entire matrix is factorized.
     Suggested value: LBLOCK = .TRUE. unless the matrix is known
     to be irreducible.

13:  GROW -- LOGICAL                                           Input
     On entry: if GROW = .TRUE., then on exit W(1) contains an
     estimate (an upper bound) of the increase in size of
     elements encountered during the factorization. If the matrix
     is well-scaled (see Section 8.2), then a high value for W(1)
     indicates that the LU factorization may be inaccurate and
     the user should be wary of the results and perhaps increase
     the parameter PIVOT for subsequent runs (see Section 7).
     Suggested value: GROW = .TRUE..

14:  ABORT(4) -- LOGICAL array                                 Input
     On entry:
          if ABORT(1) = .TRUE., the routine will exit
          immediately on detecting a structural singularity (one
          that depends on the pattern of non-zeros) and return
```

```
          IFAIL = 1; otherwise it will complete the
          factorization (see Section 8.3).

          If ABORT(2) = .TRUE., the routine will exit
          immediately on detecting a numerical singularity (one
          that depends on the numerical values) and return IFAIL
          = 2; otherwise it will complete the factorization (see
          Section 8.3).

          If ABORT(3) = .TRUE., the routine will exit
          immediately (with IFAIL = 5) when the arrays A and ICN
          are filled up by the previously factorized, active and
          unfactorized parts of the matrix; otherwise it
          continues so that better guidance on necessary array
          sizes can be given in IDISP(6) and IDISP(7), and will
          exit with IFAIL in the range 4 to 6. Note that there
          is always an immediate error exit if the array IRN is
          too small.

          If ABORT(4) = .TRUE., the routine exits immediately
          (with IFAIL = 13) if it finds duplicate elements in
          the input matrix. If ABORT(4) = .FALSE., the routine
          proceeds using a value equal to the sum of the
          duplicate elements. In either case details of each
          duplicate element are output on the current advisory
          message unit (see X04ABF), unless suppressed by the
          value of IFAIL on entry.
      Suggested values:
          ABORT(1) = .TRUE.

          ABORT(2) = .TRUE.

          ABORT(3) = .FALSE.

          ABORT(4) = .TRUE..

15:  IDISP(10) -- INTEGER array                              Output
     On exit: IDISP is used to communicate information about the
     factorization to the user and also between a call of F01BRF
     and subsequent calls to F01BSF or F04AXF.
          IDISP(1) and IDISP(2), indicate the position in arrays
          A and ICN of the first and last elements in the LU
          factorization of the diagonal blocks. (IDISP(2) gives
          the number of non-zeros in the factorization.)

          IDISP(3) and IDISP(4), monitor the adequacy of 'elbow
          room' in the arrays IRN and A/ICN respectively, by
          giving the number of times that the data in these
          arrays has been compressed during the factorization to
          release more storage. If either IDISP(3) or IDISP(4)
```

```
            is quite large (say greater than 10), it will probably
            pay the user to increase the size of the corresponding
            array(s) for subsequent runs. If either is very low or
            zero, then the user can perhaps save storage by
            reducing the size of the corresponding array(s).

            IDISP(5), gives an upper bound on the rank of the
            matrix.

            IDISP(6) and IDISP(7), give the minimum size of arrays
            IRN and A/ICN respectively which would enable a
            successful run on an identical matrix (but some '
            elbow-room' should be allowed - see Section 8).

            IDISP(8) to (10), are only used if LBLOCK = .TRUE..

            IDISP(8), gives the structural rank of the matrix.

            IDISP(9), gives the number of diagonal blocks.

            IDISP(10), gives the size of the largest diagonal
            block.

            IDISP(1) and IDISP(2), must not be changed by the user
            between a call of F01BRF and subsequent calls to
            F01BSF or F04AXF.

16: IFAIL -- INTEGER                                  Input/Output
    For this routine, the normal use of IFAIL is extended to
    control the printing of error and warning messages as well
    as specifying hard or soft failure (see the Essential
    Introduction).

    Before entry, IFAIL must be set to a value with the decimal
    expansion cba, where each of the decimal digits c, b and a
    must have a value of 0 or 1.
    a=0   specifies hard failure, otherwise soft failure;

    b=0   suppresses error messages, otherwise error messages
          will be printed (see Section 6);

    c=0   suppresses warning messages, otherwise warning
          messages will be printed (see Section 6).
    The recommended value for inexperienced users is 110 (i.e.,
    hard failure with all messages printed).

    Unless the routine detects an error (see Section 6), IFAIL
    contains 0 on exit.

6. Error Indicators and Warnings
```

```
Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL=-2
     Successful factorization of a numerically singular matrix
     (which may also be structurally singular) (see Section 8.3).

IFAIL=-1
     Successful factorization of a structurally singular matrix
     (see Section 8.3).

IFAIL= 1
     The matrix is structurally singular and the factorization
     has been abandoned (ABORT(1) was .TRUE. on entry).

IFAIL= 2
     The matrix is numerically singular and the factorization has
     been abandoned (ABORT(2) was .TRUE. on entry).

IFAIL= 3
     LIRN is too small: there is not enough space in the array
     IRN to continue the factorization. The user is recommended
     to try again with LIRN (and the length of IRN) equal to at
     least IDISP(6) + N/2.

IFAIL= 4
     LICN is much too small: there is much too little space in
     the arrays A and ICN to continue the factorization.

IFAIL= 5
     LICN is too small: there is not enough space in the arrays A
     and ICN to store the factorization. If ABORT(3) was .FALSE.
     on entry, the factorization has been completed but some of
     the LU factors have been discarded to create space, IDISP(7)
     then gives the minimum value of LICN (i.e., the minimum
     length of A and ICN) required for a successful factorization
     of the same matrix.

IFAIL= 6
     LICN and LIRN are both too small: effectively this is a
     combination of IFAIL = 3 and IFAIL = 5 (with ABORT(3) = .
     FALSE.).

IFAIL= 7
     LICN is too small: there is not enough space in the arrays A
     and ICN for the permutation to block triangular form.
```

```
IFAIL= 8
     On entry N <= 0.

IFAIL= 9
     On entry NZ <= 0.

IFAIL= 10
     On entry LICN < NZ.

IFAIL= 11
     On entry LIRN < NZ.

IFAIL= 12
     On entry an element of the input matrix has a row or column
     index (i.e., an element of IRN or ICN) outside the range 1
     to N.

IFAIL= 13
     Duplicate elements have been found in the input matrix and
     the factorization has been abandoned (ABORT(4) = .TRUE. on
     entry).

7. Accuracy

The factorization obtained is exact for a perturbed matrix whose
(i,j)th element differs from a   by less than 3(epsilon)(rho)m
                               ij                             ij
where (epsilon) is the machine precision, (rho) is the growth
value returned in W(1) if GROW = .TRUE., and m   the number of
                                              ij
Gaussian elimination operations applied to element (i,j). The
value of m   is not greater than n and is usually much less.
         ij
Small (rho) values therefore guarantee accurate results, but
unfortunately large (rho) values may give a very pessimistic
indication of accuracy.

8. Further Comments

8.1. Timing

The time required may be estimated very roughly from the number
(tau) of non-zeros in the factorized form (output as IDISP(2))
and for this routine and its associates is

                  2
     F01BRF: 5(tau) /n units

                 2
```

F01BSF: (tau) /n units

F04AXF: 2(tau) units

where our unit is the time for the inner loop of a full matrix
code (e.g. solving a full set of equations takes about $\frac{1}{3}n^3$ units). Note that the faster F01BSF time makes it well worthwhile to use this for a sequence of problems with the same pattern.

It should be appreciated that (tau) varies widely from problem to problem. For network problems it may be little greater than NZ, the number of non-zeros in A; for discretisation of 2-dimensional and 3-dimensional partial differential equations it may be about $3n\log_2 n$ and $\frac{1}{2}n^{5/3}$, respectively.

The time taken to find the block lower triangular form (LBLOCK = it is not found (LBLOCK = .FALSE.). If the matrix is irreducible (IDISP(9) = 1 after a call with LBLOCK = .TRUE.) then this time is wasted. Otherwise, particularly if the largest block is small (IDISP(10)<<n), the consequent savings are likely to be greater.

The time taken to estimate growth (GROW = .TRUE.) is typically under 2% of the overall time.

The overall time may be substantially increased if there is inadequate 'elbow-room' in the arrays A, IRN and ICN. When the sizes of the arrays are minimal (IDISP(6) and IDISP(7)) it can execute as much as three times slower. Values of IDISP(3) and IDISP(4) greater than about 10 indicate that it may be worthwhile to increase array sizes.

8.2. Scaling

The use of a relative pivot tolerance PIVOT essentially presupposes that the matrix is well-scaled, i.e., that the matrix elements are broadly comparable in size. Practical problems are often naturally well-scaled but particular care is needed for problems containing mixed types of variables (for example millimetres and neutron fluxes).

8.3. Singular and Rectangular Systems

It is envisaged that this routine will almost always be called for square non-singular matrices and that singularity indicates an error condition. However, even if the matrix is singular it is possible to complete the factorization. It is even possible for

```
F04AXF to solve a set of equations whose matrix is singular
provided the set is consistent.

Two forms of singularity are possible. If the matrix would be
singular for any values of the non-zeros (e.g. if it has a whole
row of zeros), then we say it is structurally singular, and
continue only if ABORT(1) = .FALSE.. If the matrix is non-
singular by virtue of the particular values of the non-zeros,
then we say that it is numerically singular and continue only if
ABORT(2) = .FALSE..

Rectangular matrices may be treated by setting N to the larger of
the number of rows and numbers of columns and setting ABORT(1) =

Note: the soft failure option should be used (last digit of IFAIL
= 1) if the user wishes to factorize singular matrices with ABORT
(1) or ABORT(2) set to .FALSE..

8.4. Duplicated Non-zeros

The matrix A may consist of a sum of contributions from different
sub-systems (for example finite elements). In such cases the user
may rely on this routine to perform assembly, since duplicated
elements are summed.

9. Example

To factorize the real sparse matrix:

                      ( 5  0  0  0 0  0)
                      ( 0  2 -1  2 0  0)
                      ( 0  0  3  0 0  0)
                      (-2  0  0  1 1  0).
                      (-1  0  0 -1 2 -3)
                      (-1 -1  0  0 0  6)

This example program simply prints out some information about the
factorization as returned by F01BRF in W(1) and IDISP. Normally
the call of F01BRF would be followed by a call of F04AXF (see
Example for F04AXF).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations                                  F01BSF
       F01BSF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01BSF factorizes a real sparse matrix using the pivotal sequence previously obtained by F01BRF when a matrix of the same sparsity pattern was factorized.

2. Specification

```
      SUBROUTINE F01BSF (N, NZ, A, LICN, IVECT, JVECT, ICN,
     1                   IKEEP, IW, W, GROW, ETA, RPMIN, ABORT,
     2                   IDISP, IFAIL)
      INTEGER           N, NZ, LICN, IVECT(NZ), JVECT(NZ), ICN
     1                  (LICN), IKEEP(5*N), IW(8*N), IDISP(2),
     2                  IFAIL
      DOUBLE PRECISION A(LICN), W(N), ETA, RPMIN
      LOGICAL           GROW, ABORT
```

3. Description

This routine accepts as input a real sparse matrix of the same sparsity pattern as a matrix previously factorized by a call of F01BRF. It first applies to the matrix the same permutations as were used by F01BRF, both for permutation to block triangular form and for pivoting, and then performs Gaussian elimination to obtain the LU factorization of the diagonal blocks.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve
$$Ax=b \quad \text{or} \quad A^{T}x=b$$
sparse systems of linear equations.

F01BSF is much faster than F01BRF and in some applications it is expected that there will be many calls of F01BSF for each call of F01BRF.

The method is fully described in Duff [1].

4. References

[1] Duff I S (1977) MA28 -- a set of Fortran subroutines for sparse unsymmetric linear equations. A.E.R.E. Report R.8730. HMSO.

```
5. Parameters

 1:  N -- INTEGER                                              Input
     On entry: n, the order of the matrix A. Constraint: N > 0.

 2:  NZ -- INTEGER                                             Input
     On entry: the number of non-zeros in the matrix A.
     Constraint: NZ > 0.

 3:  A(LICN) -- DOUBLE PRECISION array                  Input/Output
     On entry: A(i), for i = 1,2,...,NZ must contain the non-
     zero elements of the sparse matrix A. They can be in any
     order since the routine will reorder them. On exit: the non-
     zero elements in the factorization. The array must not be
     changed by the user between a call of this routine and a
     call of F04AXF.

 4:  LICN -- INTEGER                                           Input
     On entry:
     the dimension of the arrays A and ICN as declared in the
     (sub)program from which F01BSF is called.
     It should have the same value as it had for F01BRF.
     Constraint: LICN >= NZ.

 5:  IVECT(NZ) -- INTEGER array                                Input

 6:  JVECT(NZ) -- INTEGER array                                Input
     On entry: IVECT(i) and JVECT(i), for i = 1,2,...,NZ must
     contain the row index and the column index respectively of
     the non-zero element stored in A(i).

 7:  ICN(LICN) -- INTEGER array                                Input
     On entry: the same information as output by F01BRF. It must
     not be changed by the user between a call of this routine
     and a call of F04AXF.

 8:  IKEEP(5*N) -- INTEGER array                               Input
     On entry: the same indexing information about the
     factorization as output from F01BRF. It must not be changed
     between a call of this routine and a call of F04AXF.

 9:  IW(8*N) -- INTEGER array                               Workspace

10:  W(N) -- DOUBLE PRECISION array                           Output
     On exit: if GROW = .TRUE., W(1) contains an estimate (an
     upper bound) of the increase in size of elements encountered
     during the factorization (see GROW); the rest of the array
     is used as workspace.

     If GROW = .FALSE., the array is not used.
```

```
11:  GROW -- LOGICAL                                          Input
     On entry: if GROW = .TRUE., then on exit W(1) contains an
     estimate (an upper bound) of the increase in size of
     elements encountered during the factorization. If the matrix
     is well-scaled (see Section 8.2), then a high value for W(1)
     indicates that the LU factorization may be inaccurate and
     the user should be wary of the results and perhaps increase
     the parameter PIVOT for subsequent runs (see Section 7).

12:  ETA -- DOUBLE PRECISION                                  Input
     On entry: the relative pivot threshold below which an error
     diagnostic is provoked and IFAIL is set to 7. If ETA is
     greater than 1.0, then no check on pivot size is made.
                                -4
     Suggested value: ETA = 10  .

13:  RPMIN -- DOUBLE PRECISION                               Output
     On exit: if ETA is less than 1.0, then RPMIN gives the
     smallest ratio of the pivot to the largest element in the
     row of the corresponding upper triangular factor thus
     monitoring the stability of the factorization. If RPMIN is
     very small it may be advisable to perform a new
     factorization using F01BRF.

14:  ABORT -- LOGICAL                                         Input
     On entry: if ABORT = .TRUE., the routine exits immediately
     (with IFAIL = 8) if it finds duplicate elements in the input
     matrix. If ABORT = .FALSE., the routine proceeds using a
     value equal to the sum of the duplicate elements. In either
     case details of each duplicate element are output on the
     current advisory message unit (see X04ABF), unless
     suppressed by the value of IFAIL on entry. Suggested value:
     ABORT = .TRUE..

15:  IDISP(2) -- INTEGER array                                Input
     On entry: IDISP(1) and IDISP(2) must be unchanged since the
     previous call of F01BRF.

16:  IFAIL -- INTEGER                                  Input/Output
     For this routine, the normal use of IFAIL is extended to
     control the printing of error and warning messages as well
     as specifying hard or soft failure (see the Essential
     Introduction).

     Before entry, IFAIL must be set to a value with the decimal
     expansion cba, where each of the decimal digits c, b and a
     must have a value of 0 or 1.
     a=0   specifies hard failure, otherwise soft failure;
```

```
     b=0   suppresses error messages, otherwise error messages
           will be printed (see Section 6);

     c=0   suppresses warning messages, otherwise warning
           messages will be printed (see Section 6).
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
     On entry N <= 0.

IFAIL= 2
     On entry NZ <= 0.

IFAIL= 3
     On entry LICN < NZ.

IFAIL= 4
     On entry an element of the input matrix has a row or column
     index (i.e., an element of IVECT or JVECT) outside the range
     1 to N.

IFAIL= 5
     The input matrix is incompatible with the matrix factorized
     by the previous call of F01BRF (see Section 8).

IFAIL= 6
     The input matrix is numerically singular.

IFAIL= 7
     A very small pivot has been detected (see Section 5, ETA).
     The factorization has been completed but is potentially
     unstable.

IFAIL= 8
     Duplicate elements have been found in the input matrix and
     the factorization has been abandoned (ABORT = .TRUE. on
     entry).
```

```
7. Accuracy

The factorization obtained is exact for a perturbed matrix whose
(i,j)th element differs from a    by less than 3(epsilon)(rho)m
                              ij                                ij
where (epsilon) is the machine precision, (rho) is the growth
value returned in W(1) if GROW = .TRUE., and m    the number of
                                              ij
Gaussian elimination operations applied to element (i,j).

If (rho) = W(1) is very large or RPMIN is very small, then a
fresh call of F01BRF is recommended.

8. Further Comments

If the user has a sequence of problems with the same sparsity
pattern then this routine is recommended after F01BRF has been
called for one such problem. It is typically 4 to 7 times faster
but is potentially unstable since the previous pivotal sequence
is used. Further details on timing are given in document F01BRF.

If growth estimation is performed (GROW = .TRUE.), then the time
increases by between 5% and 10%. Pivot size monitoring (ETA <= 1.
0) involves a similar overhead.

We normally expect this routine to be entered with a matrix
having the same pattern of non-zeros as was earlier presented to
F01BRF. However there is no record of this pattern, but rather a
record of the pattern including all fill-ins. Therefore we permit
additional non-zeros in positions corresponding to fill-ins.

If singular matrices are being treated then it is also required
that the present matrix be sufficiently like the previous one for
the same permutations to be suitable for factorization with the
same set of zero pivots.

9. Example

To factorize the real sparse matrices

                      ( 5  0  0  0 0  0)
                      ( 0  2 -1  2 0  0)
                      ( 0  0  3  0 0  0)
                      (-2  0  0  1 1  0)
                      (-1  0  0 -1 2 -3)
                      (-1 -1  0  0 0  6)

and

                     (10  0  0  0  0  0)
```

```
                        ( 0 12 -3 -1  0  0)
                        ( 0  0 15  0  0  0)
                        (-2  0  0 10 -1  0).
                        (-1  0  0 -5  1 -1)
                        (-1 -2  0  0  0  6)
```

This example program simply prints the values of W(1) and RPMIN returned by F01BSF. Normally the calls of F01BRF and F01BSF would be followed by calls of F04AXF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F01 -- Matrix Factorizations F01MAF
F01MAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01MAF computes an incomplete Cholesky factorization of a real sparse symmetric positive-definite matrix A.

2. Specification

```
      SUBROUTINE F01MAF (N, NZ, A, LICN, IRN, LIRN, ICN, DROPTL,
     1                   DENSW, WKEEP, IKEEP, IWORK, ABORT,
     2                   INFORM, IFAIL)
      INTEGER          N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN),
     1                 IKEEP(2*N), IWORK(6*N), INFORM(4), IFAIL
      DOUBLE PRECISION A(LICN), DROPTL, DENSW, WKEEP(3*N)
      LOGICAL          ABORT(3)
```

3. Description

F01MAF computes an incomplete Cholesky factorization

$$C=PLDL^{T}P^{T}, \quad WAW=C+E$$

for the sparse symmetric positive-definite matrix A, where P is a permutation matrix, L is a unit lower triangular matrix, D is a diagonal matrix with positive diagonal elements, E is an error matrix representing elements dropped during the factorization and

diagonal elements that have been modified to ensure that C is positive-definite, and W is a diagonal matrix, chosen to make the diagonal elements of WAW unity.

$W^{-1}CW^{-1}$ is a pre-conditioning matrix for A, and the factorization of C is intended to be used by F04MAF to solve systems of linear equations Ax=b.

The permutation matrix P is chosen to reduce the amount of fill-in that occurs in L and the user-supplied parameter DROPTL can also be used to control the amount of fill-in that occurs.

Full details on the factorization can be found in Munksgaard [1].

F01MAF is based on the Harwell Library routine MA31A.

4. References

[1] Munksgaard N (1980) Solving Sparse Symmetric Sets of Linear Equations by Pre-conditioned Conjugate Gradients. ACM Trans. Math. Softw. 6 206--219.

5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix A. Constraint: N >= 1.

2: NZ -- INTEGER Input
On entry: the number of non-zero elements in the upper triangular part of the matrix A, including the number of elements on the leading diagonal. Constraint: NZ >= N.

3: A(LICN) -- DOUBLE PRECISION array Input/Output
On entry: the first NZ elements of the array A must contain the non-zero elements of the upper triangular part of the sparse positive-definite symmetric matrix A, including the elements on the leading diagonal. On exit: the first (NZ-N) elements of A contain the elements above the diagonal of the matrix WAW, where W is a diagonal matrix whose ith diagonal element is $w_i = a_{ii}^{-1/2}$. These elements are returned in order by rows and the value returned in ICN(k) gives the column index of the element returned in A(k). The value w_i is returned in the ith element of the array WKEEP. The remaining LROW-NZ+N elements of A, where LROW is the value returned in INFORM(1), return details of the factorization for use by F04MAF.

4: LICN -- INTEGER Input
On entry:
the dimension of the array A as declared in the (sub)program from which F01MAF is called.
If fill-in is expected during the factorization, then a larger value of LICN will allow fewer elements to be dropped during the factorization, thus giving a more accurate factorization, which in turn will almost certainly mean that fewer iterations will be required by F04MAF. Constraint: LICN>=2*NZ.

5: IRN(LIRN) -- INTEGER array Input/Output
On entry: IRN(k), for k = 1,2,...,NZ must contain the row index of the non-zero element of the matrix A supplied in A (k). On exit: the first LCOL elements of IRN, where LCOL is the value returned in INFORM(2), return details of the factorization for use by F04MAF.

6: LIRN -- INTEGER Input
On entry:
the dimension of the array IRN as declared in the (sub)program from which F01MAF is called.
LIRN must be at least NZ, but, as with LICN, if fill-in is expected then a larger value of LIRN will allow a more accurate factorization. For this purpose LIRN should exceed NZ by the same amount that LICN exceeds 2*NZ. Constraint: LIRN >= NZ.

7: ICN(LICN) -- INTEGER array Input/Output
On entry: ICN(k), for k = 1,2,...,NZ must contain the column index of the non-zero element of the matrix A supplied in A (k). Thus a_{ij} =A(k), where i = IRN(k) and j = ICN(k). On exit: the first (NZ-N) elements of ICN give the column indices of the first (NZ-N) elements returned in A. The remaining LROW - NZ + N elements of ICN return details of the factorization for use by F04MAF.

8: DROPTL -- DOUBLE PRECISION Input/Output
On entry: a value in the range [-1.0,1.0] to be used as a tolerance in deciding whether or not to drop elements during the factorization. At the kth pivot step the element $a_{ij}^{(k+1)}$ is dropped if it would cause fill-in and if

$$|a_{ij}^{(k+1)}| < |\mathrm{DROPTL}| * \sqrt{a_{ii}^{(k)} a_{jj}^{(k)}}.$$

```
     If DROPTL is supplied as negative, then it is not altered
     during the factorization and so is unchanged on exit, but if
     DROPTL is supplied as positive then it may be altered by the
     routine with the aim of obtaining an accurate factorization
     in the space available. If DROPTL is supplied as -1.0, then
     no fill-in will occur during the factorization; and if
     DROPTL is supplied as 0.0 then a complete factorization is
     performed. On exit: may be overwritten with the value used
     by the routine in order to obtain an accurate factorization
     in the space available, if DROPTL > 0.0 on entry.

 9:  DENSW -- DOUBLE PRECISION                          Input/Output
     On entry: a value in the range [0.0,1.0] to be used in
     deciding whether or not to regard the active part of the
     matrix at the kth pivot step as being full. If the ratio of
     non-zero elements to the total number of elements is greater
     than or equal to DENSW, then the active part is regarded as
     full. If DENSW < 1.0, then the storage used is likely to
     increase compared to the case where DENSW = 0, but the
     execution time is likely to decrease. Suggested value: DENSW
     = 0.8. On exit: if on entry DENSW is not in the range
     [0.0,1.0], then it is set to 0.8. Otherwise it is unchanged.

10:  WKEEP(3*N) -- DOUBLE PRECISION array                   Output
     On exit: information which must be passed unchanged to
     F04MAF. The first N elements contain the values w , for
                                                      i
     i=1,2,...,n, and the next N elements contain the diagonal
     elements of D.

11:  IKEEP(2*N) -- INTEGER array                            Output
     On exit: information which must be passed unchanged to
     F04MAF.

12:  IWORK(6*N) -- INTEGER array                          Workspace

13:  ABORT(3) -- LOGICAL array                              Input
     On entry:
          if ABORT(1) = .TRUE., the routine will exit
          immediately on detecting duplicate elements and return
          IFAIL = 5. Otherwise when ABORT(1) = .FALSE., the
          calculations will continue using the sum of the
          duplicate entries. In either case details of the
          duplicate elements are output on the current advisory
          message unit (see X04ABF), unless suppressed by the
          value of IFAIL on entry.

          If ABORT(2) = .TRUE., the routine will exit
          immediately on detecting a zero or negative pivot
          element and return IFAIL = 6. Otherwise when ABORT(2)
```

```
          = .FALSE., the zero or negative pivot element will be
          modified to ensure positive-definiteness and a message
          will be printed on the current advisory message unit,
          unless suppressed by the value of IFAIL on entry.

          If ABORT(3) = .TRUE., the routine will exit
          immediately if the arrays A and ICN have been filled
          up and return IFAIL = 7. Otherwise when ABORT(3) = .
          FALSE., the data in the arrays is compressed to
          release more storage and a message will be printed on
          the current advisory message unit, unless suppressed
          by the value of IFAIL on entry. If DROPTL is positive
          on entry, it may be modified in order to allow a
          factorization to be completed in the available space.
     Suggested values:
          ABORT(1) = .TRUE.,

          ABORT(2) = .TRUE.,

          ABORT(3) = .TRUE..

14:  INFORM(4) -- INTEGER array                          Output
     On exit:
          INFORM(1) returns the number of elements of A and ICN
          that have been used by the routine. Thus at least the
          first INFORM(1) elements of A and of ICN must be
          supplied to F04MAF.

          Similarly, INFORM(2) returns the number of elements of
          IRN that have been used by the routine and so at least
          the first INFORM(2) elements must be supplied to
          F04MAF.

          INFORM(3) returns the number of entries supplied in A
          that corresponded to diagonal and duplicate elements.
          If no duplicate entries were found, then INFORM(3)
          will return the value of N.

          INFORM(4) returns the value k of the pivot step from
          which the active matrix was regarded as full.
     INFORM must be passed unchanged to F04MAF.

15:  IFAIL -- INTEGER                                Input/Output
     For this routine, the normal use of IFAIL is extended to
     control the printing of error and warning messages as well
     as specifying hard or soft failure (see the Essential
     Introduction).

     Before entry, IFAIL must be set to a value with the decimal
     expansion cba, where each of the decimal digits c, b and a
```

```
     must have a value of 0 or 1.
     a=0   specifies hard failure, otherwise soft failure;

     b=0   suppresses error messages, otherwise error messages
           will be printed (see Section 6);

     c=0   suppresses warning messages, otherwise warning
           messages will be printed (see Section 6).
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
     On entry N < 1,

     or       NZ < N,

     or       LIRN < NZ,

     or       LICN<2*NZ.

IFAIL= 2
     One of the conditions 0 < IRN(k) <= ICN(k) <= N is not
     satisfied so that A(k) is not in the upper triangle of the
     matrix. No further computation is attempted.

IFAIL= 3
     One of the diagonal elements of the matrix A is zero or
     negative so that A is not positive-definite. No further
     computation is attempted.

IFAIL= 4
     The available space has been used and no further
     compressions are possible. The user should either increase
     DROPTL, or allocate more space to A, IRN and ICN.

     For all the remaining values of IFAIL the computations will
     continue in the case of soft failure, so that more than one
     advisory message may be printed.
```

```
IFAIL= 5
     Duplicate elements have been detected and ABORT(1) = .TRUE..

IFAIL= 6
     A zero or negative pivot element has been detected during
     the factorization and ABORT(2) = .TRUE..

     This should not happen if A is an M-matrix (see Munksgaard
     [1]), but may occur for other types of positive-definite
     matrix.

IFAIL= 7
     The available space has been used and ABORT(3) = .TRUE..

7. Accuracy

The accuracy of the factorization will be determined by the size
of the elements that are dropped and the size of the
modifications made to the diagonal elements. If these sizes are
small then the computed factors will correspond to a matrix close
to A and the number of iterations required by F04MAF will be
small. The more incomplete the factorization, the higher the
number of iterations required by F04MAF.

8. Further Comments

The time taken by the routine will depend upon the sparsity
pattern of the matrix and the number of fill-ins that occur
during the factorization. At the very least the time taken can be
expected to be roughly proportional to n(tau), where (tau) is the
number of non-zeros.
The routine is intended for use with positive-definite matrices,
but the user is warned that it will not necessarily detect non-
positive-definiteness. Indeed the routine may return a
factorization that can satisfactorily be used by F04MAF even when
A is not positive-definite, but this should not be relied upon as
F04MAF may not converge.

9. Example

The example program illustrates the use of F01MAF in conjunction
with F04MAF to solve the 16 linear equations Ax=b, where

              (1 z     z                    )
              (z 1 z     z                  )
              (  z 1 z     z                )
              (    z 1 0     z              )
              (z     0 1 z     z            )
              (  z     z 1 z     z          )
              (    z     z 1 z     z        )
```

$$
A=\begin{pmatrix}
z & z1 & 0 & z & & \\
z & 0 & 1z & z & & \\
 & z & z1z & z & & \\
 & z & z1z & z & & \\
 & & z & z1z & z & \\
 & & z & 01z & & \\
 & & & z & z1z & \\
 & & & z & z1z & \\
 & & & & z & z1z \\
 & & & & z & z1
\end{pmatrix}.
$$

$$
b=\left(\begin{array}{cccccccccccccc}
T & 1 & 1 & 1 & 1 & 1 & & 1 & 1 & & 1 & 1 & 1 & 1 & 1 \\
 & - & - & - & - & - & 0 & - & - & 0 & - & - & - & - & - \\
 & 2 & 4 & 4 & 2 & 4 & & 4 & 4 & & 4 & 2 & 4 & 4 & 2
\end{array}\right),
$$

$$
\text{where } z=-\frac{1}{4}.
$$

The n by n matrix A arises in the solution of Laplace's equation in a unit square, using a 5-point formula with a 6 by 6 discretisation, with unity on the boundaries.

The drop tolerance, DROPTL, is taken as 0.1, and the density factor, DENSW, is taken as 0.8. The value IFAIL = 111 is used so that advisory and error messages will be printed, but soft failure would occur if IFAIL were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F01 -- Matrix Factorizations F01MCF
F01MCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01MCF computes the Cholesky factorization of a real symmetric

```
positive-definite variable-bandwidth matrix.

2. Specification

       SUBROUTINE F01MCF (N, A, LAL, NROW, AL, D, IFAIL)
       INTEGER          N, LAL, NROW(N), IFAIL
       DOUBLE PRECISION A(LAL), AL(LAL), D(N)

3. Description

This routine determines the unit lower triangular matrix L and
                                                          T
the diagonal matrix D in the Cholesky factorization A=LDL  of a
symmetric positive-definite variable-bandwidth matrix A of order
n. (Such a matrix is sometimes called a 'sky-line' matrix.)

The matrix A is represented by the elements lying within the
envelope of its lower triangular part, that is, between the first
non-zero of each row and the diagonal (see Section 9 for an
example). The width NROW(i) of the ith row is the number of
elements between the first non-zero element and the element on
the diagonal, inclusive. Although, of course, any matrix
possesses an envelope as defined, this routine is primarily
intended for the factorization of symmetric positive-definite
matrices with an average bandwidth which is small compared with n
(also see Section 8).

The method is based on the property that during Cholesky
factorization there is no fill-in outside the envelope.

The determination of L and D is normally the first of two steps
in the solution of the system of equations Ax=b. The remaining
                               T
step, viz. the solution of LDL x=b may be carried out using
F04MCF.

4. References

[1]   Jennings A (1966) A Compact Storage Scheme for the Solution
      of Symmetric Linear Simultaneous Equations. Comput. J. 9
      281--285.

[2]   Wilkinson J H and Reinsch C (1971) Handbook for Automatic
      Computation II, Linear Algebra. Springer-Verlag.

5. Parameters

 1:  N -- INTEGER                                          Input
     On entry: n, the order of the matrix A. Constraint: N >= 1.
```

2: A(LAL) -- DOUBLE PRECISION array Input
On entry: the elements within the envelope of the lower triangle of the positive-definite symmetric matrix A, taken in row by row order. The following code assigns the matrix elements within the envelope to the correct elements of the array:

```
K = 0
DO 20 I = 1, N
DO 10 J = I-NROW(I)+1, I
K = K + 1
A(K) = matrix (I,J)
10    CONTINUE
20 CONTINUE
```

See also Section 8.

3: LAL -- INTEGER Input
On entry: the smaller of the dimensions of the arrays A and AL as declared in the calling (sub)program from which F01MCF is called. Constraint: LAL >= NROW(1) + NROW(2) +... + NROW(n).

4: NROW(N) -- INTEGER array Input
On entry: NROW(i) must contain the width of row i of the matrix A, i.e., the number of elements between the first (leftmost) non-zero element and the element on the diagonal, inclusive. Constraint: 1 <= NROW(i) <= i, for i=1,2,...,n.

5: AL(LAL) -- DOUBLE PRECISION array Output
On exit: the elements within the envelope of the lower triangular matrix L, taken in row by row order. The envelope of L is identical to that of the lower triangle of A. The unit diagonal elements of L are stored explicitly. See also Section 8.

6: D(N) -- DOUBLE PRECISION array Output
On exit: the diagonal elements of the the diagonal matrix D . Note that the determinant of A is equal to the product of these diagonal elements. If the value of the determinant is required it should not be determined by forming the product explicitly, because of the possibility of overflow or underflow. The logarithm of the determinant may safely be formed from the sum of the logarithms of the diagonal elements.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

```
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry N < 1,

     or       for some i, NROW(i) < 1 or NROW(i) > i,

     or       LAL < NROW(1) + NROW(2) +... + NROW(N).

IFAIL= 2
     A is not positive-definite, or this property has been
     destroyed by rounding errors. The factorization has not been
     completed.

IFAIL= 3
     A is not positive-definite, or this property has been
     destroyed by rounding errors. The factorization has been
     completed but may be very inaccurate (see Section 7).

7. Accuracy

If IFAIL = 0 on exit, then the computed L and D satisfy the
            T
relation LDL =A+F, where

                                2
                  ||F|| <=km (epsilon)max a
                       2                 i ii

and

                            2
                  ||F|| <=km (epsilon)||A|| ,
                       2                   2

where k is a constant of order unity, m is the largest value of
NROW(i), and (epsilon) is the machine precision. See Wilkinson
and Reinsch [2], pp 25--27, 54--55. If IFAIL = 3 on exit, then
the factorization has been completed although the matrix was not
positive-definite. However the factorization may be very
inaccurate and should be used only with great caution. For
instance, if it is used to solve a set of equations Ax=b using
F04MCF, the residual vector b-Ax should be checked.
```

```
8. Further Comments

The time taken by the routine is approximately proportional to
the sum of squares of the values of NROW(i).

The distribution of row widths may be very non-uniform without
undue loss of efficiency. Moreover, the routine has been designed
to be as competitive as possible in speed with routines designed
for full or uniformly banded matrices, when applied to such
matrices.

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters A and AL, in which case L
overwrites the lower triangle of A. However this is not standard
Fortran 77 and may not work in all implementations.

9. Example

To obtain the Cholesky factorization of the symmetric matrix,
whose lower triangle is:

                        (1                )
                        (2  5             )
                        (0  3 13          )
                        (0  0  0 16       ).
                        (5 14 18  8 55    )
                        (0  0  0 24 17 77)

For this matrix, the elements of NROW must be set to 1, 2, 2, 1,
5, 3, and the elements within the envelope must be supplied in
row order as:

1, 2, 5, 3, 13, 16, 5, 14, 18, 8, 55, 24, 17, 77.


The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations                                    F01QCF
       F01QCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.
```

1. Purpose

F01QCF finds the QR factorization of the real m by n matrix A,
where m>=n.

2. Specification

```
      SUBROUTINE F01QCF (M, N, A, LDA, ZETA, IFAIL)
      INTEGER          M, N, LDA, IFAIL
      DOUBLE PRECISION A(LDA,*), ZETA(*)
```

3. Description

The m by n matrix A is factorized as

$$A = \begin{pmatrix} R \\ 0 \end{pmatrix} Q \quad \text{when } m>n,$$

$$A = QR \quad \text{when } m=n,$$

where Q is an m by m orthogonal matrix and R is an n by n upper
triangular matrix. The factorization is obtained by Householder's
method. The kth transformation matrix, Q_k, which is used to
introduce zeros into the kth column of A is given in the form

$$Q_k = \begin{pmatrix} I & 0 \\ 0 & T_k \end{pmatrix}$$

where

$$T_k = I - u_k u_k^T,$$

$$u_k = \begin{pmatrix} (\zeta)_k \\ z_k \end{pmatrix},$$

$(\zeta)_k$ is a scalar and z_k is an (m-k) element vector. $(\zeta)_k$
and z_k are chosen to annihilate the elements below the triangular
part of A.

The vector u_k is returned in the kth element of the array ZETA
and in the kth column of A, such that $(\zeta)_k$ is in ZETA(k) and

```
                                                    k
the elements of z  are in A(k+1,k),...,A(m,k). The elements of R
                k
are returned in the upper triangular part of A.

Q is given by

                                      T
                    Q=(Q Q   ...Q ) .
                        n n-1    1

Good background descriptions to the QR factorization are given in
Dongarra et al [1] and Golub and Van Loan [2], but note that this
routine is not based upon LINPACK routine DQRDC.

4. References

[1]   Dongarra J J, Moler C B, Bunch J R and Stewart G W (1979)
      LINPACK Users' Guide. SIAM, Philadelphia.

[2]   Golub G H and Van Loan C F (1989) Matrix Computations (2nd
      Edition). Johns Hopkins University Press, Baltimore,
      Maryland.

[3]   Wilkinson J H (1965) The Algebraic Eigenvalue Problem.
      Oxford University Press.

5. Parameters

 1:  M -- INTEGER                                              Input
     On entry: m, the number of rows of A. Constraint: M >= N.

 2:  N -- INTEGER                                              Input
     On entry: n, the number of columns of A.

     When N = 0 then an immediate return is effected.
     Constraint: N >= 0.

 3:  A(LDA,*) -- DOUBLE PRECISION array              Input/Output
     Note: the second dimension of the array A must be at least
     max(1,n).
     On entry: the leading m by n part of the array A must
     contain the matrix to be factorized. On exit: the n by n
     upper triangular part of A will contain the upper triangular
     matrix R and the m by n strictly lower triangular part of A
     will contain details of the factorization as described in
     Section 3.

 4:  LDA -- INTEGER                                            Input
     On entry:
```

```
     the first dimension of the array A as declared in the
     (sub)program from which F01QCF is called.
     Constraint: LDA >= max(1,M).

 5:  ZETA(*) -- DOUBLE PRECISION array                      Output
     Note: the dimension of the array ZETA must be at least max
     (1,n) On exit: ZETA(k) contains the scalar (zeta)  for the k
                                                        k
     th transformation. If T =I then ZETA(k)=0.0, otherwise ZETA(
                            k
     k) contains (zeta)  as described in Section 3 and (zeta)  is
                     k                                      k

     always in the range (1.0, \/2.0).

 6:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     On entry M < N,

     or       N < 0,

     or       LDA < M.

7. Accuracy

The computed factors Q and R satisfy the relation

                              (R)
                            Q(0)=A+E,

where

                    |||E|||<=c(epsilon)|||A|||,

and (epsilon) is the machine precision (see X02AJF(*)), c is a
modest function of m and n and |||.||| denotes the spectral (two)
```

norm.

8. Further Comments

The approximate number of floating-point operations is given by
  2
2n (3m-n)/3.

Following the use of this routine the operations

$$B:=QB \quad \text{and} \quad B:=Q^{T}B,$$

where B is an m by k matrix, can be performed by calls to F01QDF. The operation B:=QB can be obtained by the call:

```
      IFAIL = 0
      CALL F01QDF('No transpose', 'Separate', M, N, A, LDA, ZETA,
     *            K, B, LDB, WORK, IFAIL)
```

  T
and B:=Q B can be obtained by the call:

```
      IFAIL = 0
      CALL F01QDF('Transpose', 'Separate', M, N, A, LDA, ZETA,
     *            K, B, LDB, WORK, IFAIL)
```

In both cases WORK must be a k element array that is used as workspace. If B is a one-dimensional array (single column) then the parameter LDB can be replaced by M. See F01QDF for further details.

The first k columns of the orthogonal matrix Q can either be obtained by setting B to the first k columns of the unit matrix and using the first of the above two calls, or by calling F01QEF, which overwrites the k columns of Q on the first k columns of the array A. Q is obtained by the call:

```
     CALL F01QEF('Separate', M, N, K, A, LDA, ZETA, WORK, IFAIL)
```

As above WORK must be a k element array. If k is larger than N, then A must have been declared to have at least k columns.

Operations involving the matrix R can readily be performed by the Level 2 BLAS routines DTRSV and DTRMV (see Chapter F06), but note that no test for near singularity of R is incorporated in DTRSV. If R is singular, or nearly singular then F02WUF(*) can be used

```
to determine the singular value decomposition of R.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

                          (2.0  2.5  2.5)
                          (2.0  2.5  2.5)
                        A=(1.6 -0.4  2.8).
                          (2.0 -0.5  0.5)
                          (1.2 -0.3 -2.9)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F01 -- Matrix Factorizations                                F01QDF
        F01QDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F01QDF performs one of the transformations

                                     T
                          B:=QB or B:=Q B,

where B is an m by ncolb real matrix and Q is an m by m
orthogonal matrix, given as the product of Householder
transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. Specification

       SUBROUTINE F01QDF (TRANS, WHERET, M, N, A, LDA, ZETA,
      1                   NCOLB, B, LDB, WORK, IFAIL)
       INTEGER          M, N, LDA, NCOLB, LDB, IFAIL
       DOUBLE PRECISION A(LDA,*), ZETA(*), B(LDB,*), WORK(*)
       CHARACTER*1      TRANS, WHERET

3. Description

Q is assumed to be given by
```

$$Q=(Q_n Q_{n-1} \ldots Q_1)^T,$$

Q_k being given in the form

$$Q_k = \begin{pmatrix} I & 0 \\ 0 & T \end{pmatrix}$$

where

$$T_k = I - u_k u_k^T,$$

$$u_k = \begin{pmatrix} (\text{zeta})_k \\ z_k \end{pmatrix},$$

$(\text{zeta})_k$ is a scalar and z_k is an (m-k) element vector. z_k must be supplied in the kth column of A in elements A(k+1,k),...,A(m,k) and $(\text{zeta})_k$ must be supplied either in A(k,k) or in ZETA(k), depending upon the parameter WHERET.

To obtain Q explicitly B may be set to I and pre-multiplied by Q^T. This is more efficient than obtaining Q .

4. References

[1] Golub G H and Van Loan C F (1989) Matrix Computations (2nd Edition). Johns Hopkins University Press, Baltimore, Maryland.

[2] Wilkinson J H (1965) The Algebraic Eigenvalue Problem. Oxford University Press.

5. Parameters

1: TRANS -- CHARACTER*1 Input
On entry: the operation to be performed as follows:
TRANS = 'N' (No transpose)
Perform the operation B:=QB.

```
     TRANS = 'T' or 'C' (Transpose)
                                      T
           Perform the operation B:=Q B.
      Constraint: TRANS must be one of 'N', 'T' or 'C'.

2:   WHERET -- CHARACTER*1                                        Input
     On entry: indicates where the elements of (zeta) are to be
     found as follows:
     WHERET = 'I' (In A)
           The elements of (zeta) are in A.

     WHERET = 'S' (Separate)
           The elements of (zeta) are separate from A, in ZETA.
     Constraint: WHERET must be one of 'I' or 'S'.

3:   M -- INTEGER                                                 Input
     On entry: m, the number of rows of A. Constraint: M >= N.

4:   N -- INTEGER                                                 Input
     On entry: n, the number of columns of A.

     When N = 0 then an immediate return is effected.
     Constraint: N >= 0.

5:   A(LDA,*) -- DOUBLE PRECISION array                           Input
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the leading m by n strictly lower triangular part
     of the array A must contain details of the matrix Q. In
     addition, when WHERET = 'I', then the diagonal elements of A
     must contain the elements of (zeta) as described under the
     argument ZETA below.

     When WHERET = 'S', the diagonal elements of the array A are
     referenced, since they are used temporarily to store the
     (zeta) , but they contain their original values on return.
           k

6:   LDA -- INTEGER                                               Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F01QDF is called.
     Constraint: LDA >= max(1,M).

7:   ZETA(*) -- DOUBLE PRECISION array                            Input
     Note: when WHERET = 'S', the dimension of the array ZETA
     must be greater than or equal to max(1,N). On entry: if
     WHERET = 'S', the array ZETA must contain the elements of
     (zeta). If ZETA(k) = 0.0 then T  is assumed to be I
                                    k
```

```
     otherwise ZETA(k) is assumed to contain (zeta) .
                                                    k

     When WHERET = 'I', ZETA is not referenced.

 8:  NCOLB -- INTEGER                                          Input
     On entry: ncolb, number of columns of B.

     When NCOLB = 0 then an immediate return is effected.
     Constraint: NCOLB >= 0.

 9:  B(LDB,*) -- DOUBLE PRECISION array                  Input/Output
     Note: the second dimension of the array B must be at least
     max(1,NCOLB).
     On entry: the leading m by ncolb part of the array B must
     contain the matrix to be transformed. On exit: B is
     overwritten by the transformed matrix.

10:  LDB -- INTEGER                                           Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F01QDF is called.
     Constraint: LDB >= max(1,M).

11:  WORK(*) -- DOUBLE PRECISION array                      Workspace
     Note: the dimension of the array WORK must be at least
     max(1,NCOLB).

12:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     On entry TRANS /= 'N', 'T' or 'C',

     or       WHERET /= 'I' or 'S',

     or       M < N,
```

```
     or       N < 0,

     or       LDA < M,

     or       NCOLB < 0,

     or       LDB < M.

7. Accuracy

                                        T
Letting C denote the computed matrix Q B, C satisfies the
relation

                                QC=B+E,

where

                        ||E||<=c(epsilon)||B||,

and (epsilon) the machine precision (see X02AJF(*)), c is a
modest function of m and |||.||| denotes the spectral (two) norm.
An equivalent result holds for the computed matrix QB. See also
Section 7 of F01QCF.

8. Further Comments

The approximate number of floating-point operations is given by
2n(2m-n)ncolb.

9. Example

                         T
To obtain the matrix Q B for the matrix B given by

                              ( 1.1  0.00)
                              ( 0.9  0.00)
                            B=( 0.6  1.32)
                              ( 0.0  1.10)
                              (-0.8 -0.26)

following the QR factorization of the 5 by 3 matrix A given by

                            (2.0  2.5  2.5)
                            (2.0  2.5  2.5)
                          A=(1.6 -0.4  2.8).
                            (2.0 -0.5  0.5)
                            (1.2 -0.3 -2.9)

The example program is not reproduced here. The source code for
```

```
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations                                    F01QEF
       F01QEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F01QEF returns the first ncolq columns of the real m by m
orthogonal matrix Q, where Q is given as the product of
Householder transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. Specification

      SUBROUTINE F01QEF (WHERET, M, N, NCOLQ, A, LDA, ZETA,
     1                   WORK, IFAIL)
      INTEGER          M, N, NCOLQ, LDA, IFAIL
      DOUBLE PRECISION A(LDA,*), ZETA(*), WORK(*)
      CHARACTER*1      WHERET

3. Description

Q is assumed to be given by

                                    T
                      Q=(Q Q   ...Q ) ,
                          n n-1    1

Q  being given in the form
 k

                             (I 0 )
                          Q =(0 T )
                           k (   k)

where

                                  T
                         T =I-u u ,
                          k    k k
```

$$u_k = \begin{pmatrix} (\zeta)_k \\ z_k \end{pmatrix},$$

$(\zeta)_k$ is a scalar and z_k is an (m-k) element vector. z_k must be supplied in the kth column of A in elements A(k+1,k),...,A(m,k) and $(\zeta)_k$ must be supplied either in A(k,k) or in ZETA(k), depending upon the parameter WHERET.

4. References

[1] Golub G H and Van Loan C F (1989) Matrix Computations (2nd Edition). Johns Hopkins University Press, Baltimore, Maryland.

[2] Wilkinson J H (1965) The Algebraic Eigenvalue Problem. Oxford University Press.

5. Parameters

1: WHERET -- CHARACTER*1 Input
On entry: indicates where the elements of (zeta) are to be found as follows:
WHERET = 'I' (In A)
The elements of (zeta) are in A.

WHERET = 'S' (Separate)
The elements of (zeta) are separate from A, in ZETA.
Constraint: WHERET must be one of 'I' or 'S'.

2: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

3: N -- INTEGER Input
On entry: n, the number of columns of A. Constraint: N >= 0.

4: NCOLQ -- INTEGER Input
On entry: ncolq, the required number of columns of Q.
Constraint: 0 <= NCOLQ <= M.

When NCOLQ = 0 then an immediate return is effected.

5: A(LDA,*) -- DOUBLE PRECISION array Input/Output
Note: the second dimension of the array A must be at least max(1,N,NCOLQ).
On entry: the leading m by n strictly lower triangular part

of the array A must contain details of the matrix Q. In addition, when WHERET = 'I', then the diagonal elements of A must contain the elements of (zeta) as described under the argument ZETA below. On exit: the first NCOLQ columns of the array A are overwritten by the first NCOLQ columns of the m by m orthogonal matrix Q. When N = 0 then the first NCOLQ columns of A are overwritten by the first NCOLQ columns of the identity matrix.

6: LDA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the (sub)program from which F01QEF is called.
Constraint: LDA >= max(1,M).

7: ZETA(*) -- DOUBLE PRECISION array Input
Note: the dimension of the array ZETA must be at least max(1,N).
On entry: with WHERET = 'S', the array ZETA must contain the elements of (zeta). If ZETA(k) = 0.0 then T_k is assumed to be I, otherwise ZETA(k) is assumed to contain $(\text{zeta})_k$.

When WHERET = 'I', the array ZETA is not referenced.

8: WORK(*) -- DOUBLE PRECISION array Workspace
Note: the dimension of the array WORK must be at least max(1,NCOLQ).

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1
On entry WHERET /= 'I' or 'S',

or M < N,

```
     or       N < 0,

     or       NCOLQ < 0 or NCOLQ > M,

     or       LDA < M.

7. Accuracy

The computed matrix Q satisfies the relation

                              Q=P+E,

where P is an exactly orthogonal matrix and

                         ||E||<=c(epsilon)

(epsilon) is the machine precision (see X02AJF(*)), c is a modest
function of m and |||.||| denotes the spectral (two) norm. See
also Section 7 of F01QCF.

8. Further Comments

The approximate number of floating-point operations required is
given by

     2
     -n{(3m-n)(2ncolq-n)-n(ncolq-n)},    ncolq>n,
     3

     2     2
     -ncolq (3m-ncolq),                 ncolq<=n.
     3

9. Example

To obtain the 5 by 5 orthogonal matrix Q following the QR
factorization of the 5 by 3 matrix A given by

                        (2.0  2.5  2.5)
                        (2.0  2.5  2.5)
                      A=(1.6 -0.4  2.8).
                        (2.0 -0.5  0.5)
                        (1.2 -0.3 -2.9)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

F01 -- Matrix Factorizations F01RCF
F01RCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01RCF finds the QR factorization of the complex m by n matrix A, where m>=n.

2. Specification

```
SUBROUTINE   F01RCF (M, N, A, LDA, THETA, IFAIL)
INTEGER              M, N, LDA, IFAIL
COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*)
```

3. Description

The m by n matrix A is factorized as

$$A = \begin{pmatrix} R \\ 0 \end{pmatrix} \quad \text{when } m>n,$$

$$A = QR \quad \text{when } m=n,$$

where Q is an m by m unitary matrix and R is an n by n upper triangular matrix with real diagonal elements.

The factorization is obtained by Householder's method. The kth transformation matrix, Q_k, which is used to introduce zeros into the kth column of A is given in the form

$$Q_k = \begin{pmatrix} I & 0 \\ 0 & T_k \end{pmatrix},$$

where

$$T_k = I - (\text{gamma}) u_k u_k^H,$$

$$u_k = \begin{pmatrix} (\text{zeta})_k \\ z_k \end{pmatrix},$$

(gamma) is a scalar for which Re (gamma) =1.0, (zeta) is a real
k k k
scalar and z is an (m-k) element vector. (gamma) , (zeta) and
k k k
z are chosen to annihilate the elements below the triangular
k
part of A and to make the diagonal elements real.

The scalar (gamma) and the vector u are returned in the kth
k k
element of the array THETA and in the kth column of A, such that
(theta) , given by
k

(theta) =((zeta) ,Im(gamma)),
k k k

is in THETA(k) and the elements of z are in a ,...,a . The
k k+1,k m,k
elements of R are returned in the upper triangular part of A.

Q is given by

H
Q=(Q Q ...Q) .
n n-1 1

A good background description to the QR factorization is given in
Dongarra et al [1], but note that this routine is not based upon
LINPACK routine ZQRDC.

4. References

[1] Dongarra J J, Moler C B, Bunch J R and Stewart G W (1979)
LINPACK Users' Guide. SIAM, Philadelphia.

[2] Wilkinson J H (1965) The Algebraic Eigenvalue Problem.
Oxford University Press.

5. Parameters

1: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER Input
On entry: n, the number of columns of A. Constraint: N >=
0.

When N = 0 then an immediate return is effected.

```
 3:  A(LDA,*) -- COMPLEX(KIND(1.0D0)) array          Input/Output
     Note: the second dimension of the array A must be at least
     max(1,N).
     On entry: the leading m by n part of the array A must
     contain the matrix to be factorized. On exit: the n by n
     upper triangular part of A will contain the upper triangular
     matrix R, with the imaginary parts of the diagonal elements
     set to zero, and the m by n strictly lower triangular part
     of A will contain details of the factorization as described
     above.

 4:  LDA -- INTEGER                                        Input
     On entry:
     the first dimension of the array A as declared in the
     (sub)program from which F01RCF is called.
     Constraint: LDA >= max(1,M).

 5:  THETA(*) -- COMPLEX(KIND(1.0D)) array                 Output
     Note: the dimension of the array THETA must be at least
     max(1,N).
     On exit: the scalar (theta)  for the kth transformation. If
                                k
     T =I then THETA(k) = 0.0; if
      k
                     ((alpha) 0)
                  T =(      0 I)   Re(alpha)<0.0,
                   k
     then THETA(k)=(alpha); otherwise THETA(k) contains THETA(k)
     as described in Section 3 and Re(THETA(k)) is always in the


     range (1.0, \/2.0).

 6:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
```

```
     On entry M < N,

     or       N < 0,

     or       LDA < M.

7. Accuracy
The computed factors Q and R satisfy the relation

                              (R)
                            Q(0)=A+E,

where

                     ||E||<=c(epsilon)||A||,

(epsilon) being the machine precision, c is a modest function of
m and n and ||.|| denotes the spectral (two) norm.

8. Further Comments

The approximate number of real floating-point operations is given
     2
by 8n (3m-n)/3.

Following the use of this routine the operations

                                        H
                        B:=QB  and  B:=Q B,

where B is an m by k matrix, can be performed by calls to F01RDF.
The operation B:=QB can be obtained by the call:


      IFAIL = 0
      CALL F01RDF('No conjugate', 'Separate', M, N, A, LDA, THETA,
     *            K, B, LDB, WORK, IFAIL)

        H
and B:=Q B can be obtained by the call:


      IFAIL = 0
      CALL F01RDF('Conjugate', 'Separate', M, N, A, LDA, THETA,
     *            K, B, LDB, WORK, IFAIL)

In both cases WORK must be a k element array that is used as
workspace. If B is a one-dimensional array (single column) then
the parameter LDB can be replaced by M. See F01RDF for further
details.
```

The first k columns of the unitary matrix Q can either be obtained by setting B to the first k columns of the unit matrix and using the first of the above two calls, or by calling F01REF, which overwrites the k columns of Q on the first k columns of the array A. Q is obtained by the call:

```
CALL F01REF('Separate', M, N, K, A, LDA, THETA, WORK, IFAIL)
```

As above, WORK must be a k element array. If k is larger than n, then A must have been declared to have at least k columns.

Operations involving the matrix R can readily be performed by the Level 2 BLAS routines ZTRSV and ZTRMV (see Chapter F06), but note that no test for near singularity of R is incorporated in ZTRSV. If R is singular, or nearly singular, then F02XUF(*) can be used to determine the singular value decomposition of R.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

$$A=\begin{pmatrix} 0.5i & -0.5+1.5i & -1.0+1.0i \\ 0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\ 0.4 & -0.4+0.4i & 1.8 \\ 0.3-0.4i & 0.1+0.7i & 0.0 \\ -0.3i & 0.3+0.3i & 2.4i \end{pmatrix}.$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

F01 -- Matrix Factorizations F01RDF

F01RDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01RDF performs one of the transformations

$$B:=QB \text{ or } B:=Q^H B,$$

```
where B is an m by ncolb complex matrix and Q is an m by m
unitary matrix, given as the product of Householder
transformation matrices.

This routine is intended for use following F01RCF or F01RFF(*).

2. Specification

      SUBROUTINE F01RDF (TRANS, WHERET, M, N, A, LDA, THETA,
     1                   NCOLB, B, LDB, WORK, IFAIL)
      INTEGER              M, N, LDA, NCOLB, LDB, IFAIL
      COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), B(LDB,*), WORK(*)
      CHARACTER*1          TRANS, WHERET

3. Description

The unitary matrix Q is assumed to be given by

                                  H
                       Q=(Q Q   ...Q ) ,
                           n n-1    1

Q  being given in the form
 k

                             (I 0 )
                          Q =(0 T ),
                           k (   k)

where

                                  H
                        T =I-(gamma) u u ,
                         k          k k k

                              ((zeta) )
                              (      k)
                           u =(z      ),
                            k ( k     )

(gamma)  is a scalar for which Re (gamma) =1.0, (zeta)  is a real
       k                                  k            k
scalar and z  is an (m-k) element vector.
           k

z  must be supplied in the kth column of A in elements
 k
a     ,...,a    and (theta) , given by
 k+1,k      m,k            k
```

$$(\theta)_k = (\zeta_k, \mathrm{Im}(\gamma_k)),$$

must be supplied either in $a_{k,k}$ or in THETA(k), depending upon the parameter WHERET.

To obtain Q explicitly B may be set to I and pre-multiplied by Q^H. This is more efficient than obtaining Q^H. Alternatively, F01REF may be used to obtain Q overwritten on A.

4. References

[1] Wilkinson J H (1965) The Algebraic Eigenvalue Problem. Oxford University Press.

5. Parameters

1: TRANS -- CHARACTER*1 Input
On entry: the operation to be performed as follows:
TRANS = 'N' (No transpose)
Perform the operation B:=QB.

TRANS = 'C' (Conjugate transpose)
Perform the operation $B:=Q^H B$.
Constraint: TRANS must be one of 'N' or 'C'.

2: WHERET -- CHARACTER*1 Input
On entry: the elements of (theta) are to be found as follows:
WHERET = 'I' (In A)
The elements of (theta) are in A.

WHERET = 'S' (Separate)
The elements of (theta) are separate from A, in THETA.
Constraint: WHERET must be one of 'I' or 'S'.

3: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

4: N -- INTEGER Input
On entry: n, the number of columns of A. Constraint: N >= 0.

When N = 0 then an immediate return is effected.

5: A(LDA,*) -- COMPLEX(KIND(1.0D)) array Input
Note: the second dimension of the array A must be at least

```
    max(1,N).
    On entry: the leading m by n strictly lower triangular part
    of the array A must contain details of the matrix Q. In
    addition, when WHERET = 'I', then the diagonal elements of A
    must contain the elements of (theta) as described under the
    argument THETA below.

    When WHERET = 'S', then the diagonal elements of the array A
    are referenced, since they are used temporarily to store the
    (zeta) , but they contain their original values on return.
          k

6:  LDA -- INTEGER                                          Input
    On entry:
    the first dimension of the array A as declared in the
    (sub)program from which F01RDF is called.
    Constraint: LDA >= max(1,M).

7:  THETA(*) -- COMPLEX(KIND(1.0D)) array                   Input
    Note: the dimension of the array THETA must be at least
    max(1,N).
    On entry: with WHERET = 'S', the array THETA must contain
    the elements of (theta). If THETA(k)=0.0 then T  is assumed
                                                   k
    to be I; if THETA(k)=(alpha), with Re(alpha)<0.0, then T  is
                                                            k
    assumed to be of the form
                              ((alpha) 0)
                         T =(0       I);
                          k
    otherwise THETA(k) is assumed to contain (theta)  given by
                                                      k
    (theta) =((zeta) ,Im(gamma) ).
           k        k          k

    When WHERET = 'I', the array THETA is not referenced, and
    may be dimensioned of length 1.

8:  NCOLB -- INTEGER                                        Input
    On entry: ncolb, the number of columns of B. Constraint:
    NCOLB >= 0.
    When NCOLB = 0 then an immediate return is effected.

9:  B(LDB,*) -- COMPLEX(KIND(1.0D)) array           Input/Output
    Note: the second dimension of the array B must be at least
    max(1,NCOLB).
    On entry: the leading m by ncolb part of the array B must
    contain the matrix to be transformed. On exit: B is
    overwritten by the transformed matrix.
```

```
10:  LDB -- INTEGER                                          Input
     On entry:
     the first dimension of the array B as declared in the
     (sub)program from which F01RDF is called.
     Constraint: LDB >= max(1,M).

11:  WORK(*) -- COMPLEX(KIND(1.0D)) array                 Workspace
     Note: the dimension of the array WORK must be at least
     max(1,NCOLB).

12:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     On entry TRANS /= 'N' or 'C',

     or       WHERET /= 'I' or 'S',

     or       M < N,

     or       N < 0,

     or       LDA < M,

     or       NCOLB < 0,

     or       LDB < M.

7. Accuracy

Letting C denote the computed matrix Q B, C satisfies the
relation

                              QC=B+E,

where

                     ||E||<=c(epsilon)||B||,
```

```
(epsilon) being the machine precision, c is a modest function of
m and |||.||| denotes the spectral (two) norm. An equivalent
result holds for the computed matrix QB. See also Section 7 of
F01RCF.

8. Further Comments

The approximate number of real floating-point operations is given
by 8n(2m-n)ncolb.

9. Example

                      H
To obtain the matrix Q B for the matrix B given by

                     (-0.55+1.05i  0.45+1.05i)
                     ( 0.49+0.93i  1.09+0.13i)
                  B=( 0.56-0.16i  0.64+0.16i)
                     ( 0.39+0.23i -0.39-0.23i)
                     ( 1.13+0.83i -1.13+0.77i)

following the QR factorization of the 5 by 3 matrix A given by

                    (  0.5i   -0.5+1.5i -1.0+1.0i)
                    (0.4+0.3i  0.9+1.3i  0.2+1.4i)
                 A=(  0.4    -0.4+0.4i  1.8     ).
                    (0.3-0.4i  0.1+0.7i  0.0     )
                    ( -0.3i    0.3+0.3i      2.4i)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
F01 -- Matrix Factorizations                                F01REF
        F01REF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F01REF returns the first ncolq columns of the complex m by m
unitary matrix Q, where Q is given as the product of Householder
transformation matrices.
```

```
This routine is intended for use following F01RCF.

2. Specification

      SUBROUTINE F01REF (WHERET, M, N, NCOLQ, A, LDA, THETA,
     1                   WORK, IFAIL)
      INTEGER              M, N, NCOLQ, LDA, IFAIL
      COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), WORK(*)
      CHARACTER*1          WHERET

3. Description

The unitary matrix Q is assumed to be given by

                                    H
                        Q=(Q Q   ...Q ) ,
                            n n-1    1

Q  being given in the form
 k

                              (I 0 )
                            Q =(0 T ),
                             k (   k)

where

                                    H
                         T =I-(gamma) u u ,
                          k           k k k

                               ((zeta) )
                               (       k)
                             u =(z       ),
                              k ( k      )

(gamma)  is a scalar for which Re (gamma) =1.0, (zeta)  is a real
       k                                  k            k

scalar and z  is an (m-k) element vector.
            k

z  must be supplied in the kth column of A in elements
 k
a     ,...,a    and (theta) , given by
 k+1,k      m,k            k

                  (theta) =((zeta) ,Im (gamma) ),
                         k        k           k
```

must be supplied either in a or in THETA(k) depending upon the
k,k
parameter WHERET.

4. References

[1] Wilkinson J H (1965) The Algebraic Eigenvalue Problem.
Oxford University Press.

5. Parameters

1: WHERET -- CHARACTER*1 Input
On entry: the elements of (theta) are to be found as follows:
WHERET = 'I' (In A)
The elements of (theta) are in A.

WHERET = 'S' (Separate)
The elements of (theta) are separate from A, in THETA.
Constraint: WHERET must be one of 'I' or 'S'.

2: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

3: N -- INTEGER Input
On entry: n, the number of columns of A. Constraint: N >= 0.

4: NCOLQ -- INTEGER Input
On entry: ncolq, the required number of columns of Q.
Constraint: 0 <= NCOLQ <= M.

When NCOLQ = 0 then an immediate return is effected.

5: A(LDA,*) -- COMPLEX(KIND(1.0D)) array Input/Output
Note: the second dimension of the array A must be at least max(1,N,NCOLQ).
On entry: the leading m by n strictly lower triangular part of the array A must contain details of the matrix Q. In addition, when WHERET = 'I', then the diagonal elements of A must contain the elements of (theta) as described under the argument THETA below. On exit: the first NCOLQ columns of the array A are overwritten by the first NCOLQ columns of the m by m unitary matrix Q. When N = 0 then the first NCOLQ columns of A are overwritten by the first NCOLQ columns of the unit matrix.

6: LDA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the

```
     (sub)program from which F01REF is called.
     Constraint: LDA >= max(1,M).

 7:  THETA(*) -- COMPLEX(KIND(1.0D)) array                    Input
     Note: the dimension of the array THETA must be at least
     max(1,N).
     On entry: if WHERET = 'S', the array THETA must contain the
     elements of (theta). If THETA(k)=0.0 then T  is assumed to
                                                k
     be I; if THETA(k)=(alpha), with Re(alpha)<0.0, then T  is
                                                          k
     assumed to be of the form
                              ((alpha) 0)
                         T =(   0    I);
                          k
     otherwise THETA(k) is assumed to contain (theta)  given by
                                                   k
     (theta) =((zeta) ,Im(gamma) ).
            k        k          k

     When WHERET = 'I', the array THETA is not referenced.

 8:  WORK(*) -- COMPLEX(KIND(1.0D)) array                 Workspace
     Note: the dimension of the array WORK must be at least
     max(1,NCOLQ).

 9:  IFAIL -- INTEGER                                     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
     On entry WHERET /= 'I' or 'S',

     or       M < N,

     or       N < 0,

     or       NCOLQ < 0 or NCOLQ > M,
```

```
     or       LDA < M.

7. Accuracy

The computed matrix Q satisfies the relation

                              Q=P+E,

where P is an exactly unitary matrix and

                        ||E||<=c(epsilon),

(epsilon) being the machine precision, c is a modest function of
m and |||.||| denotes the spectral (two) norm. See also Section 7
of F01RCF.

8. Further Comments

The approximate number of real floating-point operations required
is given by

     8
     -n{(3m-n)(2ncolq-n)-n(ncolq-n)},  ncolq>n
     3

     8     2
     -ncolq (3m-ncolq),                ncolq<=n
     3

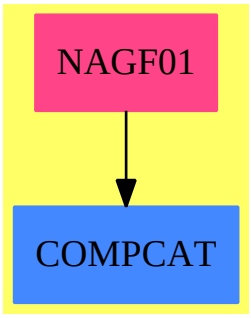
9. Example

To obtain the 5 by 5 unitary matrix Q following the QR
factorization of the 5 by 3 matrix A given by

                  (    0.5i -0.5+1.5i -1.0+1.4i)
                  (0.4+0.3i  0.9+1.3i  0.2+1.4i)
                A=(0.4      -0.4+0.4i  1.8     ).
                  (0.3-0.4i  0.1+0.7i  0.0     )
                  (   -0.3i  0.3+0.3i      2.4i)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.8.1 NagMatrixOperationsPackage (NAGF01)



Exports:

f01brf	f01bsf	f01maf	f01mcf	f01qcf
f01qdf	f01qef	f01rcf	f01rdf	f01ref

— package NAGF01 NagMatrixOperationsPackage —

```
)abbrev package NAGF01 NagMatrixOperationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:15 1994
++ Description:
++ This package uses the NAG Library to provide facilities for matrix
++ factorizations and associated transformations.

NagMatrixOperationsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    f01brf : (Integer,Integer,Integer,Integer,_
        DoubleFloat,Boolean,Boolean,List Boolean,Matrix DoubleFloat,_
        Matrix Integer,Matrix Integer,Integer) -> Result
     ++ f01brf(n,nz,licn,lirn,pivot,lblock,grow,abort,a,irn,icn,ifail)
     ++ factorizes a real sparse matrix. The routine either forms
     ++ the LU factorization of a permutation of the entire matrix, or,
     ++ optionally, first permutes the matrix to block lower triangular
     ++ form and then only factorizes the diagonal blocks.
     ++ See \downlink{Manual Page}{manpageXXf01brf}.
    f01bsf : (Integer,Integer,Integer,Matrix Integer,_
        Matrix Integer,Matrix Integer,Matrix Integer,Boolean,DoubleFloat,_
        Boolean,Matrix Integer,Matrix DoubleFloat,Integer) -> Result
     ++ f01bsf(n,nz,licn,ivect,jvect,icn,ikeep,grow,
     ++        eta,abort,idisp,avals,ifail)
     ++ factorizes a real sparse matrix using the pivotal sequence
     ++ previously obtained by F01BRF when a matrix of the same sparsity
```

```
 ++ pattern was factorized.
 ++ See \downlink{Manual Page}{manpageXXf01bsf}.
f01maf : (Integer,Integer,Integer,Integer,_
    List Boolean,Matrix DoubleFloat,Matrix Integer,Matrix Integer,_
    DoubleFloat,DoubleFloat,Integer) -> Result
 ++ f01maf(n,nz,licn,lirn,abort,avals,irn,icn,droptl,densw,ifail)
 ++ computes an incomplete Cholesky factorization of a real
 ++ sparse symmetric positive-definite matrix A.
 ++ See \downlink{Manual Page}{manpageXXf01maf}.
f01mcf : (Integer,Matrix DoubleFloat,Integer,Matrix Integer,_
    Integer) -> Result
 ++ f01mcf(n,avals,lal,nrow,ifail)
 ++ computes the Cholesky factorization of a real symmetric
 ++ positive-definite variable-bandwidth matrix.
 ++ See \downlink{Manual Page}{manpageXXf01mcf}.
f01qcf : (Integer,Integer,Integer,Matrix DoubleFloat,_
    Integer) -> Result
 ++ f01qcf(m,n,lda,a,ifail)
 ++ finds the QR factorization of the real m by n matrix A,
 ++ where m>=n.
 ++ See \downlink{Manual Page}{manpageXXf01qcf}.
f01qdf : (String,String,Integer,Integer,_
    Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,Integer,_
    Matrix DoubleFloat,Integer) -> Result
 ++ f01qdf(trans,wheret,m,n,a,lda,zeta,ncolb,ldb,b,ifail)
 ++ performs one of the transformations
 ++ See \downlink{Manual Page}{manpageXXf01qdf}.
f01qef : (String,Integer,Integer,Integer,_
    Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
 ++ f01qef(wheret,m,n,ncolq,lda,zeta,a,ifail)
 ++ returns the first ncolq columns of the real m by m
 ++ orthogonal matrix Q, where Q is given as the product of
 ++ Householder transformation matrices.
 ++ See \downlink{Manual Page}{manpageXXf01qef}.
f01rcf : (Integer,Integer,Integer,Matrix Complex DoubleFloat,_
    Integer) -> Result
 ++ f01rcf(m,n,lda,a,ifail)
 ++ finds the QR factorization of the complex m by n matrix A,
 ++ where m>=n.
 ++ See \downlink{Manual Page}{manpageXXf01rcf}.
f01rdf : (String,String,Integer,Integer,_
    Matrix Complex DoubleFloat,Integer,Matrix Complex DoubleFloat,_
    Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
 ++ f01rdf(trans,wheret,m,n,a,lda,theta,ncolb,ldb,b,ifail)
 ++ performs one of the transformations
 ++ See \downlink{Manual Page}{manpageXXf01rdf}.
f01ref : (String,Integer,Integer,Integer,_
    Integer,Matrix Complex DoubleFloat,Matrix Complex DoubleFloat,_
    Integer) -> Result
 ++ f01ref(wheret,m,n,ncolq,lda,theta,a,ifail)
```

```
    ++ returns the first ncolq columns of the complex m by m
    ++ unitary matrix Q, where Q is given as the product of Householder
    ++ transformation matrices.
    ++ See \downlink{Manual Page}{manpageXXf01ref}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Any
    import Record
    import Integer
    import Matrix DoubleFloat
    import Boolean
    import NAGLinkSupportPackage
    import AnyFunctions1(Integer)
    import AnyFunctions1(DoubleFloat)
    import AnyFunctions1(Boolean)
    import AnyFunctions1(String)
    import AnyFunctions1(List Boolean)
    import AnyFunctions1(Matrix DoubleFloat)
    import AnyFunctions1(Matrix Complex DoubleFloat)
    import AnyFunctions1(Matrix Integer)


    f01brf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
        lirnArg:Integer,pivotArg:DoubleFloat,lblockArg:Boolean,_
        growArg:Boolean,abortArg:List Boolean,aArg:Matrix DoubleFloat,_
        irnArg:Matrix Integer,icnArg:Matrix Integer,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01brf",_
        ["n"::S,"nz"::S,"licn"::S,"lirn"::S,"pivot"::S_
        ,"lblock"::S,"grow"::S,"ifail"::S,"abort"::S,"ikeep"::S,_
        "w"::S,"idisp"::S,"a"::S_
        ,"irn"::S,"icn"::S,"iw"::S]$Lisp,_
        ["ikeep"::S,"w"::S,"idisp"::S,"iw"::S]$Lisp,_
        [["double"::S,"pivot"::S,["w"::S,"n"::S]$Lisp_
        ,["a"::S,"licn"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"nz"::S,"licn"::S,"lirn"::S_
        ,["ikeep"::S,["*"::S,5$Lisp,"n"::S]$Lisp]$Lisp,_
        ["idisp"::S,10$Lisp]$Lisp,["irn"::S,"lirn"::S]$Lisp,_
        ["icn"::S,"licn"::S]$Lisp_
        ,"ifail"::S,["iw"::S,["*"::S,8$Lisp,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["logical"::S,"lblock"::S,"grow"::S,["abort"::S,4$Lisp]$Lisp]$Lisp_
        ]$Lisp,_
        ["ikeep"::S,"w"::S,"idisp"::S,"a"::S,"irn"::S,_
        "icn"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,nzArg::Any,licnArg::Any,lirnArg::Any,pivotArg::Any,_
        lblockArg::Any,growArg::Any,ifailArg::Any,abortArg::Any,_
        aArg::Any,irnArg::Any,icnArg::Any ])_
```

```
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01bsf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
        ivectArg:Matrix Integer,jvectArg:Matrix Integer,icnArg:Matrix Integer,_
        ikeepArg:Matrix Integer,growArg:Boolean,etaArg:DoubleFloat,_
        abortArg:Boolean,idispArg:Matrix Integer,avalsArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01bsf",_
        ["n"::S,"nz"::S,"licn"::S,"grow"::S,"eta"::S_
        ,"abort"::S,"rpmin"::S,"ifail"::S,"ivect"::S,"jvect"::S,_
        "icn"::S,"ikeep"::S,"idisp"::S_
        ,"w"::S,"avals"::S,"iw"::S]$Lisp,_
        ["w"::S,"rpmin"::S,"iw"::S]$Lisp,_
        [["double"::S,"eta"::S,["w"::S,"n"::S]$Lisp_
        ,"rpmin"::S,["avals"::S,"licn"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"nz"::S,"licn"::S,["ivect"::S,"nz"::S]$Lisp_
        ,["jvect"::S,"nz"::S]$Lisp,["icn"::S,"licn"::S]$Lisp,_
        ["ikeep"::S,["*"::S,5$Lisp,"n"::S]$Lisp]$Lisp_
        ,["idisp"::S,2$Lisp]$Lisp,"ifail"::S,_
        ["iw"::S,["*"::S,8$Lisp,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["logical"::S,"grow"::S,"abort"::S]$Lisp_
        ]$Lisp,_
        ["w"::S,"rpmin"::S,"avals"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,nzArg::Any,licnArg::Any,growArg::Any,etaArg::Any,_
        abortArg::Any,ifailArg::Any,ivectArg::Any,jvectArg::Any,icnArg::Any,_
        ikeepArg::Any,idispArg::Any,avalsArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01maf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
        lirnArg:Integer,abortArg:List Boolean,avalsArg:Matrix DoubleFloat,_
        irnArg:Matrix Integer,icnArg:Matrix Integer,droptlArg:DoubleFloat,_
        denswArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01maf",_
        ["n"::S,"nz"::S,"licn"::S,"lirn"::S,"droptl"::S_
        ,"densw"::S,"ifail"::S,"abort"::S,"wkeep"::S,"ikeep"::S,_
        "inform"::S,"avals"::S_
        ,"irn"::S,"icn"::S,"iwork"::S]$Lisp,_
        ["wkeep"::S,"ikeep"::S,"inform"::S,"iwork"::S]$Lisp,_
        [["double"::S,["wkeep"::S,["*"::S,3$Lisp,"n"::S]$Lisp]$Lisp_
        ,["avals"::S,"licn"::S]$Lisp,"droptl"::S,"densw"::S]$Lisp_
        ,["integer"::S,"n"::S,"nz"::S,"licn"::S,"lirn"::S_
        ,["ikeep"::S,["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,_
        ["inform"::S,4$Lisp]$Lisp,["irn"::S,"lirn"::S]$Lisp,_
        ["icn"::S,"licn"::S]$Lisp_
        ,"ifail"::S,["iwork"::S,["*"::S,6$Lisp,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["logical"::S,["abort"::S,3$Lisp]$Lisp]$Lisp_
```

```
    ]$Lisp,_
    ["wkeep"::S,"ikeep"::S,"inform"::S,"avals"::S,"irn"::S,_
    "icn"::S,"droptl"::S,"densw"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,nzArg::Any,licnArg::Any,lirnArg::Any,droptlArg::Any,_
    denswArg::Any,ifailArg::Any,abortArg::Any,avalsArg::Any,_
    irnArg::Any,icnArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f01mcf(nArg:Integer,avalsArg:Matrix DoubleFloat,lalArg:Integer,_
    nrowArg:Matrix Integer,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f01mcf",_
    ["n"::S,"lal"::S,"ifail"::S,"avals"::S,"nrow"::S,"al"::S,"d"::S]$Lisp,_
    ["al"::S,"d"::S]$Lisp,_
    [["double"::S,["avals"::S,"lal"::S]$Lisp,["al"::S,"lal"::S]$Lisp_
    ,["d"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"n"::S,"lal"::S,["nrow"::S,"n"::S]$Lisp_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["al"::S,"d"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,lalArg::Any,ifailArg::Any,avalsArg::Any,nrowArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f01qcf(mArg:Integer,nArg:Integer,ldaArg:Integer,_
    aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f01qcf",_
    ["m"::S,"n"::S,"lda"::S,"ifail"::S,"zeta"::S,"a"::S]$Lisp,_
    ["zeta"::S]$Lisp,_
    [["double"::S,["zeta"::S,"n"::S]$Lisp,["a"::S,"lda"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"lda"::S,"ifail"::S_
    ]$Lisp_
    ]$Lisp,_
    ["zeta"::S,"a"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

f01qdf(transArg:String,wheretArg:String,mArg:Integer,_
    nArg:Integer,aArg:Matrix DoubleFloat,ldaArg:Integer,_
    zetaArg:Matrix DoubleFloat,ncolbArg:Integer,ldbArg:Integer,_
    bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "f01qdf",_
    ["trans"::S,"wheret"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S,"ldb"::S,_
    "ifail"::S,"a"::S,"zeta"::S,"b"::S,"work"::S]$Lisp,_
    ["work"::S]$Lisp,_
```

```
        [["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp,["zeta"::S,"n"::S]$Lisp,_
        ["b"::S,"ldb"::S,"ncolb"::S]$Lisp,["work"::S,"ncolb"::S]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S_
        ,"ldb"::S,"ifail"::S]$Lisp_
        ,["character"::S,"trans"::S,"wheret"::S]$Lisp_
        ]$Lisp,_
        ["b"::S,"ifail"::S]$Lisp,_
        [([transArg::Any,wheretArg::Any,mArg::Any,nArg::Any,ldaArg::Any,_
        ncolbArg::Any,ldbArg::Any,ifailArg::Any,aArg::Any,_
        zetaArg::Any,bArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01qef(wheretArg:String,mArg:Integer,nArg:Integer,_
        ncolqArg:Integer,ldaArg:Integer,zetaArg:Matrix DoubleFloat,_
        aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01qef",_
        ["wheret"::S,"m"::S,"n"::S,"ncolq"::S,"lda"::S_
        ,"ifail"::S,"zeta"::S,"a"::S,"work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["double"::S,["zeta"::S,"n"::S]$Lisp,_
        ["a"::S,"lda"::S,"ncolq"::S]$Lisp_
        ,["work"::S,"ncolq"::S]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ncolq"::S,"lda"::S_
        ,"ifail"::S]$Lisp_
        ,["character"::S,"wheret"::S]$Lisp_
        ]$Lisp,_
        ["a"::S,"ifail"::S]$Lisp,_
        [([wheretArg::Any,mArg::Any,nArg::Any,ncolqArg::Any,ldaArg::Any,_
        ifailArg::Any,zetaArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01rcf(mArg:Integer,nArg:Integer,ldaArg:Integer,_
        aArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01rcf",_
        ["m"::S,"n"::S,"lda"::S,"ifail"::S,"theta"::S,"a"::S]$Lisp,_
        ["theta"::S]$Lisp,_
        [["integer"::S,"m"::S,"n"::S,"lda"::S,"ifail"::S]$Lisp_
        ,["double complex"::S,["theta"::S,"n"::S]$Lisp,_
        ["a"::S,"lda"::S,"n"::S]$Lisp]$Lisp]$Lisp,_
        ["theta"::S,"a"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01rdf(transArg:String,wheretArg:String,mArg:Integer,_
        nArg:Integer,aArg:Matrix Complex DoubleFloat,ldaArg:Integer,_
```

```
        thetaArg:Matrix Complex DoubleFloat,ncolbArg:Integer,ldbArg:Integer,_
        bArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01rdf",_
        ["trans"::S,"wheret"::S,"m"::S,"n"::S,"lda"::S_
        ,"ncolb"::S,"ldb"::S,"ifail"::S,"a"::S,"theta"::S,_
        "b"::S,"work"::S]$Lisp,["work"::S]$Lisp,_
        [["integer"::S,"m"::S,"n"::S,"lda"::S,"ncolb"::S_
        ,"ldb"::S,"ifail"::S]$Lisp_
        ,["character"::S,"trans"::S,"wheret"::S]$Lisp_
        ,["double complex"::S,["a"::S,"lda"::S,"n"::S]$Lisp,_
        ["theta"::S,"n"::S]$Lisp,["b"::S,"ldb"::S,"ncolb"::S]$Lisp,_
        ["work"::S,"ncolb"::S]$Lisp]$Lisp]$Lisp,_
        ["b"::S,"ifail"::S]$Lisp,_
        [([transArg::Any,wheretArg::Any,mArg::Any,nArg::Any,_
        ldaArg::Any,ncolbArg::Any,ldbArg::Any,ifailArg::Any,aArg::Any,_
        thetaArg::Any,bArg::Any ])@List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    f01ref(wheretArg:String,mArg:Integer,nArg:Integer,_
        ncolqArg:Integer,ldaArg:Integer,thetaArg:Matrix Complex DoubleFloat,_
        aArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "f01ref",_
        ["wheret"::S,"m"::S,"n"::S,"ncolq"::S,"lda"::S_
        ,"ifail"::S,"theta"::S,"a"::S,"work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["integer"::S,"m"::S,"n"::S,"ncolq"::S,"lda"::S_
        ,"ifail"::S]$Lisp_
        ,["character"::S,"wheret"::S]$Lisp_
        ,["double complex"::S,["theta"::S,"n"::S]$Lisp,_
        ["a"::S,"lda"::S,"n"::S]$Lisp,["work"::S,"ncolq"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["a"::S,"ifail"::S]$Lisp,_
        [([wheretArg::Any,mArg::Any,nArg::Any,ncolqArg::Any,ldaArg::Any,_
        ifailArg::Any,thetaArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— NAGF01.dotabb —

```
"NAGF01" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF01"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NAGF01" -> "COMPCAT"
```

———-

15.9 package NAGE04 NagOptimisationPackage

— NagOptimisationPackage.help —

```
E04 -- Minimizing or Maximizing a Function    Introduction -- E04
                          Chapter E04
              Minimizing or Maximizing a Function

Contents of this Introduction:

1.     Scope of the Chapter

2.     Background to the Problems

2.1.   Types of Optimization Problems

2.1.1. Unconstrained minimization

2.1.2. Nonlinear least-squares problems

2.1.3. Minimization subject to bounds on the variables

2.1.4. Minimization subject to linear constraints

2.1.5. Minimization subject to nonlinear constraints

2.2.   Geometric Representation and Terminology

2.2.1. Gradient vector

2.2.2. Hessian matrix

2.2.3. Jacobian matrix; matrix of constraint normals

2.3.   Sufficient Conditions for a Solution

2.3.1. Unconstrained minimization

2.3.2. Minimization subject to bounds on the variables

2.3.3. Linearly-constrained minimization

2.3.4. Nonlinearly-constrained minimization

2.4.   Background to Optimization Methods

2.4.1. Methods for unconstrained optimization
```

2.4.2. Methods for nonlinear least-squares problems

2.4.3. Methods for handling constraints

2.5. Scaling

2.5.1. Transformation of variables

2.5.2. Scaling the objective function

2.5.3. Scaling the constraints

2.6. Analysis of Computed Results

2.6.1. Convergence criteria

2.6.2. Checking results

2.6.3. Monitoring progress

2.6.4. Confidence intervals for least-squares solutions

2.7. References

3. Recommendations on Choice and Use of Routines

3.1. Choice of Routine

3.2. Service Routines

3.3. Function Evaluations at Infeasible Points

3.4. Related Problems

1. Scope of the Chapter

An optimization problem involves minimizing a function (called the objective function) of several variables, possibly subject to restrictions on the values of the variables defined by a set of constraint functions. The routines in the NAG Foundation Library are concerned with function minimization only, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by -1.

This introduction is only a brief guide to the subject of optimization designed for the casual user. Anyone with a difficult or protracted problem to solve will find it beneficial to consult a more detailed text, such as Gill et al [5] or

Fletcher [3].

Readers who are unfamiliar with the mathematics of the subject may find some sections difficult at first reading; if so, they should concentrate on Sections 2.1, 2.2, 2.5, 2.6 and 3.

2. Background to the Problems

2.1. Types of Optimization Problems

Solution of optimization problems by a single, all-purpose, method is cumbersome and inefficient. Optimization problems are therefore classified into particular categories, where each category is defined by the properties of the objective and constraint functions, as illustrated by some examples below.

Properties of Objective Function	Properties of Constraints
Nonlinear	Nonlinear
Sums of squares of nonlinear functions	Sparse linear
Quadratic	Linear
Sums of squares of linear functions	Bounds
Linear	None

For instance, a specific problem category involves the minimization of a nonlinear objective function subject to bounds on the variables. In the following sections we define the particular categories of problems that can be solved by routines contained in this Chapter.

2.1.1. Unconstrained minimization

In unconstrained minimization problems there are no constraints on the variables. The problem can be stated mathematically as follows:

$$\underset{x}{\text{minimize}}\ F(x)$$

where x is in R^n, that is, $x=(x_1,x_2,\ldots,x_n)^T$.

2.1.2. Nonlinear least-squares problems

Special consideration is given to the problem for which the function to be minimized can be expressed as a sum of squared functions. The least-squares problem can be stated mathematically as follows:

$$\underset{x}{\text{minimize}} \left\{ f \;\middle|\; f = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2 \right\}, \quad x \text{ is in } R^n$$

where the ith element of the m-vector f is the function $f_i(x)$.

2.1.3. Minimization subject to bounds on the variables

These problems differ from the unconstrained problem in that at least one of the variables is subject to a simple restriction on its value, e.g.$x_5 \leq 10$, but no constraints of a more general form are present.

The problem can be stated mathematically as follows:

$$\underset{x}{\text{minimize}}\ F(x), \quad x \text{ is in } R^n$$

subject to $l_i \leq x_i \leq u_i$, i=1,2,...,n.

This format assumes that upper and lower bounds exist on all the variables. By conceptually allowing $u_i = \text{infty}$ and $l_i = -\text{infty}$ all the variables need not be restricted.

2.1.4. Minimization subject to linear constraints

A general linear constraint is defined as a constraint function that is linear in more than one of the variables, e.g. $3x_1 + 2x_2 \geq 4$

The various types of linear constraint are reflected in the following mathematical statement of the problem:

$$\underset{x}{\text{minimize}}\ F(x), \quad x \text{ is in } R^n$$

subject to the

$$
\begin{array}{lll}
 & T & \\
\text{equality} & a\,x=b & i=1,2,\ldots,m\ ; \\
\text{constraints:} & a_i\ x & \\
 & T & \\
\text{inequality} & a\,x\geq b & i=m_1+1,m_2+2,\ldots,m\ ; \\
\text{constraints:} & a_i\ x & \\
 & T & \\
 & a\,x\leq b & i=m_2+1,m_3+2,\ldots,m\ ; \\
 & a_i\ x & \\
 & T & \\
\text{range} & s\leq a\,x\leq t & i=m_3+1,m_4+2,\ldots,m\ ; \\
\text{constraints:} & s_j\ a_i\ x\ t_j & j=1,2,\ldots,m_4-m_3\ ; \\
\text{bounds} & l\leq x\leq u & i=1,2,\ldots,n \\
\text{constraints:} & l_i\ x\ u_i &
\end{array}
$$

where each a_i is a vector of length n; b_i, s_j and t_j are constant scalars; and any of the categories may be empty.

Although the bounds on x_i could be included in the definition of general linear constraints, we prefer to distinguish between them for reasons of computational efficiency.

If F(x) is a linear function, the linearly-constrained problem is termed a linear programming problem (LP problem); if F(x) is a quadratic function, the problem is termed a quadratic programming problem (QP problem). For further discussion of LP and QP problems, including the dual formulation of such problems, see Dantzig [2].

2.1.5. Minimization subject to nonlinear constraints

A problem is included in this category if at least one constraint function is nonlinear, e.g. $x_1+x_3+x_4^2-2\geq 0$. The mathematical statement of the problem is identical to that for the linearly-constrained case, except for the addition of the following constraints:

```
equality       c (x)=0        i=1,2,...,m ;
constraints:    i                        5

inequality     c (x)>=0       i=m +1,m +2,...,m ;
constraints:    i               5    5        6

range          v <=c (x)<=w   i=m +1,m +2,...,m ,
constraints:    j   i      j     6    6        7
                               j=1,2,...,m -m
                                          7  6
```

where each c_i is a nonlinear function; v_j and w_j are constant scalars; and any category may be empty. Note that we do not include a separate category for constraints of the form $c_i(x)<=0$, since this is equivalent to $-c_i(x)>=0$.

2.2. Geometric Representation and Terminology

To illustrate the nature of optimization problems it is useful to consider the following example in two dimensions

$$F(x)=e^{x_1}(4x_1^2+2x_2^2+4x_1x_2+2x_2+1).$$

(This function is used as the example function in the documentation for the unconstrained routines.)

Figure 1

Please see figure in printed Reference Manual

Figure 1 is a contour diagram of F(x). The contours labelled $F_0, F_1, \ldots, F_4$ are isovalue contours, or lines along which the function F(x) takes specific constant values. The point x^* is a local unconstrained minimum, that is, the value of $F(x^*)$ is less than at all the neighbouring points. A function may have several such minima. The lowest of the local minima is termed a global minimum. In the problem illustrated in Figure 1, x^* is the only

local minimum. The point x is said to be a saddle point because

it is a minimum along the line AB, but a maximum along CD.

If we add the constraint x >=0 to the problem of minimizing F(x),
                          1
the solution remains unaltered. In Figure 1 this constraint is
represented by the straight line passing through x =0, and the
                                                  1
shading on the line indicates the unacceptable region. The region
    n
in R satisfying the constraints of an optimization problem is
termed the feasible region. A point satisfying the constraints is
defined as a feasible point.

If we add the nonlinear constraint x +x -x x -1.5>=0, represented
                                    1 2 1 2
                                              *
by the curved shaded line in Figure 1, then x is not a feasible
                                             ^
point. The solution of the new constrained problem is x, the
feasible point with the smallest function value.

2.2.1. Gradient vector

The vector of first partial derivatives of F(x) is called the
gradient vector, and is denoted by g(x), i.e.,

$$g(x)=\left[\frac{dF(x)}{dx_1}, \frac{dF(x)}{dx_2}, \ldots, \frac{dF(x)}{dx_n}\right]^T .$$

For the function illustrated in Figure 1,

$$g(x)=\begin{bmatrix} x_1 \\ F(x)+e^{x_1}(8x_1+4x_2) \\ x_1 \\ e^{x_1}(4x_1+4x_2+2) \end{bmatrix}.$$

The gradient vector is of importance in optimization because it
must be zero at an unconstrained minimum of any function with
continuous first derivatives.

2.2.2. Hessian matrix

The matrix of second partial derivatives of a function is termed
its Hessian matrix. The Hessian matrix of F(x) is denoted by G(x)

and its (i,j)th element is given by $\frac{d^2 F(x)}{dx_i dx_j}$. If F(x) has continuous second derivatives, then G(x) must be positive semi-definite at any unconstrained minimum of F.

2.2.3. Jacobian matrix; matrix of constraint normals

In nonlinear least-squares problems, the matrix of first partial derivatives of the vector-valued function f(x) is termed the Jacobian matrix of f(x) and its (i,j)th component is $\frac{df_i}{dx_j}$.

The vector of first partial derivatives of the constraint $c_i(x)$ is denoted by

$$\hat{a}_i(x) = \left[\frac{dc_i(x)}{dx_1}, \ldots, \frac{dc_i(x)}{dx_n} \right]^T .$$

At a point, x, the vector $\hat{a}_i(x)$ is orthogonal (normal) to the isovalue contour of $c_i(x)$ passing through x; this relationship is illustrated for a two-dimensional function in Figure 2.

Figure 2
Please see figure in printed Reference Manual

The matrix whose columns are the vectors $\{\hat{a}_i\}$ is termed the matrix of constraint normals. Note that if $c_i(x)$ is a linear constraint involving $a_i^T x$, then its vector of first partial derivatives is simply the vector $\hat{a}_i$.

2.3. Sufficient Conditions for a Solution

All nonlinear functions will be assumed to have continuous second derivatives in the neighbourhood of the solution.

2.3.1. Unconstrained minimization

The following conditions are sufficient for the point x^* to be an unconstrained local minimum of F(x):

(i) $|||g(x^*)|||=0$; and

(ii) $G(x^*)$ is positive-definite,

where |||g||| denotes the Euclidean length of g.

2.3.2. Minimization subject to bounds on the variables

At the solution of a bounds-constrained problem, variables which are not on their bounds are termed free variables. If it is known in advance which variables are on their bounds at the solution, the problem can be solved as an unconstrained problem in just the free variables; thus, the sufficient conditions for a solution are similar to those for the unconstrained case, applied only to the free variables.

Sufficient conditions for a feasible point x^* to be the solution of a bound-constrained problem are as follows:

(i) $|||g(x^*)|||=0$; and

(ii) $G(x^*)$ is positive-definite; and

(iii) $g_j(x^*)<0, x_j=u_j$; $g_j(x^*)>0, x_j=l_j$,

where g(x) is the gradient of F(x) with respect to the free

variables, and G(x) is the Hessian matrix of F(x) with respect to the free variables. The extra condition (iii) ensures that F(x) cannot be reduced by moving off one or more of the bounds.

2.3.3. Linearly-constrained minimization

For the sake of simplicity, the following description does not

```
include a specific treatment of bounds or range constraints,
since the results for general linear inequality constraints can
be applied directly to these cases.

                *
At a solution x , of a linearly-constrained problem, the
constraints which hold as equalities are called the active or
binding constraints. Assume that there are t active constraints
                 *          ^
at the solution x , and let A denote the matrix whose columns are
                                                                 ^
the columns of A corresponding to the active constraints, with b
the vector similarly obtained from b; then

                              ^T * ^
                              A x =b.

The matrix Z is defined as an n by (n-t) matrix satisfying:

                           ^T        T
                           A Z=0;   Z Z=I.

The columns of Z form an orthogonal basis for the set of vectors
                                ^
orthogonal to the columns of A.

Define

            T
     g (x)=Z g(x), the projected gradient vector of F(x);
      z

            T
     G (x)=Z G(x)Z, the projected Hessian matrix of F(x).
      z

At the solution of a linearly-constrained problem, the projected
gradient vector must be zero, which implies that the gradient
          *
vector g(x ) can be written as a linear combination of the
                              t
           ^           *      --          ^  ^
columns of A, i.e., g(x )= >  (lambda) a =A(lambda). The scalar
                              --         i i
                              i=1
(lambda)  is defined as the Lagrange multiplier corresponding to
        i
the ith active constraint. A simple interpretation of the ith
Lagrange multiplier is that it gives the gradient of F(x) along
the ith active constraint normal; a convenient definition of the
```

Lagrange multiplier vector (although not a recommended method for computation) is:

$$(\lambda) = (A\hat{}T\hat{} A)^{-1} A\hat{}T\hat{} g(x^*).$$

Sufficient conditions for x^* to be the solution of a linearly-constrained problem are:

(i) x^* is feasible, and $A\hat{}T\hat{} x^* = b$; and

(ii) $|||g(x^*)|||_z = 0$, or equivalently, $g(x^*) = A\hat{}T\hat{}(\lambda)$; and

(iii) $G_z(x^*)$ is positive-definite; and

(iv) $(\lambda)_i > 0$ if $(\lambda)_i$ corresponds to a constraint $A\hat{}T\hat{}_i x^* \geq b_i$;

$(\lambda)_i < 0$ if $(\lambda)_i$ corresponds to a constraint $A\hat{}T\hat{}_i x^* \leq b_i$.

The sign of $(\lambda)_i$ is immaterial for equality constraints, which by definition are always active.

2.3.4. Nonlinearly-constrained minimization

For nonlinearly-constrained problems, much of the terminology is defined exactly as in the linearly-constrained case. The set of active constraints at x again means the set of constraints that hold as equalities at x, with corresponding definitions of $\hat{c}$ and $\hat{A}$: the vector $\hat{c}(x)$ contains the active constraint functions, and the columns of $\hat{A}(x)$ are the gradient vectors of the active

constraints. As before, Z is defined in terms of A(x) as a matrix such that:

$$\hat{A}^T Z = 0; \quad Z^T Z = I$$

where the dependence on x has been suppressed for compactness.

The projected gradient vector $g_z(x)$ is the vector $Z^T g(x)$. At the solution x^* of a nonlinearly-constrained problem, the projected gradient must be zero, which implies the existence of Lagrange multipliers corresponding to the active constraints, i.e., $g(x^*) = \hat{A}(x^*)(\lambda)$.

The Lagrangian function is given by:

$$L(x,(\lambda)) = F(x) - (\lambda)^{\hat{T}} c(x).$$

We define $g_L(x)$ as the gradient of the Lagrangian function; $G_L(x)$ as its Hessian matrix, and $\hat{G}_L(x)$ as its projected Hessian matrix, i.e., $\hat{G}_L = Z^T G_L Z$.

Sufficient conditions for x^* to be a solution of nonlinearly-constrained problem are:

(i) x^* is feasible, and $\hat{c}(x^*) = 0$; and

(ii) $|||g_z(x^*)||| = 0$, or, equivalently, $g(x^*) = \hat{A}(x^*)(\lambda)$; and

(iii) $\hat{G}_L(x^*)$ is positive-definite; and

(iv) $(\lambda)_i > 0$ if $(\lambda)_i$ corresponds to a constraint of the

```
      ^
form c >=0; the sign of (lambda)  is immaterial for an
      i                         i
equality constraint.
```

Note that condition (ii) implies that the projected gradient of
*
the Lagrangian function must also be zero at x , since the
T   ^ *
application of Z annihilates the matrix A(x).

2.4. Background to Optimization Methods

All the algorithms contained in this Chapter generate an
(k)   *
iterative sequence {x   } that converges to the solution x in the limit, except for some special problem categories (i.e., linear and quadratic programming). To terminate computation of the sequence, a convergence test is performed to determine whether the current estimate of the solution is an adequate approximation. The convergence tests are discussed in Section 2.6

(k)
Most of the methods construct a sequence {x   } satisfying:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)},$$

(k)
where the vector p   is termed the direction of search, and
(k)   (k)
(alpha)   is the steplength. The steplength (alpha)   is chosen
(k+1)   (k)
so that F(x   )<F(x   ).

2.4.1. Methods for unconstrained optimization

The distinctions among methods arise primarily from the need to use varying levels of information about derivatives of F(x) in defining the search direction. We describe three basic approaches to unconstrained problems, which may be extended to other problem categories. Since a full description of the methods would fill several volumes, the discussion here can do little more than allude to the processes involved, and direct the reader to other sources for a full explanation.

(a) Newton-type Methods (Modified Newton Methods)

(k)

Newton-type methods use the Hessian matrix G(x (k)), or a finite difference approximation to G(x (k)), to define the search direction. The routines in the Library either require a subroutine that computes the elements of G(x (k)), or they approximate G(x (k)) by finite differences.

Newton-type methods are the most powerful methods available for general problems and will find the minimum of a quadratic function in one iteration. See Sections 4.4. and 4.5.1. of Gill et al [5].

(b) Quasi-Newton Methods

Quasi-Newton methods approximate the Hessian G(x (k)) by a matrix B (k) which is modified at each iteration to include information obtained about the curvature of F along the latest search direction. Although not as robust as Newton-type methods, quasi-Newton methods can be more efficient because G(x (k)) is not computed, or approximated by finite-differences. Quasi-Newton methods minimize a quadratic function in n iterations. See Section 4.5.2 of Gill et al [5].

(c) Conjugate-Gradient Methods

Unlike Newton-type and quasi-Newton methods, conjugate gradient methods do not require the storage of an n by n matrix and so are ideally suited to solve large problems. Conjugate-gradient type methods are not usually as reliable or efficient as Newton-type, or quasi-Newton methods. See Section 4.8.3 of Gill et al [5].

2.4.2. Methods for nonlinear least-squares problems

These methods are similar to those for unconstrained optimization, but exploit the special structure of the Hessian matrix to give improved computational efficiency.

Since

$$F(x) = \sum_{} f_{}^{m} (x)^{2}$$

$$\sum_{i=1}^{m}$$

the Hessian matrix G(x) is of the form

$$G(x)=2[J(x)^T J(x)+\sum_{i=1}^{m} f_i(x)G_i(x)],$$

where J(x) is the Jacobian matrix of f(x), and $G_i(x)$ is the Hessian matrix of $f_i(x)$.

In the neighbourhood of the solution, $|||f(x)|||$ is often small compared to $|||J(x)^T J(x)|||$ (for example, when f(x) represents the goodness of fit of a nonlinear model to observed data). In such cases, $2J(x)^T J(x)$ may be an adequate approximation to G(x), thereby avoiding the need to compute or approximate second derivatives of $\{f_i(x)\}$. See Section 4.7 of Gill et al [5].

2.4.3. Methods for handling constraints

Bounds on the variables are dealt with by fixing some of the variables on their bounds and adjusting the remaining free variables to minimize the function. By examining estimates of the Lagrange multipliers it is possible to adjust the set of variables fixed on their bounds so that eventually the bounds active at the solution should be correctly identified. This type of method is called an active set method. One feature of such methods is that, given an initial feasible point, all approximations $x^{(k)}$ are feasible. This approach can be extended to general linear constraints. At a point, x, the set of constraints which hold as equalities being used to predict, or approximate, the set of active constraints is called the working set.

Nonlinear constraints are more difficult to handle. If at all possible, it is usually beneficial to avoid including nonlinear constraints during the formulation of the problem. The methods currently implemented in the Library handle nonlinearly constrained problems either by transforming them into a sequence of bound constraint problems, or by transforming them into a sequence of quadratic programming problems. A feature of almost

all methods for nonlinear constraints is that $x^{(k)}$ is not guaranteed to be feasible except in the limit, and this is certainly true of the routines currently in the Library. See Chapter 6, particularly Section 6.4 and Section 6.5 of Gill et al [5].

Anyone interested in a detailed description of methods for optimization should consult the references.

2.5. Scaling

Scaling (in a broadly defined sense) often has a significant influence on the performance of optimization methods. Since convergence tolerances and other criteria are necessarily based on an implicit definition of 'small' and 'large', problems with unusual or unbalanced scaling may cause difficulties for some algorithms. Nonetheless, there are currently no scaling routines in the Library, although the position is under constant review. In light of the present state of the art, it is considered that sensible scaling by the user is likely to be more effective than any automatic routine. The following sections present some general comments on problem scaling.

2.5.1. Transformation of variables

One method of scaling is to transform the variables from their original representation, which may reflect the physical nature of the problem, to variables that have certain desirable properties in terms of optimization. It is generally helpful for the following conditions to be satisfied:

(i) the variables are all of similar magnitude in the region of interest;

(ii) a fixed change in any of the variables results in similar changes in F(x). Ideally, a unit change in any variable produces a unit change in F(x);

(iii) the variables are transformed so as to avoid cancellation error in the evaluation of F(x).

Normally, users should restrict themselves to linear transformations of variables, although occasionally nonlinear transformations are possible. The most common such transformation (and often the most appropriate) is of the form

$$x_{new} = D x_{old},$$

where D is a diagonal matrix with constant coefficients. Our experience suggests that more use should be made of the transformation

$$x_{new} = Dx_{old} + v,$$

where v is a constant vector.

Consider, for example, a problem in which the variable x_3 represents the position of the peak of a Gaussian curve to be fitted to data for which the extreme values are 150 and 170; therefore x_3 is known to lie in the range 150--170. One possible scaling would be to define a new variable x_3, given by

$$x_3 = \frac{x_3}{170}.$$

A better transformation, however, is given by defining x_3 as

$$x_3 = \frac{x_3 - 160}{10}.$$

Frequently, an improvement in the accuracy of evaluation of F(x) can result if the variables are scaled before the routines to evaluate F(x) are coded. For instance, in the above problem just mentioned of Gaussian curve fitting, x_3 may always occur in terms of the form $(x_3 - x_m)$, where x_m is a constant representing the mean peak position.

2.5.2. Scaling the objective function

The objective function has already been mentioned in the discussion of scaling the variables. The solution of a given problem is unaltered if F(x) is multiplied by a positive

constant, or if a constant value is added to F(x). It is generally preferable for the objective function to be of the order of unity in the region of interest; thus, if in the original formulation F(x) is always of the order of 10^{+5} (say), then the value of F(x) should be multiplied by 10^{-5} when evaluating the function within the optimization routines. If a constant is added or subtracted in the computation of F(x), usually it should be omitted - i.e., it is better to formulate $F(x)$ as $x_1^2 + x_2^2$ rather than as $x_1^2 + x_2^2 + 1000$ or even $x_1^2 + x_2^2 + 1$. The inclusion of such a constant in the calculation of F(x) can result in a loss of significant figures.

2.5.3. Scaling the constraints

The solution of a nonlinearly-constrained problem is unaltered if the ith constraint is multiplied by a positive weight w_i. At the approximation of the solution determined by a Library routine, the active constraints will not be satisfied exactly, but will have 'small' values (for example, $c_1 = 10^{-8}$, $c_2 = 10^{-6}$, etc.). In general, this discrepancy will be minimized if the constraints are weighted so that a unit change in x produces a similar change in each constraint.

A second reason for introducing weights is related to the effect of the size of the constraints on the Lagrange multiplier estimates and, consequently, on the active set strategy. Additional discussion is given in Gill et al [5].

2.6. Analysis of Computed Results

2.6.1. Convergence criteria

The convergence criteria inevitably vary from routine to routine, since in some cases more information is available to be checked (for example, is the Hessian matrix positive-definite?), and different checks need to be made for different problem categories (for example, in constrained minimization it is necessary to verify whether a trial solution is feasible). Nonetheless, the underlying principles of the various criteria are the same; in non-mathematical terms, they are:

(i) is the sequence $\{x^{(k)}\}$ converging?

(k)
(ii) is the sequence {F } converging?

(iii) are the necessary and sufficient conditions for the solution satisfied?

The decision as to whether a sequence is converging is necessarily speculative. The criterion used in the present routines is to assume convergence if the relative change occurring between two successive iterations is less than some prescribed quantity. Criterion (iii) is the most reliable but often the conditions cannot be checked fully because not all the required information may be available.

2.6.2. Checking results

Little a priori guidance can be given as to the quality of the solution found by a nonlinear optimization algorithm, since no guarantees can be given that the methods will always work. Therefore, it is necessary for the user to check the computed solution even if the routine reports success. Frequently a ' solution' may have been found even when the routine does not report a success. The reason for this apparent contradiction is that the routine needs to assess the accuracy of the solution. This assessment is not an exact process and consequently may be unduly pessimistic. Any 'solution' is in general only an approximation to the exact solution, and it is possible that the accuracy specified by the user is too stringent.

Further confirmation can be sought by trying to check whether or not convergence tests are almost satisfied, or whether or not some of the sufficient conditions are nearly satisfied. When it is thought that a routine has returned a non-zero value of IFAIL only because the requirements for 'success' were too stringent it may be worth restarting with increased convergence tolerances.

For nonlinearly-constrained problems, check whether the solution returned is feasible, or nearly feasible; if not, the solution returned is not an adequate solution.

Confidence in a solution may be increased by resolving the problem with a different initial approximation to the solution. See Section 8.3 of Gill et al [5] for further information.

2.6.3. Monitoring progress

Many of the routines in the Chapter have facilities to allow the user to monitor the progress of the minimization process, and users are encouraged to make use of these facilities. Monitoring

information can be a great aid in assessing whether or not a satisfactory solution has been obtained, and in indicating difficulties in the minimization problem or in the routine's ability to cope with the problem.

The behaviour of the function, the estimated solution and first derivatives can help in deciding whether a solution is acceptable and what to do in the event of a return with a non-zero value of IFAIL.

2.6.4. Confidence intervals for least-squares solutions

When estimates of the parameters in a nonlinear least-squares problem have been found, it may be necessary to estimate the variances of the parameters and the fitted function. These can be calculated from the Hessian of F(x) at the solution.

In many least-squares problems, the Hessian is adequately approximated at the solution by $G=2J^{T}J$ (see Section 2.4.3). The Jacobian, J, or a factorization of J is returned by all the comprehensive least-squares routines and, in addition, a routine is supplied in the Library to estimate variances of the parameters following the use of most of the nonlinear least-squares routines, in the case that $G=2J^{T}J$ is an adequate approximation.

Let H be the inverse of G, and S be the sum of squares, both

calculated at the solution x; an unbiased estimate of the variance of the ith parameter x_i is

$$\text{var } x_i = \frac{2S}{m-n} H_{ii}$$

and an unbiased estimate of the covariance of x_i and x_j is

$$\text{covar}(x_i, x_j) = \frac{2S}{m-n} H_{ij} .$$

*

If x is the true solution, then the 100(1-(beta)) confidence

interval on x is

$$x_i - \sqrt{\mathrm{var}\ x_i}\ .\ t_{(1-(\mathrm{beta})/2,m-n)} < x_i < x_i^* + \sqrt{\mathrm{var}\ x_i}\ .\ t_{(1-(\mathrm{beta})/2,m-n)}, \quad i=1,2,\ldots,n$$

where $t_{(1-(\mathrm{beta})/2,m-n)}$ is the 100(1-(beta))/2 percentage point of the t-distribution with m-n degrees of freedom.

In the majority of problems, the residuals f_i, for i=1,2,...,m, contain the difference between the values of a model function (phi)(z,x) calculated for m different values of the independent variable z, and the corresponding observed values at these points. The minimization process determines the parameters, or

constants x, of the fitted function (phi)(z,x). For any value, z, of the independent variable z, an unbiased estimate of the variance of (phi) is

$$\mathrm{var}\ (\mathrm{phi}) = \frac{2S}{m-n} \sum_{i=1}^{n} \sum_{j=1}^{n} \left[\frac{\mathrm{d}(\mathrm{phi})}{\mathrm{d}x_i}\right]_z \left[\frac{\mathrm{d}(\mathrm{phi})}{\mathrm{d}x_j}\right]_z H_{ij}.$$

The 100(1-(beta)) confidence interval on F at the point z is

$$(\mathrm{phi})(z,x) - \sqrt{\mathrm{var}\ (\mathrm{phi})}.t_{((\mathrm{beta})/2,m-n)} < (\mathrm{phi})(z,x^*) < (\mathrm{phi})(z,x) + \sqrt{\mathrm{var}\ (\mathrm{phi})}.t_{((\mathrm{beta})/2,m-n)}.$$

((beta)/2,m-n)

For further details on the analysis of least-squares solutions see Bard [1] and Wolberg [7].

2.7. References

[1] Bard Y (1974) Nonlinear Parameter Estimation. Academic Press.

[2] Dantzig G B (1963) Linear Programming and Extensions. Princeton University Press.

[3] Fletcher R (1987) Practical Methods of Optimization. Wiley (2nd Edition).

[4] Gill P E and Murray W (eds) (1974) Numerical Methods for Constrained Optimization. Academic Press.

[5] Gill P E, Murray W and Wright M H (1981) Practical Optimization. Academic Press.

[6] Murray W (ed) (1972) Numerical Methods for Unconstrained Optimization. Academic Press.

[7] Wolberg J R (1967) Prediction Analysis. Van Nostrand.

3. Recommendations on Choice and Use of Routines

The choice of routine depends on several factors: the type of problem (unconstrained, etc.); the level of derivative information available (function values only, etc.); the experience of the user (there are easy-to-use versions of some routines); whether or not storage is a problem; and whether computational time has a high priority.

3.1. Choice of Routine

Routines are provided to solve the following types of problem:

Nonlinear Programming	E04UCF
Quadratic Programming	E04NAF
Linear Programming	E04MBF
Nonlinear Function (using 1st derivatives)	E04DGF
Nonlinear Function, unconstrained or simple bounds (using function values only)	E04JAF
Nonlinear least-squares (using function values only)	E04FDF

Nonlinear least-squares E04GCF
(using function values and 1st derivatives)

E04UCF can be used to solve unconstrained, bound-constrained and linearly-constrained problems.

E04NAF can be used as a comprehensive linear programming solver; however, in most cases the easy-to-use routine E04MBFwill be adequate.

E04MBF can be used to obtain a feasible point for a set of linear constraints.

E04DGF can be used to solve large scale unconstrained problems.

The routines can be used to solve problems in a single variable.

3.2. Service Routines

One of the most common errors in use of optimization routines is that the user's subroutines incorrectly evaluate the relevant partial derivatives. Because exact gradient information normally enhances efficiency in all areas of optimization, the user should be encouraged to provide analytical derivatives whenever possible. However, mistakes in the computation of derivatives can result in serious and obscure run-time errors, as well as complaints that the Library routines are incorrect.

E04UCF incorporates a check on the gradients being supplied and users are encouraged to utilize this option; E04GCF also incorporates a call to a derivative checker.

E04YCF estimates selected elements of the variance-covariance matrix for the computed regression parameters following the use of a nonlinear least-squares routine.

3.3. Function Evaluations at Infeasible Points

Users must not assume that the routines for constrained problems will require the objective function to be evaluated only at points which satisfy the constraints, i.e., feasible points. In the first place some of the easy-to-use routines call a service routine which will evaluate the objective function at the user-supplied initial point, and at neighbouring points (to check user-supplied derivatives or to estimate intervals for finite differencing). Apart from this, all routines will ensure that any evaluations of the objective function occur at points which approximately satisfy any simple bounds or linear constraints. Satisfaction of such constraints is only approximate because:

(a) routines which have a parameter FEATOL may allow such constraints to be violated by a margin specified by FEATOL;

(b) routines which estimate derivatives by finite differences may require function evaluations at points which just violate such constraints even though the current iteration just satisfies them.

There is no attempt to ensure that the current iteration satisfies any nonlinear constraints. Users who wish to prevent their objective function being evaluated outside some known region (where it may be undefined or not practically computable), may try to confine the iteration within this region by imposing suitable simple bounds or linear constraints (but beware as this may create new local minima where these constraints are active).

Note also that some routines allow the user-supplied routine to return a parameter (MODE) with a negative value to force an immediate clean exit from the minimization when the objective function cannot be evaluated.

3.4. Related Problems

Apart from the standard types of optimization problem, there are other related problems which can be solved by routines in this or other chapters of the Library.

E04MBF can be used to find a feasible point for a set of linear constraints and simple bounds.

Two routines in Chapter F04 solve linear least-squares problems,

$$\text{i.e., minimize } \sum_{i=1}^{m} r_i(x)^2 \quad \text{where } r_i(x) = b_i - \sum_{j=1}^{n} a_{ij} x_j .$$

E02GAF solves an overdetermined system of linear equations in the

$$l_1 \text{ norm, i.e., minimizes } \sum_{i=1}^{m} |r_i(x)|, \text{ with } r_i \text{ as above.}$$

E04 -- Minimizing or Maximizing a Function Contents -- E04 Chapter E04

Minimizing or Maximizing a Function

```
E04DGF  Unconstrained minimum, pre-conditioned conjugate gradient
        algorithm, function of several variables using 1st
        derivatives

E04DJF  Read optional parameter values for E04DGF from external
        file

E04DKF  Supply optional parameter values to E04DGF

E04FDF  Unconstrained minimum of a sum of squares, combined
        Gauss-Newton and modified Newton algorithm using function
        values only

E04GCF  Unconstrained minimum of a sum of squares, combined
        Gauss-Newton and quasi-Newton algorithm, using 1st
        derivatives

E04JAF  Minimum, function of several variables, quasi-Newton
        algorithm, simple bounds, using function values only

E04MBF  Linear programming problem

E04NAF  Quadratic programming problem

E04UCF  Minimum, function of several variables, sequential QP
        method, nonlinear constraints, using function values and
        optionally 1st derivatives

E04UDF  Read optional parameter values for E04UCF from external
        file

E04UEF  Supply optional parameter values to E04UCF

E04YCF  Covariance matrix for nonlinear least-squares problem
```

%%

```
E04 -- Minimizing or Maximizing a Function                E04DGF
        E04DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine
has been enhanced for use with AXIOM and is slightly different to
that offered in the standard version of the Foundation Library.  In
particular, the optional parameters of the NAG routine are now
included in the parameter list.  These are described in section
```

5.1.2, below.

1. Purpose

E04DGF minimizes an unconstrained nonlinear function of several variables using a pre-conditioned, limited memory quasi-Newton conjugate gradient method. First derivatives are required. The routine is intended for use on large scale problems.

2. Specification

```
      SUBROUTINE E04DGF(N,OBJFUN,ITER,OBJF,OBJGRD,X,IWORK,WORK,IUSER,
     1                  USER,ES,FU,IT,LIN,LIST,MA,OP,PR,STA,STO,
     2                  VE,IFAIL)
      INTEGER           N, ITER, IWORK(N+1), IUSER(*),
     1                  IT, PR, STA, STO, VE, IFAIL
      DOUBLE PRECISION OBJF, OBJGRD(N), X(N), WORK(13*N), USER(*)
     1                  ES, FU, LIN, OP, MA
      LOGICAL           LIST
      EXTERNAL          OBJFUN
```

3. Description

E04DGF uses a pre-conditioned conjugate gradient method and is based upon algorithm PLMA as described in Gill and Murray [1] and Gill et al [2] Section 4.8.3.

The algorithm proceeds as follows:

Let x_0 be a given starting point and let k denote the current iteration, starting with k=0. The iteration requires g_k, the gradient vector evaluated at x_k, the kth estimate of the minimum. At each iteration a vector p_k (known as the direction of search) is computed and the new estimate x_{k+1} is given by $x_k + (\alpha)_k p_k$ where $(\alpha)_k$ (the step length) minimizes the function $F(x_k + (\alpha)_k p_k)$ with respect to the scalar $(\alpha)_k$. A choice of initial step $(\alpha)_0$ is taken as

$$(\alpha)_0 = \min\{1, 2|F_0 - F_{k\,est}| / g_k^T g_k\}$$

where F is a user-supplied estimate of the function value at
est
the solution. If F is not specified, the software always
est
chooses the unit step length for (alpha) . Subsequent step length
0
estimates are computed using cubic interpolation with safeguards.

A quasi-Newton method can be used to compute the search direction
p by updating the inverse of the approximate Hessian (H) and
k k
computing

$$p_{k+1} = -H_{k+1}\, g_{k+1} \tag{1}$$

The updating formula for the approximate inverse is given by

$$H_{k+1} = H_k - \frac{1}{y_k^T s_k}\left(H_k y_k s_k^T + s_k y_k^T H_k\right) + \frac{1}{y_k^T s_k}\left(1 + \frac{y_k^T H_k y_k}{y_k^T s_k}\right) s_k s_k^T \tag{2}$$

where $y_k = g_k - g_{k-1}$ and $s_k = x_k - x_{k-1} = (\text{alpha})\, p_k$.

The method used by E04DGF to obtain the search direction is based
upon computing p as -H g where H is a matrix obtained
k+1 k+1 k+1 k+1
by updating the identity matrix with a limited number of quasi-
Newton corrections. The storage of an n by n matrix is avoided by
storing only the vectors that define the rank two corrections -
hence the term limited-memory quasi-Newton method. The precise
method depends upon the number of updating vectors stored. For
example, the direction obtained with the 'one-step' limited
memory update is given by (1) using (2) with H equal to the
k
identity matrix, viz.

$$p_{k+1} = -g_{k+1} + \frac{1}{y_k^T s_k}\left(s_k^T g_{k+1} + y_k^T g_{k+1}\right) s_k - \frac{s_k^T g_{k+1}}{y_k^T s_k}\left(1 + \frac{y_k^T y_k}{y_k^T s_k}\right) s_k$$

E04DGF uses a two-step method described in detail in Gill and

Murray [1] in which restarts and pre-conditioning are incorporated. Using a limited-memory quasi-Newton formula, such as the one above, guarantees p_{k+1} to be a descent direction if T all the inner products y_k are positive for all vectors y_k and s_k used in the updating formula.

The termination criterion of E04DGF is as follows:

Let τ_F specify a parameter that indicates the number of correct figures desired in F_k (τ_F is equivalent to Optimality Tolerance in the optional parameter list, see Section 5.1). If the following three conditions are satisfied

(i) $F_{k-1} - F_k < \tau_F (1+|F_k|)$

(ii) $||x_{k-1} - x_k|| < \sqrt{\tau_F} (1+||x_k||)$

(iii) $||g_k|| \leq 3 \sqrt{\tau_F} (1+|F_k|)$ or $||g_k|| < \epsilon_A$, where ϵ_A is the absolute error associated with computing the objective function

then the algorithm is considered to have converged. For a full discussion on termination criteria see Gill et al [2] Chapter 8.

4. References

[1] Gill P E and Murray W (1979) Conjugate-gradient Methods for Large-scale Nonlinear Optimization. Technical Report SOL 79-15. Department of Operations Research, Stanford University.

[2] Gill P E, Murray W and Wright M H (1981) Practical Optimization. Academic Press.

5. Parameters

1: N -- INTEGER Input
On entry: the number n of variables. Constraint: N >= 1.

2: OBJFUN -- SUBROUTINE, supplied by the user.

External Procedure

OBJFUN must calculate the objective function F(x) and its gradient for a specified n element vector x.

Its specification is:

```
      SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD,
     1                   NSTATE, IUSER, USER)
      INTEGER          MODE, N, NSTATE, IUSER(*)
      DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)
```

1: MODE -- INTEGER Input/Output
MODE is a flag that the user may set within OBJFUN to indicate a failure in the evaluation of the objective function. On entry: MODE is always non-negative. On exit: if MODE is negative the execution of E04DGF is terminated with IFAIL set to MODE.

2: N -- INTEGER Input
On entry: the number n of variables.

3: X(N) -- DOUBLE PRECISION array Input
On entry: the point x at which the objective function is required.

4: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function F at the current point x.

5: OBJGRD(N) -- DOUBLE PRECISION array Output
On exit: OBJGRD(i) must contain the value of $\frac{\partial F}{\partial x_i}$ at the point x, for i=1,2,...,n.

6: NSTATE -- INTEGER Input
On entry: NSTATE will be 1 on the first call of OBJFUN by E04DGF, and is 0 for all subsequent calls. Thus, if the user wishes, NSTATE may be tested within OBJFUN in order to perform certain calculations once only. For example the user may read data or initialise COMMON blocks when NSTATE = 1.

7: IUSER(*) -- INTEGER array User Workspace

8: USER(*) -- DOUBLE PRECISION array User Workspace
OBJFUN is called from E04DGF with the parameters IUSER and USER as supplied to E04DGF. The user is free to use arrays IUSER and USER to supply information to OBJFUN

```
     as an alternative to using COMMON.
    OBJFUN must be declared as EXTERNAL in the (sub)program
    from which E04DGF is called. Parameters denoted as
    Input must not be changed by this procedure.

 3: ITER -- INTEGER                                        Output
    On exit: the number of iterations performed.

 4: OBJF -- DOUBLE PRECISION                               Output
    On exit: the value of the objective function F(x) at the
    final iterate.

 5: OBJGRD(N) -- DOUBLE PRECISION array                    Output
    On exit: the objective gradient at the final iterate.

 6: X(N) -- DOUBLE PRECISION array                   Input/Output
    On entry: an initial estimate of the solution. On exit: the
    final estimate of the solution.

 7: IWORK(N+1) -- INTEGER array                         Workspace

 8: WORK(13*N) -- DOUBLE PRECISION array                Workspace

 9: IUSER(*) -- INTEGER array                      User Workspace
    Note: the dimension of the array IUSER must be at least 1.
    This array is not used by E04DGF, but is passed directly to
    routine OBJFUN and may be used to supply information to
    OBJFUN.

10: USER(*) -- DOUBLE PRECISION array              User Workspace
    Note: the dimension of the array USER must be at least 1.
    This array is not used by E04DGF, but is passed directly to
    routine OBJFUN and may be used to supply information to
    OBJFUN.

11: IFAIL -- INTEGER                                 Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.

    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).

    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit.


5.1. Optional Input Parameters
```

Several optional parameters in E04DGF define choices in the behaviour of the routine. In order to reduce the number of formal parameters of E04DGF these optional parameters have associated default values (see Section 5.1.3) that are appropriate for most problems. Therefore the user need only specify those optional parameters whose values are to be different from their default values.

The remainder of this section can be skipped by users who wish to use the default values for all optional parameters. A complete list of optional parameters and their default values is given in Section 5.1.3.

5.1.1. Specification of the Optional Parameters

Optional parameters may be specified by calling one, or both, of E04DJF and E04DKF prior to a call to E04DGF.

E04DJF reads options from an external options file, with Begin and End as the first and last lines respectively and each intermediate line defining a single optional parameter. For example,

```
Begin
  Print Level = 1
End
```

The call

```
CALL E04DJF(IOPTNS, INFORM)
```

can then be used to read the file on unit IOPTNS. INFORM will be zero on successful exit. E04DJF should be consulted for a full description of this method of supplying optional parameters.

E04DKF can be called to supply options directly, one call being necessary for each optional parameter.

For example,

```
CALL E04DKF('Print level = 1')
```

E04DKF should be consulted for a full description of this method of supplying optional parameters.

All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user

are unaltered by E04DGF (unless they define invalid values) and so remain in effect for subsequent calls to E04DGF, unless altered by the user.

5.1.2. Description of the Optional Parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and real values required with certain options. The number (epsilon) is a generic notation for machine precision, and (ϵ_R) denotes the relative precision of the objective function (the optional parameter Function Precision; see below).

Defaults

This special keyword may be used to reset the default values following a call to E04DGF.

Estimated Optimal Function Value r

(Axiom parameter ES)

This value of r specifies the user-supplied guess of the optimum objective function value. This value is used by E04DGF to calculate an initial step length (see Section 3). If the value of r is not specified by the user (the default), then this has the effect of setting the initial step length to unity. It should be noted that for badly scaled functions a unit step along the steepest descent direction will often compute the function at very large values of x.

Function Precision r Default = $(\epsilon)^{0.9}$

(Axiom parameter FU)

The parameter defines (ϵ_R), which is intended to be a measure of the accuracy with which the problem function F can be computed. The value of (ϵ_R) should reflect the relative precision of $1+|F(x)|$; i.e. (ϵ_R) acts as a relative precision when $|F|$ is large, and as an absolute precision when

```
|F| is small. For example, if F(x) is typically of order 1000 and
the first six significant digits are known to be correct, an
appropriate value for (epsilon)  would be 1.0E-6. In contrast, if
                                R
                               -4
F(x) is typically of order 10   and the first six significant
digits are known to be correct, an appropriate value for
(epsilon)  would be 1.0E-10. The choice of (epsilon)  can be
         R                                            R
quite complicated for badly scaled problems; see Chapter 8 of
Gill and Murray [2], for a discussion of scaling techniques. The
default value is appropriate for most simple functions that are
computed with full accuracy. However when the accuracy of the
computed function values is known to be significantly worse than
full precision, the value of (epsilon)  should be large enough so
                                      R
that E04DGF will not attempt to distinguish between function
values that differ by less than the error inherent in the
calculation. If 0<=r<(epsilon), where (epsilon) is the machine
precision then the default value is used.

Iteration Limit  i Default = max(50,5n)

Iters

Itns

   (Axiom parameter IT)

The value i (i>=0) specifies the maximum number of iterations
allowed before termination. If i<0 the default value is used. See
Section 8 for further information.

Linesearch Tolerance  r Default = 0.9

   (Axiom parameter LIN)

The value r (0<=r<1) controls the accuracy with which the step
(alpha) taken during each iteration approximates a minimum of the
function along the search direction (the smaller the value of r,
the more accurate the linesearch). The default value r=0.9
requests an inaccurate search, and is appropriate for most
problems. A more accurate search may be appropriate when it is
desirable to reduce the number of iterations - for example, if
the objective function is cheap to evaluate.

List     Default =  List
Nolist

   (Axiom parameter LIST)
```

```
Normally each optional parameter specification is printed as it
is supplied. Nolist may be used to suppress the printing and List
may be used to restore printing.
                                     10
Maximum Step Length  r Default = 10

   (Axiom parameter MA)

The value r (r>0) defines the maximum allowable step length for
the line search. If r<=0 the default value is used.

                                           0.8
Optimality Tolerance  r Default = (epsilon)

   (Axiom parameter OP)
                                           R

The parameter r ((epsilon) <=r<1) specifies the accuracy to which
                          R
the user wishes the final iterate to approximate a solution of
the problem. Broadly speaking, r indicates the number of correct
figures desired in the objective function at the solution. For
                  - 6
example, if r is 10    and E04DGF terminates successfully, the
final value of F should have approximately six correct figures.
E04DGF will terminate successfully if the iterative sequence of x
-values is judged to have converged and the final point satisfies
the termination criteria (see Section 3, where (tau)  represents
                                                       F
Optimality Tolerance).

Print Level  i Default = 10

   (Axiom parameter PR)

The value i controls the amount of printout produced by E04DGF.
The following levels of printing are available.

i     Output.

0     No output.

1     The final solution.

5     One line of output for each iteration.

10    The final solution and one line of output for each
      iteration.
```

```
Start Objective Check at Variable  i Default = 1

   (Axiom parameter STA)

Stop Objective Check at Variable  i Default = n

   (Axiom parameter STO)

These keywords take effect only if Verify Level > 0 (see below).
They may be used to control the verification of gradient elements
computed by subroutine OBJFUN. For example if the first 30
variables appear linearly in the objective, so that the
corresponding gradient elements are constant, then it is
reasonable to specify Start Objective Check at Variable  31.

Verify Level  i Default = 0

Verify                          No

Verify Level                    -1

Verify Level                    0

Verify

Verify                          Yes

Verify Objective Gradients

Verify Gradients

Verify Level                    1

   (Axiom parameter VE)

These keywords refer to finite-difference checks on the gradient
elements computed by the user-provided subroutine OBJFUN. It is
possible to set Verify Level in several ways, as indicated above.
For example, the gradients will be verified if Verify, Verify
Yes, Verify Gradients, Verify Objective Gradients or Verify Level
= 1 is specified.

If i<0 then no checking will be performed. If i>0 then the
gradients will be verified at the user-supplied point. If i=0
only a 'cheap' test will be performed, requiring one call to
OBJFUN. If i=1, a more reliable (but more expensive) check will
be made on individual gradient components, within the ranges
specified by the Start and Stop keywords as described above. A
result of the form OK or BAD? is printed by E04DGF to indicate
whether or not each component appears to be correct.
```

5.1.3. Optional parameter checklist and default values

For easy reference, the following sample list shows all valid keywords and their default values. The default options Function Precision and Optimality Tolerance depend upon (epsilon), the machine precision.

Optional Parameters	Default Values
Estimated Optimal Function Value	
Function precision	0.9 (epsilon)
Iterations	max(50,5n)
Linesearch Tolerance	0.9
Maximum Step Length	10 10
List/Nolist	List
Optimality Tolerance	0.8 (epsilon)
Print Level	10
Start Objective Check at Variable	1
Stop Objective Check at Variable	n
Verify Level	0

5.2. Description of Printed Output

The level of printed output from E04DGF is controlled by the user (see the description of Print Level in Section 5.1).

When Print Level >= 5, the following line of output is produced at each iteration.

Itn	is the iteration count.
Step	is the step (alpha) taken along the computed search direction. On reasonably well-behaved

```
                problems, the unit step will be taken as the
                solution is approached.

Nfun            is the cumulated number of evaluations of the
                objective function needed for the linesearch.
                Evaluations needed for the estimation of the
                gradients by finite differences are not included.
                Nfun is printed as a guide to the amount of work
                required for the linesearch. E04DGF will perform
                at most 16 function evaluations per iteration.

Objective       is the value of the objective function.

Norm G          is the Euclidean norm of the gradient of the
                objective function.

Norm X          is the Euclidean norm of x.

Norm (X(k-1)-X(k)) is the Euclidean norm of x   -x .
                                             k-1  k

When Print Level = 1 or Print Level >= 10 then the solution at
the end of execution of E04DGF is printed out.

The following describes the printout for each variable:

Variable        gives the name (VARBL) and index j (j = 1 to n) of
                the variable

Value           is the value of the variable at the final iterate

Gradient Value  is the value of the gradient of the objective
                function with respect to the jth variable at the
                final iterate

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

On exit from E04DGF, IFAIL should be tested. If Print Level > 0
then a short description of IFAIL is printed.

Errors and diagnostics indicated by IFAIL from E04DGF are as
follows:

IFAIL< 0
     A negative value of IFAIL indicates an exit from E04DGF
```

```
     because the user set MODE negative in routine OBJFUN. The
     value of IFAIL will be the same as the user's setting of
     MODE.

IFAIL= 1
     Not used by this routine.

IFAIL= 2
     Not used by this routine.

IFAIL= 3
     The maximum number of iterations has been performed. If the
     algorithm appears to be making progress the iterations value
     may be too small (see Section 5.1.2) so the user should
     increase iterations and rerun E04DGF. If the algorithm seems
     to be 'bogged down',the user should check for incorrect
     gradients or ill-conditioning as described below under IFAIL
     = 6.

IFAIL= 4
     The computed upper bound on the step length taken during the
     linesearch was too small. A rerun with an increased value of
     the Maximum Step Length ((rho) say) may be successful unless
              10
     (rho)>=10   (the default value), in which case the current
     point cannot be improved upon.

IFAIL= 5
     Not used by this routine.

IFAIL= 6
     A sufficient decrease in the function value could not be
     attained during the final linesearch. If the subroutine
     OBJFUN computes the function and gradients correctly, then
     this may occur because an overly stringent accuracy has been
     requested, i.e., Optimality Tolerance is too small or if the
     minimum lies close to a step length of zero. In this case
     the user should apply the four tests described in Section 3
     to determine whether or not the final solution is acceptable
     (the user will need to set Print Level >= 5). For a
     discussion of attainable accuracy see Gill and Murray [2].

     If many iterations have occurred in which essentially no
     progress has been made or E04DGF has failed to move from the
     initial point, subroutine OBJFUN may be incorrect. The user
     should refer to the comments below under IFAIL = 7 and check
     the gradients using the Verify parameter. Unfortunately,
     there may be small errors in the objective gradients that
     cannot be detected by the verification process. Finite-
     difference approximations to first derivatives are
```

```
     catastrophically affected by even small inaccuracies.

IFAIL= 7
     Large errors were found in the derivatives of the objective
     function. This value of IFAIL will occur if the verification
     process indicated that at least one gradient component had
     no correct figures. The user should refer to the printed
     output to determine which elements are suspected to be in
     error.

     As a first step, the user should check that the code for the
     objective values is correct - for example, by computing the
     function at a point where the correct value is known.
     However, care should be taken that the chosen point fully
     tests the evaluation of the function. It is remarkable how
     often the values x=0 or x=1 are used to test function
     evaluation procedures, and how often the special properties
     of these numbers make the test meaningless.

     Special care should be used in this test if computation of
     the objective function involves subsidiary data communicated
     in COMMON storage. Although the first evaluation of the
     function may be correct, subsequent calculations may be in
     error because some of the subsidiary data has accidentally
     been overwritten.

     Errors in programming the function may be quite subtle in
     that the function value is 'almost' correct. For example,
     the function may not be accurate to full precision because
     of the inaccurate calculation of a subsidiary quantity, or
     the limited accuracy of data upon which the function
     depends. A common error on machines where numerical
     calculations are usually performed in double precision is to
     include even one single-precision constant in the
     calculation of the function; since some compilers do not
     convert such constants to double precision, half the correct
     figures may be lost by such a seemingly trivial error.

IFAIL= 8
     The gradient (g) at the starting point is too small. The
            T
     value g g is less than (epsilon) |F(x )|, where (epsilon)
                                      m    o                  m
     is the machine precision.

     The problem should be rerun at a different starting point.

IFAIL= 9
     On entry N < 1.
```

```
7. Accuracy

On successful exit the accuracy of the solution will be as
defined by the optional parameter Optimality Tolerance.

8. Further Comments

Problems whose Hessian matrices at the solution contain sets of
clustered eigenvalues are likely to be minimized in significantly
fewer than n iterations. Problems without this property may
require anything between n and 5n iterations, with approximately
2n iterations being a common figure for moderately difficult
problems.

9. Example

To find a minimum of the function

                        x
                         1   2   2
                    F=e  (4x +2x +4x x +2x +1).
                           1   2   1 2   2

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
  E04 -- Minimizing or Maximizing a Function                E04DJF
          E04DJF -- NAG Foundation Library Routine Document

  Note: Before using this routine, please read the Users' Note for
  your implementation to check implementation-dependent details.
  The symbol (*) after a NAG routine name denotes a routine that is
  not included in the Foundation Library.

  1. Purpose

  To supply optional parameters to E04DGF from an external file.

  2. Specification

         SUBROUTINE E04DJF (IOPTNS, INFORM)
         INTEGER          IOPTNS, INFORM

  3. Description

  E04DJF may be used to supply values for optional parameters to
  E04DGF. E04DJF reads an external file and each line of the file
```

defines a single optional parameter. It is only necessary to supply values for those parameters whose values are to be different from their default values.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

```
Print level = 1
```

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

The file containing the options must start with begin and must finish with end An example of a valid options file is:

```
Begin * Example options file
  Print level = 10
End
```

Normally each line of the file is printed as it is read, on the current advisory message unit (see X04ABF), but printing may be suppressed using the keyword nolist. To suppress printing of begin, nolist must be the first option supplied as in the file:

```
Begin
  Nolist
  Print level = 10
End
```

Printing will automatically be turned on again after a call to E04DGF and may be turned on again at any time by the user by using the keyword list.

```
Optional parameter settings are preserved following a call to
E04DGF, and so the keyword defaults is provided to allow the user
to reset all the optional parameters to their default values
prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations,
synonyms and default values is given in Section 5.1 of the
routine document for E04DGF.

4. References

None.

5. Parameters

 1:  IOPTNS -- INTEGER                                          Input
     On entry: IOPTNS must be the unit number of the options
     file. Constraint: 0 <= IOPTNS <= 99.

 2:  INFORM -- INTEGER                                         Output
     On exit: INFORM will be zero if an options file with the
     correct structure has been read. Otherwise INFORM will be
     positive. Positive values of INFORM indicate that an options
     file may not have been successfully read as follows:
     INFORM = 1
           IOPTNS is not in the range [0,99].

     INFORM = 2
           begin was found, but end-of-file was found before end
           was found.

     INFORM = 3
           end-of-file was found before begin was found.

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning
message is output on the current advisory message unit (see
X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04DKF may also be used to supply optional parameters to E04DGF.

9. Example
```

```
          See the example for E04DGF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          E04 -- Minimizing or Maximizing a Function                E04DKF
                  E04DKF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          To supply individual optional parameters to E04DGF.

          2. Specification

                 SUBROUTINE E04DKF (STRING)
                 CHARACTER*(*)    STRING

          3. Description

          E04DKF may be used to supply values for optional parameters to
          E04DGF. It is only necessary to call E04DKF for those parameters
          whose values are to be different from their default values. One
          call to E04DKF sets one parameter value.

          Each optional parameter is defined by a single character string
          of up to 72 characters, consisting of one or more items. The
          items associated with a given option must be separated by spaces,
          or equal signs (=). Alphabetic characters may be upper or lower
          case. The string

                Print Level = 1

          is an example of a string used to set an optional parameter. For
          each option the string contains one or more of the following
          items:

          (a)   A mandatory keyword.

          (b)   A phrase that qualifies the keyword.

          (c)   A number that specifies an INTEGER or real value. Such
                numbers may be up to 16 contiguous characters in Fortran
                77's I, F, E or D formats, terminated by a space if this is
                not the last item on the line.
```

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist Thus the statement

```
CALL E04DKF ('Nolist')
```

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04DGF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to E04DGF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

```
CALL E04DKF ('Defaults')
```

prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the routine document for E04DGF.

4. References

None.

5. Parameters

1: STRING -- CHARACTER*(*) Input
On entry: STRING must be a single valid option string. See Section 3 above, and Section 5.1 of the routine document for E04DGF.

6. Error Indicators and Warnings

If the parameter STRING is not recognised as a valid option string, then a warning message is output on the current advisory message unit (see X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04DJF may also be used to supply optional parameters to E04DGF.

9. Example

See the example for E04DGF.

%%

E04 -- Minimizing or Maximizing a Function E04FDF
E04FDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04FDF is an easy-to-use algorithm for finding an unconstrained minimum of a sum of squares of m nonlinear functions in n variables (m>=n). No derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```
SUBROUTINE E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
INTEGER          M, N, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION X(N), FSUMSQ, W(LW)
```

3. Description

This routine is essentially identical to the subroutine LSNDN1 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

$$\text{Minimize } F(x) = \sum_{i=1}^{m} [f_i(x)]^2$$

where $x=(x_1, x_2, \ldots, x_n)^T$ and m>=n. (The functions $f_i(x)$ are often referred to as 'residuals'.) The user must supply a subroutine

LSFUN1 to evaluate functions f (x) at any point x.
                              i

From a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of the sum of squares. These points are generated using estimates of the curvature of F(x).

4. References

[1] Gill P E and Murray W (1978) Algorithms for the Solution of the Nonlinear Least-squares Problem. SIAM J. Numer. Anal. 15 977--992.

5. Parameters

1: M -- INTEGER Input

2: N -- INTEGER Input
On entry: the number m of residuals f (x), and the number n
                                      i
of variables, x . Constraint: 1 <= N <= M.
               j

3: X(N) -- DOUBLE PRECISION array Input/Output
On entry: X(j) must be set to a guess at the jth component of the position of the minimum, for j=1,2,...,n. On exit: the lowest point found during the calculations. Thus, if IFAIL = 0 on exit, X(j) is the jth component of the position of the minimum.

4: FSUMSQ -- DOUBLE PRECISION Output
On exit: the value of the sum of squares, F(x), corresponding to the final point stored in X.

5: IW(LIW) -- INTEGER array Workspace

6: LIW -- INTEGER Input
On entry: the length of IW as declared in the (sub)program from which E04FDF has been called. Constraint: LIW >= 1.

7: W(LW) -- DOUBLE PRECISION array Workspace

8: LW -- INTEGER Input
On entry: the length of W as declared in the (sub)program from which E04FDF is called. Constraints:
LW >= N*(7 + N + 2*M + (N-1)/2) + 3*M, if N > 1,

LW >= 9 + 5*>M, if N = 1.

```
 9:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

5.1. Optional Parameters

     LSFUN1 -- SUBROUTINE, supplied by the user.
                                              External Procedure
     This routine must be supplied by the user to calculate the
     vector of values f (x) at any point x. Since the routine is
                      i
     not a parameter to E04FDF, it must be called LSFUN1. It
     should be tested separately before being used in conjunction
     with E04FDF (see the Chapter Introduction).

     Its specification is:

            SUBROUTINE LSFUN1 (M, N, XC, FVECC)
            INTEGER          M, N
            DOUBLE PRECISION XC(N), FVECC(M)

      1:  M -- INTEGER                                        Input

      2:  N -- INTEGER                                        Input
          On entry: the numbers m and n of residuals and
          variables, respectively.

      3:  XC(N) -- DOUBLE PRECISION array                     Input
          On entry: the point x at which the values of the f
                                                               i
          are required.

      4:  FVECC(M) -- DOUBLE PRECISION array                 Output
          On exit: FVECC(i) must contain the value of f  at the
                                                       i
          point x, for i=1,2,...,m.
     LSFUN1 must be declared as EXTERNAL in the (sub)program
     from which E04FDF is called. Parameters denoted as
     Input must not be changed by this procedure.

6. Error Indicators and Warnings
```

```
Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 1,

     or       M < N,

     or       LIW < 1,

     or       LW < N*(7 + N + 2*M + (N-1)/2) + 3*M, when N > 1,

     or       LW < 9 + 5*>M, when N = 1.

IFAIL= 2
     There have been 400*n calls of LSFUN1, yet the algorithm
     does not seem to have converged. This may be due to an
     awkward function or to a poor starting point, so it is worth
     restarting E04FDF from the final point held in X.

IFAIL= 3
     The final point does not satisfy the conditions for
     acceptance as a minimum, but no lower point could be found.

IFAIL= 4
     An auxiliary routine has been unable to complete a singular
     value decomposition in a reasonable number of sub-
     iterations.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
     There is some doubt about whether the point x found by
     E04FDF is a minimum of F(x). The degree of confidence in the
     result decreases as IFAIL increases. Thus when IFAIL = 5, it
     is probable that the final x gives a good estimate of the
     position of a minimum, but when IFAIL = 8 it is very
     unlikely that the routine has found a minimum.

If the user is not satisfied with the result (e.g. because IFAIL
lies between 3 and 8), it is worth restarting the calculations
from a different starting point (not the point at which the
failure occurred) in order to avoid the region which caused the
```

failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made, then, for a computer with a mantissa of t decimals, one would expect to get about t/2-1 decimals accuracy in the components of x and between t-1 (if F(x) is of order 1 at the minimum) and 2t-2 (if F(x) is close to zero at the minimum) decimals accuracy in F(x).

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the distance of the starting point from the solution. The number of multiplications performed per iteration of E04FDF varies, but for $m \gg n$ is approximately $n*m + O(n^2)$. In addition, each iteration makes at least n+1 calls of LSFUN1. So, unless the residuals can be evaluated very quickly, the run time will be dominated by the time spent in LSFUN1.

Ideally, the problem should be scaled so that the minimum value of the sum of squares is in the range (0,1), and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04FDF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array W. See E04YCF for further details.

9. Example

To find least-squares estimates of x_1, x_2 and x_3 in the model

$$y = x_1 + \frac{t_1}{x_2 t_1 + x_3 t_1}$$

$$2\ 2 \quad 3\ 3$$

using the 15 sets of data given in the following table.

y	t 1	t 2	t 3
0.14	1.0	15.0	1.0
0.18	2.0	14.0	2.0
0.22	3.0	13.0	3.0
0.25	4.0	12.0	4.0
0.29	5.0	11.0	5.0
0.32	6.0	10.0	6.0
0.35	7.0	9.0	7.0
0.39	8.0	8.0	8.0
0.37	9.0	7.0	7.0
0.58	10.0	6.0	6.0
0.73	11.0	5.0	5.0
0.96	12.0	4.0	4.0
1.34	13.0	3.0	3.0
2.10	14.0	2.0	2.0
4.39	15.0	1.0	1.0

The program uses (0.5, 1.0, 1.5) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E04 -- Minimizing or Maximizing a Function E04GCF
E04GCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04GCF is an easy-to-use quasi-Newton algorithm for finding an unconstrained minimum of a sum of squares of m nonlinear functions in n variables (m>=n). First derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```
      SUBROUTINE E04GCF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
      INTEGER          M, N, IW(LIW), LIW, LW, IFAIL
      DOUBLE PRECISION X(N), FSUMSQ, W(LW)
```

3. Description

This routine is essentially identical to the subroutine LSFDQ2 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

$$\text{Minimize} \quad F(x) = \sum_{i=1}^{m} [f_i(x)]^2$$

where $x=(x_1, x_2, \ldots, x_n)^T$ and $m \geq n$. (The functions $f_i(x)$ are often referred to as 'residuals'.) The user must supply a subroutine LSFUN2 to evaluate the residuals and their first derivatives at any point x.

Before attempting to minimize the sum of squares, the algorithm checks LSFUN2 for consistency. Then, from a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of the sum of squares. These points are generated using estimates of the curvature of F(x).

4. References

[1] Gill P E and Murray W (1978) Algorithms for the Solution of the Nonlinear Least-squares Problem. SIAM J. Numer. Anal. 15 977--992.

5. Parameters

1: M -- INTEGER Input

2: N -- INTEGER Input
On entry: the number m of residuals $f_i(x)$, and the number n of variables, x_j. Constraint: $1 \leq N \leq M$.

3: X(N) -- DOUBLE PRECISION array Input/Output
On entry: X(j) must be set to a guess at the jth component of the position of the minimum, for j=1,2,...,n. The routine

checks the first derivatives calculated by LSFUN2 at the starting point, and so is more likely to detect an error in the user's routine if the initial X(j) are non-zero and mutually distinct. On exit: the lowest point found during the calculations. Thus, if IFAIL = 0 on exit, X(j) is the j th component of the position of the minimum.

4: FSUMSQ -- DOUBLE PRECISION Output
On exit: the value of the sum of squares, F(x), corresponding to the final point stored in X.

5: IW(LIW) -- INTEGER array Workspace

6: LIW -- INTEGER Input
On entry: the length of IW as declared in the (sub)program from which E04GCF is called. Constraint: LIW >= 1.

7: W(LW) -- DOUBLE PRECISION array Workspace

8: LW -- INTEGER Input
On entry: the length of W as declared in the (sub)program from which E04GCF is called. Constraints:
LW >= 2*N*(4 + N + M) + 3*M, if N > 1,

LW >= 11 + 5*M, if N = 1.

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

5.1. Optional Parameters

LSFUN2 -- SUBROUTINE, supplied by the user.
External Procedure
This routine must be supplied by the user to calculate the vector of values $f_i(x)$ and the Jacobian matrix of first derivatives $\frac{ddf_i}{ddx}$ at any point x. Since the routine is not a

```
                j
parameter to E04GCF, it must be called LSFUN2. It should be
tested separately before being used in conjunction with
E04GCF (see the Chapter Introduction).

Its specification is:

      SUBROUTINE LSFUN2 (M, N, XC, FVECC, FJACC, LJC)
      INTEGER          M, N, LJC
      DOUBLE PRECISION XC(N), FVECC(M), FJACC(LJC,N)
Important: The dimension declaration for FJACC must
contain the variable LJC, not an integer constant.

 1:  M -- INTEGER                                        Input

 2:  N -- INTEGER                                        Input
     On entry: the numbers m and n of residuals and
     variables, respectively.

 3:  XC(N) -- DOUBLE PRECISION array                     Input
     On entry: the point x at which the values of the f
                                                           i
              ddf
                 i
     and the  ---- are required.
              ddx
                 j

 4:  FVECC(M) -- DOUBLE PRECISION array                 Output
     On exit: FVECC(i) must contain the value of f  at the
                                                   i
     point x, for i=1,2,...,m.

 5:  FJACC(LJC,N) -- DOUBLE PRECISION array              Output
                                                        ddf
                                                           i
     On exit: FJACC(i,j) must contain the value of  ---- at
                                                        ddx
                                                           j
     the point x, for i=1,2,...,m; j=1,2,...,n.

 6:  LJC -- INTEGER                                      Input
     On entry: the first dimension of the array FJACC.
LSFUN2 must be declared as EXTERNAL in the (sub)program
from which E04GCF is called. Parameters denoted as
Input must not be changed by this procedure.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:
```

```
If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry N < 1,

     or       M < N,

     or       LIW < 1,

     or       LW < 2*N*(4 + N + M) + 3*M, when N > 1,

     or       LW < 9 + 5*>M, when N = 1.

IFAIL= 2
     There have been 50*n calls of LSFUN2, yet the algorithm does
     not seem to have converged. This may be due to an awkward
     function or to a poor starting point, so it is worth
     restarting E04GCF from the final point held in X.

IFAIL= 3
     The final point does not satisfy the conditions for
     acceptance as a minimum, but no lower point could be found.

IFAIL= 4
     An auxiliary routine has been unable to complete a singular
     value decomposition in a reasonable number of sub-
     iterations.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
     There is some doubt about whether the point X found by
     E04GCF is a minimum of F(x). The degree of confidence in the
     result decreases as IFAIL increases. Thus, when IFAIL = 5,
     it is probable that the final x gives a good estimate of the
     position of a minimum, but when IFAIL = 8 it is very
     unlikely that the routine has found a minimum.

IFAIL= 9
     It is very likely that the user has made an error in forming
                     ddf
                        i
     the derivatives  ---- in LSFUN2.
                     ddx
```

j

If the user is not satisfied with the result (e.g. because IFAIL lies between 3 and 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made then, for a computer with a mantissa of t decimals, one would expect to get t/2-1 decimals accuracy in the components of x and between t-1 (if F(x) is of order 1 at the minimum) and 2t-2 (if F(x) is close to zero at the minimum) decimals accuracy in F(x).

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the distance of the starting point from the solution. The number of multiplications performed per iteration of E04GCF varies, but for

$$2 \quad 3$$

m>>n is approximately n*m +O(n). In addition, each iteration makes at least one call of LSFUN2. So, unless the residuals and their derivatives can be evaluated very quickly, the run time will be dominated by the time spent in LSFUN2.

Ideally the problem should be scaled so that the minimum value of the sum of squares is in the range (0,1) and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04GCF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array W. See E04YCF for further details.

9. Example

To find the least-squares estimates of x , x and x in the model

```
                                   1   2      3

                                        t
                                         1
                                y=x + ---------
                                   1  x t +x t
                                       2 2  3 3
```

using the 15 sets of data given in the following table.

```
      y    t   t    t
           1   2    3
   0.14  1.0 15.0 1.0
   0.18  2.0 14.0 2.0
   0.22  3.0 13.0 3.0
   0.25  4.0 12.0 4.0
   0.29  5.0 11.0 5.0
   0.32  6.0 10.0 6.0
   0.35  7.0  9.0 7.0
   0.39  8.0  8.0 8.0
   0.37  9.0  7.0 7.0
   0.58 10.0  6.0 6.0
   0.73 11.0  5.0 5.0
   0.96 12.0  4.0 4.0
   1.34 13.0  3.0 3.0
   2.10 14.0  2.0 2.0
   4.39 15.0  1.0 1.0
```

The program uses (0.5, 1.0, 1.5) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E04 -- Minimizing or Maximizing a Function E04JAF
E04JAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04JAF is an easy-to-use quasi-Newton algorithm for finding a minimum of a function F(x ,x ,...,x), subject to fixed upper and
                                                1 2   n

lower bounds of the independent variables x ,x ,...,x , using
1 2 n
function values only.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```
SUBROUTINE E04JAF (N, IBOUND, BL, BU, X, F, IW, LIW, W,
1                  LW, IFAIL)
INTEGER          N, IBOUND, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION BL(N), BU(N), X(N), F, W(LW)
```

3. Description

This routine is applicable to problems of the form:

Minimize F(x ,x ,...,x) subject to l <=x <=u , j=1,2,...,n
1 2 n j j j

when derivatives of F(x) are unavailable.

Special provision is made for problems which actually have no bounds on the x , problems which have only non-negativity bounds
j
and problems in which l =l =...=l and u =u =...=u . The user
1 2 n 1 2 n
must supply a subroutine FUNCT1 to calculate the value of F(x) at any point x.

From a starting point supplied by the user there is generated, on the basis of estimates of the gradient and the curvature of F(x), a sequence of feasible points which is intended to converge to a local minimum of the constrained function. An attempt is made to verify that the final point is a minimum.

4. References

[1] Gill P E and Murray W (1976) Minimization subject to bounds on the variables. Report NAC 72. National Physical Laboratory.

5. Parameters

1: N -- INTEGER Input
On entry: the number n of independent variables.
Constraint: N >= 1.

2: IBOUND -- INTEGER Input
On entry: indicates whether the facility for dealing with bounds of special forms is to be used.

It must be set to one of the following values:
IBOUND = 0
if the user will be supplying all the l_j and u_j individually.

IBOUND = 1
if there are no bounds on any x_j.

IBOUND = 2
if all the bounds are of the form $0 \leq x_j$.

IBOUND = 3
if $l_1 = l_2 = \ldots = l_n$ and $u_1 = u_2 = \ldots = u_n$.

3: BL(N) -- DOUBLE PRECISION array Input/Output
On entry: the lower bounds l_j.

If IBOUND is set to 0, the user must set BL(j) to l_j, for $j=1,2,\ldots,n$. (If a lower bound is not specified for a particular x_j, the corresponding BL(j) should be set to -10.)

If IBOUND is set to 3, the user must set BL(1) to l_1; E04JAF will then set the remaining elements of BL equal to BL(1).
On exit: the lower bounds actually used by E04JAF.

4: BU(N) -- DOUBLE PRECISION array Input/Output
On entry: the upper bounds u_j.

If IBOUND is set to 0, the user must set BU(j) to u_j, for $j=1,2,\ldots,n$. (If an upper bound is not specified for a particular x_j, the corresponding BU(j) should be set to 10.)

If IBOUND is set to 3, the user must set BU(1) to u_1; E04JAF

```
                                                          1
     will then set the remaining elements of BU equal to BU(1).
     On exit: the upper bounds actually used by E04JAF.

  5: X(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: X(j) must be set to an estimate of the jth
     component of the position of the minimum, for j=1,2,...,n.
     On exit: the lowest point found during the calculations.
     Thus, if IFAIL = 0 on exit, X(j) is the jth component of the
     position of the minimum.

  6: F -- DOUBLE PRECISION                                Output
     On exit: the value of F(x) corresponding to the final point
     stored in X.

  7: IW(LIW) -- INTEGER array                          Workspace

  8: LIW -- INTEGER                                       Input
     On entry: the length of IW as declared in the (sub)program
     from which E04JAF is called. Constraint: LIW >= N + 2.

  9: W(LW) -- DOUBLE PRECISION array                   Workspace

 10: LW -- INTEGER                                        Input
     On entry: the length of W as declared in the (sub)program
     from which E04JAF is called. Constraint: LW>=max(N*(N-
     1)/2+12*N,13).

 11: IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

5.1. Optional Parameters

     FUNCT1 -- SUBROUTINE, supplied by the user.
                                          External Procedure
     This routine must be supplied by the user to calculate the
     value of the function F(x) at any point x. Since this
     routine is not a parameter to E04JAF, it must be called
```

```
     FUNCT1. It should be tested separately before being used in
     conjunction with E04JAF (see the Chapter Introduction).

     Its specification is:

            SUBROUTINE FUNCT1 (N, XC, FC)
            INTEGER          N
            DOUBLE PRECISION XC(N), FC

      1:  N -- INTEGER                                          Input
          On entry: the number n of variables.

      2:  XC(N) -- DOUBLE PRECISION array                       Input
          On entry: the point x at which the function value is
          required.

      3:  FC -- DOUBLE PRECISION                               Output
          On exit: the value of the function F at the current
          point x.
     FUNCT1 must be declared as EXTERNAL in the (sub)program
     from which E04JAF is called. Parameters denoted as
     Input must not be changed by this procedure.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
     On entry N < 1,

     or       IBOUND < 0,

     or       IBOUND > 3,

     or       IBOUND = 0 and BL(j) > BU(j) for some j,

     or       IBOUND = 3 and BL(1) > BU(1),

     or       LIW < N + 2,

     or       LW<max(13,12*N+N*(N-1)/2).

IFAIL= 2
     There have been 400*n function evaluations, yet the
     algorithm does not seem to be converging. The calculations
     can be restarted from the final point held in X. The error
     may also indicate that F(x) has no minimum.

IFAIL= 3
     The conditions for a minimum have not all been met but a
```

```
     lower point could not be found and the algorithm has failed.

IFAIL= 4
     An overflow has occurred during the computation. This is an
     unlikely failure, but if it occurs the user should restart
     at the latest point given in X.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
     There is some doubt about whether the point x found by
     E04JAF is a minimum. The degree of confidence in the result
     decreases as IFAIL increases. Thus, when IFAIL = 5 it is
     probable that the final x gives a good estimate of the
     position of a minimum, but when IFAIL = 8 it is very
     unlikely that the routine has found a minimum.

IFAIL= 9
     In the search for a minimum, the modulus of one of the
                                          6
     variables has become very large  (~10 ).  This indicates
     that there is a mistake in FUNCT1, that the user's problem
     has no finite solution, or that the problem needs rescaling
     (see Section 8).

If the user is dissatisfied with the result (e.g. because IFAIL =
5, 6, 7 or 8), it is worth restarting the calculations from a
different starting point (not the point at which the failure
occurred) in order to avoid the region which caused the failure.
If persistent trouble occurs and the gradient can be calculated,
it may be advisable to change to a routine which uses gradients
(see the Chapter Introduction).

7. Accuracy

When a successful exit is made then, for a computer with a
mantissa of t decimals, one would expect to get about t/2-1
decimals accuracy in x and about t-1 decimals accuracy in F,
provided the problem is reasonably well scaled.

8. Further Comments

The number of iterations required depends on the number of
variables, the behaviour of F(x) and the distance of the starting
point from the solution. The number of operations performed in an
                                                    2
```

iteration of E04JAF is roughly proportional to n . In addition, each iteration makes at least m+1 calls of FUNCT1, where m is the number of variables not fixed on bounds. So, unless F(x) can be evaluated very quickly, the run time will be dominated by the time spent in FUNCT1.

Ideally the problem should be scaled so that at the solution the value of F(x) and the corresponding values of $x_1, x_2, \ldots, x_n$ are each in the range (-1,+1), and so that at points a unit distance away from the solution, F is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04JAF will take less computer time.

9. Example

To minimize

$$F=(x_1+10x_2)^2+5(x_3-x_4)^2+(x_2-2x_3)^4+10(x_1-x_4)^4$$

subject to

$$1 \leq x_1 \leq 3$$

$$-2 \leq x_2 \leq 0$$

$$1 \leq x_4 \leq 3,$$

starting from the initial guess (3, - 1, 0, 1).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E04 -- Minimizing or Maximizing a Function E04MBF
E04MBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is

not included in the Foundation Library.

1. Purpose

E04MBF is an easy-to-use routine for solving linear programming problems, or for finding a feasible point for such problems. It is not intended for large sparse problems.

2. Specification

```
 SUBROUTINE E04MBF (ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1                   A, BL, BU, CVEC, LINOBJ, X, ISTATE,
2                   OBJLP, CLAMDA, IWORK, LIWORK, WORK,
3                   LWORK, IFAIL)
 INTEGER           ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1                  ISTATE(NCTOTL), IWORK(LIWORK), LIWORK,
2                  LWORK, IFAIL
 DOUBLE PRECISION A(NROWA,N), BL(NCTOTL), BU(NCTOTL), CVEC
1                  (N), X(N), OBJLP, CLAMDA(NCTOTL), WORK
2                  (LWORK)
 LOGICAL           LINOBJ
```

3. Description

E04MBF solves linear programming (LP) problems of the form

$$\text{Minimize}_{x \text{ is in } R^n} \; c^T x \quad \text{subject to} \quad l \le \begin{pmatrix} x \\ Ax \end{pmatrix} \le u \qquad \text{(LP)}$$

where c is an n element vector and A is an m by n matrix i.e., there are n variables and m general linear constraints. m may be zero in which case the LP problem is subject only to bounds on the variables. Notice that upper and lower bounds are specified for all the variables and constraints. This form allows full generality in specifying other types of constraints. For example the ith constraint may be specified as equality by setting $l_i = u_i$. If certain bounds are not present the associated elements of l or u can be set to special values that will be treated as -infty or +infty.

The routine allows the linear objective function to be omitted in which case a feasible point for the set of constraints is sought.

The user must supply an initial estimate of the solution.

Users who wish to exercise additional control and users with problems whose solution would benefit from additional flexibility

```
should consider using the comprehensive routine E04NAF.

4. References

[1]   Gill P E, Murray W and Wright M H (1981) Practical
      Optimization. Academic Press.

[2]   Gill P E, Murray W, Saunders M A and Wright M H (1983)
      User's Guide for SOL/QPSOL. Report SOL 83-7. Department of
      Operations Research, Stanford University.

5. Parameters

 1:  ITMAX -- INTEGER                                          Input
     On entry: an upper bound on the number of iterations to be
     taken. If ITMAX is not positive, then the value 50 is used
     in place of ITMAX.

 2:  MSGLVL -- INTEGER                                         Input
     On entry: indicates whether or not printout is required at
     the final solution. When printing occurs the output is on
     the advisory message channel (see X04ABF). A description of
     the printed output is given in Section 5.1. The level of
     printing is determined as follows:
     MSGLVL < 0
           No printing.

     MSGLVL = 0
           Printing only if an input parameter is incorrect, or
           if the problem is so ill-conditioned that subsequent
           overflow is likely. This setting is strongly
           recommended in preference to MSGLVL < 0.

     MSGLVL = 1
           Printing at the solution.

     MSGLVL > 1
           Values greater than 1 should normally be used only at
           the direction of NAG; such values may generate large
           amounts of printed output.

 3:  N -- INTEGER                                              Input
     On entry: the number n of variables. Constraint: N >= 1.

 4:  NCLIN -- INTEGER                                          Input
     On entry: the number of general linear constraints in the
     problem. Constraint: NCLIN >= 0.

 5:  NCTOTL -- INTEGER                                         Input
     On entry: the value (N+NCLIN).
```

```
 6: NROWA -- INTEGER                                          Input
    On entry:
    the first dimension of the array A as declared in the
    (sub)program from which E04MBF is called.
    Constraint: NROWA >= max(1,NCLIN).

 7: A(NROWA,N) -- DOUBLE PRECISION array                      Input
    On entry: the leading NCLIN by n part of A must contain the
    NCLIN general constraints, with the coefficients of the ith
    constraint in the ith row of A. If NCLIN = 0, then A is not
    referenced.

 8: BL(NCTOTL) -- DOUBLE PRECISION array                      Input
    On entry: the first n elements of BL must contain the lower
    bounds on the n variables, and when NCLIN > 0, the next
    NCLIN elements of BL must contain the lower bounds on the
    NCLIN general linear constraints. To specify a non-existent
    lower bound (l =-infty), set BL(j)<=-1.0E+20.
                 j

 9: BU(NCTOTL) -- DOUBLE PRECISION array                      Input
    On entry: the first n elements of BU must contain the upper
    bounds on the n variables, and when NCLIN > 0, the next
    NCLIN elements of BU must contain the upper bounds on the
    NCLIN general linear constraints. To specify a non-existent
    upper bound (u =+infty), set BU(j)>=1.0E+20. Constraint:
                 j
    BL(j)<=BU(j), for j=1,2,...,NCTOTL.

10: CVEC(N) -- DOUBLE PRECISION array                         Input
    On entry: with LINOBJ = .TRUE., CVEC must contain the
    coefficients of the objective function. If LINOBJ = .FALSE.,
    then CVEC is not referenced.

11: LINOBJ -- LOGICAL                                         Input
    On entry: indicates whether or not a linear objective
    function is present. If LINOBJ = .TRUE., then the full LP
    problem is solved, but if LINOBJ = .FALSE., only a feasible
    point is found and the array CVEC is not referenced.

12: X(N) -- DOUBLE PRECISION array                    Input/Output
    On entry: an estimate of the solution, or of a feasible
    point. Even when LINOBJ = .TRUE. it is not necessary for the
    point supplied in X to be feasible. In the absence of better
    information all elements of X may be set to zero. On exit:
    the solution to the LP problem when LINOBJ = .TRUE., or a
    feasible point when LINOBJ = .FALSE..

    When no feasible point exists (see IFAIL = 1 in Section 6)
```

then X contains the point for which the sum of the infeasibilities is a minimum. On return with IFAIL = 2, 3 or 4, X contains the point at which E04MBF terminated.

13: ISTATE(NCTOTL) -- INTEGER array Output
On exit: with IFAIL < 5, ISTATE indicates the status of every constraint at the final point. The first n elements of ISTATE refer to the upper and lower bounds on the variables and when NCLIN > 0 the next NCLIN elements refer to the general constraints.

Their meaning is:
ISTATE(j) Meaning

-2 The constraint violates its lower bound. This value cannot occur for any element of ISTATE when a feasible point has been found.

-1 The constraint violates its upper bound. This value cannot occur for any element of ISTATE when a feasible point has been found.

0 The constraint is not in the working set (is not active) at the final point. Usually this means that the constraint lies strictly between its bounds.

1 This inequality constraint is in the working set (is active) at its lower bound.

2 This inequality constraint is in the working set (is active) at its upper bound.

3 This constraint is included in the working set (is active) as an equality. This value can only occur when BL(j) = BU(j).

14: OBJLP -- DOUBLE PRECISION Output
On exit: when LINOBJ = .TRUE., then on successful exit, OBJLP contains the value of the objective function at the solution, and on exit with IFAIL = 2, 3 or 4, OBJLP contains the value of the objective function at the point returned in X.

When LINOBJ = .FALSE., then on successful exit OBJLP will be zero and on return with IFAIL = 1, OBJLP contains the minimum sum of the infeasibilities corresponding to the point returned in X.

15: CLAMDA(NCTOTL) -- DOUBLE PRECISION array Output

```
     On exit: when LINOBJ = .TRUE., then on successful exit, or
     on exit with IFAIL = 2, 3, or 4, CLAMDA contains the
     Lagrange multipliers (reduced costs) for each constraint
     with respect to the working set. The first n components of
     CLAMDA contain the multipliers for the bound constraints on
     the variables and the remaining NCLIN components contain the
     multipliers for the general linear constraints.

     If ISTATE(j) = 0 so that the jth constraint is not in the
     working set then CLAMDA(j) is zero. If X is optimal and
     ISTATE(j) = 1, then CLAMDA(j) should be non-negative, and if
     ISTATE(j) = 2, then CLAMDA(j) should be non-positive.

     When LINOBJ = .FALSE., all NCTOTL elements of CLAMDA are
     returned as zero.

16:  IWORK(LIWORK) -- INTEGER array                      Workspace

17:  LIWORK -- INTEGER                                       Input
     On entry: the length of the array IWORK as declared in the
     (sub)program from which E04MBF is called. Constraint:
     LIWORK>=2*N.

18:  WORK(LWORK) -- DOUBLE PRECISION array               Workspace

19:  LWORK -- INTEGER                                        Input
     On entry: the length of the array WORK as declared in the
     (sub)program from which E04MBF is called. Constraints:
     when N <= NCLIN then
                    2
           LWORK>=2*N +6*N+4*NCLIN+NROWA;

     when 0 <= NCLIN < N then
                           2
           LWORK>=2*(NCLIN+1) +4*NCLIN+6*N+NROWA.

20:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.
```

```
5.1. Description of the Printed Output

When MSGLVL = 1, then E04MBF will produce output on the advisory
message channel (see X04ABF ), giving information on the final
point. The following describes the printout associated with each
variable.

Output         Meaning

VARBL          The name (V) and index j, for j=1,2,...,n, of the
               variable.

STATE          The state of the variable. (FR if neither bound is
               in the working set, EQ for a fixed variable, LL if
               on its lower bound, UL if on its upper bound and
               TB if held on a temporary bound.) If the value of
               the variable lies outside the upper or lower bound
               then STATE will be ++ or -- respectively.

VALUE          The value of the variable at the final iteration.

LOWER BOUND    The lower bound specified for the variable.

UPPER BOUND    The upper bound specified for the variable.

LAGR MULT      The value of the Lagrange multiplier for the
               associated bound.

RESIDUAL       The difference between the value of the variable
               and the nearer of its bounds.

For each of the general constraints the printout is as above with
refers to the jth element of Ax, except that VARBL is replaced
by:

LNCON          The name (L) and index j, for j = 1,2,...,NCLIN of
               the constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

Note: when MSGLVL=1 a short description of the error is printed.

IFAIL= 1
     No feasible point could be found. Moving violated
     constraints so that they are satisfied at the point returned
     in X gives the minimum moves necessary to make the LP
     problem feasible.
```

```
IFAIL= 2
     The solution to the LP problem is unbounded.

IFAIL= 3
     A total of 50 changes were made to the working set without
     altering x. Cycling is probably occurring. The user should
     consider using E04NAF with MSGLVL >= 5 to monitor constraint
     additions and deletions in order to determine whether or not
     cycling is taking place.

IFAIL= 4
     The limit on the number of iterations has been reached.
     Increase ITMAX or consider using E04NAF to monitor progress.

IFAIL= 5
     An input parameter is invalid. Unless MSGLVL < 0 a message
     will be printed.

IFAILOverflow
     If the printed output before the overflow occurred contains
     a warning about serious ill-conditioning in the working set
     when adding the jth constraint, then either the user should
     try using E04NAF and experiment with the magnitude of FEATOL
     (j) in that routine, or the offending linearly dependent
     constraint (with index j) should be removed from the
     problem.

7. Accuracy

The routine implements a numerically stable active set strategy
and returns solutions that are as accurate as the condition of
the LP problem warrants on the machine.

8. Further Comments

The time taken by each iteration is approximately proportional to
     2      2
min(n ,NCLIN ).

Sensible scaling of the problem is likely to reduce the number of
iterations required and make the problem less sensitive to
perturbations in the data, thus improving the condition of the LP
problem. In the absence of better information it is usually
sensible to make the Euclidean lengths of each constraint of
comparable magnitude. See Gill et al [1] for further information
and advice.

Note that the routine allows constraints to be violated by an
absolute tolerance equal to the machine precision (see X02AJF(*))
```

```
9. Example

To minimize the function

          -0.02x -0.2x -0.2x -0.2x -0.2x +0.04x +0.04x
                1     2     3     4     5      6      7

subject to the bounds

                      -0.01 <= x  <= 0.01
                                1
                      -0.1 <= x  <= 0.15,
                               2
                      -0.01 <= x  <= 0.03,
                                3
                      -0.04 <= x  <= 0.02,
                                4
                      -0.1 <= x  <= 0.05,
                               5
                      -0.01 <= x
                                6
                      -0.01 <= x
                                7

and the general constraints

                    x +x +x +x +x +x +x =-0.13
                     1  2  3  4  5  6  7

    0.15x +0.04x +0.02x +0.04x +0.02x +0.01x +0.03x <=-0.0049
         1      2      3      4      5      6      7

       0.03x +0.05x +0.08x +0.02x +0.06x +0.01x <=-0.0064
            1      2      3      4      5      6

          0.02x +0.04x +0.01x +0.02x +0.02x <=-0.0037
               1      2      3      4      5

                 0.02x +0.03x +0.01x <=-0.0012
                      1      2      5

       -0.0992<=0.70x +0.75x +0.80x +0.75x +0.80x +0.97x
                     1      2      3      4      5      6

 -0.003<=0.02x +0.06x +0.08x +0.12x +0.02x +0.01x +0.97x <=0.002
             1      2      3      4      5      6      7

The initial point, which is infeasible, is
```

```
                                                          T
        x =(-0.01, -0.03, 0.0, -0.01, -0.1, 0.02, 0.01) .
         0
```

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

E04 -- Minimizing or Maximizing a Function E04NAF
E04NAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04NAF is a comprehensive routine for solving quadratic programming (QP) or linear programming (LP) problems. It is not intended for large sparse problems.

2. Specification

```
      SUBROUTINE E04NAF (ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
     1                   NROWH, NCOLH, BIGBND, A, BL, BU, CVEC,
     2                   FEATOL, HESS, QPHESS, COLD, LP, ORTHOG,
     3                   X, ISTATE, ITER, OBJ, CLAMDA, IWORK,
     4                   LIWORK, WORK, LWORK, IFAIL)
      INTEGER           ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
     1                  NROWH, NCOLH, ISTATE(NCTOTL), ITER, IWORK
     2                  (LIWORK), LIWORK, LWORK, IFAIL
      DOUBLE PRECISION BIGBND, A(NROWA,N), BL(NCTOTL),
     1                  BU(NCTOTL), CVEC(N), FEATOL(NCTOTL), HESS
     2                  (NROWH,NCOLH), X(N), OBJ, CLAMDA(NCTOTL),
     3                  WORK(LWORK)
      LOGICAL           COLD, LP, ORTHOG
      EXTERNAL          QPHESS
```

3. Description

E04NAF is essentially identical to the subroutine SOL/QPSOL described in Gill et al [1].

E04NAF is designed to solve the quadratic programming (QP) problem - the minimization of a quadratic function subject to a set of linear constraints on the variables. The problem is assumed to be stated in the following form:

$$\text{Minimize} \quad c^T x + \frac{1}{2} x^T H x \quad \text{subject to} \quad l \leq \begin{pmatrix} x \\ Ax \end{pmatrix} \leq u \ , \tag{1}$$

where c is a constant n-vector and H is a constant n by n symmetric matrix; note that H is the Hessian matrix (matrix of second partial derivatives) of the quadratic objective function. The matrix A is m by n, where m may be zero; A is treated as a dense matrix.

The constraints involving A will be called the general constraints. Note that upper and lower bounds are specified for all the variables and for all the general constraints. The form of (1) allows full generality in specifying other types of constraints. In particular, an equality constraint is specified by setting $l_i = u_i$. If certain bounds are not present, the associated elements of l or u can be set to special values that will be treated as -infty or +infty.

The user must supply an initial estimate of the solution to (1), and a subroutine that computes the product Hx for any given vector x. If H is positive-definite or positive semi-definite, E04NAF will obtain a global minimum; otherwise, the solution obtained will be a local minimum (which may or may not be a global minimum). If H is defined as the zero matrix, E04NAF will solve the resulting linear programming (LP) problem; however, this can be accomplished more efficiently by setting a logical variable in the call of the routine (see the parameter LP in Section 5).

E04NAF allows the user to provide the indices of the constraints that are believed to be exactly satisfied at the solution. This facility, known as a warm start, can lead to significant savings in computational effort when solving a sequence of related problems.

The method has two distinct phases. In the first (the LP phase), an iterative procedure is carried out to determine a feasible point. In this context, feasibility is defined by a user-provided array FEATOL; the jth constraint is considered satisfied if its violation does not exceed FEATOL(j). The second phase (the QP phase) generates a sequence of feasible iterates in order to minimize the quadratic objective function. In both phases, a subset of the constraints - called the working set - is used to define the search direction at each iteration; typically, the working set includes constraints that are satisfied to within the corresponding tolerances in the FEATOL array.

We now briefly describe a typical iteration in the QP phase. Let
x_k denote the estimate of the solution at the kth iteration; the
next iterate is defined by

$$x_{k+1} = x_k + (\alpha)_k p_k$$

where p_k is an n-dimensional search direction and $(\alpha)_k$ is a
scalar step length. Assume that the working (active) set contains
t_k linearly independent constraints, and let C_k denote the matrix
of coefficients of the bounds and general constraints in the
current working set.

Let Z_k denote a matrix whose columns form a basis for the null
space of C_k, so that $C_k Z_k = 0$. (Note that Z_k has n_z columns, where
$n_z = n - t_k$.) The vector $Z_k^T (c + Hx_k)$ is called the projected gradient
at x_k. If the projected gradient is zero at x_k (i.e., x_k is a
constrained stationary point in the subspace defined by Z_k),
Lagrange multipliers $(\lambda)_k$ are defined as the solution of the
compatible overdetermined system

$$C_k^T (\lambda)_k = c + Hx_k \qquad (2)$$

The Lagrange multiplier $(\lambda)_k$ corresponding to an inequality
constraint in the working set is said to be optimal if
$(\lambda)_k \leq 0$ when the associated constraint is at its upper bound,
or if $(\lambda)_k \geq 0$ when the associated constraint is at its lower
bound. If a multiplier is non-optimal, the objective function can
be reduced by deleting the corresponding constraint (with index
JDEL, see Section 5.1) from the working set.

If the projected gradient at x_k is non-zero, the search direction
p_k is defined as

$$p_k = Z_k p_z \qquad (3)$$

where p_z is an n$_z$-vector. In effect, the constraints in the working set are treated as equalities, by constraining p_k to lie within the subspace of vectors orthogonal to the rows of C_k. This definition ensures that $C_k p_k = 0$, and hence the values of the constraints in the working set are not altered by any move along p_k.

The vector p_z is obtained by solving the equations

$$Z_k^T H Z_k p_z = -Z_k^T (c + H x_k) \tag{4}$$

(The matrix $Z_k^T H Z_k$ is called the projected Hessian matrix.) If the projected Hessian is positive-definite, the vector defined by (3) and (4) is the step to the minimum of the quadratic function in the subspace defined by Z_k.

If the projected Hessian is positive-definite and $x_k + p_k$ is feasible, α_k will be taken as unity. In this case, the projected gradient at x_{k+1} will be zero (see NORM ZTG in Section 5.1), and Lagrange multipliers can be computed (see Gill et al [2]). Otherwise, α_k is set to the step to the 'nearest' constraint (with index JADD, see Section 5.1), which is added to the working set at the next iteration.

The matrix Z_k is obtained from the TQ factorization of C_k, in which C_k is represented as

$$C_k Q = (0 \; T_k) \tag{5}$$

where T_k is reverse-triangular. It follows from (5) that Z_k may

be taken as the first n_z columns of Q. If the projected Hessian is positive-definite, (3) is solved using the Cholesky factorization

$$Z_k^T H Z_k = R_k^T R_k$$

where R_k is upper triangular. These factorizations are updated as constraints enter or leave the working set (see Gill et al [2] for further details).

An important feature of E04NAF is the treatment of indefiniteness in the projected Hessian. If the projected Hessian is positive-definite, it may become indefinite only when a constraint is deleted from the working set. In this case, a temporary modification (of magnitude HESS MOD, see Section 5.1) is added to the last diagonal element of the Cholesky factor. Once a modification has occurred, no further constraints are deleted from the working set until enough constraints have been added so that the projected Hessian is again positive-definite. If equation (1) has a finite solution, a move along the direction obtained by solving (4) with the modified Cholesky factor must encounter a constraint that is not already in the working set.

In order to resolve indefiniteness in this way, we must ensure that the projected Hessian is positive-definite at the first iterate in the QP phase. Given the n_z by n_z projected Hessian, a step-wise Cholesky factorization is performed with symmetric interchanges (and corresponding rearrangement of the columns of Z_z), terminating if the next step would cause the matrix to become indefinite. This determines the largest possible positive-definite principal sub-matrix of the (permuted) projected Hessian. If n_R steps of the Cholesky factorization have been successfully completed, the relevant projected Hessian is an n_R by n_R positive-definite matrix $Z_R^T H Z_R$, where Z_R comprises the first n_R columns of Z. The quadratic function will subsequently be minimized within subspaces of reduced dimension until the full projected Hessian is positive-definite.

If a linear program is being solved and there are fewer general constraints than variables, the method moves from one vertex to

another while minimizing the objective function. When necessary, an initial vertex is defined by temporarily fixing some of the variables at their initial values.

Several strategies are used to control ill-conditioning in the working set. One such strategy is associated with the FEATOL array. Allowing the jth constraint to be violated by as much as FEATOL(j) often provides a choice of constraints that could be added to the working set. When a choice exists, the decision is based on the conditioning of the working set. Negative steps are occasionally permitted, since x_k may violate the constraint to be added.

4. References

[1] Gill P E, Murray W, Saunders M A and Wright M H (1983) User's Guide for SOL/QPSOL. Report SOL 83-7. Department of Operations Research, Stanford University.

[2] Gill P E, Murray W, Saunders M A and Wright M H (1982) The design and implementation of a quadratic programming algorithm. Report SOL 82-7. Department of Operations Research, Stanford University.

[3] Gill P E, Murray W and Wright M H (1981) Practical Optimization. Academic Press.

5. Parameters

1: ITMAX -- INTEGER Input
On entry: an upper bound on the number of iterations to be taken during the LP phase or the QP phase. If ITMAX is not positive, then the value 50 is used in place of ITMAX.

2: MSGLVL -- INTEGER Input
On entry: MSGLVL must indicate the amount of intermediate output desired (see Section 5.1 for a description of the printed output). All output is written to the current advisory message unit (see X04ABF). For MSGLVL >= 10, each level includes the printout for all lower levels.

Value	Definition
<0	No printing.
0	Printing only if an input parameter is incorrect, or if the working set is so ill-conditioned that subsequent overflow is likely. This setting is strongly recommended in preference to MSGLVL < 0.

```
        1       The final solution only.

        5       One brief line of output for each constraint
                addition or deletion (no printout of the final
                solution).

        >=10    The final solution and one brief line of output for
                each constraint addition or deletion.

        >=15    At each iteration, X, ISTATE, and the indices of the
                free variables (i.e.,the variables not currently
                held on a bound).

        >=20    At each iteration, the Lagrange multiplier estimates
                and the general constraint values.

        >=30    At each iteration, the diagonal elements of the
                matrix T associated with the TQ factorization of the
                working set, and the diagonal elements of the
                Cholesky factor R of the projected Hessian.

        >=80    Debug printout.

        99      The arrays CVEC and HESS.

3:  N -- INTEGER                                              Input
    On entry: the number, n, of variables. Constraint: N >= 1.

4:  NCLIN -- INTEGER                                          Input
    On entry: the number of general linear constraints in the
    problem. Constraint: NCLIN >= 0.

5:  NCTOTL -- INTEGER                                         Input
    On entry: the value (N+NCLIN).

6:  NROWA -- INTEGER                                          Input
    On entry:
    the first dimension of the array A as declared in the
    (sub)program from which E04NAF is called.
    Constraint: NROWA >= max(1,NCLIN).

7:  NROWH -- INTEGER                                          Input
    On entry: the first dimension of the array HESS as declared
    in the (sub)program from which E04NAF is called.
    Constraint: NROWH >= 1.

8:  NCOLH -- INTEGER                                          Input
    On entry: the column dimension of the array HESS as declared
    in the (sub)program from which E04NAF is called.
    Constraint: NCOLH >= 1.
```

```
 9:  BIGBND -- DOUBLE PRECISION                                  Input
     On entry: BIGBND must denote an 'infinite' component of l
     and u. Any upper bound greater than or equal to BIGBND will
     be regarded as plus infinity, and a lower bound less than or
     equal to -BIGBND will be regarded as minus infinity.
     Constraint: BIGBND > 0.0.

10:  A(NROWA,N) -- DOUBLE PRECISION array                        Input
     On entry: the leading NCLIN by n part of A must contain the
     NCLIN general constraints, with the ith constraint in the i
     th row of A. If NCLIN = 0, then A is not referenced.

11:  BL(NCTOTL) -- DOUBLE PRECISION array                        Input
     On entry: the lower bounds for all the constraints, in the
     following order. The first n elements of BL must contain the
     lower bounds on the variables. If NCLIN > 0, the next NCLIN
     elements of BL must contain the lower bounds for the general
     linear constraints. To specify a non-existent lower bound
     (i.e., l =-infty), the value used must satisfy BL(j)<=-
             j
     BIGBND To specify the jth constraint as an equality, the
     user must set BL(j) = BU(j). Constraint: BL(j) <= BU(j),
     j=1,2,...,NCTOTL.

12:  BU(NCTOTL) -- DOUBLE PRECISION array                        Input
     On entry: the upper bounds for all the constraints, in the
     following order. The first n elements of BU must contain the
     upper bounds on the variables. If NCLIN > 0, the next NCLIN
     elements of BU must contain the upper bounds for the general
     linear constraints. To specify a non-existent upper bound
     (i.e., u =+infty), the value used must satisfy BU(j) >=
             j
     BIGBND. To specify the jth constraint as an equality, the
     user must set BU(j) = BL(j). Constraint: BU(j) >= BL(j),
     j=1,2,...,NCTOTL.

13:  CVEC(N) -- DOUBLE PRECISION array                           Input
     On entry: the coefficients of the linear term of the
     objective function (the vector c in equation  (1)).

14:  FEATOL(NCTOTL) -- DOUBLE PRECISION array                    Input
     On entry: a set of positive tolerances that define the
     maximum permissible absolute violation in each constraint in
     order for a point to be considered feasible, i.e., if the
     violation in constraint j is less than FEATOL(j), the point
     is considered to be feasible with respect to the jth
     constraint. The ordering of the elements of FEATOL is the
     same as that described above for BL.
```

```
     The elements of FEATOL should not be too small and a warning
     message will be printed on the current advisory message
     channel if any element of FEATOL is less than the machine
     precision (see X02AJF(*)). As the elements of FEATOL
     increase, the algorithm is less likely to encounter
     difficulties with ill-conditioning and degeneracy. However,
     larger values of FEATOL(j) mean that constraint j could be
     violated by a significant amount. It is recommended that
     FEATOL(j) be set to a value equal to the largest acceptable
     violation for constraint j. For example, if the data
     defining the constraints are of order unity and are correct
     to about 6 decimal digits, it would be appropriate to choose
                   -6
     FEATOL(j) as 10   for all relevant j. Often the square root
     of the machine precision is a reasonable choice if the
     constraint is well scaled.

15:  HESS(NROWH,NCOLH) -- DOUBLE PRECISION array            Input
     On entry: HESS may be used to store the Hessian matrix H of
     equation (1) if desired. HESS is accessed only by the
     subroutine QPHESS and is not accessed if LP = .TRUE.. Refer
     to the specification of QPHESS (below) for further details
     of how HESS may be used to pass data to QPHESS.

16:  QPHESS -- SUBROUTINE, supplied by the user.
                                              External Procedure
     QPHESS must define the product of the Hessian matrix H and a
     vector x. The elements of H need not be defined explicitly.
     QPHESS is not accessed if LP is set to .TRUE. and in this
     case QPHESS may be the dummy routine E04NAN. (E04NAN is
     included in the NAG Foundation Library and so need not be
     supplied by the user. Its name may be implementation-
     dependent: see the Users' Note for your implementation for
     details.)

     Its specification is:

           SUBROUTINE QPHESS (N, NROWH, NCOLH, JTHCOL,
          1                   HESS, X, HX)
           INTEGER          N, NROWH, NCOLH, JTHCOL
           DOUBLE PRECISION HESS(NROWH,NCOLH), X(N), HX(N)

      1:  N -- INTEGER                                      Input
          On entry: the number n of variables.

      2:  NROWH -- INTEGER                                  Input
          On entry: the row dimension of the array HESS.

      3:  NCOLH -- INTEGER                                  Input
          On entry: the column dimension of the array HESS.
```

```
 4:  JTHCOL -- INTEGER                                        Input
     The input parameter JTHCOL is included to allow
     flexibility for the user in the special situation when
     x is the jth co-ordinate vector (i.e.,the jth column of
     the identity matrix). This may be of interest because
     the product Hx is then the jth column of H, which can
     sometimes be computed very efficiently. The user may
     code QPHESS to take advantage of this case. On entry:
     if JTHCOL = j, where j>0, HX must contain column JTHCOL
     of H, and hence special code may be included in QPHESS
     to test JTHCOL if desired. However, special code is not
     necessary, since the vector x always contains column
     JTHCOL of the identity matrix whenever QPHESS is called
     with JTHCOL > 0.

 5:  HESS(NROWH,NCOLH) -- DOUBLE PRECISION array              Input
     On entry: the Hessian matrix H.

     In some cases, it may be desirable to use a one-
     dimensional array to transmit data or workspace to
     QPHESS; HESS should then be declared with dimension
     (NROWH) in the (sub)program from which E04NAF is called
     and the parameter NCOLH must be 1.

     In other situations, it may be desirable to compute Hx
     without accessing HESS - for example, if H is sparse or
     has special structure. (This is illustrated in the
     subroutine QPHES1 in the example program in Section 9.)
     The parameters HESS, NROWH and NCOLH may then refer to
     any convenient array.

     When MSGLVL = 99, the (possibly undefined) contents of
     HESS will be printed, except if NROWH and NCOLH are
     both 1. Also printed are the results of calling QPHESS
     with JTHCOL = 1,2,...,n.

 6:  X(N) -- DOUBLE PRECISION array                           Input
     On entry: the vector x.

 7:  HX(N) -- DOUBLE PRECISION array                         Output
     On exit: HX must contain the product Hx.
QPHESS must be declared as EXTERNAL in the (sub)program
from which E04NAF is called. Parameters denoted as
Input must not be changed by this procedure.

17:  COLD -- LOGICAL                                         Input
     On entry: COLD must indicate whether the user has specified
     an initial estimate of the active set of constraints. If
     COLD is set to .TRUE., the initial working set is determined
```

by E04NAF. If COLD is set to .FALSE. (a 'warm start'), the user must define the ISTATE array which gives the status of each constraint with respect to the working set. E04NAF will override the user's specification of ISTATE if necessary, so that a poor choice of working set will not cause a fatal error.

The warm start option is particularly useful when E04NAF is called repeatedly to solve related problems.

18: LP -- LOGICAL Input
On entry: if LP = .FALSE., E04NAF will solve the specified quadratic programming problem. If LP = .TRUE., E04NAF will treat H as zero and solve the resulting linear programming problem; in this case, the parameters HESS and QPHESS will not be referenced.

19: ORTHOG -- LOGICAL Input
On entry: ORTHOG must indicate whether orthogonal transformations are to be used in computing and updating the TQ factorization of the working set

$$A_s Q=(0\ T),$$

where A_s is a sub-matrix of A and T is reverse-triangular.
If ORTHOG = .TRUE., the TQ factorization is computed using Householder reflections and plane rotations, and the matrix Q is orthogonal. If ORTHOG = .FALSE., stabilized elementary transformations are used to maintain the factorization, and Q is not orthogonal. A rule of thumb in making the choice is that orthogonal transformations require more work, but provide greater numerical stability. Thus, we recommend setting ORTHOG to .TRUE. if the problem is reasonably small or the active set is ill-conditioned. Otherwise, setting ORTHOG to .FALSE. will often lead to a reduction in solution time with negligible loss of reliability.

20: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an estimate of the solution. In the absence of better information all elements of X may be set to zero. On exit: from E04NAF, X contains the best estimate of the solution.

21: ISTATE(NCTOTL) -- INTEGER array Input/Output
On entry: with COLD as .FALSE., ISTATE must indicate the status of every constraint with respect to the working set. The ordering of ISTATE is as follows; the first n elements of ISTATE refer to the upper and lower bounds on the variables and elements n+1 through n + NCLIN refer to the upper and lower bounds on Ax. The significance of each

```
possible value of ISTATE(j) is as follows:
ISTATE(j) Meaning

-2        The constraint violates its lower bound by more
          than FEATOL(j). This value of ISTATE cannot occur
          after a feasible point has been found.

-1        The constraint violates its upper bound by more
          than FEATOL(j). This value of ISTATE cannot occur
          after a feasible point has been found.

0         The constraint is not in the working set. Usually,
          this means that the constraint lies strictly
          between its bounds.

1         This inequality constraint is included in the
          working set at its lower bound. The value of the
          constraint is within FEATOL(j) of its lower bound.

2         This inequality constraint is included in the
          working set at its upper bound. The value of the
          constraint is within FEATOL(j) of its upper bound.

3         The constraint is included in the working set as
          an equality. This value of ISTATE can occur only
          when BL(j) = BU(j). The corresponding constraint
          is within FEATOL(j) of its required value.
If COLD = .TRUE., ISTATE need not be set by the user.
However, when COLD = .FALSE., every element of ISTATE must
be set to one of the values given above to define a
suggested initial working set (which will be changed by
E04NAF if necessary). The most likely values are:
ISTATE(j) Meaning

0         The corresponding constraint should not be in the
          initial working set.

1         The constraint should be in the initial working
          set at its lower bound.

2         The constraint should be in the initial working
          set at its upper bound.

3         The constraint should be in the initial working
          set as an equality. This value must not be
          specified unless BL(j) = BU(j). The values 1, 2 or
          3 all have the same effect when BL(j) = BU(j).
Note that if E04NAF has been called previously with the same
values of N and NCLIN, ISTATE already contains satisfactory
values. On exit: when E04NAF exits with IFAIL set to 0, 1 or
```

```
     3, the values in the array ISTATE indicate the status of the
     constraints in the active set at the solution. Otherwise,
     ISTATE indicates the composition of the working set at the
     final iterate.

22:  ITER -- INTEGER                                          Output
     On exit: the number of iterations performed in either the LP
     phase or the QP phase, whichever was last entered.

     Note that ITER is reset to zero after the LP phase.

23:  OBJ -- DOUBLE PRECISION                                  Output
     On exit: the value of the quadratic objective function at x
     if x is feasible (IFAIL <= 5), or the sum of infeasibilities
     at x otherwise (6 <= IFAIL <= 8).

24:  CLAMDA(NCTOTL) -- DOUBLE PRECISION array                 Output
     On exit: the values of the Lagrange multiplier for each
     constraint with respect to the current working set. The
     ordering of CLAMDA is as follows; the first n components
     contain the multipliers for the bound constraints on the
     variables, and the remaining components contain the
     multipliers for the general linear constraints. If ISTATE(j)
     = 0 (i.e.,constraint j is not in the working set), CLAMDA(j)
     is zero. If x is optimal and ISTATE(j) = 1, CLAMDA(j) should
     be non-negative; if ISTATE(j) = 2, CLAMDA(j) should be non-
     positive.

25:  IWORK(LIWORK) -- INTEGER array                        Workspace

26:  LIWORK -- INTEGER                                        Input
     On entry:
     the dimension of the array IWORK as declared in the
     (sub)program from which E04NAF is called.
     Constraint: LIWORK>=2*N.

27:  WORK(LWORK) -- DOUBLE PRECISION array                 Workspace

28:  LWORK -- INTEGER                                         Input
     On entry:
     the dimension of the array WORK as declared in the
     (sub)program from which E04NAF is called.
     Constrai if LP = .FALSE. or NCLIN >= N then
     nts:                2
              LWORK>=2*N +4*N*NCLIN+NROWA.

              if LP = .TRUE. and NCLIN < N then
                                 2
              LWORK>=2*(NCLIN+1) +4*N+2*NCLIN+NROWA.
     If MSGLVL > 0, the amount of workspace provided and the
```

amount of workspace required are output on the current advisory message unit (as defined by X04ABF). As an alternative to computing LWORK from the formula given above, the user may prefer to obtain an appropriate value from the output of a preliminary run with a positive value of MSGLVL and LWORK set to 1 (E04NAF will then terminate with IFAIL = 9).

29: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.

IFAIL contains zero on exit if x is a strong local minimum. i.e., the projected gradient is neglible, the Lagrange multipliers are optimal, and the projected Hessian is positive-definite. In some cases, a zero value of IFAIL means that x is a global minimum (e.g. when the Hessian matrix is positive-definite).

5.1. Description of the Printed Output

When MSGLVL >= 5, a line of output is produced for every change in the working set (thus, several lines may be printed during a single iteration).

To aid interpretation of the printed results, we mention the convention for numbering the constraints: indices 1 through to n refer to the bounds on the variables, and when NCLIN > 0 indices n+1 through to n + NCLIN refer to the general constraints. When the status of a constraint changes, the index of the constraint is printed, along with the designation L (lower bound), U (upper bound) or E (equality).

In the LP phase, the printout includes the following:

ITN is the iteration count.

JDEL is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was

deleted.

JADD is the index of the constraint added to the working set. If JADD is zero, no constraint was added.

STEP is the step taken along the computed search direction.

COND T is a lower bound on the condition number of the matrix of predicted active constraints.

NUMINF is the number of violated constraints (infeasibilities).

SUMINF is a weighted sum of the magnitudes of the constraint violations.

LPOBJ is the value of the linear objective function $c^T x$. It is printed only if LP = .TRUE..

During the QP phase, the printout includes the following:

ITN is the iteration count (reset to zero after the LP phase).

JDEL is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was deleted.

JADD is the index of the constraint added to the working set. If JADD is zero, no constraint was added.

STEP is the step (alpha) $_k$ taken along the direction of search (if STEP is 1.0, the current point is a minimum in the subspace defined by the current working set).

NHESS is the number of calls to subroutine QPHESS.

OBJECTIVE is the value of the quadratic objective function.

NCOLZ is the number of columns of Z (see Section 3). In general, it is the dimension of the subspace in which the quadratic is currently being minimized.

NORM GFREE is the Euclidean norm of the gradient of the

objective function with respect to the free variables, i.e. variables not currently held at a bound (NORM GFREE is not printed if ORTHOG = .FALSE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

NORM QTG is a weighted norm of the gradient of the objective function with respect to the free variables (NORM QTG is not printed if ORTHOG = .TRUE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

NORM ZTG is the Euclidean norm of the projected gradient (see Section 3).

COND T is a lower bound on the condition number of the matrix of constraints in the working set.

COND ZHZ is a lower bound on the condition number of the projected Hessian matrix.

HESS MOD is the correction added to the diagonal of the projected Hessian to ensure that a satisfactory Cholesky factorization exists (see Section 3). When the projected Hessian is sufficiently positive-definite, HESS MOD will be zero.

When MSGLVL = 1 or MSGLVL >= 10, the summary printout at the end of execution of E04NAF includes a listing of the status of every constraint. Note that default names are assigned to all variables and constraints.

The following describes the printout for each variable.

VARBL is the name (V) and index j, j=1,2,...,n, of the variable.

STATE gives the state of the variable (FR if neither bound is in the working set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound, TB if held on a temporary bound). If VALUE lies outside the upper or lower bounds by more than FEATOL(j), STATE will be ++ or -- respectively.

```
VALUE          is the value of the variable at the final
               iteration.

LOWER BOUND    is the lower bound specified for the variable.

UPPER BOUND    is the upper bound specified for the variable.

LAGR MULT      is the value of the Lagrange multiplier for the
               associated bound constraint. This will be zero if
               STATE is FR. If x is optimal and STATE is LL, the
               multiplier should be non-negative; if STATE is UL,
               the multiplier should be non-positive.

RESIDUAL       is the difference between the variable and the
               nearer of its bounds BL(j) and BU(j).

For each of the general constraints the printout is as above with
refers to the jth element of Ax, except that VARBL is replaced by

LNCON          The name (L) and index j, j=1,2,...,NCLIN, of the
               constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
     x is a weak local minimum (the projected gradient is
     negligible, the Lagrange multipliers are optimal, but the
     projected Hessian is only semi-definite). This means that
     the solution is not unique.

IFAIL= 2
     The solution appears to be unbounded, i.e., the quadratic
     function is unbounded below in the feasible region. This
     value of IFAIL occurs when a step of infinity would have to
     be taken in order to continue the algorithm.

IFAIL= 3
     x appears to be a local minimum, but optimality cannot be
     verified because some of the Lagrange multipliers are very
     small in magnitude.

     E04NAF has probably found a solution. However, the presence
     of very small Lagrange multipliers means that the predicted
     active set may be incorrect, or that x may be only a
     constrained stationary point rather than a local minimum.
     The method in E04NAF is not guaranteed to find the correct
     active set when there are very small multipliers. E04NAF
```

```
     attempts to delete constraints with zero multipliers, but
     this does not necessarily resolve the issue. The
     determination of the correct active set is a combinatorial
     problem that may require an extremely large amount of time.
     The occurrence of small multipliers often (but not always)
     indicates that there are redundant constraints.

IFAIL= 4
     The iterates of the QP phase could be cycling, since a total
     of 50 changes were made to the working set without altering
     x.

     This value will occur if 50 iterations are performed in the
     QP phase without changing x. The user should check the
     printed output for a repeated pattern of constraint
     deletions and additions. If a sequence of constraint changes
     is being repeated, the iterates are probably cycling.
     (E04NAF  does not contain a method that is guaranteed to
     avoid cycling, which would be combinatorial in nature.)
     Cycling may occur in two circumstances: at a constrained
     stationary point where there are some small or zero Lagrange
     multipliers (see the discussion of IFAIL = 3); or at a point
     (usually a vertex) where the constraints that are satisfied
     exactly are nearly linearly dependent. In the latter case,
     the user has the option of identifying the offending
     dependent constraints and removing them from the problem, or
     restarting the run with larger values of FEATOL for nearly
     dependent constraints. If E04NAF terminates with IFAIL = 4,
     but no suspicious pattern of constraint changes can be
     observed, it may be worthwhile to restart with the final x
     (with or without the warm start option).

IFAIL= 5
     The limit of ITMAX iterations was reached in the QP phase
     before normal termination occurred.

     The value of ITMAX may be too small. If the method appears
     to be making progress (e.g. the objective function is being
     satisfactorily reduced), increase ITMAX and rerun E04NAF
     (possibly using the warm start facility to specify the
     initial working set). If ITMAX is already large, but some of
     the constraints could be nearly linearly dependent, check
     the output for a repeated pattern of constraints entering
     and leaving the working set. (Near-dependencies are often
     indicated by wide variations in size in the diagonal
     elements of the T matrix, which will be printed if MSGLVL >=
     30.) In this case, the algorithm could be cycling (see the
     comments for IFAIL = 4).

IFAIL= 6
```

```
     The LP phase terminated without finding a feasible point,
     and hence it is not possible to satisfy all the constraints
     to within the tolerances specified by the FEATOL array. In
     this case, the final iterate will reveal values for which
     there will be a feasible point (e.g. a feasible point will
     exist if the feasibility tolerance for each violated
     constraint exceeds its RESIDUAL at the final point). The
     modified problem (with altered values in FEATOL) may then be
     solved using a warm start.

     The user should check that there are no constraint
     redundancies. If the data for the jth constraint are
     accurate only to the absolute precision (delta), the user
     should ensure that the value of FEATOL(j) is greater than
     (delta). For example, if all elements of A are of order
     unity and are accurate only to three decimal places, every
                                                  -3
     component of FEATOL should be at least 10  .

IFAIL= 7
     The iterates may be cycling during the LP phase; see the
     comments above under IFAIL = 4.

IFAIL= 8
     The limit of ITMAX iterations was reached during the LP
     phase. See comments above under IFAIL = 5.

IFAIL= 9
     An input parameter is invalid.

Overflow
     If the printed output before the overflow error contains a
     warning about serious ill-conditioning in the working set
     when adding the jth constraint, it may be possible to avoid
     the difficulty by increasing the magnitude of FEATOL(j) and
     rerunning the program. If the message recurs even after this
     change, the offending linearly dependent constraint (with
     index j) must be removed from the problem. If a warning
     message did not precede the fatal overflow, the user should
     contact NAG.

7. Accuracy

The routine implements a numerically stable active set strategy
and returns solutions that are as accurate as the condition of
the QP problem warrants on the machine.

8. Further Comments

The number of iterations depends upon factors such as the number
```

of variables and the distances of the starting point from the solution. The number of operations performed per iteration is
                                2
roughly proportional to (NFREE) , where NFREE (NFREE<=n) is the number of variables fixed on their upper or lower bounds.

Sensible scaling of the problem is likely to reduce the number of iterations required and make the problem less sensitive to perturbations in the data, thus improving the condition of the QP problem. See the Chapter Introduction and Gill et al [1] for further information and advice.

9. Example

                          T   1 T
To minimize the function c x+ -x Hx, where
                              2

                                                    T
          c=[-0.02,-0.2,-0.2,-0.2,-0.2,0.04,0.04]

                        [2 0 0 0 0 0 0]
                        [0 2 0 0 0 0 0]
                        [0 0 2 2 0 0 0]
                      H=[0 0 2 2 0 0 0]
                        [0 0 0 0 2 0 0]
                        [0 0 0 0 0 -2 -2]
                        [0 0 0 0 0 -2 -2]

subject to the bounds

%%%

E04 -- Minimizing or Maximizing a Function E04UCF
E04UCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library. In particular, the optional parameters of the NAG routine are now included in the parameter list. These are described in section 5.1.2, below.

1. Purpose

```
E04UCF is designed to minimize an arbitrary smooth function
subject to constraints, which may include simple bounds on the
variables, linear constraints and smooth nonlinear constraints.
(E04UCF  may be used for unconstrained, bound-constrained and
linearly constrained optimization.) The user must provide
subroutines that define the objective and constraint functions
and as many of their first partial derivatives as possible.
Unspecified derivatives are approximated by finite differences.
All matrices are treated as dense, and hence E04UCF is not
intended for large sparse problems.

E04UCF uses a sequential quadratic programming (SQP) algorithm in
which the search direction is the solution of a quadratic
programming (QP) problem. The algorithm treats bounds, linear
constraints and nonlinear constraints separately.

2. Specification

      SUBROUTINE E04UCF (N, NCLIN, NCNLN, NROWA, NROWJ, NROWR,
     1                   A, BL, BU, CONFUN, OBJFUN, ITER,
     2                   ISTATE, C, CJAC, CLAMDA, OBJF, OBJGRD,
     3                   R, X, IWORK, LIWORK, WORK, LWORK,
     4                   IUSER, USER, STA, CRA, DER, FEA, FUN,
     5                   HES, INFB, INFS, LINF, LINT, LIST,
     6                   MAJI, MAJP, MINI, MINP, MON, NONF,
     7                   OPT, STE, STAO, STAC, STOO, STOC, VE,
     8                   IFAIL)
      INTEGER           N, NCLIN, NCNLN, NROWA, NROWJ, NROWR,
     1                  ITER, ISTATE(N+NCLIN+NCNLN), IWORK(LIWORK)
     2                  , LIWORK, LWORK, IUSER(*), DER, MAJI,
     3                  MAJP, MINI, MINP, MON, STAO, STAC, STOO,
     4                  STOC, VE, IFAIL
      DOUBLE PRECISION  A(NROWA,*), BL(N+NCLIN+NCNLN), BU
     1                  (N+NCLIN+NCNLN), C(*), CJAC(NROWJ,*),
     2                  CLAMDA(N+NCLIN+NCNLN), OBJF, OBJGRD(N), R
     3                  (NROWR,N), X(N), WORK(LWORK), USER(*),
     4                  CRA, FEA, FUN,  INFB, INFS, LINF, LINT,
     5                  NONF, OPT, STE
      LOGICAL           LIST, STA, HES
      EXTERNAL          CONFUN, OBJFUN

3. Description

E04UCF is designed to solve the nonlinear programming problem --
the minimization of a smooth nonlinear function subject to a set
of constraints on the variables. The problem is assumed to be
stated in the following form:

                                          { x  }
      Minimize      F(x)   subject to  l<={A x }<=u,          (1)
```

```
                n                          { L  }
       x is in R                           {c(x)}
```

where F(x), the objective function, is a nonlinear function, A is an n by n constant matrix, and c(x) is an n element vector of nonlinear constraint functions. (The matrix A and the vector c(x) may be empty.) The objective function and the constraint functions are assumed to be smooth, i.e., at least twice-continuously differentiable. (The method of E04UCF will usually solve (1) if there are only isolated discontinuities away from the solution.)

This routine is essentially identical to the subroutine SOL/NPSOL described in Gill et al [8].

Note that upper and lower bounds are specified for all the variables and for all the constraints.

An equality constraint can be specified by setting l =u . If certain bounds are not present, the associated elements of l or u can be set to special values that will be treated as -infty or +infty.

If there are no nonlinear constraints in (1) and F is linear or quadratic then one of E04MBF, E04NAF or E04NCF(*) will generally be more efficient. If the problem is large and sparse the MINOS package (see Murtagh and Saunders [13]) should be used, since E04UCF treats all matrices as dense.

The user must supply an initial estimate of the solution to (1), together with subroutines that define F(x), c(x) and as many first partial derivatives as possible; unspecified derivatives are approximated by finite differences.

The objective function is defined by subroutine OBJFUN, and the nonlinear constraints are defined by subroutine CONFUN. On every call, these subroutines must return appropriate values of the objective and nonlinear constraints. The user should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see Section 5.1 for a discussion of the optional parameter Derivative Level. Just before either OBJFUN or CONFUN is called, each element of the current gradient array OBJGRD or CJAC is initialised to a special value. On exit, any element that retains the value is estimated by finite differences. Note that if there are nonlinear costraints, then the first call to CONFUN will precede the first

call to OBJFUN.

For maximum reliability, it is preferable for the user to provide all partial derivatives (see Chapter 8 of Gill et al [10], for a detailed discussion). If all gradients cannot be provided, it is similarly advisable to provide as many as possible. While developing the subroutines OBJFUN and CONFUN, the optional parameter Verify (see Section 5.1) should be used to check the calculation of any known gradients.

E04UCF implements a sequential quadratic programming (SQP) method. The document for E04NCF(*) should be consulted in conjunction with this document.

In the rest of this section we briefly summarize the main features of the method of E04UCF. Where possible, explicit reference is made to the names of variables that are parameters of subroutines E04UCF or appear in the printed output (see Section 5.2).

At a solution of (1), some of the constraints will be active, i.e., satisfied exactly. An active simple bound constraint implies that the corresponding variable is fixed at its bound, and hence the variables are partitioned into fixed and free variables. Let C denote the m by n matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by n_{FX}, with n_{FR} ($n_{FR}=n-n_{FX}$) the number of free variables. The subscripts 'FX' and 'FR' on a vector or matrix will denote the vector or matrix composed of the components corresponding to fixed or free variables.

A point x is a first-order Kuhn-Tucker point for (1) (see, e.g., Powell [14]) if the following conditions hold:

(i) x is feasible;

(ii) there exist vectors (xi) and (lambda) (the Lagrange multiplier vectors for the bound and general constraints) such that

$$g = C^{T}(\lambda) + (\xi), \qquad (2)$$

where g is the gradient of F evaluated at x, and $(\xi)_j = 0$ if the jth variable is free.

(iii) The Lagrange multiplier corresponding to an inequality constraint active at its lower bound must be non-negative, and non-positive for an inequality constraint active at its upper bound.

Let Z denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of C_{FR} ; i.e., $C_{FR} Z = 0$. An equivalent statement of the condition (2) in terms of Z is

$$Z^T g_{FR} = 0.$$

The vector $Z^T g_{FR}$ is termed the projected gradient of F at x. Certain additional conditions must be satisfied in order for a first-order Kuhn-Tucker point to be a solution of (1) (see, e.g., Powell [14]).

The method of E04UCF is a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher [5], Gill et al [10] and Powell [15].

The basic structure of E04UCF involves major and minor iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to x^*, a first-order Kuhn-Tucker point of (1).

At a typical major iteration, the new iterate $\bar{x}$ is defined by

$$\bar{x} = x + (\alpha) p \tag{3}$$

where x is the current iterate, the non-negative scalar (α) is the step length, and p is the search direction. (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Also associated with each major iteration are estimates of the Lagrange multipliers and a prediction of the active set.

The search direction p in (3) is the solution of a quadratic programming subproblem of the form

$$\underset{p}{\text{Minimize}} \quad g^T p + \frac{1}{2} p^T H p, \qquad \text{subject to} \quad l \leq \begin{Bmatrix} p \\ A_L p \\ A_N p \end{Bmatrix} \leq \bar{u}, \tag{4}$$

where g is the gradient of F at x, the matrix H is a positive-definite quasi-Newton approximation to the Hessian of the

Lagrangian function (see Section 8.3), and A is the Jacobian
N
matrix of c evaluated at x. (Finite-difference estimates may be used for g and A ; see the optional parameter Derivative Level in
N
Section 5.1.) Let l in (1) be partitioned into three sections: l , l and l , corresponding to the bound, linear and nonlinear
B L N
constraints. The vector $\bar{l}$ in (4) is similarly partitioned, and is defined as

$$\bar{l}_B = l_B - x, \quad \bar{l}_L = l_L - A_L x, \quad \text{and} \quad \bar{l}_N = l_N - c,$$

where c is the vector of nonlinear constraints evaluated at x.

The vector $\bar{u}$ is defined in an analogous fashion.

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities Bnd, Lin and Nln in the printed output of E04UCF.) In E04UCF, (4) is solved using E04NCF(*). Since solving a quadratic program as an iterative procedure, the minor iterations of E04UCF are the iterations of E04NCF(*). (More details about solving the subproblem are given in Section 8.1.)

Certain matrices associated with the QP subproblem are relevant in the major iterations. Let the subscripts 'FX' and 'FR' refer to the predicted fixed and free variables, and let C denote the m by n matrix of gradients of the general linear and nonlinear constraints in the predicted active set. First, we have available the TQ factorization of C_{FR} :

$$C_{FR} Q_{FR} = (0 \; T), \tag{5}$$

where T is a nonsingular m by m reverse-triangular matrix (i.e., $t_{ij} = 0$ if $i+j<m$), and the non-singular n_{FR} by n_{FR} matrix Q_{FR} is the product of orthogonal transformations (see Gill et al [6]). Second, we have the upper-triangular Cholesky factor R of the transformed and re-ordered Hessian matrix

$$R_{FR}^T R_{FR} = Q_{FR}^T \tilde{H}_{FR} Q_{FR}, \tag{6}$$

Q

~
where H is the Hessian H with rows and columns permuted so that
the free variables are first, and Q is the n by n matrix

$$Q=\begin{pmatrix} Q_{FR} & \\ & I \end{pmatrix}, \qquad (7)$$

with I_{FX} the identity matrix of order n_{FX}. If the columns of Q_{FR} are partitioned so that

$$Q_{FR}=(Z\ Y),$$

the n_z ($n_z = n_{FR} - m$) columns of Z form a basis for the null space of C_{FR}. The matrix Z is used to compute the projected gradient $Z^T g_{FR}$ at the current iterate. (The values Nz, Norm Gf and Norm Gz printed by E04UCF give n_z and the norms of g_{FR} and $Z^T g_{FR}$.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (4) is identical to the correct active set in a neighbourhood of x^*. In E04UCF, this feature is exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints in E04UCF also saves computation in factorizing C_{FR} and H_Q.

Once p has been computed, the major iteration proceeds by determining a step length (alpha) that produces a 'sufficient decrease' in an augmented Lagrangian merit function (see Section 8.2). Finally, the approximation to the transformed Hessian matrix H_Q is updated using a modified BFGS quasi-Newton update (see Section 8.3) to incorporate new curvature information obtained in the move from x to $\bar{x}$.

On entry to E04UCF, an iterative procedure from E04NCF(*) is executed, starting with the user-provided initial point, to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by Linear Feasibility Tolerance see Section 5.1). If no feasible point exists for the bound and linear constraints, (1) has no solution and E04UCF terminates. Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite-difference interval (see the discussion of Difference Interval in Section 5.1). In contrast to the bounds and linear constraints, it must be emphasised that the nonlinear constraints will not generally be satisfied until an optimal point is reached.

Facilities are provided to check whether the user-provided gradients appear to be correct (see the optional parameter Verify in Section 5.1). In general, the check is provided at the first point that is feasible with respect to the linear constraints and bounds. However, the user may request that the check be performed at the initial point.

In summary, the method of E04UCF first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem;

(b) a linesearch with an augmented Lagrangian merit function; and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian function.

These three procedures are described in more detail in Section 8.

4. References

[1] Dennis J E Jr and More J J (1977) Quasi-Newton Methods, Motivation and Theory. SIAM Review. 19 46--89.

[2] Dennis J E Jr and Schnabel R B (1981) A New Derivation of Symmetric Positive-Definite Secant Updates. Nonlinear Programming 4. (ed O L Mangasarian, R R Meyer and S M. Robinson) Academic Press. 167--199.

[3] Dennis J E Jr and Schnabel R B (1983) Numerical Methods for Unconstrained Optimixation and Nonlinear Equations. Prentice-Hall.

[4] Dongarra J J, Du Croz J J, Hammarling S and Hanson R J (1985) A Proposal for an Extended set of Fortran Basic Linear Algebra Subprograms. SIGNUM Newsletter. 20 (1) 2--18.

[5] Fletcher R (1981) Practical Methods of Optimization, Vol 2. Constrained Optimization. Wiley.

[6] Gill P E, Murray W, Saunders M A and Wright M H (1984) User's Guide for SOL/QPSOL Version 3.2. Report SOL 84-5. Department of Operations Research, Stanford University.

[7] Gill P E, Murray W, Saunders M A and Wright M H (1984) Procedures for Optimization Problems with a Mixture of Bounds and General Linear Constraints. ACM Trans. Math. Softw. 10 282--298.

[8] Gill P E, Hammarling S, Murray W, Saunders M A and Wright M H (1986) User's Guide for LSSOL (Version 1.0). Report SOL 86-1. Department of Operations Research, Stanford University.

[9] Gill P E, Murray W, Saunders M A and Wright M H (1986) Some Theoretical Properties of an Augmented Lagrangian Merit Function. Report SOL 86-6R. Department of Operations Research, Stanford University.

[10] Gill P E, Murray W and Wright M H (1981) Practical Optimization. Academic Press.

[11] Hock W and Schittkowski K (1981) Test Examples for Nonlinear Programming Codes. Lecture Notes in Economics and Mathematical Systems. 187 Springer-Verlag.

[12] Lawson C L, Hanson R J, Kincaid D R and Krogh F T (1979) Basic Linear Algebra Subprograms for Fortran Usage. ACM Trans. Math. Softw. 5 308--325.

[13] Murtagh B A and Saunders M A (1983) MINOS 5.0 User's Guide. Report SOL 83-20. Department of Operations Research, Stanford University.

[14] Powell M J D (1974) Introduction to Constrained Optimization. Numerical Methods for Constrained Optimization. (ed P E Gill and W Murray) Academic Press. 1--28.

[15] Powell M J D (1983) Variable Metric Methods in Constrained Optimization. Mathematical Programming: The State of the Art. (ed A Bachem, M Groetschel and B Korte) Springer-

```
      Verlag. 288--311.

   5. Parameters

    1: N -- INTEGER                                              Input
       On entry: the number, n, of variables in the problem.
       Constraint: N > 0.

    2: NCLIN -- INTEGER                                          Input
       On entry: the number, n , of general linear constraints in
                              L
       the problem. Constraint: NCLIN >= 0.

    3: NCNLN -- INTEGER                                          Input
       On entry: the number, n , of nonlinear constraints in the
                              N
       problem. Constraint: NCNLN >= 0.

    4: NROWA -- INTEGER                                          Input
       On entry:
       the first dimension of the array A as declared in the
       (sub)program from which E04UCF is called.
       Constraint: NROWA >= max(1,NCLIN).

    5: NROWJ -- INTEGER                                          Input
       On entry:
       the first dimension of the array CJAC as declared in the
       (sub)program from which E04UCF is called.
       Constraint: NROWJ >= max(1,NCNLN).

    6: NROWR -- INTEGER                                          Input
       On entry:
       the first dimension of the array R as declared in the
       (sub)program from which E04UCF is called.
       Constraint: NROWR >= N.

    7: A(NROWA,*) -- DOUBLE PRECISION array                      Input
       The second dimension of the array A must be >= N for NCLIN >
       0. On entry: the ith row of the array A must contain the ith
       row of the matrix A  of general linear constraints in (1).
                          L
       That is, the ith row contains the coefficients of the ith
       general linear constraint, for i = 1,2,...,NCLIN.

       If NCLIN = 0 then the array A is not referenced.

    8: BL(N+NCLIN+NCNLN) -- DOUBLE PRECISION array               Input
       On entry: the lower bounds for all the constraints, in the
       following order. The first n elements of BL must contain the
       lower bounds on the variables. If NCLIN > 0, the next n
```

L
elements of BL must contain the lower bounds on the general linear constraints. If NCNLN > 0, the next n elements of BL
N
must contain the lower bounds for the nonlinear constraints. To specify a non-existent lower bound (i.e., l =-infty), the
j
value used must satisfy BL(j)<=-BIGBND, where BIGBND is the value of the optional parameter Infinite Bound Size whose
10
default value is 10 (see Section 5.1). To specify the jth constraint as an equality, the user must set BL(j) = BU(j) = (beta), say, where |(beta)|<BIGBND. Constraint: BL(j) <= BU(j), for j=1,2,...,N+NCLIN+NCNLN.

9: BU(N+NCLIN+NCNLN) -- DOUBLE PRECISION array Input
On entry: the upper bounds for all the constraints in the following order. The first n elements of BU must contain the upper bounds on the variables. If NCLIN > 0, the next n
L
elements of BU must contain the upper bounds on the general linear constraints. If NCNLN > 0, the next n elements of BU
N
must contain the upper bounds for the nonlinear constraints. To specify a non-existent upper bound (i.e., u =+infty), the
j
value used must satisfy BU(j) >= BIGBND, where BIGBND is the value of the optional parameter Infinite Bound Size, whose
10
default value is 10 (see Section 5.1). To specify the jth constraint as an equality, the user must set BU(j) = BL(j) = (beta), say, where |(beta)| < BIGBND. Constraint: BU(j) >= BL(j), for j=1,2,...,N+NCLIN+NCNLN.

10: CONFUN -- SUBROUTINE, supplied by the user.
External Procedure
CONFUN must calculate the vector c(x) of nonlinear constraint functions and (optionally) its Jacobian for a specified n element vector x. If there are no nonlinear constraints (NCNLN=0), CONFUN will never be called by E04UCF and CONFUN may be the dummy routine E04UDM. (E04UDM is included in the NAG Foundation Library and so need not be supplied by the user. Its name may be implementation-dependent: see the Users' Note for your implementation for details.) If there are nonlinear constraints, the first call to CONFUN will occur before the first call to OBJFUN.

Its specification is:

```
SUBROUTINE CONFUN (MODE, NCNLN, N, NROWJ, NEEDC,
```

```
     1                  X, C, CJAC, NSTATE, IUSER,
     2                  USER)
      INTEGER          MODE, NCNLN, N, NROWJ, NEEDC
     1                 (NCNLN), NSTATE, IUSER(*)
      DOUBLE PRECISION X(N), C(NCNLN), CJAC(NROWJ,N),
     1                 USER(*)
```

1: MODE -- INTEGER Input/Output
On entry: MODE indicates the values that must be assigned during each call of CONFUN. MODE will always have the value 2 if all elements of the Jacobian are available, i.e., if Derivative Level is either 2 or 3 (see Section 5.1). If some elements of CJAC are unspecified, E04UCF will call CONFUN with MODE = 0, 1, or 2:

If MODE = 2, only the elements of C corresponding to positive values of NEEDC must be set (and similarly for the available components of the rows of CJAC).

If MODE = 1, the available components of the rows of CJAC corresponding to positive values in NEEDC must be set. Other rows of CJAC and the array C will be ignored.

If MODE = 0, the components of C corresponding to positive values in NEEDC must be set. Other components and the array CJAC are ignored. On exit: MODE may be set to a negative value if the user wishes to terminate the solution to the current problem. If MODE is negative on exit from CONFUN then E04UCF will terminate with IFAIL set to MODE.

2: NCNLN -- INTEGER Input
On entry: the number, n , of nonlinear constraints.
N

3: N -- INTEGER Input
On entry: the number, n, of variables.

4: NROWJ -- INTEGER Input
On entry: the first dimension of the array CJAC.

5: NEEDC(NCNLN) -- INTEGER array Input
On entry: the indices of the elements of C or CJAC that must be evaluated by CONFUN. If NEEDC(i)>0 then the ith element of C and/or the ith row of CJAC (see parameter MODE above) must be evaluated at x.

6: X(N) -- DOUBLE PRECISION array Input

On entry: the vector x of variables at which the constraint functions are to be evaluated.

7: C(NCNLN) -- DOUBLE PRECISION array Output
On exit: if NEEDC(i)>0 and MODE = 0 or 2, C(i) must contain the value of the ith constraint at x. The remaining components of C, corresponding to the non-positive elements of NEEDC, are ignored.

8: CJAC(NROWJ,N) -- DOUBLE PRECISION array Output
On exit: if NEEDC(i)>0 and MODE = 1 or 2, the ith row of CJAC must contain the available components of the vector (nabla)c given by

$$(\nabla c_i)^T = \left(\frac{\partial c_i}{\partial x_1}, \frac{\partial c_i}{\partial x_2}, \ldots, \frac{\partial c_i}{\partial x_n} \right),$$

where $\frac{\partial c_i}{\partial x_j}$ is the partial derivative of the ith constraint with respect to the jth variable, evaluated at the point x. See also the parameter NSTATE below. The remaining rows of CJAC, corresponding to non-positive elements of NEEDC, are ignored.

If all constraint gradients (Jacobian elements) are known (i.e., Derivative Level = 2 or 3; see Section 5.1) any constant elements may be assigned to CJAC one time only at the start of the optimization. An element of CJAC that is not subsequently assigned in CONFUN will retain its initial value throughout. Constant elements may be loaded into CJAC either before the call to E04UCF or during the first call to CONFUN (signalled by the value NSTATE = 1). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case CJAC may be initialised to zero and non-zero elements may be reset by CONFUN.

Note that constant non-zero elements do affect the values of the constraints. Thus, if CJAC(i,j) is set to a constant value, it need not be reset in subsequent calls to CONFUN, but the value CJAC(i,j)*X(j) must nonetheless be added to C(i).

It must be emphasized that, if Derivative Level < 2, unassigned elements of CJAC are not treated as

```
     constant; they are estimated by finite differences, at
     non-trivial expense. If the user does not supply a
     value for Difference Interval (see Section 5.1), an
     interval for each component of x is computed
     automatically at the start of the optimization. The
     automatic procedure can usually identify constant
     elements of CJAC, which are then computed once only by
     finite differences.

 9:  NSTATE -- INTEGER                                  Input
     On entry: if NSTATE = 1 then E04UCF is calling CONFUN
     for the first time. This parameter setting allows the
     user to save computation time if certain data must be
     read or calculated only once.

10:  IUSER(*) -- INTEGER array              User Workspace

11:  USER(*) -- DOUBLE PRECISION array      User Workspace
     CONFUN is called from E04UCF with the parameters IUSER
     and USER as supplied to E04UCF. The user is free to use
     the arrays IUSER and USER to supply information to
     CONFUN as an alternative to using COMMON.
CONFUN must be declared as EXTERNAL in the (sub)program
from which E04UCF is called. Parameters denoted as
Input must not be changed by this procedure.

11:  OBJFUN -- SUBROUTINE, supplied by the user.
                                           External Procedure
OBJFUN must calculate the objective function F(x) and
(optionally) the gradient g(x) for a specified n element
vector x.

Its specification is:

       SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD,
      1                   NSTATE, IUSER, USER)
       INTEGER          MODE, N, NSTATE, IUSER(*)
       DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)

 1:  MODE -- INTEGER                            Input/Output
     On entry: MODE indicates the values that must be
     assigned during each call of OBJFUN.

     MODE will always have the value 2 if all components of
     the objective gradient are specified by the user, i.e.,
     if Derivative Level is either 1 or 3. If some gradient
     elements are unspecified, E04UCF will call OBJFUN with
     MODE = 0, 1 or 2.
          If MODE = 2, compute OBJF and the available
          components of OBJGRD.
```

```
                If MODE = 1, compute all available components of
                OBJGRD; OBJF is not required.

                If MODE = 0, only OBJF needs to be computed;
                OBJGRD is ignored.
            On exit: MODE may be set to a negative value if the
            user wishes to terminate the solution to the current
            problem. If MODE is negative on exit from OBJFUN, then
            E04UCF will terminate with IFAIL set to MODE.

        2:  N -- INTEGER                                        Input
            On entry: the number, n, of variables.

        3:  X(N) -- DOUBLE PRECISION array                      Input
            On entry: the vector x of variables at which the
            objective function is to be evaluated.

        4:  OBJF -- DOUBLE PRECISION                           Output
            On exit: if MODE = 0 or 2, OBJF must be set to the
            value of the objective function at x.

        5:  OBJGRD(N) -- DOUBLE PRECISION array                Output
            On exit: if MODE = 1 or 2, OBJGRD must return the
            available components of the gradient evaluated at x.

        6:  NSTATE -- INTEGER                                   Input
            On entry: if NSTATE = 1 then E04UCF is calling OBJFUN
            for the first time. This parameter setting allows the
            user to save computation time if certain data must be
            read or calculated only once.

        7:  IUSER(*) -- INTEGER array                  User Workspace

        8:  USER(*) -- DOUBLE PRECISION array          User Workspace
            OBJFUN is called from E04UCF with the parameters IUSER
            and USER as supplied to E04UCF. The user is free to use
            the arrays IUSER and USER to supply information to
            OBJFUN as an alternative to using COMMON.
        OBJFUN must be declared as EXTERNAL in the (sub)program
        from which E04UCF is called. Parameters denoted as
        Input must not be changed by this procedure.

    12: ITER -- INTEGER                                        Output
        On exit: the number of iterations performed.

    13: ISTATE(N+NCLIN+NCNLN) -- INTEGER array           Input/Output
        On entry: ISTATE need not be initialised if E04UCF is called
        with (the default) Cold Start option. The ordering of ISTATE
        is as follows. The first n elements of ISTATE refer to the
```

upper and lower bounds on the variables, elements n+1
through n+n refer to the upper and lower bounds on A x, and
          L   L
elements n+n +1 through n+n +n refer to the upper and lower
           L   L N
bounds on c(x). When a Warm Start option is chosen, the
elements of ISTATE corresponding to the bounds and linear
constraints define the initial working set for the procedure
that finds a feasible point for the linear constraints and
bounds. The active set at the conclusion of this procedure
and the elements of ISTATE corresponding to nonlinear
constraints then define the initial working set for the
first QP subproblem. Possible values for ISTATE(j) are:

ISTATE(j) Meaning

0 The corresponding constraint is not in the initial QP working set.

1 This inequality constraint should be in the working set at its lower bound.

2 This inequality constraint should be in the working set at its upper bound.

3 This equality constraint should be in the initial working set. This value must not be specified unless BL(j) = BU(j). The values 1,2 or 3 all have the same effect when BL(j) = BU(j).

Note that if E04UCF has been called previously with the same values of N, NCLIN and NCNLN, ISTATE already contains satisfactory values. If necessary, E04UCF will override the user's specification of ISTATE so that a poor choice will not cause the algorithm to fail. On exit: with IFAIL = 0 or 1, the values in the array ISTATE correspond to the active set of the final QP subproblem, and are a prediction of the status of the constraints at the solution of the problem. Otherwise, ISTATE indicates the composition of the QP working set at the final iterate. The significance of each possible value of ISTATE(j) is as follows:

-2 This constraint violates its lower bound by more than the appropriate feasibility tolerance (see the optional parameters LinearFeasibility Tolerance and Nonlinear Feasibility Tolerance in Section 5.1). This value can occur only when no feasible point can be found for a QP subproblem.

-1 This constraint violates its upper bound by more than the appropriate feasibility tolerance (see the optional parameters Linearear Feasibility

```
                Tolerance and Nonlinear Feasibility Tolerance in
                Section 5.1). This value can occur only when no
                feasible point can be found for a QP subproblem.

           0    The constraint is satisfied to within the
                feasibility tolerance, but is not in the working
                set.

           1    This inequality constraint is included in the QP
                working set at its upper bound.

           2    This inequality constraint is included in the QP
                working set at its upper bound.

           3    This constraint is included in the QP working set
                as an equality. This value of ISTATE can occur
                only when BL(j) = BU(j).

     14:  C(*) -- DOUBLE PRECISION array                        Output
          Note: the dimension of the array C must be at least
          max(1,NCNLN).
          On exit: if NCNLN > 0, C(i) contains the value of the ith
          nonlinear constraint function c  at the final iterate, for
                                         i
          i=1,2,...,NCNLN. If NCNLN = 0, then the array C is not
          referenced.

     15:  CJAC(NROWJ,*) -- DOUBLE PRECISION array         Input/Output
          Note: the second dimension of the array CJAC must be at
          least N for NCNLN >0 and 1 otherwise On entry: in general,
          CJAC need not be initialised before the call to E04UCF.
          However, if Derivative Level = 3, the user may optionally
          set the constant elements of CJAC (see parameter NSTATE in
          the description of CONFUN). Such constant elements need not
          be re-assigned on subsequent calls to CONFUN. If NCNLN = 0,
          then the array CJAC is not referenced. On exit: if NCNLN >
          0, CJAC contains the Jacobian matrix of the nonlinear
          constraint functions at the final iterate, i.e., CJAC(i,j)
          contains the partial derivative of the ith constraint
          function with respect to the jth variable, for i=1,2,...,
          NCNLN; j = 1,2,...,N. (See the discussion of parameter CJAC
          under CONFUN.)

     16:  CLAMDA(N+NCLIN+NCNLN) -- DOUBLE PRECISION array Input/Output
          On entry: CLAMDA need not be initialised if E04UCF is called
          with the (default) Cold Start option. With the Warm Start
          option, CLAMDA must contain a multiplier estimate for each
          nonlinear constraint with a sign that matches the status of
          the constraint specified by the ISTATE array (as above). The
          ordering of CLAMDA is as follows; the first n elements
```

contain the multipliers for the bound constraints on the variables, elements n+1 through n+n contain the multipliers
                                  L
for the general linear constraints, and elements n+n +1
                                                    L
through n+n +n contain the multipliers for the nonlinear
          L N
constraints. If the jth constraint is defined as 'inactive' by the initial value of the ISTATE array, CLAMDA(j) should be zero; if the jth constraint is an inequality active at its lower bound, CLAMDA(j) should be non-negative; if the j th constraint is an inequality active at its upper bound, CLAMDA(j) should be non-positive. On exit: the values of the QP multipliers from the last QP subproblem. CLAMDA(j) should be non-negative if ISTATE(j) = 1 and non-positive if ISTATE(j) = 2.

17: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function, F(x), at the final iterate.

18: OBJGRD(N) -- DOUBLE PRECISION array Output
On exit: the gradient (or its finite-difference approximation) of the objective function at the final iterate.

19: R(NROWR,N) -- DOUBLE PRECISION array Input/Output
On entry: R need not be initialised if E04UCF is called with a Cold Start option (the default), and will be taken as the identity. With a Warm Start R must contain the upper-triangular Cholesky factor R of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements not in the upper-triangular part of R are assumed to be zero and need not be assigned. On exit: if Hessian = No, (the default; see Section 5.1), R
                                                    T~
contains the upper-triangular Cholesky factor R of Q HQ, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (6) in Section 3). If Hessian = Yes, R contains the upper-triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

20: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an initial estimate of the solution. On exit: the final estimate of the solution.

21: IWORK(LIWORK) -- INTEGER array Workspace

22: LIWORK -- INTEGER Input

```
     On entry:
     the dimension of the array IWORK as declared in the
     (sub)program from which E04UCF is called.
     Constraint: LIWORK>=3*N+NCLIN+2*NCNLN.

23:  WORK(LWORK) -- DOUBLE PRECISION array              Workspace

24:  LWORK -- INTEGER                                      Input
     On entry:
     the dimension of the array WORK as declared in the
     (sub)program from which E04UCF is called.
     Constraints:
     if NCLIN = NCNLN = 0 then
           LWORK >=20*N

     if NCNLN = 0 and NCLIN > 0 then
                      2
           LWORK >=2*N +20*N+11*NCLIN

     if NCNLN > 0 and NCLIN >= 0 then
                    2
           LWORK>=2*N +N*NCLIN+20*N*NCNLN+20*N+ 11*NCLIN+21*NCNLN

     If Major Print Level > 0, the required amounts of workspace
     are output on the current advisory message channel (see
     X04ABF). As an alternative to computing LIWORK and LWORK
     from the formulas given above, the user may prefer to obtain
     appropriate values from the output of a preliminary run with
     a positive value of Major Print Level and LIWORK and LWORK
     set to 1. (E04UCF will then terminate with IFAIL = 9.)

25:  IUSER(*) -- INTEGER array                      User Workspace
     Note: the dimension of the array IUSER must be at least 1.
     IUSER is not used by E04UCF, but is passed directly to
     routines CONFUN and OBJFUN and may be used to pass
     information to those routines.

26:  USER(*) -- DOUBLE PRECISION array              User Workspace
     Note: the dimension of the array USER must be at least 1.
     USER is not used by E04UCF, but is passed directly to
     routines CONFUN and OBJFUN and may be used to pass
     information to those routines.

27:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).
```

```
For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit.

E04UCF returns with IFAIL = 0 if the iterates have
converged to a point x that satisfies the first-order Kuhn-
Tucker conditions to the accuracy requested by the optional
parameter Optimality Tolerance (see Section 5.1), i.e., the
projected gradient and active constraint residuals are
negligible at x.

The user should check whether the following four conditions
are satisfied:
(i)   the final value of Norm Gz is significantly less than
      that at the starting point;

(ii)  during the final major iterations, the values of Step
      and ItQP are both one;

(iii) the last few values of both Norm Gz and Norm C become
      small at a fast linear rate;

(iv)  Cond Hz is small.
If all these conditions hold, x is almost certainly a local
minimum of (1). (See Section 9 for a specific example.)
```

5.1. Optional Input Parameters

Several optional parameters in E04UCF define choices in the behaviour of the routine. In order to reduce the number of formal parameters of E04UCF these optional parameters have associated default values (see Section 5.1.3) that are appropriate for most problems. Therefore the user need only specify those optional parameters whose values are to be different from their default values.

The remainder of this section can be skipped by users who wish to use the default values for all optional prameters. A complete list of optional parameters and their default values is given in Section 5.1.3

5.1.1. Specification of the optional parameters

Optional parameters may be specified by calling one, or both, of E04UDF and E04UEF prior to a call to E04UCF.

E04UDF reads options from an external options file, with Begin and End as the first and last lines respectively and each

intermediate line defining a single optional parameter. For example,

```
Begin
  Print Level = 1
End
```

The call

```
CALL E04UDF (IOPTNS, INFORM)
```

can then be used to read the file on unit IOPTNS. INFORM will be zero on successful exit. E04UDF should be consulted for a full description of this method of supplying optional parameters.

E04UEF can be called directly to supply options, one call being necessary for each optional parameter. For example,

```
CALL E04UEF ('Print level = 1')
```

E04UEF should be consulted for a full description of this method of supplying optional parameters.

All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user are unaltered by E04UCF (unless they define invalid values) and so remain in effect for subsequent calls to E04UCF, unless altered by the user.

5.1.2. Description of the optional parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and DOUBLE PRECISION values required with certain options. The number (epsilon) is a generic notation for machine precision (see X02AJF(*)), and $(\text{epsilon})^{R}$ denotes the relative precision of the objective function (the optional parameter Function Precision see below).

Central Difference Interval r Default values are computed

If the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate, the value of r is used as the difference interval for every

component of x. The use of finite-differences is discussed
further below under the optional parameter Difference Interval.

Cold Start Default = Cold Start

Warm Start

(AXIOM parameter STA, warm start when .TRUE.)

This option controls the specification of the initial working set
in both the procedure for finding a feasible point for the linear
constraints and bounds, and in the first QP subproblem
thereafter. With a Cold Start, the first working set is chosen by
E04UCF based on the values of the variables and constraints at
the initial point. Broadly speaking, the initial working set will
include equality constraints and bounds or inequality constraints
that violate or 'nearly' satisfy their bounds (within Crash
Tolerance; see below). With a Warm Start, the user must set the
ISTATE array and define CLAMDA and R as discussed in Section 5.
ISTATE values associated with bounds and linear constraints
determine the initial working set of the procedure to find a
feasible point with respect to the bounds and linear constraints.
ISTATE values associated with nonlinear constraints determine the
initial working set of the first QP subproblem after such a
feasible point has been found. E04UCF will override the user's
specification of ISTATE if necessary, so that a poor choice of
the working set will not cause a fatal error. A warm start will
be advantageous if a good estimate of the initial working set is
available - for example, when E04UCF is called repeatedly to
solve related problems.

Crash Tolerance r Default = 0.01

(AXIOM parameter CRA)

This value is used in conjunction with the optional parameter
Cold Start (the default value). When making a cold start, the QP
algorithm in E04UCF must select an initial working set. When r>=0
, the initial working set will include (if possible) bounds or
general inequality constraints that lie within r of their bounds.
                                        T
In particular, a constraint of the form a x>=l will be included
                                        j
                                  T
in the initial working set if |a x-l|<=r(1+|l|). If r<0 or r>1,
                                  j
the default value is used.

Defaults

```
This special keyword may be used to reset the default values
following a call to E04UCF.

Derivative Level i Default = 3

   (AXIOM parameter DER)

This parameter indicates which derivatives are provided by the
user in subroutines OBJFUN and CONFUN. The possible choices for i
are the following.

    i    Meaning

    3    All objective and constraint gradients are provided by
         the user.

    2    All of the Jacobian is provided, but some components of
         the objective gradient are not specified by the user.

    1    All elements of the objective gradient are known, but
         some elements of the Jacobian matrix are not specified
         by the user.

    0    Some elements of both the objective gradient and the
         Jacobian matrix are not specified by the user.

The value i=3 should be used whenever possible, since E04UCF is
more reliable and will usually be more efficient when all
derivatives are exact.

If i=0 or 2, E04UCF will estimate the unspecified components of
the objective gradient, using finite differences. The computation
of finite-difference approximations usually increases the total
run-time, since a call to OBJFUN is required for each unspecified
element. Furthermore, less accuracy can be attained in the
solution (see Chapter 8 of Gill et al [10], for a discussion of
limiting accuracy).

If i=0 or 1, E04UCF will approximate unspecified elements of the
Jacobian. One call to CONFUN is needed for each variable for
which partial derivatives are not available. For example, if the
Jacobian has the form

                          (* * * *)
                          (* ? ? *)
                          (* * ? *)
                          (* * * *)

where '*' indicates an element provided by the user and '?'
indicates an unspecified element, E04UCF will call CONFUN twice:
```

once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to CONFUN.)

At times, central differences are used rather than forward differences, in which case twice as many calls to OBJFUN and CONFUN are needed. (The switch to central differences is not under the user's control.)

Difference Interval r Default values are computed

(AXIOM parameter DIF)

This option defines an interval used to estimate gradients by finite differences in the following circumstances:

(a) For verifying the objective and/or constraint gradients (see the description of Verify, below).

(b) For estimating unspecified elements of the objective gradient of the Jacobian matrix.

In general, a derivative with respect to the jth variable is approximated using the interval $(delta)_j$, where $(delta)_j = r(1+|\hat{x}_j|)$ with $\hat{x}$ the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to O(r).

See Gill et al [10] for a discussion of the accuracy in finite-difference approximations.

If a difference interval is not specified by the user, a finite-difference interval will be computed automatically for each variable by a procedure that requires up to six calls of CONFUN and OBJFUN for each component. This option is recommended if the function is badly scaled or the user wishes to have E04UCF determine constant elements in the objective and constraint gradients (see the descriptions of CONFUN and OBJFUN in Section 5).

Feasibility Tolerance r Default = $\sqrt{(epsilon)}$

(AXIOM parameter FEA)

The scalar r defines the maximum acceptable absolute violations in linear and nonlinear constraints at a 'feasible' point; i.e.,

a constraint is considered satisfied if its violation does not exceed r. If r<(epsilon) or r>=1, the default value is used. Using this keyword sets both optional parameters Linear Feasibility Tolerance and Nonlinear Feasibility Tolerance to r, if (epsilon)<=r<1. (Additional details are given below under the descriptions of these parameters.)

Function Precision r Default = (epsilon) $^{0.9}$

(AXIOM parameter FUN)

This parameter defines (epsilon) $_{R}$, which is intended to be a measure of the accuracy with which the problem functions f and c can be computed. If r<(epsilon) or r>=1, the default value is used. The value of (epsilon) $_{R}$ should reflect the relative precision of 1+|F(x)|; i.e., (epsilon) $_{R}$ acts as a relative precision when |F| is large, and as an absolute precision when |F| is small. For example, if F(x) is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for (epsilon) $_{R}$ would be 1.0E-6. In contrast, if F(x) is typically of order 10^{-4} and the first six significant digits are known to be correct, an appropriate value for (epsilon) $_{R}$ would be 1.0E-10. The choice of (epsilon) $_{R}$ can be quite complicated for badly scaled problems; see Chapter 8 of Gill et al [10] for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of (epsilon) $_{R}$ should be large enough so that E04UCF will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Hessian No Default = No

Hessian Yes

(No AXIOM parameter - fixed as Yes)

This option controls the contents of the upper-triangular matrix R (see Section 5). E04UCF works exclusively with the transformed and re-ordered Hessian H (6), and hence extra computation is

```
                        Q
required to form the Hessian itself. If Hessian = No, R contains
the Cholesky factor of the transformed and re-ordered Hessian. If
Hessian = Yes the Cholesky factor of the approximate Hessian
itself is formed and stored in R. The user should select Hessian
= Yes if a warm start will be used for the next call to E04UCF.

                                   10
Infinite Bound Size r Default = 10

   (AXIOM parameter INFB)

If r>0, r defines the 'infinite' bound BIGBND in the definition
of the problem constraints. Any upper bound greater than or equal
to BIGBND will be regarded as plus infinity (and similarly for a
lower bound less than or equal to -BIGBND). If r<=0, the default
value is used.

                                              10
Infinite Step Size r Default = max(BIGBND,10  )

   (AXIOM parameter INFS)

If r>0, r specifies the magnitude of the change in variables that
is treated as a step to an unbounded solution. If the change in x
during an iteration would exceed the value of Infinite Step Size,
the objective function is considered to be unbounded below in the
feasible region. If r<=0, the default value is used.

Iteration limit i Default = max(50,3(n+n )+10n )
                                        L     N

See Major Iteration Limit below.


                                            ---------
Linear Feasibility Tolerance r  Default = \/(epsilon)
                              1

   (AXIOM parameter LINF)


                                              ---------
Nonlinear Feasibility Tolerance r  Default = \/(epsilon) if
                                 2

   (AXIOM parameter NONF)
                                   0.33
Derivative Level >= 2 and (epsilon)     otherwise

The scalars r  and r  define the maximum acceptable absolute
             1      2
```

violations in linear and nonlinear constraints at a 'feasible' point; i.e., a linear constraint is considered satisfied if its violation does not exceed r_1, and similarly for a nonlinear constraint and r_2. If $r_i <$ (epsilon) or $r_i \geq 1$, the default value is used, for i=1,2.

On entry to E04UCF, an iterative procedure is executed in order to find a point that satisfies the linear constraint and bounds on the variables to within the tolerance r_1. All subsequent iterates will satisfy the linear constraints to within the same tolerance (unless r_1 is comparable to the finite-difference interval).

For nonlinear constraints, the feasibility tolerance r_2 defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, the value of Nonlinear Feasibility Tolerance acts as a partial termination criterion for the iterative sequence generated by E04UCF (see the discussion of Optimality Tolerance).

These tolerances should reflect the precision of the corresponding constraints. For example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify r_1 as 10^{-6}.

Linesearch Tolerance r Default = 0.9

(AXIOM parameter LINT)

The value r ($0 \leq r < 1$) controls the accuracy with which the step (alpha) taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of r, the more accurate the linesearch). The default value r=0.9 requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If there are no nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations - for example, if the objective function is cheap to evaluate, or if a substantial number of gradients are

```
unspecified.

List Default = List

Nolist

   (AXIOM parameter LIST)

Normally each optional parameter specification is printed as it
is supplied. Nolist may be used to suppress the printing and List
may be used to restore printing.

Major Iteration Limit i Default = max(50,3(n+n )+10n )
                                                L     N

Iteration Limit

Iters

Itns

   (AXIOM parameter MAJI)

The value of i specifies the maximum number of major iterations
allowed before termination. Setting i=0 and Major Print Level> 0
means that the workspace needed will be computed and printed, but
no iterations will be performed.

Major Print level i Default = 10

Print Level

   (AXIOM parameter MAJP)

The value of i controls the amount of printout produced by the
major iterations of E04UCF. (See also Minor Print level below.)
The levels of printing are indicated below.

i        Output

0        No output.

1        The final solution only.

5        One line for each major iteration (no printout of the
         final solution).

>=10     The final solution and one line of output for each
         iteration.
```

```
>=20     At each major iteration, the objective function, the
         Euclidean norm of the nonlinear constraint violations,
         the values of the nonlinear constraints (the vector c),
         the values of the linear constraints (the vector A x),
                                                             L
         and the current values of the variables (the vector x).

>=30     At each major iteration, the diagonal elements of the
         matrix T associated with the TQ factorization (5) of the
         QP working set, and the diagonal elements of R, the
         triangular factor of the transformed and re-ordered
         Hessian (6).

Minor Iteration Limit i Default = max(50,3(n+n +n ))
                                           L  N

   (AXIOM parameter MINI)

The value of i specifies the maximum number of iterations for the
optimality phase of each QP subproblem.

Minor Print Level i Default = 0

   (AXIOM parameter MINP)

The value of i controls the amount of printout produced by the
minor iterations of E04UCF, i.e., the iterations of the quadratic
programming algorithm. (See also Major Print Level, above.) The
following levels of printing are available.

i        Output

0        No output.

1        The final QP solution.

5        One line of output for each minor iteration (no printout
         of the final QP solution).

>=10     The final QP solution and one brief line of output for
         each minor iteration.

>=20     At each minor iteration, the current estimates of the QP
         multipliers, the current estimate of the QP search
         direction, the QP constraint values, and the status of
         each QP constraint.

>=30     At each minor iteration, the diagonal elements of the
         matrix T associated with the TQ factorization (5) of the
         QP working set, and the diagonal elements of the
```

Cholesky factor R of the transformed Hessian (6).

Nonlinear Feasibility Tolerance r Default = $\sqrt{(\text{epsilon})}$

See Linear Feasibility Tolerance, above.

Optimality Tolerance r Default = $(\text{epsilon})^{0.8}$

(AXIOM parameter OPT)

The parameter r ($(\text{epsilon}) \le r < 1$) specifies the accuracy to which R the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, r indicates the number of correct figures desired in the objective function at the solution. For example, if r is 10^{-6} and E04UCF terminates successfully, the final value of F should have approximately six correct figures. If $r < (\text{epsilon})_R$ or $r \ge 1$ the default value is used.

E04UCF will terminate successfully if the iterative sequence of x -values is judged to have converged and the final point satisfies the first-order Kuhn-Tucker conditions (see Section 3). The sequence of iterates is considered to have converged at x if

$$(\text{alpha})\ ||p|| \le \sqrt{r}(1+||x||), \tag{8a}$$

where p is the search direction and (alpha) the step length from (3). An iterate is considered to satisfy the first-order conditions for a minimum if

$$||Z_{FR}^T g_{FR}|| \le \sqrt{r}(1+\max(1+|F(x)|, ||g_{FR}||)) \tag{8b}$$

and

$$|res_j| \le \text{ftol for all } j, \tag{8c}$$

where $Z_{FR}^T g_{FR}$ is the projected gradient (see Section 3), g_{FR} is the gradient of F(x) with respect to the free variables, res_j is the violation of the jth active nonlinear constraint, and ftol is the

Nonlinear Feasibility Tolerance.

Step Limit r Default = 2.0

(AXIOM parameter STE)

If r>0, r specifies the maximum change in variables at the first
step of the linesearch. In some cases, such as $F(x)=ae^{bx}$ or $F(x)=ax^{b}$, even a moderate change in the components of x can lead to floating-point overflow. The parameter r is therefore used to encourage evaluation of the problem functions at meaningful points. Given any major iterate x, the first point $\tilde{x}$ at which F and c are evaluated during the linesearch is restricted so that

$$||\tilde{x}-x||_2 \leq r(1+||x||_2).$$

The linesearch may go on and evaluate F and c at points further from x if this will result in a lower value of the merit function. In this case, the character L is printed at the end of the optional line of printed output, (see Section 5.2). If L is printed for most of the iterations, r should be set to a larger value.

Wherever possible, upper and lower bounds on x should be used to prevent evaluation of nonlinear functions at wild values. The default value Step Limit = 2.0 should not affect progress on well-behaved functions, but values 0.1 or 0.01 may be helpful when rapidly varying functions are present. If a small value of Step Limit is selected, a good starting point may be required. An important application is to the class of nonlinear least-squares problems. If r<=0, the default value is used.

Start Objective Check At Variable k Default = 1

(AXIOM parameter STAO)

Start Constraint Check At Variable k Default = 1

(AXIOM parameter STAC)

Stop Objective Check At Variable l Default = n

(AXIOM parameter STOO)

Stop Constraint Check At Variable l Default = n

```
   (AXIOM parameter STOC)

These keywords take effect only if Verify Level > 0 (see below).
They may be used to control the verification of gradient elements
computed by subroutines OBJFUN and CONFUN. For example, if the
first 30 components of the objective gradient appeared to be
correct in an earlier run, so that only component 31 remains
questionable, it is reasonable to specify Start Objective Check
At Variable 31. If the first 30 variables appear linearly in the
objective, so that the corresponding gradient elements are
constant, the above choice would also be appropriate.

Verify Level i Default = 0

Verify No

Verify Level - 1

Verify Level 0

Verify Objective Gradients

Verify Level 1

Verify Constraint Gradients

Verify Level 2

Verify

Verify Yes

Verify Gradients

Verify Level 3

   (AXIOM parameter VE)

These keywords refer to finite-difference checks on the gradient
elements computed by the user-provided subroutines OBJFUN and
CONFUN. (Unspecified gradient components are not checked.) It is
possible to specify Verify Levels 0-3 in several ways, as
indicated above. For example, the nonlinear objective gradient
(if any) will be verified if either Verify Objective Gradients or
Verify Level 1 is specified. Similarly, the objective and the
constraint gradients will be verified if Verify Yes or Verify
Level 3 or Verify is specified.

If 0<=i<=3, gradients will be verified at the first point that
```

satisfies the linear constraints and bounds. If i=0, only a ' cheap' test will be performed, requiring one call to OBJFUN and one call to CONFUN. If 1<=i<=3, a more reliable (but more expensive) check will be made on individual gradient components, within the ranges specified by the Start and Stop keywords described above. A result of the form OK or BAD? is printed by E04UCF to indicate whether or not each component appears to be correct.

If 10<=i<=13, the action is the same as for i - 10, except that it will take place at the user-specified initial value of x.

We suggest that Verify Level 3 be specified whenever a new function routine is being developed.

5.1.3. Optional parameter checklist and default values

For easy reference, the following list shows all the valid keywords and their default values. The symbol (epsilon) represents the machine precision (see X02AJF(*)).

Optional Parameters	Default Values
Central difference interval	Computed automatically
Cold/Warm start	Cold start
Crash tolerance	0.01
Defaults	
Derivative level	3
Difference interval	Computed automatically
Feasibility tolerance	$\sqrt{(\text{epsilon})}$
Function precision	$(\text{epsilon})^{0.9}$
Hessian	No
Infinite bound size	10^{10}

```
Infinite step size       10

                           ---------
Linear feasibility       \/(epsilon)
tolerance

Linesearch tolerance     0.9

List/Nolist              List

Major iteration limit    max(50,3(n+n )+10n )
                                   L     N

Major print level        10

Minor iteration limit    max(50,3(n+n +n ))
                                   L  N

Minor print level        0

                           ---------
Nonlinear feasibility    \/(epsilon) if Derivative Level >= 2
tolerance                                  0.33
                         otherwise (epsilon)

                                  0.8
Optimality tolerance     (epsilon)
                                  R

Step limit               2.0

Start objective check    1

Start constraint check   1

Stop objective check     n

Stop constraint check    n

Verify level             0
```

5.2. Description of Printed Output

The level of printed output from E04UCF is controlled by the user (see the description of Major Print Level and Minor Print Level in Section 5.1). If Minor Print Level > 0, output is obtained from the subroutines that solve the QP subproblem. For a detailed description of this information the reader should refer to E04NCF(*).

```
When Major Print Level >= 5, the following line of output is
produced at every major iteration of E04UCF. In all cases, the
values of the quantities printed are those in effect on
completion of the given iteration.

Itn            is the iteration count.

ItQP           is the sum of the iterations required by the
               feasibility and optimality phases of the QP
               subproblem. Generally, ItQP will be 1 in the later
               iterations, since theoretical analysis predicts
               that the correct active set will be identified
               near the solution (see Section 3).

               Note that ItQP may be greater than the Minor
               Iteration Limit if some iterations are required
               for the feasibility phase.

Step           is the step (alpha) taken along the computed
               search direction. On reasonably well-behaved
               problems, the unit step will be taken as the
               solution is approached.

Nfun           is the cumulative number of evaluations of the
               objective function needed for the linesearch.
               Evaluations needed for the estimation of the
               gradients by finite differences are not included.
               Nfun is printed as a guide to the amount of work
               required for the linesearch.

Merit          is the value of the augmented Lagrangian merit
               function (12) at the current iterate. This
               function will decrease at each iteration unless it
               was necessary to increase the penalty parameters
               (see Section 8.2). As the solution is approached,
               Merit will converge to the value of the objective
               function at the solution.

               If the QP subproblem does not have a feasible
               point (signified by I at the end of the current
               output line), the merit function is a large
               multiple of the constraint violations, weighted by
               the penalty parameters. During a sequence of major
               iterations with infeasible subproblems, the
               sequence of Merit values will decrease
               monotonically until either a feasible subproblem
               is obtained or E04UCF terminates with IFAIL = 3
               (no feasible point could be found for the
               nonlinear constraints).
```

If no nonlinear constraints are present (i.e., NCNLN = 0), this entry contains Objective, the value of the objective function F(x). The objective function will decrease monotonically to its optimal value when there are no nonlinear constraints.

Bnd — is the number of simple bound constraints in the predicted active set.

Lin — is the number of general linear constraints in the predicted active set.

Nln — is the number of nonlinear constraints in the predicted active set (not printed if NCNLN is zero).

Nz — is the number of columns of Z (see Section 8.1). The value of Nz is the number of variables minus the number of constraints in the predicted active set; i.e., Nz = n-(Bnd + Lin + Nln).

Norm Gf — is the Euclidean norm of g_{FR}, the gradient of the objective function with respect to the free variables, i.e.,variables not currently held at a bound.

Norm Gz — is $||Z^T g_{FR}||$, the Euclidean norm of the projected gradient (see Section 8.1). Norm Gz will be approximately zero in the neighbourhood of a solution.

Cond H — is a lower bound on the condition number of the Hessian approximation H.

Cond Hz — is a lower bound on the condition number of the projected Hessian approximation H_z ($H_z = Z^T H Z = R^T R$; see (6) and (12) in Sections 3 and 8.1). The larger this number, the more difficult the problem.

Cond T — is a lower bound on the condition number of the matrix of predicted active constraints.

Norm C — is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if NCNLN is zero). Norm C will be approximately zero in the neighbourhood of a solution.

Penalty — is the Euclidean norm of the vector of penalty parameters used in the augumented Lagrangian merit function (not printed if NCNLN is zero).

Conv — is a three-letter indication of the status of the three convergence tests (8a)-(8c) defined in the description of the optional parameter Optimality Tolerance in Section 5.1 Each letter is T if the test is satisfied, and F otherwise. The three tests indicate whether:

(a) the sequence of iterates has converged;

(b) the projected gradient (Norm Gz) is sufficiently small; and

(c) the norm of the residuals of constraints in the predicted active set (Norm C) is small enough.

If any of these indicators is F when E04UCF terminates with IFAIL = 0, the user should check the solution carefully.

M — is printed if the Quasi-Newton update was modified to ensure that the Hessian approximation is positive-definite (see Section 8.3).

I — is printed if the QP subproblem has no feasible point.

C — is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of Step is zero, the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is resolved with the central-difference gradient and Jacobian.) If the value of Step is non-zero, central differences were computed because Norm Gz and Norm C imply that x is close to a Kuhn-Tucker point.

L — is printed if the linesearch has produced a relative change in x greater than the value defined by the optional parameter Step Limit. If this output occurs frequently during later

```
                iterations of the run, Step Limit should be set to
                a larger value.

R               is printed if the approximate Hessian has been
                refactorized. If the diagonal condition estimator
                of R indicates that the approximate Hessian is
                badly conditioned, the approximate Hessian is
                refactorized using column interchanges. If
                necessary, R is modified so that its diagonal
                condition estimator is bounded.

When Major Print Level = 1 or Major Print Level >= 10, the
summary printout at the end of execution of E04UCF includes a
listing of the status of every variable and constraint. Note that
default names are assigned to all variables and constraints.

The following describes the printout for each variable.

Varbl           gives the name (V) and index j=1,2,...,n of the
                variable.

State           gives the state of the variable in the predicted
                active set (FR if neither bound is in the active
                set, EQ if a fixed variable, LL if on its lower
                bound, UL if on its upper bound). If the variable
                is predicted to lie outside its upper or lower
                bound by more than the feasibility tolerance,
                State will be ++ or -- respectively. (The latter
                situation can occur only when there is no feasible
                point for the bounds and linear constraints.)

Value           is the value of the variable at the final
                iteration.

Lower bound     is the lower bound specified for the variable.
                (None indicates that BL(j)<=- BIGBND.)

Upper bound     is the upper bound specified for the variable.
                (None indicates that BL(j)>=BIGBND.)

Lagr Mult       is the value of the Lagrange-multiplier for the
                associated bound constraint. This will be zero if
                State is FR. If x is optimal, the multiplier
                should be non-negative if State is LL, and non-
                positive if State is UL.

Residual        is the difference between the variable Value and
                the nearer of its bounds BL(j) and BU(j).

The printout for general constraints is the same as for
```

```
variables, except for the following:

L Con      is the name (L) and index i, for i = 1,2,...,NCLIN of
           a linear constraint.

N Con      is the name (N) and index i, for i = 1,2,...,NCNLN of
           a nonlinear constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

The input data for E04UCF should always be checked (even if
E04UCF terminates with IFAIL=0).

Note that when Print Level>0, a short description of IFAIL is
printed.

Errors and diagnostics indicated by IFAIL, together with some
recommendations for recovery are indicated below.

IFAIL= 1
     The final iterate x satisfies the first-order Kuhn-Tucker
     conditions to the accuracy requested, but the sequence of
     iterates has not yet converged. E04UCF was terminated
     because no further improvement could be made in the merit
     function.

     This value of IFAIL may occur in several circumstances. The
     most common situation is that the user asks for a solution
     with accuracy that is not attainable with the given
     precision of the problem (as specified by Function Precision
     see Section 5). This condition will also occur if, by
     chance, an iterate is an 'exact' Kuhn-Tucker point, but the
     change in the variables was significant at the previous
     iteration. (This situation often happens when minimizing
     very simple functions, such as quadratics.)

     If the four conditions listed in Section 5 for IFAIL = 0 are
     satisfied, x is likely to be a solution of (1) even if IFAIL
     = 1.

IFAIL= 2
     E04UCF has terminated without finding a feasible point for
     the linear constraints and bounds, which means that no
     feasible point exists for the given value of Linear
     Feasibility Tolerance (see Section 5.1). The user should
```

```
     check that there are no constraint redundancies. If the data
     for the constraints are accurate only to an absolute
     precision (sigma), the user should ensure that the value of
     the optional parameter Linear Feasibility Tolerance is
     greater than (sigma). For example, if all elements of A are
     of order unity and are accurate to only three decimal
                                                                -3
     places, Linear Feasibility Tolerance should be at least 10  .

IFAIL= 3
     No feasible point could be found for the nonlinear
     constraints. The problem may have no feasible solution. This
     means that there has been a sequence of QP subproblems for
     which no feasible point could be found (indicated by I at
     the end of each terse line of output). This behaviour will
     occur if there is no feasible point for the nonlinear
     constraints. (However, there is no general test that can
     determine whether a feasible point exists for a set of
     nonlinear constraints.) If the infeasible subproblems occur
     from the very first major iteration, it is highly likely
     that no feasible point exists. If infeasibilities occur when
     earlier subproblems have been feasible, small constraint
     inconsistencies may be present. The user should check the
     validity of constraints with negative values of ISTATE. If
     the user is convinced that a feasible point does exist,
     E04UCF should be restarted at a different starting point.

IFAIL= 4
     The limiting number of iterations (determined by the
     optional parameter Major Iteration Limit see Section 5.1)
     has been reached.

     If the algorithm appears to be making progress, Major
     Iteration Limit may be too small. If so, increase its value
     and rerun E04UCF (possibly using the Warm Start option). If
     the algorithm seems to be 'bogged down', the user should
     check for incorrect gradients or ill-conditioning as
     described below under IFAIL = 6.

     Note that ill-conditioning in the working set is sometimes
     resolved automatically by the algorithm, in which case
     performing additional iterations may be helpful. However,
     ill-conditioning in the Hessian approximation tends to
     persist once it has begun, so that allowing additional
     iterations without altering R is usually inadvisable. If the
     quasi-Newton update of the Hessian approximation was
     modified during the latter iterations (i.e., an M occurs at
     the end of each terse line), it may be worthwhile to try a
     warm start at the final point as suggested above.
```

```
IFAIL= 6
     x does not satisfy the first-order Kuhn-Tucker conditions,
     and no improved point for the merit function could be found
     during the final line search.

     A sufficient decrease in the merit function could not be
     attained during the final line search. This sometimes occurs
     because an overly stringent accuracy has been requested,
     i.e., Optimality Tolerance is too small. In this case the
     user should apply the four tests described under IFAIL = 0
     above to determine whether or not the final solution is
     acceptable (see Gill et al [10], for a discussion of the
     attainable accuracy).

     If many iterations have occurred in which essentially no
     progress has been made and E04UCF has failed completely to
     move from the initial point then subroutines OBJFUN or
     CONFUN may be incorrect. The user should refer to comments
     below under IFAIL = 7 and check the gradients using the
     Verify parameter. Unfortunately, there may be small errors
     in the objective and constraint gradients that cannot be
     detected by the verification process. Finite-difference
     approximations to first derivatives are catastrophically
     affected by even small inaccuracies. An indication of this
     situation is a dramatic alteration in the iterates if the
     finite-difference interval is altered. One might also
     suspect this type of error if a switch is made to central
     differences even when Norm Gz and Norm C are large.

     Another possibility is that the search direction has become
     inaccurate because of ill-conditioning in the Hessian
     approximation or the matrix of constraints in the working
     set; either form of ill-conditioning tends to be reflected
     in large values of ItQP (the number of iterations required
     to solve each QP subproblem).

     If the condition estimate of the projected Hessian (Cond Hz)
     is extremely large, it may be worthwhile to rerun E04UCF
     from the final point with the Warm Start option. In this
     situation, ISTATE should be left unaltered and R should be
     reset to the identity matrix.

     If the matrix of constraints in the working set is ill-
     conditioned (i.e., Cond T is extremely large), it may be
     helpful to run E04UCF with a relaxed value of the
     Feasibility Tolerance (Constraint dependencies are often
     indicated by wide variations in size in the diagonal
     elements of the matrix T, whose diagonals will be printed
     for Major Print Level >= 30).
```

IFAIL= 7
The user-provided derivatives of the objective function and/or nonlinear constraints appear to be incorrect.

Large errors were found in the derivatives of the objective function and/or nonlinear constraints. This value of IFAIL will occur if the verification process indicated that at least one gradient or Jacobian component had no correct figures. The user should refer to the printed output to determine which elements are suspected to be in error.

As a first-step, the user should check that the code for the objective and constraint values is correct - for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values x=0 or x=1 are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Special care should be used in this test if computation of the objective function involves subsidiary data communicated in COMMON storage. Although the first evaluation of the function may be correct, subsequent calculations may be in error because some of the subsidiary data has accidently been overwritten.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single-precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

IFAIL= 9
An input parameter is invalid. The user should refer to the printed output to determine which parameter must be redefined.

IFAILOverflow
If the printed output before the overflow error contains a warning about serious ill-conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of the optional parameter Linear Feasiblity Tolerance or Nonlinear

```
     Feasiblity Tolerance, and rerunning the program. If the
     message recurs even after this change, the offending
     linearly dependent constraint (with index 'j') must be
     removed from the problem. If overflow occurs in one of the
     user-supplied routines (e.g. if the nonlinear functions
     involve exponentials or singularities), it may help to
     specify tighter bounds for some of the variables (i.e.,
     reduce the gap between appropriate l  and u ).
                                         j      j
```

7. Accuracy

If IFAIL = 0 on exit then the vector returned in the array X is an estimate of the solution to an accuracy of approximately Feasiblity Tolerance (see Section 5.1), whose default value is $(\text{epsilon})^{0.8}$, where (epsilon) is the machine precision (see X02AJF(*)).

8. Further Comments

In this section we give some further details of the method used by E04UCF.

8.1. Solution of the Quadratic Programming Subproblem

The search direction p is obtained by solving (4) using the method of E04NCF(*) (Gill et al [8]), which was specifically designed to be used within an SQP algorithm for nonlinear programming.

The method of E04UCF is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the feasibility phase), and minimizing the quadratic objective function within the feasible region (the optimality phase). The computations in both phases are perfomed by the same subroutines. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let p denote the current estimate of the solution of (4); the new iterate $\bar{p}$ is defined by

$$\bar{p}=p+(\text{sigma})d, \qquad (9)$$

where, as in (3), (sigma) is a non-negative step length and d is

a search direction.

At the beginning of each iteration of E04UCF, a working set is defined of constraints (general and bound) that are satisfied exactly. The vector d is then constructed so that the values of constraints in the working set remain unaltered for any move along d. For a bound constraint in the working set, this property is achieved by setting the corresponding component of d to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the components associated with the fixed and free variables.

Let C denote the sub-matrix of rows of

$$\begin{pmatrix} A \\ L \\ A \\ N \end{pmatrix}$$

corresponding to general constraints in the working set. The general constraints in the working set will remain unaltered if

$$C_{FR}\, d_{FR} = 0, \tag{10}$$

which is equivalent to defining d_{FR} as

$$d_{FR} = Z d_z \tag{11}$$

for some vector d_z, where Z is the matrix associated with the TQ factorization (5) of C_{FR}.

The definition of d_z in (11) depends on whether the current p is feasible. If not, d_z is zero except for a component (gamma) in the jth position, where j and (gamma) are chosen so that the sum of infeasibilities is decreasing along d. (For further details, see Gill et al [8].) In the feasible case, d_z satisfies the equations

$$R_z^T R_z d_z = -Z_{FR}^T q, \tag{12}$$

```
                                    T
where R  is the Cholesky factor of Z H  Z and q is the gradient
       z                             FR
                                                          T
of the quadratic objective function (q=g+Hp). (The vector Z q
                                                            FR
is the projected gradient of the QP.) With (12), P+d is the
minimizer of the quadratic objective function subject to treating
the constraints in the working set as equalities.

If the QP projected gradient is zero, the current point is a
constrained stationary point in the subspace defined by the
working set. During the feasiblity phase, the projected gradient
will usually be zero only at a vertex (although it may vanish at
non-vertices in the presence of constraint dependencies). During
the optimality phase, a zero projected gradient implies that p
minimizes the quadratic objective function when the constraints
in the working set are treated as equalities. In either case,
Lagrange multipliers are computed. Given a positive constant
(delta) of the order of the machine precision, the Lagrange
multiplier (mu)  corresponding to an inequality constraint in the
              j
working set at its upper bound is said to be optimal if
(mu) <=(delta) when the jth constraint is at its upper bound, or
    j
if (mu) >=-(delta) when the associated constraint is at its lower
       j
bound. If any multiplier is non-optimal, the current objective
function (either the true objective or the sum of
infeasibilities) can be reduced by deleting the corresponding
constraint from the working set.

If optimal multipliers occur during the feasibility phase and the
sum of infeasibilities is non-zero, no feasible point exists. The
QP algorithm will then continue iterating to determine the
minimum sum of infeasibilities. At this point, the Lagrange
multiplier (mu)  will satisfy -(1+(delta))<=(mu) <=(delta) for an
              j                                  j
inequality constraint at its upper bound, and
-(delta)<=(mu) <=1+(delta) for an inequality at its lower bound.
             j
The Lagrange multiplier for an equality constraint will satisfy
|(mu) |<=1+(delta).
     j

The choice of step length (sigma) in the QP iteration (9) is
based on remaining feasible with respect to the satisfied
constraints. During the optimality phase, if p+d is feasible,
(sigma) will be taken as unity. (In this case, the projected
            -
```

gradient at p will be zero.) Otherwise, (sigma) is set to
(sigma) , the step to the 'nearest'constraint, which is added to
       M
the working set at the next iteration.

Each change in the working set leads to a simple change to C :
                                                            FR
if the status of a general constraint changes, a row of C   is
                                                         FR
altered; if a bound constraint enters or leaves the working set,
a column of C   changes. Explicit representations are recurred of
            FR
                                                T   T
the matrices T, Q   and R, and of the vectors Q q and Q g.
                FR

8.2. The Merit Function

After computing the search direction as described in Section 3,
each major iteration proceeds by determining a step length
(alpha) in (3) that produces a 'sufficient decrease' in the
augmented Lagrangian merit function

                      --
L(x,(lambda),s)=F(x)- > (lambda) (c (x)-s)
                      --   i i   i
                      i

                      1 --   2
                    + - > (rho) (c (x)-s) ,   (13)
                      2 --   i i   i
                        i

where x, (lambda) and s vary during the linsearch. The summation
terms in (13) involve only the nonlinear constraints. The vector
(lambda) is an estimate of the Lagrange multipliers for the
nonlinear constraints of (1). The non-negative slack variables
{s } allow nonlinear inequality constraints to be treated without
  i
introducing discontinuities. The solution of the QP subproblem
(4) provides a vector triple that serves as a direction of search
for the three sets of variables. The non-negative vector (rho) of
penalty parameters is initialised to zero at the beginning of the
first major iteration. Thereafter, selected components are
increased whenever necessary to ensure descent for the merit
function. Thus, the sequence of norms of (rho) (the printed
quantity Penalty, see Section 5.2) is generally non-decreasing,
although each (rho) may be reduced a limited number of times.
                  i

The merit function (13) and its global convergence properties are described in Gill et al [9].

8.3. The Quasi-Newton Update

The matrix H in (4) is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel [3].) At the end of each major iteration, a new Hessian approximation $\bar{H}$ is defined as a rank-two modification of H. In E04UCF, the BFGS quasi-Newton update is used:

$$\bar{H} = H - \frac{1}{s^T H s} H s s^T H + \frac{1}{y^T s} y y^T, \qquad (14)$$

where $\bar{s} = \bar{x} - x$ (the change in x).

In E04UCF, H is required to be positive-definite. If H is positive-definite, $\bar{H}$ defined by (14) will be positive-definite if and only if $y^T s$ is positive (see, e.g. Dennis and More [1]). Ideally, y in (14) would be taken as y_L, the change in gradient of the Lagrangian function

$$y_L = \bar{g} - \bar{A}^T(\mu)_N - g + A^T(\mu)_N, \qquad (15)$$

where $(\mu)_N$ denotes the QP multipliers associated with the nonlinear constraints of the original problem. If $y^T s$ is not sufficiently positive, an attempt is made to perform the update with a vector y of the form

$$y = y + \sum_{i=1}^{m_N} (\omega)_i \left(a_i(\bar{x})c_i(\bar{x}) - a_i(x)c_i(x)\right),$$

where (omega) >=0. If no such vector can be found, the update is
           i
perfomed with a scaled y ; in this case, M is printed to indicate
                        L
that the update is modified.

Rather than modifying H itself, the Cholesky factor of the
transformed Hessian H (6) is updated, where Q is the matrix from
                     Q
(5) associated with the active set of the QP subproblem. The
update (13) is equivalent to the following update to H :
                                                      Q

$$\bar{H}_Q = H_Q - \frac{H_Q s_Q s_Q^T H_Q}{s_Q^T H_Q s_Q} + \frac{y_Q y_Q^T}{y_Q^T s_Q}, \qquad (16)$$

          T   T
where y =Q y, and s =Q s. This update may be expressed as a rank-
       Q   Q
one update to R (see Dennis and Schnabel [2]).

9. Example

This section describes one version of the so-called 'hexagon'
problem (a different formulation is given as Problem 108 in Hock
and Schittkowski [11]). The problem is to determine the hexagon
of maximum area such that no two of its vertices are more than
one unit apart (the solution is not a regular hexagon).

All constraint types are included (bounds, linear, nonlinear),
and the Hessian of the Lagrangian function is not positive-
definite at the solution. The problem has nine variables, non-
infinite bounds on seven of the variables, four general linear
constraints, and fourteen nonlinear constraints.

The objective function is

$$F(x) = -x_2 x_6 + x_1 x_7 - x_3 x_7 - x_5 x_8 + x_4 x_9 + x_3 x_8 .$$

The bounds on the variables are

$$x_1 \geq 0,\ -1 \leq x_3 \leq 1,\ x_5 \geq 0,\ x_6 \geq 0,\ x_7 \geq 0,\ x_8 \leq 0,\ \text{and } x_9 \leq 0.$$

Thus,

```
                                                          T
          l =(0,-infty,-1,-infty,0,0,0,-infty,-infty)
           B

                                                           T
         u =(infty,infty,1,infty,infty,infty,infty,0,0)
          B
```

The general linear constraints are

$$x_2 - x_1 \geq 0, \quad x_3 - x_2 \geq 0, \quad x_4 - x_3 \geq 0, \text{ and } x_5 - x_4 \geq 0.$$

Hence,

$$l_B = \begin{pmatrix} (0) \\ (0) \\ (0) \end{pmatrix}, \quad A = \begin{pmatrix} (-1\ \ 1\ 0\ \ 0\ \ 0\ 0\ 0\ 0\ 0) \\ (\ 0\ -1\ 1\ \ 0\ \ 0\ 0\ 0\ 0\ 0) \\ (\ 0\ \ 0\ 1\ -1\ \ 0\ 0\ 0\ 0\ 0) \end{pmatrix} \text{ and } u_B = \begin{pmatrix} (\text{infty}) \\ (\text{infty}) \\ (\text{infty}) \end{pmatrix}.$$

The nonlinear constraints are all of the form $c_i(x) \leq 1$, for $i = 1, 2, \ldots, 14$; hence, all components of l_N are $-\infty$, and all components of u_N are 1. The fourteen functions $\{c_i(x)\}$ are

$$c_1(x) = x_1^2 + x_6^2,$$

$$c_2(x) = (x_2 - x_1)^2 + (x_7 - x_6)^2,$$

$$c_3(x) = (x_3 - x_1)^2 + x_6^2,$$

$$c_4(x) = (x_4 - x_1)^2 + (x_8 - x_6)^2,$$

$$c_5(x) = (x_5 - x_1)^2 + (x_9 - x_6)^2,$$

$$c_6(x) = x_2^2 + x_7^2,$$

```
                2  2
      c (x)=(x -x ) +x ,
       7      3  2    7

                2        2
      c (x)=(x -x ) +(x -x ) ,
       8      4  2     8  7

                2        2
      c (x)=(x -x ) +(x -x ) ,
       9      2  5     7  9

                 2  2
      c  (x)=(x -x ) +x ,
       10      4  3    8

                 2  2
      c  (x)=(x -x ) +x ,
       11      5  3    9

              2  2
      c  (x)=x +x ,
       12     4  8

                 2        2
      c  (x)=(x -x ) +(x -x ) ,
       13      4  5     9  8

              2  2
      c  (x)=x +x .
       14     5  9
```

An optimal solution (to five figures) is

```
 *
x =(0.060947,0.59765,1.0,0.59765,0.060947,0.34377,0.5,
                T
    -0.5,0.34377) ,

       *
and F(x )=-1.34996. (The optimal objective function is unique,
but is achieved for other values of x.) Five nonlinear
                                                    *
constraints and one simple bound are active at x . The sample
solution output is given later in this section, following the
sample main program and problem definition.

Two calls are made to E04UCF in order to demonstrate some of its
```

features. For the first call, the starting point is:

$$x_0 = (0.1, 0.125, 0.666666, 0.142857, 0.111111, 0.2, 0.25, -0.2, -0.25)^T.$$

All objective and constraint derivatives are specified in the user-provided subroutines OBJFN1 and CONFN1, i.e., the default option Derivative Level =3 is used.

On completion of the first call to E04UCF, the optimal variables are perturbed to produce the initial point for a second run in which the problem functions are defined by the subroutines OBJFN2 and CONFN2. To illustrate one of the finite-difference options in E04UCF, these routines are programmed so that the first six components of the objective gradient and the constant elements of the Jacobian matrix are not specified; hence, the option Derivative Level =0 is chosen. During computation of the finite-difference intervals, the constant Jacobian elements are identified and set, and E04UCF automatically increases the derivative level to 2.

The second call to E04UCF illustrates the use of the Warm Start Level option to utilize the final active set, nonlinear multipliers and approximate Hessian from the first run. Note that Hessian = Yes was specified for the first run so that the array R would contain the Cholesky factor of the approximate Hessian of the Lagrangian.

The two calls to E04UCF illustrate the alternative methods of assigning default parameters. (There is no special significance in the order of these assignments; an options file may just as easily be used to modify parameters set by E04UEF.)

The results are typical of those obtained from E04UCF when solving well behaved (non-trivial) nonlinear problems. The approximate Hessian and working set remain relatively well-conditioned. Similarly the penalty parameters remain small and approximately constant. The numerical results illustrate much of the theoretically predicted behaviour of a quasi-Newton SQP method. As x approaches the solution, only one minor iteration is perfomed per major iteration, and the Norm Gz and Norm C columns exhibit the fast linear convergence rate mentioned in Sections 5 and 6. Note that the constraint violations converge earlier than the projected gradient. The final values of the project gradient norm and constraint norm reflect the limiting accuracy of the two quantities. It is possible to achieve almost full precision in the constraint norm but only half precision in the projected gradient norm. Note that the final accuracy in the nonlinear constraints is considerably better than the feasibility

```
tolerance, because the constraint violations are being refined
during the last few iterations while the algorithm is working to
reduce the projected gradient norm. In this problem, the
constraint values and Lagrange multipliers at the solution are '
well balanced', i.e., all the multipliers are approximately the
same order of magnitude. The behaviour is typical of a well-
scaled problem.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
E04 -- Minimizing or Maximizing a Function                E04UDF
        E04UDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

To supply optional parameters to E04UCF from an external file.

2. Specification

       SUBROUTINE E04UDF (IOPTNS, INFORM)
       INTEGER          IOPTNS, INFORM

3. Description

E04UDF may be used to supply values for optional parameters to
E04UCF. E04UDF reads an external file and each line of the file
defines a single optional parameter. It is only necessary to
supply values for those parameters whose values are to be
different from their default values.

Each optional parameter is defined by a single character string
of up to 72 characters, consisting of one or more items. The
items associated with a given option must be separated by spaces,
or equal signs (=). Alphabetic characters may be upper or lower
case. The string

       Print level = 1

is an example of a string used to set an optional parameter. For
each option the string contains one or more of the following
items:
```

```
(a)   A mandatory keyword.

(b)   A phrase that qualifies the keyword.

(c)   A number that specifies an INTEGER or real value. Such
      numbers may be up to 16 contiguous characters in Fortran
      77's I, F, E or D formats, terminated by a space if this is
      not the last item on the line.

Blank strings and comments are ignored. A comment begins with an
asterisk (*) and all subsequent characters in the string are
regarded as part of the comment.

The file containing the options must start with begin and must
finish with end An example of a valid options file is:

      Begin * Example options file
       Print level =10
      End

Normally each line of the file is printed as it is read, on the
current advisory message unit (see X04ABF), but printing may be
suppressed using the keyword nolist To suppress printing of begin,
nolist must be the first option supplied as in the file:

      Begin
        Nolist
        Print level = 10
      End

Printing will automatically be turned on again after a call to
E04UCF and may be turned on again at any time by the user by
using the keyword list.

Optional parameter settings are preserved following a call to
E04UCF, and so the keyword defaults is provided to allow the user
to reset all the optional parameters to their default values
prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations,
synonyms and default values is given in Section 5.1 of the
document for E04UCF.

4. References

None.

5. Parameters
```

```
 1:  IOPTNS -- INTEGER                                        Input
     On entry: IOPTNS must be the unit number of the options
     file. Constraint: 0 <= IOPTNS <= 99.

 2:  INFORM -- INTEGER                                       Output
     On exit: INFORM will be zero, if an options file with the
     current structure has been read. Otherwise INFORM will be
     positive. Positive values of INFORM indicate that an options
     file may not have been successfully read as follows:
     INFORM = 1
           IOPTNS is not in the range [0,99].

     INFORM = 2
           begin was found, but end-of-file was found before end
           was found.

     INFORM = 3
           end-of-file was found before begin was found.

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning
message is output on the current advisory message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UEF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.
```

%%%

```
E04 -- Minimizing or Maximizing a Function                   E04UEF
        E04UEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

To supply individual optional parameters to E04UCF.
```

2. Specification

```
SUBROUTINE E04UEF (STRING)
CHARACTER*(*)    STRING
```

3. Description

E04UEF may be used to supply values for optional parameters to E04UCF. It is only necessary to call E04UEF for those parameters whose values are to be different from their default values. One call to E04UEF sets one parameter value.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

```
Print level = 1
```

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist Thus the statement

```
CALL E04UEF ('Nolist')
```

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04UCF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to

E04UCF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

```
CALL E04UEF ('Defaults')
```

prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the document for E04UCF.

4. References

None.

5. Parameters

1: STRING -- CHARACTER*(*) Input
On entry: STRING must be a single valid option string. See Section 3 above and Section 5.1 of the routine document for E04UCF. On entry: STRING must be a single valid option string. See Section 3 above and Section 5.1 of the routine document for E04UCF.

6. Error Indicators and Warnings

If the parameter STRING is not recognised as a valid option string, then a warning message is output on the current advisory message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UDF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.

%%%

E04 -- Minimizing or Maximizing a Function E04YCF
E04YCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details.

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04YCF returns estimates of elements of the variance-covariance matrix of the estimated regression coefficients for a nonlinear least squares problem. The estimates are derived from the Jacobian of the function f(x) at the solution.

This routine may be used following any one of the nonlinear least-squares routines E04FCF(*), E04FDF, E04GBF(*), E04GCF, E04GDF(*), E04GEF(*), E04HEF(*), E04HFF(*).

2. Specification

```
      SUBROUTINE E04YCF (JOB, M, N, FSUMSQ, S, V, LV, CJ, WORK,
     1                   IFAIL)
      INTEGER          JOB, M, N, LV, IFAIL
      DOUBLE PRECISION FSUMSQ, S(N), V(LV,N), CJ(N), WORK(N)
```

3. Description

E04YCF is intended for use when the nonlinear least-squares function, $F(x)=f^T(x)f(x)$, represents the goodness of fit of a nonlinear model to observed data. The routine assumes that the Hessian of $F(x)$, at the solution, can be adequately approximated by $2J^TJ$, where J is the Jacobian of $f(x)$ at the solution. The estimated variance-covariance matrix C is then given by

$$C=(\sigma)^2 (J^TJ)^{-1} J^T \text{ } J \text{ non-singular},$$

where $(\sigma)^2$ is the estimated variance of the residual at the solution, x, given by

$$(\sigma)^2 = \frac{F(x)}{m-n},$$

m being the number of observations and n the number of variables.

The diagonal elements of C are estimates of the variances of the

estimated regression coefficients. See the Chapter Introduction E04 and Bard [1] and Wolberg [2] for further information on the use of C.

T
When $J\ J$ is singular then C is taken to be

$$C=(sigma)^2\ (J\ J)^{T\ *},$$

$T\ *$ T
where $(J\ J)$ is the pseudo-inverse of $J\ J$, but in this case the parameter IFAIL is returned as non-zero as a warning to the user that J has linear dependencies in its columns. The assumed rank of J can be obtained from IFAIL.

The routine can be used to find either the diagonal elements of C, or the elements of the jth column of C, or the whole of C.

E04YCF must be preceded by one of the nonlinear least-squares routines mentioned in Section 1, and requires the parameters FSUMSQ, S and V to be supplied by those routines. FSUMSQ is the

residual sum of squares F(x), and S and V contain the singular values and right singular vectors respectively in the singular value decomposition of J. S and V are returned directly by the comprehensive routines E04FCF(*), E04GBF(*), E04GDF(*) and E04HEF(*), but are returned as part of the workspace parameter W from the easy-to-use routines E04FDF, E04GCF, E04GEF(*) and E04HFF(*). In the case of E04FDF, S starts at W(NS), where

$$NS=6*N+2*M+M*N+1+max(1,N*(N-1)/2)$$

and in the cases of the remaining easy-to-use routines, S starts at W(NS), where

$$NS=7*N+2*M+M*N+N*(N+1)/2+1+max(1,N*(N-1)/2)$$

The parameter V starts immediately following the elements of S, so that V starts at W(NV), where

$$NV=NS+N.$$

For all the easy-to-use routines the parameter LV must be supplied as N. Thus a call to E04YCF following E04FDF can be illustrated as

```
CALL E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
```

```
      NS = 6*N + 2*M + M*N + 1 + MAX((1,(N*(N-1))/2)
      NV = NS + N
      CALL E04YCF (JOB, M, N, FSUMSQ, W(NS), W(NV),
     *             N, CJ, WORK, IFAIL)
```

where the parameters M, N, FSUMSQ and the $(n+n^2)$ elements W(NS), WS(NS+1),..., W(NV+N*N-1) must not be altered between the calls to E04FDF and E04YCF. The above illustration also holds for a call to E04YCF following a call to one of E04GCF, E04GEF(*), E04HFF(*) except that NS must be computed as

```
      NS = 7*N + 2*M + M*N + (N*(N+1))/2 + 1 + MAX((1,N*(N-1))/2)
```

4. References

[1] Bard Y (1974) Nonlinear Parameter Estimation. Academic Press.

[2] Wolberg J R (1967) Prediction Analysis. Van Nostrand.

5. Parameters

1: JOB -- INTEGER Input
On entry: which elements of C are returned as follows:
JOB = -1
The n by n symmetric matrix C is returned.

JOB = 0
The diagonal elements of C are returned.

JOB > 0
The elements of column JOB of C are returned.
Constraint: -1 <= JOB <= N.

2: M -- INTEGER Input
On entry: the number m of observations (residuals $f_i(x)$).
Constraint: M >= N.

3: N -- INTEGER Input
On entry: the number n of variables (x_j). Constraint: 1 <= N <= M.

4: FSUMSQ -- DOUBLE PRECISION Input

On entry: the sum of squares of the residuals, F(x), at the

```
    solution x, as returned by the nonlinear least-squares
    routine. Constraint: FSUMSQ >= 0.0.

 5: S(N) -- DOUBLE PRECISION array                          Input
    On entry: the n singular values of the Jacobian as returned
    by the nonlinear least-squares routine. See Section 3 for
    information on supplying S following one of the easy-to-use
    routines.

 6: V(LV,N) -- DOUBLE PRECISION array              Input/Output
    On entry: the n by n right-hand orthogonal matrix (the
    right singular vectors) of J as returned by the nonlinear
    least-squares routine. See Section 3 for information on
    supplying V following one of the easy-to-use routines. On
    exit: when JOB >= 0 then V is unchanged.

    When JOB = -1 then the leading n by n part of V is
    overwritten by the n by n matrix C. When E04YCF is called
    with JOB = -1 following an easy-to-use routine this means
                                                        2
    that C is returned, column by column, in the n  elements of
                                          2
    W given by W(NV),W(NV+1),...,W(NV+N -1). (See Section 3 for
    the definition of NV).

 7: LV -- INTEGER                                           Input
    On entry:
    the first dimension of the array V as declared in the
    (sub)program from which E04YCF is called.
    When V is passed in the workspace parameter W following one
    of the easy-to-use least-square routines, LV must be the
    value N.

 8: CJ(N) -- DOUBLE PRECISION array                        Output
    On exit: with JOB = 0, CJ returns the n diagonal elements
    of C.

    With JOB = j>0, CJ returns the n elements of the jth column
    of C.

    When JOB = -1, CJ is not referenced.

 9: WORK(N) -- DOUBLE PRECISION array                  Workspace
    When JOB = -1 or 0 then WORK is used as internal workspace.

    When JOB > 0, WORK is not referenced.

10: IFAIL -- INTEGER                               Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
```

```
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit. To suppress
     the output of an error message when soft failure occurs, set
     IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
     On entry JOB < -1,

     or       JOB > N,

     or       N < 1,

     or       M < N,

     or       FSUMSQ < 0.0.

IFAIL= 2
     The singular values are all zero, so that at the solution
     the Jacobian matrix J has rank 0.

IFAIL> 2
     At the solution the Jacobian matrix contains linear, or near
     linear, dependencies amongst its columns. In this case the
     required elements of C have still been computed based upon J
     having an assumed rank given by (IFAIL-2). The rank is
     computed by regarding singular values SV(j) that are not
     larger than 10*(epsilon)*SV(1) as zero, where (epsilon) is
     the machine precision (see X02AJF(*)). Users who expect near
     linear dependencies at the solution and are happy with this
     tolerance in determining rank should call E04YCF with IFAIL
     = 1 in order to prevent termination by P01ABF(*). It is then
     essential to test the value of IFAIL on exit from E04YCF.

IFAILOverflow
     If overflow occurs then either an element of C is very
     large, or the singular values or singular vectors have been
     incorrectly supplied.
```

7. Accuracy

The computed elements of C will be the exact covariances corresponding to a closely neighbouring Jacobian matrix J.

8. Further Comments

When JOB = -1 the time taken by the routine is approximately proportional to n^3. When JOB >= 0 the time taken by the routine is approximately proportional to n^2.

9. Example

To estimate the variance-covariance matrix C for the least-squares estimates of x_1, x_2 and x_3 in the model

$$y = x_1 + \frac{t_1}{x_2 t_2 + x_3 t_3}$$

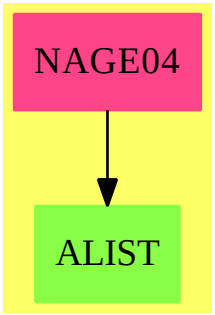
using the 15 sets of data given in the following table:

y	t_1	t_2	t_3
0.14	1.0	15.0	1.0
0.18	2.0	14.0	2.0
0.22	3.0	13.0	3.0
0.25	4.0	12.0	4.0
0.29	5.0	11.0	5.0
0.32	6.0	10.0	6.0
0.35	7.0	9.0	7.0
0.39	8.0	8.0	8.0
0.37	9.0	7.0	7.0
0.58	10.0	6.0	6.0
0.73	11.0	5.0	5.0
0.96	12.0	4.0	4.0
1.34	13.0	3.0	3.0
2.10	14.0	2.0	2.0
4.39	15.0	1.0	1.0

The program uses (0.5,1.0,1.5) as the initial guess at the position of the minimum and computes the least-squares solution using E04FDF. See the routine document E04FDF for further information.

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.9.1 NagOptimisationPackage (NAGE04)



Exports:
e04dgf e04fdf e04gcf e04jaf e04mbf
e04naf e04ucf e04ycf

— package NAGE04 NagOptimisationPackage —

```
)abbrev package NAGE04 NagOptimisationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:09 1994
++ Description:
++ This package uses the NAG Library to perform optimization.
++ An optimization problem involves minimizing a function (called
++ the objective function) of several variables, possibly subject to
++ restrictions on the values of the variables defined by a set of
++ constraint functions. The routines in the NAG Foundation Library
++ are concerned with function minimization only, since the problem
++ of maximizing a given function can be transformed into a
++ minimization problem simply by multiplying the function by -1.

NagOptimisationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    e04dgf : (Integer,DoubleFloat,DoubleFloat,Integer,_
```

```
    DoubleFloat,Boolean,DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
    Integer,Matrix DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp49(OBJFUN))) -> Result
 ++ e04dgf(n,es,fu,it,lin,list,ma,op,pr,sta,sto,ve,x,ifail,objfun)
 ++ minimizes an unconstrained nonlinear function of several
 ++ variables using a pre-conditioned, limited memory quasi-Newton
 ++ conjugate gradient method. First derivatives are required. The
 ++ routine is intended for use on large scale problems.
 ++ See \downlink{Manual Page}{manpageXXe04dgf}.
e04fdf : (Integer,Integer,Integer,Integer,_
    Matrix DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp50(LSFUN1))) -> Result
 ++ e04fdf(m,n,liw,lw,x,ifail,lsfun1)
 ++ is an easy-to-use algorithm for finding an unconstrained
 ++ minimum of a sum of squares of m nonlinear functions in n
 ++ variables (m>=n). No derivatives are required.
 ++ See \downlink{Manual Page}{manpageXXe04fdf}.
e04gcf : (Integer,Integer,Integer,Integer,_
    Matrix DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp19(LSFUN2))) -> Result
 ++ e04gcf(m,n,liw,lw,x,ifail,lsfun2)
 ++ is an easy-to-use quasi-Newton algorithm for finding an
 ++ unconstrained minimum of m nonlinear
 ++ functions in n variables (m>=n). First derivatives are required.
 ++ See \downlink{Manual Page}{manpageXXe04gcf}.
e04jaf : (Integer,Integer,Integer,Integer,_
    Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp24(FUNCT1))) -> Result
 ++ e04jaf(n,ibound,liw,lw,bl,bu,x,ifail,funct1)
 ++ is an easy-to-use quasi-Newton algorithm for finding a
 ++ minimum of a function F(x ,x ,...,x ), subject to fixed upper and
 ++                          1  2     n
 ++ lower bounds of the independent variables x ,x ,...,x , using
 ++                                            1  2     n
 ++ function values only.
 ++ See \downlink{Manual Page}{manpageXXe04jaf}.
e04mbf : (Integer,Integer,Integer,Integer,_
    Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Boolean,Integer,Integer,_
    Matrix DoubleFloat,Integer) -> Result
 ++ e04mbf(itmax,msglvl,n,nclin,nctotl,nrowa,a,bl,bu,
 ++        cvec,linobj,liwork,lwork,x,ifail)
 ++ is an easy-to-use routine for solving linear programming
 ++ problems, or for finding a feasible point for such problems. It
 ++ is not intended for large sparse problems.
 ++ See \downlink{Manual Page}{manpageXXe04mbf}.
e04naf : (Integer,Integer,Integer,Integer,Integer,Integer,Integer,_
    Integer,DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Boolean,Boolean,Boolean,Integer,Integer,_
```

```
      Matrix DoubleFloat,Matrix Integer,Integer,_
      Union(fn:FileName,fp:Asp20(QPHESS))) -> Result
   ++ e04naf(itmax,msglvl,n,nclin,nctotl,nrowa,nrowh,ncolh,bigbnd,a,bl,
   ++ bu,cvec,featol,hess,cold,lpp,orthog,liwork,lwork,x,istate,ifail,qphess)
   ++ is a comprehensive
   ++ programming (QP) or linear programming (LP) problems. It is not
   ++ intended for large sparse problems.
   ++ See \downlink{Manual Page}{manpageXXe04naf}.
  e04ucf : (Integer,Integer,Integer,Integer,Integer,Integer,_
      Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
      Integer,Boolean,DoubleFloat,Integer,DoubleFloat,DoubleFloat,Boolean,_
      DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Boolean,Integer,_
      Integer,Integer,Integer,Integer,DoubleFloat,DoubleFloat,DoubleFloat,_
      Integer,Integer,Integer,Integer,Integer,Matrix Integer,_
      Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
      Matrix DoubleFloat,Integer,Union(fn:FileName,fp:Asp55(CONFUN)),_
      Union(fn:FileName,fp:Asp49(OBJFUN))) -> Result
   ++ e04ucf(n,nclin,ncnln,nrowa,nrowj,nrowr,a,bl,bu,liwork,lwork,sta,
   ++        cra,der,fea,fun,hes,infb,infs,linf,lint,list,maji,majp,mini,
   ++        minp,mon,nonf,opt,ste,stao,stac,stoo,stoc,ve,istate,cjac,
   ++        clamda,r,x,ifail,confun,objfun)
   ++ is designed to minimize an arbitrary smooth function
   ++ subject to constraints on the
   ++ variables, linear constraints.
   ++ (E04UCF  may be used for unconstrained, bound-constrained and
   ++ linearly constrained optimization.) The user must provide
   ++ subroutines that define the objective and constraint functions
   ++ and as many of their first partial derivatives as possible.
   ++ Unspecified derivatives are approximated by finite differences.
   ++ All matrices are treated as dense, and hence E04UCF is not
   ++ intended for large sparse problems.
   ++ See \downlink{Manual Page}{manpageXXe04ucf}.
  e04ycf : (Integer,Integer,Integer,DoubleFloat,_
      Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer) -> Result
   ++ e04ycf(job,m,n,fsumsq,s,lv,v,ifail)
   ++ returns estimates of elements of the variance
   ++ matrix of the estimated regression coefficients for a nonlinear
   ++ least squares problem. The estimates are derived from the
   ++ Jacobian of the function f(x) at the solution.
   ++ See \downlink{Manual Page}{manpageXXe04ycf}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
```

```
import FortranPackage
import Union(fn:FileName,fp:Asp49(OBJFUN))
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)


e04dgf(nArg:Integer,esArg:DoubleFloat,fuArg:DoubleFloat,_
    itArg:Integer,linArg:DoubleFloat,listArg:Boolean,_
    maArg:DoubleFloat,opArg:DoubleFloat,prArg:Integer,_
    staArg:Integer,stoArg:Integer,veArg:Integer,_
    xArg:Matrix DoubleFloat,ifailArg:Integer,_
    objfunArg:Union(fn:FileName,fp:Asp49(OBJFUN))): Result ==
    pushFortranOutputStack(objfunFilename := aspFilename "objfun")$FOP
    if objfunArg case fn
      then outputAsFortran(objfunArg.fn)
      else outputAsFortran(objfunArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([objfunFilename]$Lisp,_
    "e04dgf",_
    ["n"::S,"es"::S,"fu"::S,"it"::S,"lin"::S_
    ,"list"::S,"ma"::S,"op"::S,"pr"::S,"sta"::S_
    ,"sto"::S,"ve"::S,"iter"::S,"objf"::S,"ifail"::S_
    ,"objfun"::S,"objgrd"::S,"x"::S,"iwork"::S,"work"::S,"iuser"::S_
    ,"user"::S]$Lisp,_
    ["iter"::S,"objf"::S,"objgrd"::S,"iwork"::S,"work"::S,"iuser"::S,_
    "user"::S,"objfun"::S]$Lisp,_
    [["double"::S,"es"::S,"fu"::S,"lin"::S,"ma"::S_
    ,"op"::S,"objf"::S,["objgrd"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp,_
    ["work"::S,["*"::S,13$Lisp,"n"::S]$Lisp]$Lisp,["user"::S,"*"::S]$Lisp_
    ,"objfun"::S]$Lisp_
    ,["integer"::S,"n"::S,"it"::S,"pr"::S,"sta"::S_
    ,"sto"::S,"ve"::S,"iter"::S,"ifail"::S,["iwork"::S,_
    ["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp,["iuser"::S,"*"::S]$Lisp]$Lisp_
    ,["logical"::S,"list"::S]$Lisp_
    ]$Lisp,_
    ["iter"::S,"objf"::S,"objgrd"::S,"x"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,esArg::Any,fuArg::Any,itArg::Any,linArg::Any,_
    listArg::Any,maArg::Any,opArg::Any,prArg::Any,staArg::Any,_
    stoArg::Any,veArg::Any,ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e04fdf(mArg:Integer,nArg:Integer,liwArg:Integer,_
    lwArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
    lsfun1Arg:Union(fn:FileName,fp:Asp50(LSFUN1))): Result ==
    pushFortranOutputStack(lsfun1Filename := aspFilename "lsfun1")$FOP
    if lsfun1Arg case fn
```

```
      then outputAsFortran(lsfun1Arg.fn)
      else outputAsFortran(lsfun1Arg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([lsfun1Filename]$Lisp,_
    "e04fdf",_
    ["m"::S,"n"::S,"liw"::S,"lw"::S,"fsumsq"::S_
    ,"ifail"::S,"lsfun1"::S,"w"::S,"x"::S,"iw"::S]$Lisp,_
    ["fsumsq"::S,"w"::S,"iw"::S,"lsfun1"::S]$Lisp,_
    [["double"::S,"fsumsq"::S,["w"::S,"lw"::S]$Lisp_
    ,["x"::S,"n"::S]$Lisp,"lsfun1"::S]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"liw"::S,"lw"::S_
    ,"ifail"::S,["iw"::S,"liw"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["fsumsq"::S,"w"::S,"x"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,nArg::Any,liwArg::Any,lwArg::Any,_
    ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e04gcf(mArg:Integer,nArg:Integer,liwArg:Integer,_
    lwArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
    lsfun2Arg:Union(fn:FileName,fp:Asp19(LSFUN2))): Result ==
    pushFortranOutputStack(lsfun2Filename := aspFilename "lsfun2")$FOP
    if lsfun2Arg case fn
              then outputAsFortran(lsfun2Arg.fn)
              else outputAsFortran(lsfun2Arg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([lsfun2Filename]$Lisp,_
    "e04gcf",_
    ["m"::S,"n"::S,"liw"::S,"lw"::S,"fsumsq"::S_
    ,"ifail"::S,"lsfun2"::S,"w"::S,"x"::S,"iw"::S]$Lisp,_
    ["fsumsq"::S,"w"::S,"iw"::S,"lsfun2"::S]$Lisp,_
    [["double"::S,"fsumsq"::S,["w"::S,"lw"::S]$Lisp_
    ,["x"::S,"n"::S]$Lisp,"lsfun2"::S]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"liw"::S,"lw"::S_
    ,"ifail"::S,["iw"::S,"liw"::S]$Lisp]$Lisp]$Lisp,_
    ["fsumsq"::S,"w"::S,"x"::S,"ifail"::S]$Lisp,_
    [([mArg::Any,nArg::Any,liwArg::Any,lwArg::Any,ifailArg::Any,_
    xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e04jaf(nArg:Integer,iboundArg:Integer,liwArg:Integer,_
    lwArg:Integer,blArg:Matrix DoubleFloat,buArg:Matrix DoubleFloat,_
    xArg:Matrix DoubleFloat,ifailArg:Integer,_
    funct1Arg:Union(fn:FileName,fp:Asp24(FUNCT1))): Result ==
    pushFortranOutputStack(funct1Filename := aspFilename "funct1")$FOP
    if funct1Arg case fn
      then outputAsFortran(funct1Arg.fn)
      else outputAsFortran(funct1Arg.fp)
```

```
        popFortranOutputStack()$FOP
        [(invokeNagman([funct1Filename]$Lisp,_
        "e04jaf",_
        ["n"::S,"ibound"::S,"liw"::S,"lw"::S,"f"::S_
        ,"ifail"::S,"funct1"::S,"bl"::S,"bu"::S,"x"::S,"iw"::S,"w"::S_
        ]$Lisp,_
        ["f"::S,"iw"::S,"w"::S,"funct1"::S]$Lisp,_
        [["double"::S,"f"::S,["bl"::S,"n"::S]$Lisp_
        ,["bu"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp,_
        ["w"::S,"lw"::S]$Lisp,"funct1"::S]$Lisp_
        ,["integer"::S,"n"::S,"ibound"::S,"liw"::S_
        ,"lw"::S,"ifail"::S,["iw"::S,"liw"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["f"::S,"bl"::S,"bu"::S,"x"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,iboundArg::Any,liwArg::Any,lwArg::Any,_
        ifailArg::Any,blArg::Any,buArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e04mbf(itmaxArg:Integer,msglvlArg:Integer,nArg:Integer,_
        nclinArg:Integer,nctotlArg:Integer,nrowaArg:Integer,_
        aArg:Matrix DoubleFloat,blArg:Matrix DoubleFloat,_
        buArg:Matrix DoubleFloat,_
        cvecArg:Matrix DoubleFloat,linobjArg:Boolean,liworkArg:Integer,_
        lworkArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "e04mbf",_
        ["itmax"::S,"msglvl"::S,"n"::S,"nclin"::S,"nctotl"::S_
        ,"nrowa"::S,"linobj"::S,"liwork"::S,"lwork"::S,"objlp"::S_
        ,"ifail"::S,"a"::S,"bl"::S,"bu"::S,"cvec"::S,"istate"::S_
        ,"clamda"::S,"x"::S,"iwork"::S,"work"::S]$Lisp,_
        ["istate"::S,"objlp"::S,"clamda"::S,"iwork"::S,"work"::S]$Lisp,_
        [["double"::S,["a"::S,"nrowa"::S,"n"::S]$Lisp_
        ,["bl"::S,"nctotl"::S]$Lisp,["bu"::S,"nctotl"::S]$Lisp,_
        ["cvec"::S,"n"::S]$Lisp,"objlp"::S,["clamda"::S,"nctotl"::S]$Lisp_
        ,["x"::S,"n"::S]$Lisp,["work"::S,"lwork"::S]$Lisp]$Lisp_
        ,["integer"::S,"itmax"::S,"msglvl"::S,"n"::S_
        ,"nclin"::S,"nctotl"::S,"nrowa"::S,"liwork"::S,"lwork"::S,_
        ["istate"::S,"nctotl"::S]$Lisp,"ifail"::S,_
        ["iwork"::S,"liwork"::S]$Lisp]$Lisp_
        ,["logical"::S,"linobj"::S]$Lisp]$Lisp,_
        ["istate"::S,"objlp"::S,"clamda"::S,"x"::S,"ifail"::S]$Lisp,_
        [([itmaxArg::Any,msglvlArg::Any,nArg::Any,nclinArg::Any,_
        nctotlArg::Any,nrowaArg::Any,linobjArg::Any,liworkArg::Any,_
        lworkArg::Any,ifailArg::Any,aArg::Any,blArg::Any,buArg::Any,_
        cvecArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    e04naf(itmaxArg:Integer,msglvlArg:Integer,nArg:Integer,_
```

```
    nclinArg:Integer,nctotlArg:Integer,nrowaArg:Integer,_
    nrowhArg:Integer,ncolhArg:Integer,bigbndArg:DoubleFloat,_
    aArg:Matrix DoubleFloat,blArg:Matrix DoubleFloat,_
    buArg:Matrix DoubleFloat,_
    cvecArg:Matrix DoubleFloat,featolArg:Matrix DoubleFloat,_
    hessArg:Matrix DoubleFloat,_
    coldArg:Boolean,lppArg:Boolean,orthogArg:Boolean,_
    liworkArg:Integer,lworkArg:Integer,xArg:Matrix DoubleFloat,_
    istateArg:Matrix Integer,ifailArg:Integer,_
    qphessArg:Union(fn:FileName,fp:Asp20(QPHESS))): Result ==
    pushFortranOutputStack(qphessFilename := aspFilename "qphess")$FOP
    if qphessArg case fn
      then outputAsFortran(qphessArg.fn)
      else outputAsFortran(qphessArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([qphessFilename]$Lisp,_
    "e04naf",_
    ["itmax"::S,"msglvl"::S,"n"::S,"nclin"::S,"nctotl"::S_
    ,"nrowa"::S,"nrowh"::S,"ncolh"::S,"bigbnd"::S,"cold"::S_
    ,"lpp"::S,"orthog"::S,"liwork"::S,"lwork"::S,"iter"::S_
    ,"obj"::S,"ifail"::S,"qphess"::S,"a"::S,"bl"::S,"bu"::S,_
    "cvec"::S,"featol"::S_
    ,"hess"::S,"clamda"::S,"x"::S,"istate"::S,"iwork"::S_
    ,"work"::S]$Lisp,_
    ["iter"::S,"obj"::S,"clamda"::S,"iwork"::S,"work"::S,_
    "qphess"::S]$Lisp,_
    [["double"::S,"bigbnd"::S,["a"::S,"nrowa"::S,"n"::S]$Lisp_
    ,["bl"::S,"nctotl"::S]$Lisp,["bu"::S,"nctotl"::S]$Lisp,_
    ["cvec"::S,"n"::S]$Lisp,["featol"::S,"nctotl"::S]$Lisp_
    ,["hess"::S,"nrowh"::S,"ncolh"::S]$Lisp,"obj"::S,_
    ["clamda"::S,"nctotl"::S]$Lisp,["x"::S,"n"::S]$Lisp,_
    ["work"::S,"lwork"::S]$Lisp_
    ,"qphess"::S]$Lisp_
    ,["integer"::S,"itmax"::S,"msglvl"::S,"n"::S_
    ,"nclin"::S,"nctotl"::S,"nrowa"::S,"nrowh"::S,"ncolh"::S,_
    "liwork"::S,"lwork"::S,"iter"::S,["istate"::S,"nctotl"::S]$Lisp_
    ,"ifail"::S,["iwork"::S,"liwork"::S]$Lisp]$Lisp_
    ,["logical"::S,"cold"::S,"lpp"::S,"orthog"::S]$Lisp_
    ]$Lisp,_
    ["iter"::S,"obj"::S,"clamda"::S,"x"::S,"istate"::S,"ifail"::S]$Lisp,_
    [([itmaxArg::Any,msglvlArg::Any,nArg::Any,nclinArg::Any,_
    nctotlArg::Any,nrowaArg::Any,nrowhArg::Any,ncolhArg::Any,_
    bigbndArg::Any,coldArg::Any,lppArg::Any,orthogArg::Any,_
    liworkArg::Any,lworkArg::Any,ifailArg::Any,aArg::Any,blArg::Any,_
    buArg::Any,cvecArg::Any,featolArg::Any,hessArg::Any,xArg::Any,_
    istateArg::Any ])@List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e04ucf(nArg:Integer,nclinArg:Integer,ncnlnArg:Integer,_
    nrowaArg:Integer,nrowjArg:Integer,nrowrArg:Integer,_
```

```
aArg:Matrix DoubleFloat,blArg:Matrix DoubleFloat,_
buArg:Matrix DoubleFloat,_
liworkArg:Integer,lworkArg:Integer,staArg:Boolean,_
craArg:DoubleFloat,derArg:Integer,feaArg:DoubleFloat,_
funArg:DoubleFloat,hesArg:Boolean,infbArg:DoubleFloat,_
infsArg:DoubleFloat,linfArg:DoubleFloat,lintArg:DoubleFloat,_
listArg:Boolean,majiArg:Integer,majpArg:Integer,_
miniArg:Integer,minpArg:Integer,monArg:Integer,_
nonfArg:DoubleFloat,optArg:DoubleFloat,steArg:DoubleFloat,_
staoArg:Integer,stacArg:Integer,stooArg:Integer,_
stocArg:Integer,veArg:Integer,istateArg:Matrix Integer,_
cjacArg:Matrix DoubleFloat,clamdaArg:Matrix DoubleFloat,_
rArg:Matrix DoubleFloat,_
xArg:Matrix DoubleFloat,ifailArg:Integer,_
confunArg:Union(fn:FileName,fp:Asp55(CONFUN)),_
objfunArg:Union(fn:FileName,fp:Asp49(OBJFUN))): Result ==
pushFortranOutputStack(confunFilename := aspFilename "confun")$FOP
if confunArg case fn
  then outputAsFortran(confunArg.fn)
  else outputAsFortran(confunArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(objfunFilename := aspFilename "objfun")$FOP
if objfunArg case fn
  then outputAsFortran(objfunArg.fn)
  else outputAsFortran(objfunArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([confunFilename,objfunFilename]$Lisp,_
"e04ucf",_
["n"::S,"nclin"::S,"ncnln"::S,"nrowa"::S,"nrowj"::S_
,"nrowr"::S,"liwork"::S,"lwork"::S,"sta"::S,"cra"::S_
,"der"::S,"fea"::S,"fun"::S,"hes"::S,"infb"::S_
,"infs"::S,"linf"::S,"lint"::S,"list"::S,"maji"::S_
,"majp"::S,"mini"::S,"minp"::S,"mon"::S,"nonf"::S_
,"opt"::S,"ste"::S,"stao"::S,"stac"::S,"stoo"::S_
,"stoc"::S,"ve"::S,"iter"::S,"objf"::S,"ifail"::S_
,"confun"::S,"objfun"::S,"a"::S,"bl"::S,"bu"::S,"c"::S,"objgrd"::S_
,"istate"::S,"cjac"::S,"clamda"::S,"r"::S,"x"::S_
,"iwork"::S,"work"::S,"iuser"::S,"user"::S]$Lisp,_
["iter"::S,"c"::S,"objf"::S,"objgrd"::S,"iwork"::S,"work"::S,_
"iuser"::S,"user"::S,"confun"::S,"objfun"::S]$Lisp,_
[["double"::S,["a"::S,"nrowa"::S,"n"::S]$Lisp_
,["bl"::S,["+"::S,["+"::S,"nclin"::S,"ncnln"::S]$Lisp,_
"n"::S]$Lisp]$Lisp,["bu"::S,["+"::S,["+"::S,"nclin"::S,_
"ncnln"::S]$Lisp,"n"::S]$Lisp]$Lisp_
,"cra"::S,"fea"::S,"fun"::S,"infb"::S,"infs"::S,"linf"::S,_
"lint"::S,"nonf"::S,"opt"::S,"ste"::S_
,["c"::S,"ncnln"::S]$Lisp,"objf"::S,["objgrd"::S,"n"::S]$Lisp,_
["cjac"::S,"nrowj"::S,"n"::S]$Lisp,["clamda"::S,["+"::S,_
["+"::S,"nclin"::S,"ncnln"::S]$Lisp,"n"::S]$Lisp]$Lisp_
,["r"::S,"nrowr"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp,_
```

```
    ["work"::S,"lwork"::S]$Lisp_
    ,["user"::S,1$Lisp]$Lisp,"confun"::S,"objfun"::S]$Lisp_
    ,["integer"::S,"n"::S,"nclin"::S,"ncnln"::S_
    ,"nrowa"::S,"nrowj"::S,"nrowr"::S,"liwork"::S,"lwork"::S,"der"::S,_
    "maji"::S,"majp"::S,"mini"::S,"minp"::S,"mon"::S,"stao"::S_
    ,"stac"::S,"stoo"::S,"stoc"::S,"ve"::S,"iter"::S,["istate"::S,_
    ["+"::S,["+"::S,"nclin"::S,"ncnln"::S]$Lisp,"n"::S]$Lisp]$Lisp_
    ,"ifail"::S,["iwork"::S,"liwork"::S]$Lisp,_
    ["iuser"::S,1$Lisp]$Lisp]$Lisp_
    ,["logical"::S,"sta"::S,"hes"::S,"list"::S]$Lisp_
    ]$Lisp,_
    ["iter"::S,"c"::S,"objf"::S,"objgrd"::S,"istate"::S,"cjac"::S,_
    "clamda"::S,"r"::S,"x"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,nclinArg::Any,ncnlnArg::Any,nrowaArg::Any,_
    nrowjArg::Any,nrowrArg::Any,liworkArg::Any,lworkArg::Any,_
    staArg::Any,craArg::Any,derArg::Any,feaArg::Any,funArg::Any,_
    hesArg::Any,infbArg::Any,infsArg::Any,linfArg::Any,lintArg::Any,_
    listArg::Any,majiArg::Any,majpArg::Any,miniArg::Any,minpArg::Any,_
    monArg::Any,nonfArg::Any,optArg::Any,steArg::Any,staoArg::Any,_
    stacArg::Any,stooArg::Any,stocArg::Any,veArg::Any,ifailArg::Any,_
    aArg::Any,blArg::Any,buArg::Any,istateArg::Any,cjacArg::Any,_
    clamdaArg::Any,rArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

e04ycf(jobArg:Integer,mArg:Integer,nArg:Integer,_
    fsumsqArg:DoubleFloat,sArg:Matrix DoubleFloat,lvArg:Integer,_
    vArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "e04ycf",_
    ["job"::S,"m"::S,"n"::S,"fsumsq"::S,"lv"::S_
    ,"ifail"::S,"s"::S,"cj"::S,"v"::S,"work"::S]$Lisp,_
    ["cj"::S,"work"::S]$Lisp,_
    [["double"::S,"fsumsq"::S,["s"::S,"n"::S]$Lisp_
    ,["cj"::S,"n"::S]$Lisp,["v"::S,"lv"::S,"n"::S]$Lisp,_
    ["work"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"job"::S,"m"::S,"n"::S,"lv"::S_
    ,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["cj"::S,"v"::S,"ifail"::S]$Lisp,_
    [([jobArg::Any,mArg::Any,nArg::Any,fsumsqArg::Any,lvArg::Any,_
    ifailArg::Any,sArg::Any,vArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— NAGE04.dotabb —

```
"NAGE04" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGE04"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE04" -> "ALIST"
```

———-

15.10 package NAGD02 NagOrdinaryDifferentialEquationsPackage

— NagOrdinaryDifferentialEquationsPackage.help —

```
D02(3NAG)          Foundation Library (12/10/92)          D02(3NAG)

     D02 -- Ordinary Differential Equations        Introduction -- D02
                              Chapter D02
                    Ordinary Differential Equations

     1. Scope of the Chapter

     This chapter is concerned with the numerical solution of ordinary
     differential equations. There are two main types of problem,
     those in which all boundary conditions are specified at one point
     (initial-value problems), and those in which the boundary
     conditions are distributed between two or more points (boundary-
     value problems and eigenvalue problems). Routines are available
     for initial-value problems, two-point boundary-value problems and
     Sturm-Liouville eigenvalue problems.

     2. Background to the Problems

     For most of the routines in this chapter a system of ordinary
     differential equations must be written in the form

                          y'  = f (x,y ,y ,...,y ),
                            1    1    1  2      n

                          y'  = f (x,y ,y ,...,y ),
                            2    2    1  2      n

                            .. ...

                          y'  = f (x,y ,y ,...y ),
                            n    n    1  2     n

     that is the system must be given in first-order form. The n
```

dependent variables (also, the solution) $y_1, y_2, \ldots, y_n$ are functions of the independent variable x, and the differential equations give expressions for the first derivatives $y_i' = dy_i/dx$ in terms of x and $y_1, y_2, \ldots, y_n$. For a system of n first-order equations, n associated boundary conditions are usually required to define the solution.

A more general system may contain derivatives of higher order, but such systems can almost always be reduced to the first-order form by introducing new variables. For example, suppose we have the third-order equation

$$z''' + zz'' + k(1 - z'^2) = 0.$$

We write $y_1 = z$, $y_2 = z'$, $y_3 = z''$, and the third order equation may then be written as the system of first-order equations

$$y_1' = y_2$$

$$y_2' = y_3$$

$$y_3' = -y_1 y_3 - k(1 - y_2^2).$$

For this system n = 3 and we require 3 boundary conditions in order to define the solution. These conditions must specify values of the dependent variables at certain points. For example, we have an initial-value problem if the conditions are:

$$y_1 = 0 \qquad \text{at } x = 0$$

$$y_2 = 0 \qquad \text{at } x = 0$$

$$y_3 = 0.1 \qquad \text{at } x = 0.$$

These conditions would enable us to integrate the equations numerically from the point x=0 to some specified end-point. We have a boundary-value problem if the conditions are:

```
y   =  0        at x=0
 1

y   =  0        at x=0
 2

y   =  1        at x=10.
 2
```

These conditions would be sufficient to define a solution in the range 0<=x<=10, but the problem could not be solved by direct integration (see Section 2.2). More general boundary conditions are permitted in the boundary-value case.

2.1. Initial-value Problems

To solve first-order systems, initial values of the dependent variables y_i, for i=1,2,...,n must be supplied at a given point, a. Also a point, b, at which the values of the dependent variables are required, must be specified. The numerical solution is then obtained by a step-by-step calculation which approximates values of the variables y_i, for i=1,2,...,n at finite intervals over the required range [a,b]. The routines in this chapter adjust the step length automatically to meet specified accuracy tolerances. Although the accuracy tests used are reliable over each step individually, in general an accuracy requirement cannot be guaranteed over a long range. For many problems there may be no serious accumulation of error, but for unstable systems small perturbations of the solution will often lead to rapid divergence of the calculated values from the true values. A simple check for stability is to carry out trial calculations with different tolerances; if the results differ appreciably the system is probably unstable. Over a short range, the difficulty may possibly be overcome by taking sufficiently small tolerances, but over a long range it may be better to try to reformulate the problem.

A special class of initial-value problems are those for which the solutions contain rapidly decaying transient terms. Such problems are called stiff; an alternative way of describing them is to say that certain eigenvalues of the Jacobian matrix (ddf_i/ddy_j) have large negative real parts when compared to others. These problems require special methods for efficient numerical solution; the methods designed for non-stiff problems when applied to stiff problems tend to be very slow, because they need small step lengths to avoid numerical instability. A full discussion is given in Hall and Watt [6] and a discussion of the methods for

stiff problems is given in Berzins, Brankin and Gladwell [1].

2.2. Boundary-value Problems

A full discussion of the design of the methods and codes for boundary-value problems is given in Gladwell [4]. In general, a system of nonlinear differential equations with boundary conditions given at two or more points cannot be guaranteed to have a solution. The solution has to be determined iteratively (if it exists). Finite-difference equations are set up on a mesh of points and estimated values for the solution at the grid points are chosen. Using these estimated values as starting values a Newton iteration is used to solve the finite-difference equations. The accuracy of the solution is then improved by deferred corrections or the addition of points to the mesh or a combination of both. Good initial estimates of the solution may be required in some cases but results may be difficult to compute when the solution varies very rapidly over short ranges. A discussion is given in Chapters 9 and 11 of Gladwell and Sayers [5] and Chapter 4 of Childs et al [2].

2.3. Eigenvalue Problems

Sturm-Liouville problems of the form

(p(x)y')'+q(x,(lambda))y=0

with appropriate boundary conditions given at two points, can be solved by a Scaled Pruefer method. In this method the differential equation is transformed to another which can be solved for a specified eigenvalue by a shooting method. A discussion is given in Chapter 11 of Gladwell and Sayers [5] and a complete description is given in Pryce [7].

2.6. References

[1] Berzins M, Brankin R W and Gladwell I (1987) Design of the Stiff Integrators in the NAG Library. Technical Report. TR14/87 NAG.

[2] (1979) Codes for Boundary-value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science. (ed Childs B, Scott M, Daniel J W, Denman E and Nelson P) 76 Springer-Verlag.

[3] Gladwell I (1979) Initial Value Routines in the NAG Library. ACM Trans Math Softw. 5 386--400.

[4] Gladwell I (1987) The NAG Library Boundary Value Codes. Numerical Analysis Report. 134 Manchester University.

[5] (1980) Computational Techniques for Ordinary Differential Equations. (Gladwell I and Sayers D K) Academic Press.

[6] Hall G and Watt J M (eds) (1976) Modern Numerical Methods for Ordinary Differential Equations. Clarendon Press.

[7] Pryce J D (1986) Error Estimation for Phase-function Shooting Methods for Sturm-Liouville Problems. J. Num. Anal. 6 103--123.

3. Recommendations on Choice and Use of Routines

There are no routines which deal directly with COMPLEX equations. These may however be transformed to larger systems of real equations of the required form. Split each equation into its real and imaginary parts and solve for the real and imaginary parts of each component of the solution. Whilst this process doubles the size of the system and may not always be appropriate it does make available for use the full range of routines provided presently.

3.1. Initial-value Problems

For simple first-order problems with low accuracy requirements, that is problems on a short range of integration, with derivative functions f_i which are inexpensive to calculate and where only a few correct figures are required, the best routines to use are likely to be the Runge-Kutta-Merson (RK) routines, D02BBF and D02BHF. For larger problems, over long ranges or with high accuracy requirements the variable-order, variable-step Adams routine D02CJF should usually be preferred. For stiff equations, that is those with rapidly decaying transient solutions, the Backward Differentiation Formula (BDF) variable-order, variable-step routine D02EJF should be used.

There are four routines for initial-value problems, two of which use the Runge-Kutta-Merson method:

D02BBF integrates a system of first order ordinary differential equations over a range with intermediate output and a choice of error control

D02BHF integrates a system of first order ordinary differential equations with a choice of error control until a position is determined where a function of the solution is zero.

one uses an Adams method:

D02CJF combines the functionality of D02BBF and D02BHF

and one uses a BDF method:

D02EJF combines the functionality of D02BBF and D02BHF.

3.2. Boundary-value Problems

D02GAF may be used for simple boundary-value problems with assigned boundary values. The user may find that convergence is difficult to achieve using D02GAF since only specifying the unknown boundary values and the position of the finite-difference mesh is permitted. In such cases the user may use D02RAF which permits specification of an initial estimate for the solution at all mesh points and allows the calculation to be influenced in other ways too. D02RAF is designed to solve a general nonlinear two-point boundary value problem with nonlinear boundary conditions.

A routine, D02GBF, is also supplied specifically for the general linear two-point boundary-value problem written in a standard

The user is advised to use interpolation routines from the E01 Chapter to obtain solution values at points not on the final mesh.

3.3. Eigenvalue Problems

There is one general purpose routine for eigenvalue problems, D02KEF. It may be used to solve regular or singular second-order Sturm-Liouville problems on a finite or infinite range. Discontinous coefficient functions can be treated and eigenfunctions can be computed.

D02 -- Ordinary Differential Equations Contents -- D02
Chapter D02

Ordinary Differential Equations

D02BBF ODEs, IVP, Runge-Kutta-Merson method, over a range, intermediate output

D02BHF ODEs, IVP, Runge-Kutta-Merson method, until function of solution is zero

D02CJF ODEs, IVP, Adams method, until function of solution is zero, intermediate output

D02EJF ODEs, stiff IVP, BDF method, until function of solution

```
                is zero, intermediate output

          D02GAF  ODEs, boundary value problem, finite difference technique
                  with deferred correction, simple nonlinear problem

          D02GBF  ODEs, boundary value problem, finite difference technique
                  with deferred correction, general linear problem

          D02KEF  2nd order Sturm-Liouville problem, regular/singular
                  system, finite/infinite range, eigenvalue and
                  eigenfunction, user-specified break-points

          D02RAF  ODEs, general nonlinear boundary value problem, finite
                  difference technique with deferred correction,
                  continuation facility
```

%%%

```
D02BBF(3NAG)                D02BBF                  D02BBF(3NAG)

D02 -- Ordinary Differential Equations                      D02BBF
        D02BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine
has been enhanced for use with AXIOM and is slightly different to
that offered in the standard version of the Foundation Library.

1. Purpose

D02BBF integrates a system of first-order ordinary differential
equations over an interval with suitable initial conditions,
using a Runge-Kutta-Merson method, and returns the solution at
points specified by the user.

2. Specification

       SUBROUTINE D02BBF (X, XEND, M, N, Y, TOL, IRELAB, RESULT,
      1                   FCN, OUTPUT, W, IFAIL)
       INTEGER          M, N, IRELAB, IFAIL
       DOUBLE PRECISION X, XEND, Y(N), TOL, W(N,7), RESULT(M,N)
       EXTERNAL         FCN, OUTPUT

3. Description

The routine integrates a system of ordinary differential
```

equations

$$y'_i = f_i(x, y_1, y_2, \ldots, y_n) \quad i = 1, 2, \ldots, n$$

from x = X to x = XEND using a Merson form of the Runge-Kutta method. The system is defined by a subroutine FCN supplied by the user, which evaluates f_i in terms of x and $y_1, y_2, \ldots, y_n$, and the values of $y_1, y_2, \ldots, y_n$ must be given at x = X.

The solution is returned via the user-supplied routine OUTPUT at a set of points specified by the user. This solution is obtained by quintic Hermite interpolation on solution values produced by the Runge-Kutta method.

The accuracy of the integration and, indirectly, the interpolation is controlled by the parameters TOL and IRELAB.

For a description of Runge-Kutta methods and their practical implementation see Hall and Watt [1].

4. References

[1] Hall G and Watt J M (eds) (1976) Modern Numerical Methods for Ordinary Differential Equations. Clarendon Press.

5. Parameters

1: X -- DOUBLE PRECISION Input/Output
On entry: X must be set to the initial value of the independent variable x. On exit: XEND, unless an error has occurred, when it contains the value of x at the error.

2: XEND -- DOUBLE PRECISION Input
On entry: the final value of the independent variable. If XEND < X on entry, integration will proceed in a negative direction.

3: M -- INTEGER Input
On entry: the first dimension of the array RESULT. This will usually be equal to the number of points at which the solution is required.
Constraint: M > 0.

4: N -- INTEGER Input
On entry: the number of differential equations.
Constraint: N > 0.

5: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: the initial values of the solution $y_1, y_2, \ldots, y_n$.
On exit: the computed values of the solution at the final value of X.

6: TOL -- DOUBLE PRECISION Input/Output
On entry: TOL must be set to a positive tolerance for controlling the error in the integration.

D02BBF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution at XEND. The relation between changes in TOL and the error at intermediate output points is less clear, but for TOL small enough the error at intermediate output points should also be approximately proportional to TOL. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02BBF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02BBF with TOL=10.0^{-p} and TOL=10.0^{-p-1} if p correct decimal digits in the solution are required. Constraint: TOL > 0.0. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution then, on return, TOL has its sign changed. This should be treated as a warning that the computed solution is likely to be more accurate than would be produced by using the same value of TOL on a longer range of integration.

7: IRELAB -- INTEGER Input
On entry: IRELAB determines the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:
IRELAB = 0
EST=10.0<=TOL*max{1.0,|y |,|y |,...,|y |};
1 2 n

IRELAB = 1
EST <= TOL;

IRELAB = 2
EST<=TOL*max{(epsilon),|y |,|y |,...,|y |}, where
1 2 n
(epsilon) is machine precision.
If the appropriate condition is not satisfied, the step size

```
is reduced and the solution is recomputed on the current
step.

If the user wishes to measure the error in the computed
solution in terms of the number of correct decimal places,
then IRELAB should be given the value 1 on entry, whereas if
the error requirement is in terms of the number of correct
significant digits, then IRELAB should be given the value 2.
Where there is no preference in the choice of error test
IRELAB = 0 will result in a mixed error test. Constraint: 0
<= IRELAB <= 2.

 8: RESULT(M,N) -- DOUBLE PRECISION array                Output
On exit: the computed values of the solution at the points
given by OUTPUT.

 9: FCN -- SUBROUTINE, supplied by the user.
                                                External Procedure
FCN must evaluate the functions f  (i.e., the derivatives
                                 i
y' ) for given values of its arguments x,y ,...,y .
  i                                       1      n

Its specification is:

       SUBROUTINE FCN (X, Y, F)
       DOUBLE PRECISION X, Y(n), F(n)

where n is the actual value of N in the call of D02BBF.

 1: X -- DOUBLE PRECISION                                Input
    On entry: the value of the argument x.

 2: Y(*) -- DOUBLE PRECISION array                       Input
    On entry: the value of the argument y , for
                                         i
    i=1,2,...,n.

 3: F(*) -- DOUBLE PRECISION array                       Output
    On exit: the value of f , for i=1,2,...,n.
                           i
FCN must be declared as EXTERNAL in the (sub)program
from which D02BBF is called. Parameters denoted as
Input must not be changed by this procedure.

10: OUTPUT -- SUBROUTINE, supplied by the user.
                                                External Procedure
OUTPUT allows the user to have access to intermediate values
of the computed solution at successive points specified by the
user. These solution values may be returned to the user via
```

```
     the array RESULT if desired (this is a non-standard feature
     added for use with the AXIOM system).  OUTPUT is initially
     called by D02BBF with XSOL = X (the initial value of x). The
     user must reset XSOL to the next point where OUTPUT is to be
     called, and so on at each call to OUTPUT. If, after a call
     to OUTPUT, the reset point XSOL is beyond XEND, D02BBF will
     integrate to XEND with no further calls to OUTPUT; if a call
     to OUTPUT is required at the point XSOL = XEND, then XSOL
     must be given precisely the value XEND.

     Its specification is:

          SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
          DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
          INTEGER M,N,COUNT

      1: XSOL -- DOUBLE PRECISION                    Input/Output
          On entry: the current value of the independent
          variable x. On exit: the next value of x at which
          OUTPUT is to be called.

      2: Y(N) -- DOUBLE PRECISION array                    Input
          On entry: the computed solution at the point XSOL.

      3: COUNT -- INTEGER                            Input/Output
          On entry: Zero if OUTPUT has not been called before, or
          the previous value of COUNT.
          On exit: A new value of COUNT: this can be used to keep
          track of the number of times OUTPUT has been called.

      4: M -- INTEGER                                      Input
          On entry: The first dimension of RESULT.

      5: N -- INTEGER                                      Input
          On entry: The dimension of Y.

      6: RESULT(M,N) -- DOUBLE PRECISION array     Input/Output
          On entry: the previous contents of RESULT.
          On exit: RESULT may be used to return the values of the
          intermediate solutions to the user.

     OUTPUT must be declared as EXTERNAL in the (sub)program
     from which D02BBF is called. Parameters denoted as
     Input must not be changed by this procedure.

11:  W(N,7) -- DOUBLE PRECISION array                  Workspace

12:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
```

```
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry TOL <= 0,

     or       N <= 0,

     or       IRELAB /= 0, 1 or 2.

IFAIL= 2
     With the given value of TOL, no further progress can be made
     across the integration range from the current point x = X,
     or the dependence of the error on TOL would be lost if
     further progress across the integration range were attempted
     (see Section 8 for a discussion of this error exit). The
     components Y(1),Y(2),...,Y(n) contain the computed values of
     the solution at the current point x = X.

IFAIL= 3
     TOL is too small for the routine to take an initial step
     (see Section 8). X and Y(1),Y(2),...,Y(n) retain their
     initial values.

IFAIL= 4
     X = XEND and XSOL /= X after the initial call to OUTPUT.

IFAIL= 5
     A value of XSOL returned by OUTPUT lies behind the previous
     value of XSOL in the direction of integration.

IFAIL= 6
     A serious error has occurred in an internal call to
     D02PAF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 7
     A serious error has occurred in an internal call to
     D02XAF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

7. Accuracy

The accuracy depends on TOL, on the mathematical properties of
```

```
the differential system, on the length of the range of
integration and on the method. It can be controlled by varying
TOL but the approximate proportionality of the error to TOL holds
only for a restricted range of values of TOL. For TOL too large,
the underlying theory may break down and the result of varying
TOL may be unpredictable. For TOL too small, rounding errors may
affect the solution significantly and an error exit with IFAIL =
2 or IFAIL = 3 is possible.

At the intermediate output points the same remarks apply. For
large values of TOL it is possible that the errors at some
intermediate output points may be much larger than at XEND. In
any case, it must not be expected that the error will have the
same size at all output points. At any point, it is a combination
of the errors arising from the integration of the differential
equation and the interpolation. The effect of combining these
errors will vary, though in most cases the integration error will
dominate.

The user who requires a more reliable estimate of the accuracy
achieved than can be obtained by varying TOL, is recommended to
call D02BDF(*) where both the solution and a global error
estimate are computed.

8. Further Comments

The time taken by the routine depends on the complexity and
mathematical properties of the system of differential equations
defined by FCN, on the range, the tolerance and the number of
calls to OUTPUT. There is also an overhead of the form a+b*n
where a and b are machine-dependent computing times.

If the routine fails with IFAIL = 3, then it can be called again
with a larger value of TOL (if this has not already been tried).
If the accuracy requested is really needed and cannot be obtained
with this routine, the system may be very stiff (see below) or so
badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has
been called with a value of TOL which is so small that the
solution cannot be obtained on the range X to XEND. This can
happen for well-behaved systems and very small values of TOL. The
user should, however, consider whether there is a more
fundamental difficulty. For example:

(a)   in the region of a singularity (infinite value) of the
      solution, the routine will usually stop with IFAIL = 2,
      unless overflow occurs first. If overflow occurs using
      D02BBF, D02PAF(*) can be used instead to trap the
      increasing solution before overflow occurs. In any case,
```

numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations, where the solution contains rapidly decaying components, the routine will use very small steps in x (internally to D02BBF) to preserve stability. This will usually exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Merson's method is not efficient in such cases, and the user should try using D02EBF(*) (Backward Differentiation Formula). To determine whether a problem is stiff, D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness or singularities, the Merson method should work well for low accuracy calculations (three or four figures). For higher accuracy calculations or where FCN is costly to evaluate, Merson's method may not be appropriate and a computationally less expensive method may be D02CBF(*) which uses an Adams method.

Users with problems for which D02BBF is not sufficiently general should consider using D02PAF(*) with D02XAF(*). D02PAF(*) is a more general Merson routine with many facilities including more general error control options and several criteria for interrupting the calculations. D02XAF(*) interpolates on values produced by D02PAF(*).

9. Example

To integrate the following equations (for a projectile)

$$y' = \tan(\phi)$$

$$v' = \frac{-0.032\tan(\phi)}{v} - \frac{0.02v}{\cos(\phi)}$$

$$(\phi)' = \frac{-0.032}{v^2}$$

over an interval X = 0.0 to XEND = 8.0, starting with values y=0.0, v=0.5 and (phi)=(pi)/5 and printing the solution at steps of 1.0. We write y=Y(1), v=Y(2) and (phi)=Y(3), and we set TOL=1.0E-4 and TOL=1.0E-5 in turn so that we may compare the solutions. The value of (pi) is obtained by using X01AAF(*).

Note the careful construction of routine OUT to ensure that the

```
value of XEND is printed.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
D02BHF(3NAG)      Foundation Library (12/10/92)      D02BHF(3NAG)

     D02 -- Ordinary Differential Equations                    D02BHF
            D02BHF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     D02BHF integrates a system of first-order ordinary differential
     equations over an interval with suitable initial conditions,
     using a Runge-Kutta-Merson method, until a user-specified
     function of the solution is zero.

     2. Specification

            SUBROUTINE D02BHF (X, XEND, N, Y, TOL, IRELAB, HMAX, FCN,
           1                   G, W, IFAIL)
            INTEGER          N, IRELAB, IFAIL
            DOUBLE PRECISION X, XEND, Y(N), TOL, HMAX, G, W(N,7)
            EXTERNAL         FCN, G

     3. Description

     The routine advances the solution of a system of ordinary
     differential equations

                  y' =f (x,y ,y ,...,y ),  i=1,2,...,n,
                    i  i    1  2      n

     from x = X towards x = XEND using a Merson form of the Runge-
     Kutta method. The system is defined by a subroutine FCN supplied
     by the user, which evaluates f  in terms of x and y ,y ,...,y
                                   i                     1  2      n
     (see Section 5), and the values of y ,y ,...,y  must be given at
                                         1  2      n
     x = X.

     As the integration proceeds, a check is made on the function
```

g(x,y) specified by the user, to determine an interval where it changes sign. The position of this sign change is then determined accurately by interpolating for the solution and its derivative. It is assumed that g(x,y) is a continuous function of the variables, so that a solution of g(x,y) = 0 can be determined by searching for a change in sign in g(x,y).

The accuracy of the integration and, indirectly, of the determination of the position where g(x,y) = 0, is controlled by the parameter TOL.

For a description of Runge-Kutta methods and their practical implementation see Hall and Watt [1].

4. References

[1] Hall G and Watt J M (eds) (1976) Modern Numerical Methods for Ordinary Differential Equations. Clarendon Press.

5. Parameters

1: X -- DOUBLE PRECISION Input/Output
On entry: X must be set to the initial value of the independent variable x. On exit: the point where g(x,y) = 0.0 unless an error has occurred, when it contains the value of x at the error. In particular, if g(x,y)/=0.0 anywhere on the range X to XEND, it will contain XEND on exit.

2: XEND -- DOUBLE PRECISION Input
On entry: the final value of the independent variable x.

If XEND < X on entry, integration proceeds in a negative direction.

3: N -- INTEGER Input
On entry: the number of differential equations, n.
Constraint: N > 0.

4: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: the initial values of the solution $y_1, y_2, \ldots, y_n$.
On exit: the computed values of the solution at the final point x = X.

5: TOL -- DOUBLE PRECISION Input/Output
On entry: TOL must be set to a positive tolerance for controlling the error in the integration and in the determination of the position where g(x,y) = 0.0.

D02BHF has been designed so that, for most problems, a

reduction in TOL leads to an approximately proportional reduction in the error in the solution obtained in the integration. The relation between changes in TOL and the error in the determination of the position where g(x,y) = 0. 0 is less clear, but for TOL small enough the error should be approximately proportional to TOL. However, the actual relation between TOL and the accuracy cannot be guaranteed. The user is strongly recommended to call D02BHF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge the user might compare results obtained by calling

$$\text{D02BHF with TOL}=10.0^{-p} \text{ and TOL}=10.0^{-p-1} \text{ if p correct decimal digits in the solution are required.}$$

Constraint: TOL > 0.0.

On exit: normally unchanged. However if the range from x = X to the position where g(x,y) = 0.0 (or to the final value of x if an error occurs) is so short that a small change in TOL is unlikely to make any change in the computed solution, then TOL is returned with its sign changed. To check results returned with TOL < 0.0, D02BHF should be called again with a positive value of TOL whose magnitude is considerably smaller than that of the previous call.

6: IRELAB -- INTEGER Input

On entry: IRELAB determines the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

IRELAB = 0

$$\text{EST} \le \text{TOL} \cdot \max\{1.0, |y_1|, |y_2|, \ldots, |y_n|\};$$

IRELAB = 1

$$\text{EST} \le \text{TOL};$$

IRELAB = 2

$$\text{EST} \le \text{TOL} \cdot \max\{(\text{epsilon}), |y_1|, |y_2|, \ldots, |y_n|\},$$

where (epsilon) is machine precision.

If the appropriate condition is not satisfied, the step size is reduced and the solution recomputed on the current step.

If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then IRELAB should be given the value 1 on entry, whereas if the error requirement is in terms of the number of correct significant digits, then IRELAB should be given the value 2. Where there is no preference in the choice of error test,

IRELAB = 0 will result in a mixed error test. It should be borne in mind that the computed solution will be used in evaluating g(x,y). Constraint: 0 <= IRELAB <= 2.

7: HMAX -- DOUBLE PRECISION Input
On entry: if HMAX = 0.0, no special action is taken.

If HMAX /= 0.0, a check is made for a change in sign of g(x,y) at steps not greater than |HMAX|. This facility should be used if there is any chance of 'missing' the change in sign by checking too infrequently. For example, if two changes of sign of g(x,y) are expected within a distance h, say, of each other, then a suitable value for HMAX might be HMAX = h/2. If only one change of sign in g(x,y) is expected on the range X to XEND, then the choice HMAX = 0.0 is most appropriate.

8: FCN -- SUBROUTINE, supplied by the user.
External Procedure
FCN must evaluate the functions f_i (i.e., the derivatives y_i') for given values of its arguments $x, y_1, \ldots, y_n$.

Its specification is:

```
SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)
```

1: X -- DOUBLE PRECISION Input
On entry: the value of the argument x.

2: Y(*) -- DOUBLE PRECISION array Input
On entry: the value of the argument y_i, for $i=1,2,\ldots,n$.

3: F(*) -- DOUBLE PRECISION array Output
On exit: the value of f_i, for $i=1,2,\ldots,n$.

FCN must be declared as EXTERNAL in the (sub)program from which D02BHF is called. Parameters denoted as Input must not be changed by this procedure.

9: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure
G must evaluate the function g(x,y) at a specified point.

Its specification is:

```
            DOUBLE PRECISION FUNCTION G (X, Y)
            DOUBLE PRECISION X, Y(n)
      where n is the actual value of N in the call of D02BHF.

       1:  X -- DOUBLE PRECISION                                Input
           On entry: the value of the independent variable x.

       2:  Y(*) -- DOUBLE PRECISION array                       Input
           On entry: the value of y , for i=1,2,...,n.
                                   i
      G must be declared as EXTERNAL in the (sub)program from
      which D02BHF is called. Parameters denoted as Input
      must not be changed by this procedure.

10:   W(N,7) -- DOUBLE PRECISION array                     Workspace

11:   IFAIL -- INTEGER                                   Input/Output
      On entry: IFAIL must be set to 0, -1 or 1. For users not
      familiar with this parameter (described in the Essential
      Introduction) the recommended value is 0.

      On exit: IFAIL = 0 unless the routine detects an error (see
      Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
      On entry TOL <= 0.0,

      or       N <= 0,

      or       IRELAB /= 0, 1 or 2.

IFAIL= 2
      With the given value of TOL, no further progress can be made
      across the integration range from the current point x = X,
      or dependence of the error on TOL would be lost if further
      progress across the integration range were attempted (see
      Section 8 for a discussion of this error exit). The
      components Y(1),Y(2),...,Y(n) contain the computed values of
      the solution at the current point x = X. No point at which
      g(x,y) changes sign has been located up to the point x = X.

IFAIL= 3
      TOL is too small for the routine to take an initial step
      (see Section 8). X and Y(1),Y(2),...,Y(n) retain their
      initial values.
```

```
IFAIL= 4
     At no point in the range X to XEND did the function g(x,y)
     change sign. It is assumed that g(x,y) = 0.0 has no
     solution.

IFAIL= 5
     A serious error has occurred in an internal call to
     C05AZF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 6
     A serious error has occurred in an internal call to
     D02PAF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 7
     A serious error has occurred in an internal call to
     D02XAF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

7. Accuracy

The accuracy depends on TOL, on the mathematical properties of
the differential system, on the position where g(x,y) = 0.0 and
on the method. It can be controlled by varying TOL but the
approximate proportionality of the error to TOL holds only for a
restricted range of values of TOL. For TOL too large, the
underlying theory may break down and the result of varying TOL
may be unpredictable. For TOL too small, rounding error may
affect the solution significantly and an error exit with IFAIL =
2 or IFAIL = 3 is possible.

The accuracy may also be restricted by the properties of g(x,y).
The user should try to code G without introducing any unnecessary
cancellation errors.

8. Further Comments

The time taken by the routine depends on the complexity and
mathematical properties of the system of differential equations
defined by FCN, the complexity of G, on the range, the position
of the solution and the tolerance. There is also an overhead of
the form a+b*n where a and b are machine-dependent computing
times.

For some problems it is possible that D02BHF will return IFAIL =
4 because of inaccuracy of the computed values Y, leading to
inaccuracy in the computed values of g(x,y) used in the search
for the solution of g(x,y) = 0.0. This difficulty can be overcome
by reducing TOL sufficiently, and if necessary, by choosing HMAX
```

```
sufficiently small. If possible, the user should choose XEND well
beyond the expected point where g(x,y) = 0.0; for example make
|XEND-X| about 50 larger than the expected range. As a simple
check, if, with XEND fixed, a change in TOL does not lead to a
significant change in Y at XEND, then inaccuracy is not a likely
source of error.

If the routine fails with IFAIL = 3, then it could be called
again with a larger value of TOL if this has not already been
tried. If the accuracy requested is really needed and cannot be
obtained with this routine, the system may be very stiff (see
below) or so badly scaled that it cannot be solved to the
required accuracy.

If the routine fails with IFAIL = 2, it is likely that it has
been called with a value of TOL which is so small that a solution
cannot be obtained on the range X to XEND. This can happen for
well-behaved systems and very small values of TOL. The user
should, however, consider whether there is a more fundamental
difficulty. For example:

(a)   in the region of a singularity (infinite value) of the
      solution, the routine will usually stop with IFAIL = 2,
      unless overflow occurs first. If overflow occurs using
      D02BHF, D02PAF(*) can be used instead to trap the
      increasing solution, before overflow occurs. In any case,
      numerical integration cannot be continued through a
      singularity, and analytical treatment should be considered;

(b)   for 'stiff' equations, where the solution contains rapidly
      decaying components, the routine will compute in very small
      steps in x (internally to D02BHF) to preserve stability.
      This will usually exhibit itself by making the computing
      time excessively long, or occasionally by an exit with
      IFAIL = 2. Merson's method is not efficient in such cases,
      and the user should try D02EHF(*) which uses a Backward
      Differentiation Formula method. To determine whether a
      problem is stiff, D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness
or singularities, the Merson method should work well for low
accuracy calculations (three or four figures). For high accuracy
calculations or where FCN is costly to evaluate, Merson's method
may not be appropriate and a computationally less expensive
method may be D02CHF(*) which uses an Adams method.

For problems for which D02BHF is not sufficiently general, the
user should consider D02PAF(*). D02PAF(*) is a more general
Merson routine with many facilities including more general error
control options and several criteria for interrupting the
```

calculations. D02PAF(*) can be combined with the rootfinder C05AZF(*) and the interpolation routine D02XAF(*) to solve equations involving $y_1, y_2, \ldots, y_n$ and their derivatives.

D02BHF can also be used to solve an equation involving x, $y_1, y_2, \ldots, y_n$ and the derivatives of $y_1, y_2, \ldots, y_n$. For example in Section 9, D02BHF is used to find a value of X > 0.0 where Y(1) = 0.0. It could instead be used to find a turning-point of y_1 by replacing the function g(x,y) in the program by:

```
      DOUBLE PRECISION FUNCTION G(X,Y)
      DOUBLE PRECISION X,Y(3),F(3)
      CALL FCN(X,Y,F)
      G = F(1)
      RETURN
      END
```

This routine is only intended to locate the first zero of g(x,y). If later zeros are required, users are strongly advised to construct their own more general root finding routines as discussed above.

9. Example

To find the value X > 0.0 at which y=0.0, where y, v, (phi) are defined by

$$y' = \tan(\text{phi})$$

$$v' = \frac{-0.032\tan(\text{phi})}{v} - \frac{0.02v}{\cos(\text{phi})}$$

$$(\text{phi})' = -\frac{0.032}{v^2}$$

and where at X = 0.0 we are given y=0.5, v=0.5 and (phi)=(pi)/5. We write y=Y(1), v=Y(2) and (phi)=Y(3) and we set TOL=1.0E-4 and TOL=1.0E-5 in turn so that we can compare the solutions. We expect the solution X ~= 7.3 and so we set XEND = 10.0 to avoid determining the solution of y=0.0 too near the end of the range of integration. The value of (pi) is obtained by using X01AAF(*).

The example program is not reproduced here. The source code for

```
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     D02CJF(3NAG)                 D02CJF                  D02CJF(3NAG)

     D02 -- Ordinary Differential Equations                      D02CJF
            D02CJF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     Note for users via the AXIOM system: the interface to this routine
     has been enhanced for use with AXIOM and is slightly different to
     that offered in the standard version of the Foundation Library.

     1. Purpose

     D02CJF integrates a system of first-order ordinary differential
     equations over a range with suitable initial conditions, using a
     variable-order, variable-step Adams method until a user-specified
     function, if supplied, of the solution is zero, and returns the
     solution at points specified by the user, if desired.

     2. Specification

           SUBROUTINE D02CJF (X, XEND, M, N, Y, FCN, TOL, RELABS,
          1                   RESULT, OUTPUT, G, W, IFAIL)
           INTEGER          M, N, IFAIL
           DOUBLE PRECISION X, XEND, Y(N), TOL, G, W(28+21*N), RESULT(M,N)
           CHARACTER*1      RELABS
           EXTERNAL         FCN, OUTPUT, G

     3. Description

     The routine advances the solution of a system of ordinary
     differential equations

                  y' =f (x,y ,y ,...,y ),  i=1,2,...,n,
                    i  i    1  2      n

     from x = X to x = XEND using a variable-order, variable-step
     Adams method. The system is defined by a subroutine FCN supplied
     by the user, which evaluates f  in terms of x and y ,y ,...,y .
                                   i                     1  2      n
     The initial values of y ,y ,...,y  must be given at x = X.
                           1  2      n
```

```
The solution is returned via the user-supplied routine OUTPUT at
points specified by the user, if desired: this solution is
                1
obtained by C  interpolation on solution values produced by the
method. As the integration proceeds a check can be made on the
user-specified function g(x,y) to determine an interval where it
changes sign. The position of this sign change is then determined
                 1
accurately by C  interpolation to the solution. It is assumed
that g(x,y) is a continuous function of the variables, so that a
solution of g(x,y)=0.0 can be determined by searching for a
change in sign in g(x,y). The accuracy of the integration, the
interpolation and, indirectly, of the determination of the
position where g(x,y)=0.0, is controlled by the parameters TOL
and RELABS.

For a description of Adams methods and their practical
implementation see Hall and Watt [1].

4. References

[1]   Hall G and Watt J M (eds) (1976) Modern Numerical Methods
      for Ordinary Differential Equations. Clarendon Press.

5. Parameters

 1:  X -- DOUBLE PRECISION                              Input/Output
     On entry: the initial value of the independent variable x.
     Constraint: X /= XEND. On exit: if g is supplied by the
     user, it contains the point where g(x,y)=0.0, unless
     g(x,y)/=0.0 anywhere on the range X to XEND, in which case,
     X will contain XEND. If g is not supplied by the user it
     contains XEND, unless an error has occurred, when it
     contains the value of x at the error.

 2:  XEND -- DOUBLE PRECISION                                 Input
     On entry: the final value of the independent variable. If
     XEND < X, integration proceeds in the negative direction.
     Constraint: XEND /= X.

 3:  M -- INTEGER                                             Input
     On entry: the first dimension of the array RESULT.  This
     will usually be equal to the number of points at which the
     solution is required.
     Constraint: M > 0.

 4:  N -- INTEGER                                             Input
     On entry: the number of differential equations.
     Constraint: N >= 1.
```

```
5:  Y(N) -- DOUBLE PRECISION array                  Input/Output
    On entry: the initial values of the solution y ,y ,...,y
                                                   1  2      n
    at x = X. On exit: the computed values of the solution at
    the final point x = X.

6:  FCN -- SUBROUTINE, supplied by the user.
                                            External Procedure
    FCN must evaluate the functions f  (i.e., the derivatives
                                    i
    y' ) for given values of their arguments x,y ,y ,...,y .
      i                                         1  2      n

    Its specification is:

           SUBROUTINE FCN (X, Y, F)
           DOUBLE PRECISION X, Y(n), F(n)
    where n is the actual value of N in the call of D02CJF.

     1:  X -- DOUBLE PRECISION                              Input
         On entry: the value of the independent variable x.

     2:  Y(*) -- DOUBLE PRECISION array                     Input
         On entry: the value of the variable y , for
                                              i
         i=1,2,...,n.

     3:  F(*) -- DOUBLE PRECISION array                    Output
         On exit: the value of f , for i=1,2,...,n.
                                 i
    FCN must be declared as EXTERNAL in the (sub)program
    from which D02CJF is called. Parameters denoted as
    Input must not be changed by this procedure.

7:  TOL -- DOUBLE PRECISION                                Input
    On entry: a positive tolerance for controlling the error in
    the integration. Hence TOL affects the determination of the
    position where g(x,y)=0.0, if g is supplied.

    D02CJF has been designed so that, for most problems, a
    reduction in TOL leads to an approximately proportional
    reduction in the error in the solution. However, the actual
    relation between TOL and the accuracy achieved cannot be
    guaranteed. The user is strongly recommended to call D02CJF
    with more than one value for TOL and to compare the results
    obtained to estimate their accuracy. In the absence of any
    prior knowledge, the user might compare the results obtained
                                   -p             -p-1
    by calling D02CJF with TOL=10.0   and TOL=10.0     where p
```

correct decimal digits are required in the solution.
Constraint: TOL > 0.0.

8: RELABS -- CHARACTER*1 Input
On entry: the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

$$EST = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \left(e_i / ((tau)_r * |y_i| + (tau)_a) \right)^2} <= 1.0$$

where $(tau)_r$ and $(tau)_a$ are defined by

RELABS	$(tau)_r$	$(tau)_a$
'M'	TOL	TOL
'A'	0.0	TOL
'R'	TOL	(epsilon)
'D'	TOL	TOL

where (epsilon) is a small machine-dependent number and e_i is an estimate of the local error at y_i, computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then RELABS should be set to 'A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then RELABS should be set to 'R'. If the user prefers a mixed error test, then RELABS should be set to 'M', otherwise if the user has no preference, RELABS should be set to the default 'D'. Note that in this case 'D' is taken to be 'M'. Constraint: RELABS = 'M', 'A', 'R', 'D'.

9: RESULT(M,N) -- DOUBLE PRECISION array Output
On exit: the computed values of the solution at the points given by OUTPUT.

10: OUTPUT -- SUBROUTINE, supplied by the user.
External Procedure
OUTPUT allows the user to have access to intermediate values

```
of the computed solution at successive points specified by the
user. These solution values may be returned to the user via
the array RESULT if desired (this is a non-standard feature
added for use with the AXIOM system).  OUTPUT is initially
called by D02CJF with XSOL = X (the initial value of x). The
user must reset XSOL to the next point where OUTPUT is to be
called, and so on at each call to OUTPUT. If, after a call
to OUTPUT, the reset point XSOL is beyond XEND, D02CJF will
integrate to XEND with no further calls to OUTPUT; if a call
to OUTPUT is required at the point XSOL = XEND, then XSOL
must be given precisely the value XEND.

Its specification is:

     SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
     DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
     INTEGER M,N,COUNT

 1:  XSOL -- DOUBLE PRECISION                    Input/Output
     On entry: the current value of the independent
     variable x. On exit: the next value of x at which
     OUTPUT is to be called.

 2:  Y(N) -- DOUBLE PRECISION array                     Input
     On entry: the computed solution at the point XSOL.

 3:  COUNT -- INTEGER                            Input/Output
     On entry: Zero if OUTPUT has not been called before, or
     the previous value of COUNT.
     On exit: A new value of COUNT: this can be used to keep
     track of the number of times OUTPUT has been called.

 4:  M -- INTEGER                                       Input
     On entry: The first dimension of RESULT.

 5:  N -- INTEGER                                       Input
     On entry: The dimension of Y.

 6:  RESULT(M,N) -- DOUBLE PRECISION array      Input/Output
     On entry: the previous contents of RESULT.
     On exit: RESULT may be used to return the values of the
     intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02CJF is called. Parameters denoted as
Input must not be changed by this procedure.

11:  G -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                        External Procedure
G must evaluate the function g(x,y) for specified values x,y
```

```
     . It specifies the function g for which the first position x
     where g(x,y)=0 is to be found.

     If the user does not require the root finding option, the
     actual argument G  mustbe the dummy routine D02CJW. (D02CJW
     is included in the NAG Foundation Library and so need not be
     supplied by the user).

     Its specification is:

            DOUBLE PRECISION FUNCTION G (X, Y)
            DOUBLE PRECISION X, Y(n)
     where n is the actual value of N in the call of D02CJF.

      1:  X -- DOUBLE PRECISION                                Input
          On entry: the value of the independent variable x.

      2:  Y(*) -- DOUBLE PRECISION array                       Input
          On entry: the value of the variable y , for
                                               i
          i=1,2,...,n.
     G must be declared as EXTERNAL in the (sub)program from
     which D02CJF is called. Parameters denoted as Input
     must not be changed by this procedure.

12:  W(28+21*N) -- DOUBLE PRECISION array              Workspace

13:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry TOL <= 0.0,

     or       N <= 0,

     or       RELABS /= 'M', 'A', 'R' or 'D'.

     or       X = XEND.
```

```
IFAIL= 2
     With the given value of TOL, no further progress can be made
     across the integration range from the current point x = X.
     (See Section 8 for a discussion of this error exit.) The
     components Y(1),Y(2),...,Y(N) contain the computed values of
     the solution at the current point x = X. If the user has
     supplied g, then no point at which g(x,y) changes sign has
     been located up to the point x = X.

IFAIL= 3
     TOL is too small for D02CJF to take an initial step. X and Y
     (1),Y(2),...,Y(N) retain their initial values.

IFAIL= 4
     XSOL has not been reset or XSOL lies behind X in the
     direction of integration, after the initial call to OUTPUT,
     if the OUTPUT option was selected.

IFAIL= 5
     A value of XSOL returned by OUTPUT has not been reset or
     lies behind the last value of XSOL in the direction of
     integration, if the OUTPUT option was selected.

IFAIL= 6
     At no point in the range X to XEND did the function g(x,y)
     change sign, if g was supplied. It is assumed that g(x,y)=0
     has no solution.

IFAIL= 7
     A serious error has occurred in an internal call. Check all
     subroutine calls and array sizes. Seek expert help.

7. Accuracy

The accuracy of the computation of the solution vector Y may be
controlled by varying the local error tolerance TOL. In general,
a decrease in local error tolerance should lead to an increase in
accuracy. Users are advised to choose RELABS = 'M' unless they
have a good reason for a different choice.

If the problem is a root-finding one, then the accuracy of the
root determined will depend on the properties of g(x,y). The user
should try to code G without introducing any unnecessary
cancellation errors.

8. Further Comments

If more than one root is required then D02QFF(*) should be used.
```

If the routine fails with IFAIL = 3, then it can be called again with a larger value of TOL if this has not already been tried. If the accuracy requested is really needed and cannot be obtained with this routine, the system may be very stiff (see below) or so badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has been called with a value of TOL which is so small that a solution cannot be obtained on the range X to XEND. This can happen for well-behaved systems and very small values of TOL. The user should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. Numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations where the solution contains rapidly decaying components, the routine will use very small steps in x (internally to D02CJF) to preserve stability. This will exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Adams methods are not efficient in such cases, and the user should try D02EJF.

9. Example

We illustrate the solution of four different problems. In each case the differential system (for a projectile) is

$$y' = \tan(\phi)$$

$$v' = \frac{-0.032\tan(\phi)}{v} - \frac{0.02v}{\cos(\phi)}$$

$$(\phi)' = \frac{-0.032}{v^2}$$

over an interval X = 0.0 to XEND = 10.0 starting with values y=0.5, v=0.5 and (phi)=(pi)/5. We solve each of the following problems with local error tolerances 1.0E-4 and 1.0E-5.

(i) To integrate to x=10.0 producing output at intervals of 2.0 until a point is encountered where y=0.0.

```
     (ii)  As (i) but with no intermediate output.

     (iii) As (i) but with no termination on a root-finding condition.

     (iv)  As (i) but with no intermediate output and no root-finding
           termination condition.

     The example program is not reproduced here. The source code for
     all example programs is distributed with the NAG Foundation
     Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     D02EJF(3NAG)                D02EJF                  D02EJF(3NAG)

     D02 -- Ordinary Differential Equations                      D02EJF
            D02EJF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     Note for users via the AXIOM system: the interface to this routine
     has been enhanced for use with AXIOM and is slightly different to
     that offered in the standard version of the Foundation Library.

     1. Purpose

     D02EJF integrates a stiff system of first-order ordinary
     differential equations over an interval with suitable initial
     conditions, using a variable-order, variable-step method
     implementing the Backward Differentiation Formulae (BDF), until a
     user-specified function, if supplied, of the solution is zero,
     and returns the solution at points specified by the user, if
     desired.

     2. Specification

            SUBROUTINE D02EJF (X, XEND, M, N, Y, FCN, PEDERV, TOL,
           1                   RELABS, OUTPUT, G, W, IW, RESULT, IFAIL)
            INTEGER          M, N, IW, IFAIL
            DOUBLE PRECISION X, XEND, Y(N), TOL, G, W(IW), RESULT(M,N)
            CHARACTER*1      RELABS
            EXTERNAL         FCN, PEDERV, OUTPUT, G

     3. Description

     The routine advances the solution of a system of ordinary
     differential equations
```

$$y_i' = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n,$$

from x = X to x = XEND using a variable-order, variable-step method implementing the BDF. The system is defined by a subroutine FCN supplied by the user, which evaluates f_i in terms of x and $y_1, y_2, \ldots, y_n$ (see Section 5). The initial values of $y_1, y_2, \ldots, y_n$ must be given at x = X.

The solution is returned via the user-supplied routine OUTPUT at points specified by the user, if desired: this solution is obtained by C^1 interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function g(x,y) to determine an interval where it changes sign. The position of this sign change is then determined accurately by C^1 interpolation to the solution. It is assumed that g(x,y) is a continuous function of the variables, so that a solution of g(x,y) = 0.0 can be determined by searching for a change in sign in g(x,y). The accuracy of the integration, the interpolation and, indirectly, of the determination of the position where g(x,y) = 0.0, is controlled by the parameters TOL and RELABS. The Jacobian of the system y'=f(x,y) may be supplied in routine PEDERV, if it is available.

For a description of BDF and their practical implementation see Hall and Watt [1].

4. References

[1] Hall G and Watt J M (eds) (1976) Modern Numerical Methods for Ordinary Differential Equations. Clarendon Press.

5. Parameters

1: X -- DOUBLE PRECISION Input/Output
On entry: the initial value of the independent variable x. Constraint: X /= XEND On exit: if G is supplied by the user, X contains the point where g(x,y) = 0.0, unless g(x,y) /= 0.0 anywhere on the range X to XEND, in which case, X will contain XEND. If G is not supplied X contains XEND, unless an error has occured, when it contains the value of x at the error.

2: XEND -- DOUBLE PRECISION Input

```
    On entry: the final value of the independent variable. If
    XEND < X, integration proceeds in the negative direction.
    Constraint: XEND /= X.

3:  M -- INTEGER                                            Input
    On entry: the first dimension of the array RESULT.  This
    will usually be equal to the number of points at which the
    solution is required.
    Constraint: M > 0.

4:  N -- INTEGER                                            Input
    On entry: the number of differential equations, n.
    Constraint: N >= 1.

5:  Y(N) -- DOUBLE PRECISION array                  Input/Output
    On entry: the initial values of the solution y ,y ,...,y
                                                  1  2      n
    at x = X. On exit: the computed values of the solution at
    the final point x = X.

6:  FCN -- SUBROUTINE, supplied by the user.
                                               External Procedure
    FCN must evaluate the functions f  (i.e., the derivatives
                                    i
    y' ) for given values of their arguments x,y ,y ,...,y .
      i                                         1  2      n

    Its specification is:

           SUBROUTINE FCN (X, Y, F)
           DOUBLE PRECISION X, Y(n), F(n)
    where n is the actual value of N in the call of D02EJF.

     1:  X -- DOUBLE PRECISION                              Input
         On entry: the value of the independent variable x.

     2:  Y(*) -- DOUBLE PRECISION array                     Input
         On entry: the value of the variable y , for
                                               i
         i=1,2,...,n.

     3:  F(*) -- DOUBLE PRECISION array                    Output
         On exit: the value of f , for i=1,2,...,n.
                                i
    FCN must be declared as EXTERNAL in the (sub)program
    from which D02EJF is called. Parameters denoted as
    Input must not be changed by this procedure.

7:  PEDERV -- SUBROUTINE, supplied by the user.
                                               External Procedure
```

PEDERV must evaluate the Jacobian of the system (that is,
$\frac{ddf_i}{ddy_j}$) for given values of the
variables $x, y_1, y_2, \ldots, y_n$.

Its specification is:

```
      SUBROUTINE PEDERV (X, Y, PW)
      DOUBLE PRECISION X, Y(n), PW(n,n)
```

where n is the actual value of N in the call of D02EJF.

1: X -- DOUBLE PRECISION Input
On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array Input
On entry: the value of the variable y_i, for
$i=1,2,\ldots,n$.

3: PW(n,*) -- DOUBLE PRECISION array Output
On exit: the value of $\frac{ddf_i}{ddy_j}$, for $i,j=1,2,\ldots,n$.

If the user does not wish to supply the Jacobian, the actual argument PEDERV must be the dummy routine D02EJY . (D02EJY is included in the NAG Foundation Library and so need not be supplied by the user. The name may be implementation dependent: see the User's Note for your implementation for details).
PEDERV must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

8: TOL -- DOUBLE PRECISION Input/Output
On entry: TOL must be set to a positive tolerance for controlling the error in the integration. Hence TOL affects the determination of the position where g(x,y) = 0.0, if G is supplied.

D02EJF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02EJF

with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02EJF with TOL=10^-p and TOL=10^-p-1 if p correct decimal digits are required in the solution. Constraint: TOL > 0.0. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution, then, on return, TOL has its sign changed.

9: RELABS -- CHARACTER*1 Input

On entry: the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

$$EST = \left(\sum_{i=1}^{n} \left(\frac{e_i}{(\tau)_r * |y_i| + (\tau)_a} \right)^2 \right)^{1/2} <= 1.0$$

where $(\tau)_r$ and $(\tau)_a$ are defined by

RELABS	$(\tau)_r$	$(\tau)_a$
'M'	TOL	TOL
'A'	0.0	TOL
'R'	TOL	(epsilon)
'D'	TOL	(epsilon)

where (epsilon) is a small machine-dependent number and e_i is an estimate of the local error at y_i, computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then RELABS should be set to 'A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then RELABS should be set to 'R'. If the user prefers a mixed error test, then RELABS should be set to 'M', otherwise if the user has no preference, RELABS should be set to the default 'D'. Note that in this case 'D' is taken to be 'R'. Constraint: RELABS = 'A', 'M', 'R' or 'D'.

```
10:  OUTPUT -- SUBROUTINE, supplied by the user.
                                                  External Procedure
     OUTPUT allows the user to have access to intermediate values
     of the computed solution at successive points specified by the
     user. These solution values may be returned to the user via
     the array RESULT if desired (this is a non-standard feature
     added for use with the AXIOM system).  OUTPUT is initially
     called by D02EJF with XSOL = X (the initial value of x). The
     user must reset XSOL to the next point where OUTPUT is to be
     called, and so on at each call to OUTPUT. If, after a call
     to OUTPUT, the reset point XSOL is beyond XEND, D02EJF will
     integrate to XEND with no further calls to OUTPUT; if a call
     to OUTPUT is required at the point XSOL = XEND, then XSOL
     must be given precisely the value XEND.

     Its specification is:

          SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
          DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
          INTEGER M,N,COUNT

      1:  XSOL -- DOUBLE PRECISION                    Input/Output
          On entry: the current value of the independent
          variable x. On exit: the next value of x at which
          OUTPUT is to be called.

      2:  Y(N) -- DOUBLE PRECISION array                      Input
          On entry: the computed solution at the point XSOL.

      3:  COUNT -- INTEGER                            Input/Output
          On entry: Zero if OUTPUT has not been called before, or
          the previous value of COUNT.
          On exit: A new value of COUNT: this can be used to keep
          track of the number of times OUTPUT has been called.

      4:  M -- INTEGER                                        Input
          On entry: The first dimension of RESULT.

      5:  N -- INTEGER                                        Input
          On entry: The dimension of Y.

      6:  RESULT(M,N) -- DOUBLE PRECISION array       Input/Output
          On entry: the previous contents of RESULT.
          On exit: RESULT may be used to return the values of the
          intermediate solutions to the user.

     OUTPUT must be declared as EXTERNAL in the (sub)program
     from which D02EJF is called. Parameters denoted as
     Input must not be changed by this procedure.
```

```
11:  G -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                                  External Procedure
     G must evaluate the function g(x,y) for specified values x,y
     . It specifies the function g for which the first position x
     where g(x,y) = 0 is to be found.

     Its specification is:

            DOUBLE PRECISION FUNCTION G (X, Y)
            DOUBLE PRECISION X, Y(n)
     where n is the actual value of N in the call of D02EJF.

      1:  X -- DOUBLE PRECISION                                 Input
          On entry: the value of the independent variable x.

      2:  Y(*) -- DOUBLE PRECISION array                        Input
          On entry: the value of the variable y , for
                                              i
          i=1,2,...,n.
     If the user does not require the root finding option,
     the actual argument G must be the dummy routine D02EJW.
     (D02EJW is included in the NAG Foundation Library and
     so need not be supplied by the user).
     G must be declared as EXTERNAL in the (sub)program from
     which D02EJF is called. Parameters denoted as Input
     must not be changed by this procedure.

12:  W(IW) -- DOUBLE PRECISION array                        Workspace

13:  IW -- INTEGER                                             Input
     On entry:
     the dimension of the array W as declared in the (sub)program
     from which D02EJF is called.
     Constraint: IW>=(12+N)*N+50.

14:  RESULT(M,N) -- DOUBLE PRECISION array                     Output
     On exit: the computed values of the solution at the points
     given by OUTPUT.

15:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings
```

```
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry TOL <= 0.0,

     or       X = XEND,

     or       N <= 0,

     or       RELABS /= 'M', 'A', 'R', 'D'.

     or       IW<(12+N)*N+50.

IFAIL= 2
     With the given value of TOL, no further progress can be made
     across the integration range from the current point x = X.
     (See Section 5 for a discussion of this error test.) The
     components Y(1),Y(2),...,Y(n) contain the computed values of
     the solution at the current point x = X. If the user has
     supplied G, then no point at which g(x,y) changes sign has
     been located up to the point x = X.

IFAIL= 3
     TOL is too small for D02EJF to take an initial step. X and Y
     (1),Y(2),...,Y(n) retain their initial values.

IFAIL= 4
     XSOL lies behind X in the direction of integration, after
     the initial call to OUTPUT, if the OUTPUT option was
     selected.

IFAIL= 5
     A value of XSOL returned by OUTPUT lies behind the last
     value of XSOL in the direction of integration, if the OUTPUT
     option was selected.

IFAIL= 6
     At no point in the range X to XEND did the function g(x,y)
     change sign, if G was supplied. It is assumed that g(x,y) =
     0 has no solution.

IFAIL= 7
     A serious error has occurred in an internal call to
     C05AZF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 8
```

```
     A serious error has occurred in an internal call to
     D02XKF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 9
     A serious error has occurred in an internal call to
     D02NMF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

7. Accuracy

The accuracy of the computation of the solution vector Y may be
controlled by varying the local error tolerance TOL. In general,
a decrease in local error tolerance should lead to an increase in
accuracy. Users are advised to choose RELABS = 'R' unless they
have a good reason for a different choice. It is particularly
appropriate if the solution decays.

If the problem is a root-finding one, then the accuracy of the
                                        ddg      ddg
root determined will depend strongly on --- and ----, for
                                        ddx      ddy
                                                   i
i=1,2,...,n. Large values for these quantities may imply large
errors in the root.

8. Further Comments

If more than one root is required, then to determine the second
and later roots D02EJF may be called again starting a short
distance past the previously determined roots. Alternatively the
user may construct his own root finding code using D02QDF(*) (or
the routines of the subchapter D02M-D02N), D02XKF(*) and
C05AZF(*).

If it is easy to code, the user should supply the routine PEDERV.
However, it is important to be aware that if PEDERV is coded
incorrectly, a very inefficient integration may result and
possibly even a failure to complete the integration (IFAIL = 2).

9. Example

We illustrate the solution of five different problems. In each
case the differential system is the well-known stiff Robertson
problem.

                          4
              a'= -0.04a-10 bc
                          4        7 2
              b'=  0.04a-10 bc -3*10 b
```

```
                                        7 2
                    c'=              3*10 b
```

with initial conditions a=1.0, b=c=0.0 at x=0.0. We solve each of the following problems with local error tolerances 1.0E-3 and 1.0 E-4.

(i) To integrate to x=10.0 producing output at intervals of 2.0 until a point is encountered where a=0.9. The Jacobian is calculated numerically.

(ii) As (i) but with the Jacobian calculated analytically.

(iii) As (i) but with no intermediate output.

(iv) As (i) but with no termination on a root-finding condition.

(v) Integrating the equations as in (i) but with no intermediate output and no root-finding termination condition.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

D02GAF(3NAG) Foundation Library (12/10/92) D02GAF(3NAG)

D02 -- Ordinary Differential Equations D02GAF
D02GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02GAF solves the two-point boundary-value problem with assigned boundary values for a system of ordinary differential equations, using a deferred correction technique and a Newton iteration.

2. Specification

```
      SUBROUTINE D02GAF (U, V, N, A, B, TOL, FCN, MNP, X, Y, NP,
     1                   W, LW, IW, LIW, IFAIL)
      INTEGER          N, MNP, NP, LW, IW(LIW), LIW, IFAIL
      DOUBLE PRECISION U(N,2), V(N,2), A, B, TOL, X(MNP), Y
     1                 (N,MNP), W(LW)
```

```
       EXTERNAL         FCN
```

3. Description

D02GAF solves a two-point boundary-value problem for a system of n differential equations in the interval [a,b]. The system is written in the form

$$y_i' = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1,2,\ldots,n \qquad (1)$$

and the derivatives are evaluated by a subroutine FCN supplied by the user. Initially, n boundary values of the variables y_i must be specified (assigned), some at a and some at b. The user also supplies estimates of the remaining n boundary values and all the boundary values are used in constructing an initial approximation to the solution. This approximate solution is corrected by a finite-difference technique with deferred correction allied with a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1]. The Newton iteration requires a Jacobian matrix $\frac{\partial f_i}{\partial y_j}$ and this is calculated by numerical differentiation using an algorithm described in Curtis et al [2].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user's tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If on the other hand the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References

[1] Pereyra V (1979) PASVA3: An Adaptive Finite-Difference Fortran Program for First Order Nonlinear, Ordinary Boundary Problems. Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science.

(ed B Childs, M Scott, J W Daniel, E Denman and P Nelson) 76 Springer-Verlag.

[2] Curtis A R, Powell M J D and Reid J K (1974) On the Estimation of Sparse Jacobian Matrices. J. Inst. Maths Applics. 13 117--119.

5. Parameters

1: U(N,2) -- DOUBLE PRECISION array Input
On entry: U(i,1) must be set to the known (assigned) or estimated values of y_i at a and U(i,2) must be set to the known or estimated values of y_i at b, for i=1,2,...,n.

2: V(N,2) -- DOUBLE PRECISION array Input
On entry: V(i,j) must be set to 0.0 if U(i,j) is a known (assigned) value and to 1.0 if U(i,j) is an estimated value, i=1,2,...,n; j=1,2. Constraint: precisely N of the V(i,j) must be set to 0.0, i.e., precisely N of the U(i,j) must be known values, and these must not be all at a or all at b.

3: N -- INTEGER Input
On entry: the number of equations. Constraint: N >= 2.

4: A -- DOUBLE PRECISION Input
On entry: the left-hand boundary point, a.

5: B -- DOUBLE PRECISION Input
On entry: the right-hand boundary point, b. Constraint: B > A.

6: TOL -- DOUBLE PRECISION Input
On entry: a positive absolute error tolerance. If
$$a=x_1<x_2<\ldots<x_{NP}=b$$
is the final mesh, $z_j(x_i)$ is the jth component of the approximate solution at x_i, and $y_j(x_i)$ is the jth component of the true solution of equation (1) (see Section 3) and the boundary conditions, then, except in extreme cases, it is expected that
$$|z_j(x_i)-y_j(x_i)|\leq TOL,\ i=1,2,\ldots,NP;j=1,2,\ldots,n \qquad (2)$$
Constraint: TOL > 0.0.

7: FCN -- SUBROUTINE, supplied by the user.
External Procedure

FCN must evaluate the functions f (i.e., the derivatives
                                i
y') at the general point x.
  i

Its specification is:

```
      SUBROUTINE FCN (X, Y, F)
      DOUBLE PRECISION X, Y(n), F(n)
```

where n is the actual value of N in the call of D02GAF.

1: X -- DOUBLE PRECISION Input
On entry: the value of the argument x.

2: Y(*) -- DOUBLE PRECISION array Input
On entry: the value of the argument y , for
                                    i
i=1,2,...,n.

3: F(*) -- DOUBLE PRECISION array Output
On exit: the values of f , for i=1,2,...,n.
                        i

FCN must be declared as EXTERNAL in the (sub)program from which D02GAF is called. Parameters denoted as Input must not be changed by this procedure.

8: MNP -- INTEGER Input
On entry: the maximum permitted number of mesh points.
Constraint: MNP >= 32.

9: X(MNP) -- DOUBLE PRECISION array Input/Output
On entry: if NP >= 4 (see NP below), the first NP elements must define an initial mesh. Otherwise the elements of X need not be set. Constraint:

$$A=X(1)<X(2)<\ldots<X(NP)=B \text{ for } NP\geq 4 \qquad (3)$$

On exit: X(1),X(2),...,X(NP) define the final mesh (with the returned value of NP) satisfying the relation (3).

10: Y(N,MNP) -- DOUBLE PRECISION array Output
On exit: the approximate solution z (x) satisfying (2), on
                                  j i
the final mesh, that is

$$Y(j,i)=z_j(x_i),\ i=1,2,\ldots,NP;j=1,2,\ldots,n,$$

where NP is the number of points in the final mesh.

The remaining columns of Y are not used.

11: NP -- INTEGER Input/Output
On entry: determines whether a default or user-supplied

```
     mesh is used. If NP = 0, a default value of 4 for NP and a
     corresponding equispaced mesh X(1),X(2),...,X(NP) are used.
     If NP >= 4, then the user must define an initial mesh using
     the array X as described. Constraint: NP = 0 or 4 <= NP <=
     MNP. On exit: the number of points in the final (returned)
     mesh.

12:  W(LW) -- DOUBLE PRECISION array                    Workspace

13:  LW -- INTEGER                                          Input
     On entry: the length of the array W as declared in the
                                                 2         2
     calling (sub)program. Constraint: LW>=MNP*(3N +6N+2)+4N +4N

14:  IW(LIW) -- INTEGER array                           Workspace

15:  LIW -- INTEGER                                         Input
     On entry: the length of the array IW as declared in the
                                                       2
     calling (sub)program. Constraint: LIW>=MNP*(2N+1)+N +4N+2.

16:  IFAIL -- INTEGER                                Input/Output
     For this routine, the normal use of IFAIL is extended to
     control the printing of error and warning messages as well
     as specifying hard or soft failure (see the Essential
     Introduction).

     Before entry, IFAIL must be set to a value with the decimal
     expansion cba, where each of the decimal digits c, b and a
     must have a value of 0 or 1.
     a=0   specifies hard failure, otherwise soft failure;

     b=0   suppresses error messages, otherwise error messages
           will be printed (see Section 6);

     c=0   suppresses warning messages, otherwise warning
           messages will be printed (see Section 6).
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).
```

```
IFAIL= 1
     One or more of the parameters N, TOL, NP, MNP, LW or LIW has
     been incorrectly set, or B <= A, or the condition (3) on X
     is not satisfied, or the number of known boundary values
     (specified by V) is not N.

IFAIL= 2
     The Newton iteration has failed to converge. This could be
     due to there being too few points in the initial mesh or to
     the initial approximate solution being too inaccurate. If
     this latter reason is suspected the user should use
     subroutine D02RAF instead. If the warning 'Jacobian matrix
     is singular' is printed this could be due to specifying zero
     estimated boundary values and these should be varied. This
     warning could also be printed in the unlikely event of the
     Jacobian matrix being calculated inaccurately. If the user
     cannot make changes to prevent the warning then subroutine
     D02RAF should be used.

IFAIL= 3
     The Newton iteration has reached round-off level. It could
     be, however, that the answer returned is satisfactory. This
     error might occur if too much accuracy is requested.

IFAIL= 4
     A finer mesh is required for the accuracy requested; that is
     MNP is not large enough.

IFAIL= 5
     A serious error has occurred in a call to D02GAF. Check all
     array subscripts and subroutine parameter lists in calls to
     D02GAF. Seek expert help.

7. Accuracy

The solution returned by the routine will be accurate to the
user's tolerance as defined by the relation (2) except in extreme
circumstances. If too many points are specified in the initial
mesh, the solution may be more accurate than requested and the
error may not be approximately equidistributed.

8. Further Comments

The time taken by the routine depends on the difficulty of the
problem, the number of mesh points used (and the number of
different meshes used), the number of Newton iterations and the
number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-
explanatory error messages, and also monitoring information about
```

the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

A common cause of convergence problems in the Newton iteration is the user specifying too few points in the initial mesh. Although the routine adds points to the mesh to improve accuracy it is unable to do so until the solution on the initial mesh has been calculated in the Newton iteration.

If the user specifies zero known and estimated boundary values, the routine constructs a zero initial approximation and in many cases the Jacobian is singular when evaluated for this approximation, leading to the breakdown of the Newton iteration.

The user may be unable to provide a sufficiently good choice of initial mesh and estimated boundary values, and hence the Newton iteration may never converge. In this case the continuation facility provided in D02RAF is recommended.

In the case where the user wishes to solve a sequence of similar problems, the final mesh from solving one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the differential equation

$$y'''=-yy''-(\mathrm{beta})(1-y'^{\,2})$$

with boundary conditions

$$y(0)=y'(0)=0,$$

$$y'(10)=1$$

for (beta)=0.0 and (beta)=0.2 to an accuracy specified by TOL=1.0 E-3. We solve first the simpler problem with (beta)=0.0 using an equispaced mesh of 26 points and then we solve the problem with (beta)=0.2 using the final mesh from the first problem.

Note the call to X04ABF prior to the call to D02GAF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%

D02GBF(3NAG) Foundation Library (12/10/92) D02GBF(3NAG)

D02 -- Ordinary Differential Equations D02GBF
D02GBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02GBF solves a general linear two-point boundary value problem for a system of ordinary differential equations using a deferred correction technique.

2. Specification

```
      SUBROUTINE D02GBF (A, B, N, TOL, FCNF, FCNG, C, D, GAM,
     1                   MNP, X, Y, NP, W, LW, IW, LIW, IFAIL)
      INTEGER           N, MNP, NP, LW, IW(LIW), LIW, IFAIL
      DOUBLE PRECISION A, B, TOL, C(N,N), D(N,N), GAM(N), X(MNP),
     1                 Y(N,MNP), W(LW)
      EXTERNAL          FCNF, FCNG
```

3. Description

D02GBF solves the linear two-point boundary value problem for a system of n ordinary differential equations in the interval

[a,b]. The system is written in the form

$$y'=F(x)y+g(x) \qquad (1)$$

and the boundary conditions are written in the form

$$Cy(a)+Dy(b)=(gamma) \qquad (2)$$

Here F(x), C and D are n by n matrices, and g(x) and (gamma) are n-component vectors. The approximate solution to (1) and (2) is found using a finite-difference method with deferred correction. The algorithm is a specialisation of that used in subroutine D02RAF which solves a nonlinear version of (1) and (2). The nonlinear version of the algorithm is described fully in Pereyra [1].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference

equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user's tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points, then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References

[1] Pereyra V (1979) PASVA3: An Adaptive Finite-Difference Fortran Program for First Order Nonlinear, Ordinary Boundary Problems. Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science. (ed B Childs, M Scott, J W Daniel, E Denman and P Nelson) 76 Springer-Verlag.

5. Parameters

1: A -- DOUBLE PRECISION Input
On entry: the left-hand boundary point, a.

2: B -- DOUBLE PRECISION Input
On entry: the right-hand boundary point, b. Constraint: B > A.

3: N -- INTEGER Input
On entry: the number of equations; that is n is the order of system (1). Constraint: N >= 2.

4: TOL -- DOUBLE PRECISION Input
On entry: a positive absolute error tolerance. If

$$a=x_1<x_2<\ldots<x_{NP}=b$$

is the final mesh, z(x) is the approximate solution from D02GBF and y(x) is the true solution of equations (1) and (2) then, except in extreme cases, it is expected that

$$||z-y||\leq TOL \qquad (3)$$

where

$$||u||=\max_{1\leq i\leq N}\ \max_{1\leq j\leq NP}|u_i(x_j)|.$$

Constraint: TOL > 0.0.

5: FCNF -- SUBROUTINE, supplied by the user.
External Procedure

```
     FCNF must evaluate the matrix F(x) in (1) at a general point
     x.

     Its specification is:

            SUBROUTINE FCN (X, F)
            DOUBLE PRECISION X, F(n,n)
     where n is the actual value of N in the call of D02GBF.

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the value of the independent variable x.

      2:  F(n,n) -- DOUBLE PRECISION array                   Output
          On exit: the (i,j)th element of the matrix F(x), for
          i,j=1,2,...,n. (See Section 9 for an example.)
     FCN must be declared as EXTERNAL in the (sub)program
     from which D02GBF is called. Parameters denoted as
     Input must not be changed by this procedure.

 6:  FCNG -- SUBROUTINE, supplied by the user.
                                              External Procedure
     FCNG must evaluate the vector g(x) in (1) at a general point
     x.

     Its specification is:

            SUBROUTINE FCNG (X, G)
            DOUBLE PRECISION X, G(n)
     where n is the actual value of N in the call of D02GBF.

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the value of the independent variable x.

      2:  G(*) -- DOUBLE PRECISION array                     Output
          On exit: the ith element of the vector g(x), for
          i=1,2,...,n. (See Section Section 9 for an example.)
     FCNG must be declared as EXTERNAL in the (sub)program
     from which D02GBF is called. Parameters denoted as
     Input must not be changed by this procedure.

 7:  C(N,N) -- DOUBLE PRECISION array                Input/Output

 8:  D(N,N) -- DOUBLE PRECISION array                Input/Output

 9:  GAM(N) -- DOUBLE PRECISION array                Input/Output
     On entry: the arrays C and D must be set to the matrices C
     and D in (2). GAM must be set to the vector (gamma) in (2).
     On exit: the rows of C and D and the components of GAM are
     re-ordered so that the boundary conditions are in the order:
       (i)   conditions on y(a) only;
```

```
        (ii)  condition involving y(a) and y(b); and

        (iii) conditions on y(b) only.

      The routine will be slightly more efficient if the arrays C,
      D and GAM are ordered in this way before entry, and in this
      event they will be unchanged on exit.

      Note that the problems (1) and (2) must be of boundary value
      type, that is neither C nor D may be identically zero. Note
      also that the rank of the matrix [C,D] must be n for the
      problem to be properly posed. Any violation of these
      conditions will lead to an error exit.

10:  MNP -- INTEGER                                          Input
      On entry: the maximum permitted number of mesh points.
      Constraint: MNP >= 32.

11:  X(MNP) -- DOUBLE PRECISION array               Input/Output
      On entry: if NP >= 4 (see NP below), the first NP elements
      must define an initial mesh. Otherwise the elements of x
      need not be set. Constraint:
                A=X(1)<X(2)<...<X(NP)=B, for NP>=4.          (4)
      On exit: X(1),X(2),...,X(NP) define the final mesh (with the
      returned value of NP) satisfying the relation (4).

12:  Y(N,MNP) -- DOUBLE PRECISION array                     Output
      On exit: the approximate solution z(x) satisfying (3), on
      the final mesh, that is
                Y(j,i)=z (x ) , i=1,2,...,NP;j=1,2,...,n
                        j  i
      where NP is the number of points in the final mesh.

      The remaining columns of Y are not used.

13:  NP -- INTEGER                                  Input/Output
      On entry: determines whether a default mesh or user-supplied
      mesh is used. If NP = 0, a default value of 4 for NP and a
      corresponding equispaced mesh X(1),X(2),...,X(NP) are used.
      If NP >= 4, then the user must define an initial mesh X as
      in (4) above. On exit: the number of points in the final
      (returned) mesh.

14:  W(LW) -- DOUBLE PRECISION array                      Workspace

15:  LW -- INTEGER                                           Input
      On entry: the length of the array W, Constraint:
                 2         2
      LW>=MNP*(3N +5N+2)+3N +5N.
```

```
16:  IW(LIW) -- INTEGER array                                Workspace

17:  LIW -- INTEGER                                             Input
     On entry: the length of the array IW. Constraint:
     LIW>=MNP*(2N+1)+N.

18:  IFAIL -- INTEGER                                       Input/Output
     For this routine, the normal use of IFAIL is extended to
     control the printing of error and warning messages as well
     as specifying hard or soft failure (see the Essential
     Introduction).

     Before entry, IFAIL must be set to a value with the decimal
     expansion cba, where each of the decimal digits c, b and a
     must have a value of 0 or 1.
     a=0   specifies hard failure, otherwise soft failure;

     b=0   suppresses error messages, otherwise error messages
           will be printed (see Section 6);

     c=0   suppresses warning messages, otherwise warning
           messages will be printed (see Section 6).
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
     One or more of the parameters N, TOL, NP, MNP, LW or LIW is
     incorrectly set, B <= A or the condition (4) on X is not
     satisfied.

IFAIL= 2
     There are three possible reasons for this error exit to be
     taken:
       (i)  one of the matrices C or D is identically zero (that
            is the problem is of initial value and not boundary
            value type). In this case, IW(1) = 0 on exit;

       (ii) a row of C and the corresponding row of D are
```

```
                identically zero (that is the boundary conditions are
                rank deficient). In this case, on exit IW(1) contains
                the index of the first such row encountered; and

          (iii more than n of the columns of the n by 2n matrix [C,D
          )     ] are identically zero (that is the boundary
                conditions are rank deficient). In this case, on exit
                IW(1) contains minus the number of non-identically
                zero columns.

IFAIL= 3
     The routine has failed to find a solution to the specified
     accuracy. There are a variety of possible reasons including:
       (i)   the boundary conditions are rank deficient, which may
             be indicated by the message that the Jacobian is
             singular. However this is an unlikely explanation for
             the error exit as all rank deficient boundary
             conditions should lead instead to error exits with
             either IFAIL = 2 or IFAIL = 5; see also (iv) below;

       (ii)  not enough mesh points are permitted in order to
             attain the required accuracy. This is indicated by NP
             = MNP on return from a call to D02GBF. This
             difficulty may be aggravated by a poor initial choice
             of mesh points;

       (iii) the accuracy requested cannot be attained on the
             computer being used; and

       (iv)  an unlikely combination of values of F(x) has led to
             a singular Jacobian. The error should not persist if
             more mesh points are allowed.

IFAIL= 4
     A serious error has occurred in a call to D02GBF. Check all
     array subscripts and subroutine parameter lists in calls to
     D02GBF. Seek expert help.

IFAIL= 5
     There are two possible reasons for this error exit which
     occurs when checking the rank of the boundary conditions by
     reduction to a row echelon form:
          (i) at least one row of the n by 2n matrix [C,D] is a
          linear combination of the other rows and hence the
          boundary conditions are rank deficient. The index of
          the first such row encountered is given by IW(1) on
          exit; and

          (ii) as (i) but the rank deficiency implied by this
          error exit has only been determined up to a numerical
```

```
     tolerance. Minus the index of the first such row
     encountered is given by IW(1) on exit.
In case (ii) above there is some doubt as to the rank
deficiency of the boundary conditions. However even if the
boundary conditions are not rank deficient they are not
posed in a suitable form for use with this routine.

For example, if
                                          ((gamma) )
         (1 0        )     (1 0)          (       1)
       C=(1 (epsilon)) , D=(1 0) , (gamma)=((gamma) )
                                          (       2)
and (epsilon) is small enough, this error exit is likely to
be taken. A better form for the boundary conditions in this
case would be
                           ((gamma)                     )
                           (       1                    )
  (1 0)     (1 0)          (         -1                 )
C=(0 1) , D=(0 0) , (gamma)=((epsilon)  ((gamma) -(gamma) ))
                           (                  2        1 )
```

7. Accuracy

The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (3) except in extreme circumstances. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments

The time taken by the routine depends on the difficulty of the problem, the number of mesh points (and meshes) used and the number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

In the case where the user wishes to solve a sequence of similar problems, the use of the final mesh from one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the problem (written as a first order system)

```
                              (epsilon)y''+y'=0

     with boundary conditions

                               y(0)=0,  y(1)=1

                                -1                -2
     for the cases (epsilon)=10   and (epsilon)=10   using the default
     initial mesh in the first case, and the final mesh of the first
     case as initial mesh for the second (more difficult) case. We
     give the solution and the error at each mesh point to illustrate
     the accuracy of the method given the accuracy request TOL=1.0E-3.

     Note the call to X04ABF prior to the call to D02GBF.

     The example program is not reproduced here. The source code for
     all example programs is distributed with the NAG Foundation
     Library software and should be available on-line.
```

%%

```
D02 -- Ordinary Differential Equations                      D02KEF
        D02KEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

D02KEF finds a specified eigenvalue of a regular singular second-
order Sturm-Liouville system on a finite or infinite range, using
a Pruefer transformation and a shooting method. It also reports
values of the eigenfunction and its derivatives. Provision is
made for discontinuities in the coefficient functions or their
derivatives.

2. Specification

       SUBROUTINE D02KEF (XPOINT, M, MATCH, COEFFN, BDYVAL, K,
      1                   TOL, ELAM, DELAM, HMAX, MAXIT, MAXFUN,
      2                   MONIT, REPORT, IFAIL)
       INTEGER          M, MATCH, K, MAXIT, MAXFUN, IFAIL
       DOUBLE PRECISION XPOINT(M), TOL, ELAM, DELAM, HMAX(2,M)
       EXTERNAL         COEFFN, BDYVAL, MONIT, REPORT

3. Description
```

D02KEF has essentially the same purpose as D02KDF(*) with minor modifications to enable values of the eigenfunction to be obtained after convergence to the eigenvalue has been achieved.

~~~~~~~~

It first finds a specified eigenvalue (lambda) of a Sturm-Liouville system defined by a self-adjoint differential equation of the second-order

(p(x)y')'+q(x;(lambda))y=0, a<x<b

together with the appropriate boundary conditions at the two (finite or infinite) end-points a and b. The functions p and q, which are real-valued, must be defined by a subroutine COEFFN. The boundary conditions must be defined by a subroutine BDYVAL, and, in the case of a singularity at a or b, take the form of an asymptotic formula for the solution near the relevant end-point.

~~~~~~~~

When the final estimate (lambda)=(lambda) of the eigenvalue has been found, the routine integrates the differential equation once more with that value of (lambda), and with initial conditions chosen so that the integral

$$S = \int_a^b y(x)\,\frac{dq}{d(\mathrm{lambda})}(x;(\mathrm{lambda}))\,dx$$

is approximately one. When q(x;(lambda)) is of the form (lambda)w(x)+q(x), which is the most common case, S represents the square of the norm of y induced by the inner product

$$\langle f,g\rangle = \int_a^b f(x)g(x)w(x)\,dx,$$

with respect to which the eigenfunctions are mutually orthogonal. This normalisation of y is only approximate, but experience shows that S generally differs from unity by only one or two per cent.

During this final integration the REPORT routine supplied by the user is called at each integration mesh point x. Sufficient information is returned to permit the user to compute y(x) and y'(x) for printing or plotting. For reasons described in Section 8.2, D02KEF passes across to REPORT, not y and y', but the Pru efer variables (beta), (phi) and (rho) on which the numerical

method is based. Their relationship to y and y' is given by the equations

$$p(x)y' = \sqrt{\beta}\exp\left(\frac{\rho}{2}\right)\cos\left(\frac{\phi}{2}\right);$$

$$y = \frac{1}{\sqrt{\beta}}\exp\left(\frac{\rho}{2}\right)\sin\left(\frac{\phi}{2}\right).$$

For the theoretical basis of the numerical method to be valid, the following conditions should hold on the coefficient functions:

(a) p(x) must be non-zero and of one sign throughout the interval (a,b); and,

(b) $\frac{dq}{d\lambda}$ must be of one sign throughout (a,b) for all relevant values of (lambda), and must not be identically zero as x varies, for any (lambda).

Points of discontinuity in the functions p and q or their derivatives are allowed, and should be included as 'break-points' in the array XPOINT.

A good account of the theory of Sturm-Liouville systems, with some description of Pruefer transformations, is given in Birkhoff and Rota [4], Chapter X. An introduction for the user of Pruefer transformations for the numerical solution of eigenvalue problems arising from physics and chemistry is Bailey [2].

The scaled Pruefer method is fairly recent, and is described in a short note by Pryce [6] and in some detail in the technical report [5].

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Bailey P B (1966) Sturm-Liouville Eigenvalues via a Phase Function. SIAM J. Appl. Math . 14 242--249.

[3] Banks D O and Kurowski I (1968) Computation of Eigenvalues of Singular Sturm-Liouville Systems. Math. Computing. 22

304--310.

[4] Birkhoff G and Rota G C (1962) Ordinary Differential Equations. Ginn & Co., Boston and New York.

[5] Pryce J D (1981) Two codes for Sturm-Liouville problems. Technical Report CS-81-01. Dept of Computer Science, Bristol University .

[6] Pryce J D and Hargrave B A (1977) The Scale Pruefer Method for one-parameter and multi-parameter eigenvalue problems in ODEs. Inst. Math. Appl., Numerical Analysis Newsletter. 1(3)

5. Parameters

1: XPOINT(M) -- DOUBLE PRECISION array Input
On entry: the points where the boundary conditions computed by BDYVAL are to be imposed, and also any break-points, i.e., XPOINT(1) to XPOINT(m) must contain values $x_1,\ldots,x_m$ such that

$$x_1 \le x_2 < x_3 < \ldots < x_{m-1} \le x_m$$

with the following meanings:

(a) x_1 and x_m are the left and right end-points, a and b, of the domain of definition of the Sturm-Liouville system if these are finite. If either a or b is infinite, the corresponding value x_1 or x_m may be a more-or-less arbitrarily 'large' number of appropriate sign.

(b) x_2 and x_{m-1} are the Boundary Matching Points (BMP's), that is the points at which the left and right boundary conditions computed in BDYVAL are imposed.

If the left-hand end-point is a regular point then the user should set $x_2 = x_1$ (=a), while if it is a singular point the user must set $x_2 > x_1$. Similarly $x_{m-1} = x_m$ (=b) if the right-hand end-point is regular, and $x_{m-1} < x_m$ if it is singular.

(c) The remaining m-4 points $x_3,\ldots,x_{m-2}$, if any, define 'break-points' which divide the interval $[x_3, x_{m-2}]$

2 m-1
into m-3 sub-intervals
i =[x ,x],...,i =[x ,x]
1 2 3 m-3 m-2 m-1
Numerical integration of the differential equation is stopped and restarted at each break-point. In simple cases no break-points are needed. However if p(x) or q(x;(lambda)) are given by different formulae in different parts of the range, then integration is more efficient if the range is broken up by break-points in the appropriate way. Similarly points where any jumps occur in p(x) or q(x;(lambda)), or in their derivatives up to the fifth order, should appear as break-points.
Constraint: X(1) <= X(2) < ... < X(M-1) <= X(M).

2: M -- INTEGER Input
On entry: the number of points in the array XPOINT.
Constraint: M >= 4.

3: MATCH -- INTEGER Input/Output
On entry: MATCH must be set to the index of the 'break-point ' to be used as the matching point (see Section 8.3). If MATCH is set to a value outside the range [2,m-1] then a default value is taken, corresponding to the break-point nearest the centre of the interval [XPOINT(2),XPOINT(m-1)].
On exit: the index of the break-point actually used as the matching point.

4: COEFFN -- SUBROUTINE, supplied by the user.
External Procedure
COEFFN must compute the values of the coefficient functions p(x) and q(x;(lambda)) for given values of x and (lambda). Section 3 states conditions which p and q must satisfy.

Its specification is:

```
SUBROUTINE COEFFN (P, Q, DQDL, X, ELAM, JINT)
DOUBLE PRECISION P, Q, DQDL, X, ELAM
INTEGER          JINT
```

1: P -- DOUBLE PRECISION Output
On exit: the value of p(x) for the current value of x.

2: Q -- DOUBLE PRECISION Output
On exit: the value of q(x;(lambda)) for the current value of x and the current trial value of (lambda).

3: DQDL -- DOUBLE PRECISION Output
ddq

```
          On exit: the value of  ----------(x;(lambda)) for the
                                 dd(lambda)
          current value of x and the current trial value of
          (lambda). However DQDL is only used in error estimation
          and an approximation (say to within 20%) will suffice.

          4:  X -- DOUBLE PRECISION                              Input
          On entry: the current value of x.

          5:  ELAM -- DOUBLE PRECISION                           Input
          On entry: the current trial value of the eigenvalue
          parameter (lambda).

          6:  JINT -- INTEGER                                    Input
          On entry: the index j of the sub-interval i  (see
                                                     j
          specification of XPOINT) in which x lies.
     See Section 8.4 and Section 9 for examples.

     COEFFN must be declared as EXTERNAL in the (sub)program
     from which D02KEF is called. Parameters denoted as
     Input must not be changed by this procedure.

5:   BDYVAL -- SUBROUTINE, supplied by the user.
                                        External Procedure
     BDYVAL must define the boundary conditions. For each end-
     point, BDYVAL must return (in YL or YR) values of y(x) and
     p(x)y'(x) which are consistent with the boundary conditions
     at the end-points; only the ratio of the values matters.
     Here x is a given point (XL or XR) equal to, or close to,
     the end-point.

     For a regular end-point (a, say), x=a; and a boundary
     condition of the form
                          c y(a)+c y'(a)=0
                           1      2
     can be handled by returning constant values in YL, e.g.
     YL(1)=c  and YL(2)=-c p(a).
            2             1

     For a singular end-point however, YL(1) and YL(2) will in
     general be functions of XL and ELAM, and YR(1) and YR(2)
     functions of XR and ELAM, usually derived analytically from
     a power-series or asymptotic expansion. Examples are given
     in Section 8.5 Section 9.

     Its specification is:

            SUBROUTINE BDYVAL (XL, XR, ELAM, YL, YR))
            DOUBLE PRECISION XL, XR, ELAM, YL(3), YR(3)
```

```
 1:  XL -- DOUBLE PRECISION                                  Input
     On entry: if a is a regular end-point of the system (so
     that a=x =x ), then XL contains a. If a is a singular
             1  2
     point (so that a<=x <x ), then XL contains a point x
                        1  2
     such that x <x<=x ).
                1     2

 2:  XR -- DOUBLE PRECISION                                  Input
     On entry: if b is a regular end-point of the system (so
     that x   =x =b), then XR contains b. If b is a singular
           m-1  m
     point (so that x   <x <=b), then XR contains a point x
                     m-1  m
     such that x   <=x<x .
                m-1     m

 3:  ELAM -- DOUBLE PRECISION                                Input
     On entry: the current trial value of (lambda).

 4:  YL(3) -- DOUBLE PRECISION array                        Output
     On exit: YL(1) and YL(2) should contain values of y(x)
     and p(x)y'(x) respectively (not both zero) which are
     consistent with the boundary condition at the left-hand
     end-point, given by x = XL. YL(3) should not be set.

 5:  YR(3) -- DOUBLE PRECISION array                        Output
     On exit: YR(1) and YR(2) should contain values of y(x)
     and p(x)y'(x) respectively (not both zero) which are
     consistent with the boundary condition at the right-
     hand end-point, given by x = XR. YR(3) should not be
     set.

    BDYVAL must be declared as EXTERNAL in the (sub)program
    from which D02KEF is called. Parameters denoted as
    Input must not be changed by this procedure.

 6: K -- INTEGER                                            Input
    On entry: the index k of the required eigenvalue when the
    eigenvalues are ordered
    (lambda) <(lambda) <(lambda) <...<(lambda) <....
           0         1         2             k
    Constraint: K >= 0.

 7: TOL -- DOUBLE PRECISION                                 Input
    On entry: the tolerance parameter which determines the
    accuracy of the computed eigenvalue. The error estimate held
    in DELAM on exit satisfies the mixed absolute/relative error
```

```
     test
                  DELAM<=TOL*max(1.0,|ELAM|)                  (*)
     where ELAM is the final estimate of the eigenvalue. DELAM is
     usually somewhat smaller than the right-hand side of (*) but
     not several orders of magnitude smaller. Constraint: TOL >
     0.0.

 8:  ELAM -- DOUBLE PRECISION                          Input/Output
                                                       ~~~~~~~~
     On entry: an initial estimate of the eigenvalue (lambda).
     On exit: the final computed estimate, whether or not an
     error occurred.

 9:  DELAM -- DOUBLE PRECISION                         Input/Output
     On entry: an indication of the scale of the problem in the
     (lambda)-direction. DELAM holds the initial 'search step'
     (positive or negative). Its value is not critical but the
     first two trial evaluations are made at ELAM and ELAM +
     DELAM, so the routine will work most efficiently if the
     eigenvalue lies between these values. A reasonable choice
     (if a closer bound is not known) is half the distance
     between adjacent eigenvalues in the neighbourhood of the one
     sought. In practice, there will often be a problem, similar
     to the one in hand but with known eigenvalues, which will
     help one to choose initial values for ELAM and DELAM.

     If DELAM = 0.0 on entry, it is given the default value of
     0.25*max(1.0,|ELAM|). On exit: with IFAIL = 0, DELAM holds
     an estimate of the absolute error in the computed
                          ~~~~~~~~
     eigenvalue, that is |(lambda)-ELAM|~=DELAM. (In Section 8.2
     we discuss the assumptions under which this is true.) The
     true error is rarely more than twice, or less than a tenth,
     of the estimated error.

     With IFAIL /= 0, DELAM may hold an estimate of the error, or
     its initial value, depending on the value of IFAIL. See
     Section 6 for further details.

10:  HMAX(2,M) -- DOUBLE PRECISION array              Input/Output
     On entry: HMAX(1,j) a maximum step size to be used by the
     differential equation code in the jth sub-interval i  (as
                                                           j
     described in the specification of parameter XPOINT), for
     j=1,2,...,m-3. If it is zero the routine generates a maximum
     step size internally.

     It is recommended that HMAX(1,j) be set to zero unless the
     coefficient functions p and q have features (such as a
     narrow peak) within the jth sub-interval that could be '
```

```
missed' if a long step were taken. In such a case HMAX(1,j)
should be set to about half the distance over which the
feature should be observed. Too small a value will increase
the computing time for the routine. See Section 8 for
further suggestions.

The rest of the array is used as workspace. On exit: HMAX(
1,m-1) and HMAX(1,m) contain the sensitivity coefficients
(sigma) ,(sigma) , described in Section 8.6. Other entries
       l        r
contain diagnostic output in case of an error (see Section 6
).

11:  MAXIT -- INTEGER                                Input/Output
On entry: a bound on n , the number of root-finding
                      r
iterations allowed, that is the number of trial values of
(lambda) that are used. If MAXIT <= 0, no such bound is
assumed. (See also under MAXFUN.) Suggested value: MAXIT =
0. On exit: MAXIT will have been decreased by the number of
iterations actually performed, whether or not it was
positive on entry.

12:  MAXFUN -- INTEGER                                       Input
On entry: a bound on n , the number of calls to COEFFN made
                      f
in any one root-finding iteration. If MAXFUN <= 0, no such
bound is assumed. Suggested value: MAXFUN = 0.

MAXFUN and MAXIT may be used to limit the computational cost
of a call to D02KEF, which is roughly proportional to n *n .
                                                       r  f

13:  MONIT -- SUBROUTINE, supplied by the user.
                                     External Procedure
MONIT is called by D02KEF at the end of each root-finding
iteration and allows the user to monitor the course of the
computation by printing out the parameters (see Section 8
for an example).
If no monitoring is required, the dummy subroutine D02KAY
may be used. (D02KAY is included in the NAG Foundation
Library).

Its specification is:

       SUBROUTINE MONIT (MAXIT, IFLAG, ELAM, FINFO)
       INTEGER          MAXIT, IFLAG
       DOUBLE PRECISION ELAM, FINFO(15)

 1:  MAXIT -- INTEGER                                        Input
```

```
      On entry: the current value of the parameter MAXIT of
      D02KEF; this is decreased by one at each iteration.

2:  IFLAG -- INTEGER                                    Input
      On entry: IFLAG describes what phase the computation is
      in, as follows:
      IFLAG < 0
            an error occurred in the computation of the '
            miss-distance' at this iteration;

            an error exit from D02KEF with IFAIL =-IFLAG will
            follow.

      IFLAG = 1
            the routine is trying to bracket the eigenvalue
            ~~~~~~~~
            (lambda).

      IFLAG = 2
            the routine is converging to the eigenvalue
            ~~~~~~~~
            (lambda) (having already bracketed it).

3:  ELAM -- DOUBLE PRECISION                            Input
      On entry: the current trial value of (lambda).

4:  FINFO(15) -- DOUBLE PRECISION array                 Input
      On entry: information about the behaviour of the
      shooting method, and diagnostic information in the case
      of errors. It should not normally be printed in full if
      no error has occurred (that is, if IFLAG > 0), though
      the first few components may be of interest to the
      user. In case of an error (IFLAG < 0) all the
      components of FINFO should be printed. The contents of
      FINFO are as follows:

      FINFO(1): the current value of the 'miss-distance' or '
      residual' function f((lambda)) on which the shooting
      method is based. FINFO(1) is set to zero if IFLAG < 0.

      FINFO(2): an estimate of the quantity dd(lambda)
      defined as follows. Consider the perturbation in the
      miss-distance f((lambda)) that would result if the
      local error, in the solution of the differential
      equation, were always positive and equal to its maximum
      permitted value. Then dd(lambda) is the perturbation in
      (lambda) that would have the same effect on f((lambda))
      . Thus, at the zero of f((lambda)),|dd(lambda)| is an
      approximate bound on the perturbation of the zero (that
      is the eigenvalue) caused by errors in numerical
```

```
solution. If dd(lambda) is very large then it is
possible that there has been a programming error in
COEFFN such that q is independent of (lambda). If this
is the case, an error exit with IFAIL = 5 should
follow. FINFO(2) is set to zero if IFLAG < 0.

FINFO(3): the number of internal iterations, using the
same value of (lambda) and tighter accuracy tolerances,
needed to bring the accuracy (that is the value of
dd(lambda)) to an acceptable value. Its value should
normally be 1.0, and should almost never exceed 2.0.

FINFO(4): the number of calls to COEFFN at this
iteration.

FINFO(5): the number of successful steps taken by the
internal differential equation solver at this
iteration. A step is successful if it is used to
advance the integration (cf. COUT(8) in specification
of D02PAF(*)).

FINFO(6): the number of unsuccessful steps used by the
internal integrator at this iteration (cf. COUT(9) in
specification of D02PAF(*)).

FINFO(7): the number of successful steps at the maximum
step size taken by the internal integrator at this
iteration (cf. COUT(3) in specification of D02PAF(*)).

FINFO(8): is not used.

FINFO(9) to FINFO(15): set to zero, unless IFLAG < 0 in
which case they hold the following values describing
the point of failure:

FINFO(9): contains the index of the sub-interval where
failure occurred, in the range 1 to m-3. In case of an
error in BDYVAL, it is set to 0 or m-2 depending on
whether the left or right boundary condition caused the
error.

FINFO(10): the value f the independent variable x, the
point at which error occurred. In case of an error in
BDYVAL, it is set to the value of XL or XR as
appropriate (see the specification of BDYVAL).

FINFO(11), FINFO(12), FINFO(13): the current values of
the Pruefer dependent variables (beta), (phi) and (rho)
respectively. These are set to zero in case of an error
in BDYVAL.
```

```
          FINFO(14): the local-error tolerance being used by the
          internal integrator at the point of failure. This is
          set to zero in the case of an error in BDYVAL.

          FINFO(15): the last integration mesh point. This is set
          to zero in the case of an error in BDYVAL.
     MONIT must be declared as EXTERNAL in the (sub)program
     from which D02KEF is called. Parameters denoted as
     Input must not be changed by this procedure.

14:  REPORT -- SUBROUTINE, supplied by the user.
                                          External Procedure
     This routine provides the means by which the user may
     compute the eigenfunction y(x) and its derivative at each
     integration mesh point x. (See Section 8 for an example).

     Its specification is:

            SUBROUTINE REPORT (X, V, JINT)
            INTEGER          JINT
            DOUBLE PRECISION X, V(3)

      1:  X -- DOUBLE PRECISION                              Input
          On entry: the current value of the independent variable
          x. See Section 8.3 for the order in which values of x
          are supplied.

      2:  V(3) -- DOUBLE PRECISION array                     Input
          On entry: V(1), V(2), V(3) hold the current values of
          the Pruefer variables (beta), (phi), (rho)
          respectively.

      3:  JINT -- INTEGER                                    Input
          On entry: JINT indicates the sub-interval between
          break-points in which X lies exactly as for the routine
          COEFFN, except that at the extreme left end-point (when
          x = XPOINT(2)) JINT is set to 0 and at the extreme
          right end-point (when x=x =XPOINT(m-1)) JINT is set to
                                    r
          m-2.
     REPORT must be declared as EXTERNAL in the (sub)program
     from which D02KEF is called. Parameters denoted as
     Input must not be changed by this procedure.

15:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
```

```
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     A parameter error. All parameters (except IFAIL) are left
     unchanged. The reason for the error is shown by the value of
     HMAX(2,1) as follows:
     HMAX(2,1) = 1: M < 4;

     HMAX(2,1) = 2: K < 0;

     HMAX(2,1) = 3: TOL <= 0.0;

     HMAX(2,1) = 4: XPOINT(1) to XPOINT(m) are not in ascending
                    order.

                    HMAX(2,2) gives the position i in XPOINT
                    where this was detected.

IFAIL= 2
     At some call to BDYVAL, invalid values were returned, that
     is, either YL(1) = YL(2) = 0.0, or YR(1) = YR(2) = 0.0 (a
     programming error in BDYVAL). See the last call of MONIT for
     details.

     This error exit will also occur if p(x) is zero at the point
     where the boundary condition is imposed. Probably BDYVAL was
     called with XL equal to a singular end-point a or with XR
     equal to a singular end-point b.

IFAIL= 3
     At some point between XL and XR the value of p(x) computed
     by COEFFN became zero or changed sign. See the last call of
     MONIT for details.

IFAIL= 4
     MAXIT > 0 on entry, and after MAXIT iterations the
     eigenvalue had not been found to the required accuracy.

IFAIL= 5
     The 'bracketing' phase (with parameter IFLAG of MONIT equal
     to 1) failed to bracket the eigenvalue within ten
     iterations. This is caused by an error in formulating the
     problem (for example, q is independent of (lambda)), or by
     very poor initial estimates of ELAM, DELAM.
```

```
          On exit ELAM and ELAM + DELAM give the end-points of the
          interval within which no eigenvalue was located by the
          routine.

     IFAIL= 6
          MAXFUN > 0 on entry, and the last iteration was terminated
          because more than MAXFUN calls to COEFFN were used. See the
          last call of MONIT for details.

     IFAIL= 7
          To obtain the desired accuracy the local error tolerance was
          set so small at the start of some sub-interval that the
          differential equation solver could not choose an initial
          step size large enough to make significant progress. See the
          last call of MONIT for diagnostics.

     IFAIL= 8
          At some point inside a sub-interval the step size in the
          differenital equation solver was reduced to a value too
          small to make significant progress (for the same reasons as
          with IFAIL = 7). This could be due to pathological behaviour
          of p(x) and q(x;(lambda)) or to an unreasonable accuracy
          requirement or to the current value of (lambda) making the
          equations 'stiff'. See the last call of MONIT for details.

     IFAIL= 9
          TOL is too small for the problem being solved and the
          machine precision is being used. The final value of ELAM
          should be a very good approximation to the eigenvalue.

     IFAIL= 10
          C05AZF(*), called by D02KEF, has terminated with the error
          exit corresponding to a pole of the residual function
          f((lambda)). This error exit should not occur, but if it
          does, try solving the problem again with a smaller value for
          TOL.

     IFAIL= 11
          A serious error has occurred in an internal call to D02KDY.
          Check all subroutine calls and array dimensions. Seek expert
          help.

     IFAIL= 12
          A serious error has occurred in an internal call to
          C05AZF(*). Check all subroutine calls and array dimensions.
          Seek expert help.

          HMAX(2,1) holds the failure exit number from the routine
          where the failure occurred. In the case of a failure in
          C05AZF(*), HMAX(2,2) holds the value of parameter IND of
```

```
     C05AZF(*).

Note: error exits with IFAIL = 2, 3, 6, 7, 8, 11 are caused by
being unable to set up or solve the differential equation at some
iteration, and will be immediately preceded by a call of MONIT
giving diagnostic information. For other errors, diagnostic
information is contained in HMAX(2,j), for j=1,2,...,m, where
appropriate.

7. Accuracy

See the discussion in Section 8.2.

8. Further Comments

8.1. Timing

The time taken by the routine depends on the complexity of the
coefficient functions, whether they or their derivatives are
rapidly changing, the tolerance demanded, and how many iterations
are needed to obtain convergence. The amount of work per
iteration is roughly doubled when TOL is divided by 16. To make
the most economical use of the routine, one should try to obtain
good initial values for ELAM and DELAM, and, where appropriate,
good asymptotic formulae. The boundary matching points should not
be set unnecessarily close to singular points. The extra time
needed to compute the eigenfunction is principally the cost of
one additional integration once the eigenvalue has been found.

8.2. General Description of the Algorithm

A shooting method, for differential equation problems containing
unknown parameters, relies on the construction of a 'miss-
distance function', which for given trial values of the
parameters measures how far the conditions of the problem are
from being met. The problem is then reduced to one of finding the
values of the parameters for which the miss-distance function is
zero, that is to a root-finding process. Shooting methods differ
mainly in how the miss-distance is defined.

This routine defines a miss-distance f((lambda)) based on the
rotation around the origin of the point P(x)=(p(x)y'(x),y(x)) in
the Phase Plane as the solution proceeds from a to b. The
boundary-conditions define the ray (i.e., two-sided line through
the origin) on which p(x) should start, and the ray on which it
should finish. The eigenvalue index k defines the total number of
half-turns it should make. Numerical solution is actually done by
matching point x=c. Then f((lambda)) is taken as the angle
between the rays to the two resulting points P (c) and P (c). A
                                              a         b
```

```
relative scaling of the py' and y axes, based on the behaviour of
the coefficient functions p and q, is used to improve the
numerical behaviour.



         Please see figure in printed Reference Manual

The resulting function f((lambda)) is monotonic over -
                                        ddq
infty<(lambda)<infty, increasing if  ---------->0 and decreasing
                                     dd(lambda)
       ddq
if  ----------<0, with a unique zero at the desired eigenvalue
    dd(lambda)
~~~~~~~~
(lambda). The routine measures f((lambda)) in units of a half-
turn. This means that as (lambda) increases, f((lambda)) varies
by about 1 as each eigenvalue is passed. (This feature implies
that the values of f((lambda)) at successive iterations -
especially in the early stages of the iterative process - can be
used with suitable extrapolation or interpolation to help the
choice of initial estimates for eigenvalues near to the one
currently being found.)

The routine actually computes a value for f((lambda)) with
errors, arising from the local errors of the differential
equation code and from the asymptotic formulae provided by the
user if singular points are involved. However, the error estimate
output in DELAM is usually fairly realistic, in that the actual
       ~~~~~~~~
error |(lambda)-ELAM| is within an order of magnitude of DELAM.

We pass the values of (beta), (phi), (rho) across through REPORT
rather than converting them to values of y, y' inside D02KEF, for
the following reasons. First, there may be cases where auxiliary
quantities can be more accurately computed from the Pruefer
variables than from y and y'. Second, in singular problems on an
infinite interval y and y' may underflow towards the end of the
range, whereas the Pruefer variables remain well-behaved. Third,
with high-order eigenvalues (and therefore highly oscillatory
eigenfunctions) the eigenfunction may have a complete oscillation
(or more than one oscillation) between two mesh points, so that
values of y and y' at mesh points give a very poor representation
of the curve. The probable behaviour of the Pruefer variables in
this case is that (beta) and (rho) vary slowly whilst (phi)
increases quickly: for all three Pruefer variables linear
interpolation between the values at adjacent mesh points is
probably sufficiently accurate to yield acceptable intermediate
values of (beta), (phi), (rho) (and hence of y,y') for graphical
```

purposes.

Similar considerations apply to the exponentially decaying 'tails
Here (phi) has approximately constant value whilst (rho)
increases rapidly in the direction of integration, though the
step length is generally fairly small over such a range.

If the solution is output through REPORT at x-values which are
too widely spaced, the step length can be controlled by choosing
HMAX suitably, or, preferably, by reducing TOL. Both these
choices will lead to more accurate eigenvalues and eigenfunctions
but at some computational cost.

8.3. The Position of the Shooting Matching Point c

This point is always one of the values x_i in array XPOINT. It may
be specified using the parameter MATCH. The default value is
chosen to be the value of that x_i ,2<=i<=m-1, that lies closest to
the middle of the interval [x_2 ,x_{m-1}]. If there is a tie, the
rightmost candidate is chosen. In particular if there are no
break-points then c=x_{m-1} (=x_3) - that is the shooting is from left
to right in this case. A break-point may be inserted purely to
move c to an interior point of the interval, even though the form
of the equations does not require it. This often speeds up
convergence especially with singular problems.

Note that the shooting method used by the code integrates first
from the left-hand end x_l , then from the right-hand end x_r , to
meet at the matching point c in the middle. This will of course
be reflected in printed or graphical output. The diagram shows a
possible sequence of nine mesh points (tau) through (tau) in
the order in which they appear, assuming there are just two sub-
intervals (so m=5).

Figure 1
Please see figure in printed Reference Manual

Since the shooting method usually fails to match up the two 'legs
p(x)y' or both, at the matching point c. The code in fact 'shares
large jump does not imply an inaccurate eigenvalue, but implies
either

(a) a badly chosen matching point: if q(x;(lambda)) has a '

humped' shape, c should be chosen near the maximum value of q, especially if q is negative at the ends of the interval.

(b) An inherently ill-conditioned problem, typically one where another eigenvalue is pathologically close to the one being sought. In this case it is extremely difficult to obtain an accurate eigenfunction.

In Section 9 below, we find the 11th eigenvalue and corresponding eigenfunction of the equation

$$y''+((\lambda)-x-2/x)y=0 \quad \text{on} \quad 0<x<\infty$$

the boundary conditions being that y should remain bounded as x tends to 0 and x tends to infty. The coding of this problem is discussed in detail in Section 8.5.

The choice of matching point c is open. If we choose c=30.0 as in the D02KDF(*) example program we find that the exponentially increasing component of the solution dominates and we get extremely inaccurate values for the eigenfunction (though the eigenvalue is determined accurately). The values of the eigenfunction calculated with c=30.0 are given schematically in Figure 2.

Figure 2
Please see figure in printed Reference Manual

If we choose c as the maximum of the hump in q(x;(lambda)) (see (a) above) we instead obtain the accurate results given in Figure 3.

Figure 3
Please see figure in printed Reference Manual

8.4. Examples of Coding the COEFFN Routine

Coding COEFFN is straightforward except when break-points are needed. The examples below show:

(a) a simple case,

(b) a case in which discontinuities in the coefficient functions or their derivatives necessitate break-points, and

(c) a case where break-points together with the HMAX parameter

are an efficient way to deal with a coefficient function
that is well-behaved except over one short interval.

Example A

The modified Bessel equation

$$x(xy')'+((\lambda)x^2-(\nu)^2)y=0$$

Assuming the interval of solution does not contain the origin, dividing through by x, we have $p(x)=x$, $q(x;(\lambda))=(\lambda)x-(\nu)^2/x$. The code could be

```
      SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
      P = X
      Q = ELAM*X + NU*NU/X
      DQDL = X
      RETURN
      END
```

where NU (standing for (nu)) is a real variable that might be defined in a DATA statement, or might be in user-declared COMMON so that its value could be set in the main program.

Example B

The Schroedinger equation

$$y''+((\lambda)+q(x))y=0$$

where $q(x)=\begin{cases} x^2-10 & (|x|\le 4), \\ 6/|x| & (|x|>4), \end{cases}$

over some interval 'approximating to (-infty,infty)', say [-20, 20]. Here we need break-points at +- 4, forming three sub-intervals $i_1=[-20,-4]$, $i_2=[-4,4]$, $i_3=[4,20]$. The code could be

```
      SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
      IF (JINT.EQ.2) THEN
        Q = ELAM + X*X - 10.0E0
      ELSE
        Q = ELAM + 6.0E0/ABS(X)
      ENDIF
```

```
      P = 1.0E0
      DQDL = 1.0E0
      RETURN
      END
```

The array XPOINT would contain the values x , -20.0, -4.0, +4.0,
                                           1
+20.0, x and m would be 6. The choice of appropriate values for
        6
x and x depends on the form of the asymptotic formula computed
 1   6
by BDYVAL and the technique is discussed in the next subsection.

Example C

$$y''+(\lambda)(1-2e^{-100x^2})y=0, \text{ over } -10\leq x\leq 10$$

Here q(x;(lambda)) is nearly constant over the range except for a sharp inverted spike over approximately -0.1<=x<=0.1. There is a danger that the routine will build up to a large step size and ' step over' the spike without noticing it. By using break-points - say at +- 0.5 - one can restrict the step size near the spike without impairing the efficiency elsewhere.

The code for COEFFN could be

```
      SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
      P = 1.0E0
      DQDL = 1.0E0 - 2.0E0*EXP(-100.0E0*X*X)
      Q = ELAM*DQDL
      RETURN
      END
```

XPOINT might contain -0.0, -10.0, -0.5, 0.5, 10.0, 10.0 (assuming +- 10 are regular points) and m would be 6. HMAX(1,j), j=1,2,3 might contain 0.0, 0.1 and 0.0.

8.5. Examples of Boundary Conditions at Singular Points

Quoting from Bailey [2] page 243: 'Usually... the differential equation has two essentially different types of solution near a singular point, and the boundary condition there merely serves to distinguish one kind from the other. This is the case in all the standard examples of mathematical physics.'

In most cases the behaviour of the ratio p(x)y'/y near the point is quite different for the two types of solution. Essentially what the user provides through his BDYVAL routine is an approximation to this ratio, valid as x tends to the singular point (SP).

The user must decide (a) how accurate to make this approximation or asymptotic formula, for example how many terms of a series to use, and (b) where to place the boundary matching point (BMP) at which the numerical solution of the differential equation takes over from the asymptotic formula. Taking the BMP closer to the SP will generally improve the accuracy of the asymptotic formula, but will make the computation more expensive as the Pruefer differential equations generally become progressively more ill-behaved as the SP is approached. The user is strongly recommended to experiment with placing the BMPs. In many singular problems quite crude asymptotic formulae will do. To help the user avoid needlessly accurate formulae, D02KEF outputs two 'sensitivity coefficients' (sigma) ,(sigma) which estimate how much the
                    l   r
errors at the BMP's affect the computed eigenvalue. They are described in detail below, see Section 8.6.

Example of coding BDYVAL:

The example below illustrates typical situations:

$$y''+\left((\lambda)-x-\frac{2}{x^2}\right)y=0, \text{ for } 0<x<\infty$$

the boundary conditions being that y should remain bounded as x tends to 0 and x tends to infty.

At the end x=0 there is one solution that behaves like x^2 and another that behaves like x^{-1}. For the first of these solutions p(x)y'/y is asymptotically 2/x while for the second it is asymptotically -1/x. Thus the desired ratio is specified by setting

YL(1)=x and YL(2)=2.0.

At the end x=infty the equation behaves like Airy's equation shifted through (lambda), i.e., like y''-ty=0 where t=x-(lambda), so again there are two types of solution. The solution we require behaves as

$$\exp\left(-\frac{2}{3}t^{3/2}\right)/\sqrt{4\sqrt{t}}$$

and the other as

$$\exp\left(+\frac{2}{3}t^{3/2}\right)/\sqrt{4\sqrt{t}}$$

once, the desired solution has $p(x)y'/y \sim -\sqrt{\sqrt{t}}$ so that we could set $YR(1) = 1.0$ and $YR(2) = -\sqrt{x-(\text{lambda})}$. The complete subroutine might thus be

```
      SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)
      real XL, XR, ELAM, YL(3), YR(3)
      YL(1) = XL
      YL(2) = 2.0E0
      YR(1) = 1.0E0
      YR(2) = -SQRT(XR - ELAM)
      RETURN
      END
```

Clearly for this problem it is essential that any value given by D02KEF to XR is well to the right of the value of ELAM, so that the user must vary the right-hand BMP with the eigenvalue index k
$$k$$
function Ai(x), so there is no problem in estimating ELAM.

More accurate asymptotic formulae are easily found - near x=0 by the standard Frobenius method, and near x=infty by using standard asymptotics for Ai(x), Ai'(x) (see [1], p. 448). For example, by the Frobenius method the solution near x=0 has the expansion

$$y = x^{2}\left(c_0 + c_1 x + c_2 x^{2} + \ldots\right)$$

with

$$c_0 = 1, c_1 = 0, c_2 = -\frac{(\text{lambda})}{8}, c_3 = \frac{1}{18}, \ldots, c_n = \frac{c_{n-3} - (\text{lambda})c_{n-2}}{n(n+3)}$$

This yields

$$\frac{p(x)y'}{y} = \frac{2 - \frac{2}{5}(\lambda)x^2 + \ldots}{x\left(1 - \frac{(\lambda)}{10}x^2 + \ldots\right)}.$$

8.6. The Sensitivity Parameters $(\sigma)_l$ and $(\sigma)_r$

The sensitivity parameters $(\sigma)_l$, $(\sigma)_r$ (held in HMAX(1,m-1) and HMAX(1,m) on output) estimate the effect of errors in the boundary conditions. For sufficiently small errors $(\Delta)y$, $(\Delta)py'$ in y and py' respectively, the relations

$$(\Delta)(\lambda)\tilde{}=(y.(\Delta)py'-py'.(\Delta)y)\ (\sigma)_l \qquad (\Delta)(\lambda)\tilde{}=(y.(\Delta)py'-py'.(\Delta)y)\ (\sigma)_r$$

are satisfied where the subscripts l, r denote errors committed at left- and right-hand BMP's respectively, and $(\Delta)(\lambda)$ denotes the consequent error in the computed eigenvalue.

8.7. Missed Zeros

This is a pitfall to beware of at a singular point. If the BMP is chosen so far from the SP that a zero of the desired eigenfunction lies in between them, then the routine will fail to number of zeros of its eigenfunction, the result will be that:

(a) The wrong eigenvalue will be computed for the given index k - in fact some $(\lambda)_{k+k'}$ will be found where k'>=1.

(b) The same index k can cause convergence to any of several eigenvalues depending on the initial values of ELAM and DELAM.

It is up to the user to take suitable precautions - for instance by varying the position of the BMP's in the light of his knowledge of the asymptotic behaviour of the eigenfunction at different eigenvalues.

9. Example

```
     To find the 11th eigenvalue and eigenfunction of the example of
     Section 8.5, using the simple asymptotic formulae for the
     boundary conditions.

     Comparison of the results from this example program with the
     corresponding results from D02KDF(*) example program shows that
     similar output is produced from the routine MONIT, followed by
     the eigenfunction values from REPORT, and then a further line of
     information from MONIT (corresponding to the integration to find
     the eigenfunction). Final information is printed within the
     example program exactly as with D02KDF(*).

                                                 3 _
     Note the discrepancy at the matching point c(=\/4 , the maximum
     of q(x;(lambda)), in this case) between the solutions obtained by
     integrations from left and right end-points.

     The example program is not reproduced here. The source code for
     all example programs is distributed with the NAG Foundation
     Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          D02 -- Ordinary Differential Equations                    D02RAF
                  D02RAF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          D02RAF solves the two-point boundary-value problem with general
          boundary conditions for a system of ordinary differential
          equations, using a deferred correction technique and Newton
          iteration.

          2. Specification

                 SUBROUTINE D02RAF (N, MNP, NP, NUMBEG, NUMMIX, TOL, INIT,
                1                   X, Y, IY, ABT, FCN, G, IJAC, JACOBF,
                2                   JACOBG, DELEPS, JACEPS, JACGEP, WORK,
                3                   LWORK, IWORK, LIWORK, IFAIL)
                 INTEGER           N, MNP, NP, NUMBEG, NUMMIX, INIT, IY,
                1                  IJAC, LWORK, IWORK(LIWORK), LIWORK, IFAIL
                 DOUBLE PRECISION TOL, X(MNP), Y(IY,MNP), ABT(N), DELEPS,
                1                  WORK(LWORK)
                 EXTERNAL          FCN, G, JACOBF, JACOBG, JACEPS, JACGEP
```

3. Description

D02RAF solves a two-point boundary-value problem for a system of n ordinary differential equations in the interval (a,b) with b>a. The system is written in the form

$$y_i' = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1,2,\ldots,n \qquad (1)$$

and the derivatives f_i are evaluated by a subroutine FCN supplied by the user. With the differential equations (1) must be given a system of n (nonlinear) boundary conditions

$$g_i(y(a), y(b)) = 0, \quad i = 1,2,\ldots,n$$

where

$$y(x) = [y_1(x), y_2(x), \ldots, y_n(x)]^T. \qquad (2)$$

The functions g_i are evaluated by a subroutine G supplied by the user. The solution is computed using a finite-difference technique with deferred correction allied to a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1].

The user must supply an absolute error tolerance and may also supply an initial mesh for the finite-difference equations and an initial approximate solution (alternatively a default mesh and approximation are used). The approximate solution is corrected using Newton iteration and deferred correction. Then, additional points are added to the mesh and the solution is recomputed with the aim of making the error everywhere less than the user's tolerance and of approximately equidistributing the error on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

The Newton iteration requires Jacobian matrices

$$\left(\frac{\partial f}{\partial y}\right) \quad \left(\frac{\partial g}{\partial y(a)}\right) \quad \left(\frac{\partial g}{\partial y(b)}\right)$$

```
(    i)  (     i  )     (     i  )
( ----), ( -------) and ( -------).
( ddy )  ( ddy (a))     ( ddy (b))
(    j)  (    j   )     (    j   )
```

These may be supplied by the user through subroutines JACOBF for
(ddf)
(i)
(----) and JACOBG for the others. Alternatively the Jacobians
(ddy)
(j)
may be calculated by numerical differentiation using the
algorithm described in Curtis et al [2].

For problems of the type (1) and (2) for which it is difficult to determine an initial approximation from which the Newton iteration will converge, a continuation facility is provided. The user must set up a family of problems

```
      y'=f(x,y,(epsilon)),   g(y(a),y(b),(epsilon))=0,        (3)
```

where $f=[f_1, f_2, \ldots, f_n]^T$ etc, and where (epsilon) is a continuation parameter. The choice (epsilon)=0 must give a problem (3) which is easy to solve and (epsilon)=1 must define the problem whose solution is actually required. The routine solves a sequence of problems with (epsilon) values

$$0=(\text{epsilon})_1 < (\text{epsilon})_2 < \ldots < (\text{epsilon})_p = 1. \qquad (4)$$

The number p and the values $(\text{epsilon})_i$ are chosen by the routine so that each problem can be solved using the solution of its predecessor as a starting approximation. Jacobians $\frac{\partial f}{\partial (\text{epsilon})}$ and $\frac{\partial g}{\partial (\text{epsilon})}$ are required and they may be supplied by the user via routines JACEPS and JACGEP respectively or may be computed by numerical differentiation.

4. References

[1] Pereyra V (1979) PASVA3: An Adaptive Finite-Difference Fortran Program for First Order Nonlinear, Ordinary Boundary Problems. Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science.

(ed B Childs, M Scott, J W Daniel, E Denman and P Nelson) 76 Springer-Verlag.

[2] Curtis A R, Powell M J D and Reid J K (1974) On the Estimation of Sparse Jacobian Matrices. J. Inst. Maths Applics. 13 117--119.

5. Parameters

1: N -- INTEGER Input
On entry: the number of differential equations, n.
Constraint: N > 0.

2: MNP -- INTEGER Input
On entry: MNP must be set to the maximum permitted number of points in the finite-difference mesh. If LWORK or LIWORK (see below) is too small then internally MNP will be replaced by the maximum permitted by these values. (A warning message will be output if on entry IFAIL is set to obtain monitoring information.) Constraint: MNP >= 32.

3: NP -- INTEGER Input/Output
On entry: NP must be set to the number of points to be used in the initial mesh. Constraint: 4 <= NP <= MNP. On exit: the number of points in the final mesh.

4: NUMBEG -- INTEGER Input
On entry: the number of left-hand boundary conditions (that is the number involving y(a) only). Constraint: 0 <= NUMBEG < N.

5: NUMMIX -- INTEGER Input
On entry: the number of coupled boundary conditions (that is the number involving both y(a) and y(b)). Constraint: 0 <= NUMMIX <= N - NUMBEG.

6: TOL -- DOUBLE PRECISION Input
On entry: a positive absolute error tolerance. If
$$a=x_1<x_2<\ldots<x_{NP}=b$$
is the final mesh, $z_j(x_i)$ is the jth component of the approximate solution at x_i, and $y_j(x_i)$ is the jth component of the true solution of (1) and (2), then, except in extreme circumstances, it is expected that
$$|z_j(x_i)-y_j(x_i)|\leq TOL,\ i=1,2,\ldots,NP;\ j=1,2,\ldots,n. \quad (5)$$
Constraint: TOL > 0.0.

```
 7:  INIT -- INTEGER                                          Input
     On entry: indicates whether the user wishes to supply an
     initial mesh and approximate solution (INIT /= 0) or whether
     default values are to be used, (INIT = 0).

 8:  X(MNP) -- DOUBLE PRECISION array                Input/Output
     On entry: the user must set X(1) = a and X(NP) = b. If INIT
     = 0 on entry a default equispaced mesh will be used,
     otherwise the user must specify a mesh by setting X(i)=x ,
                                                            i
     for i = 2,3,...NP-1. Constraints:
            X(1) < X(NP), if INIT = 0,

            X(1) < X(2) <... < X(NP), if INIT /= 0.
     On exit: X(1),X(2),...,X(NP) define the final mesh (with
     the returned value of NP) and X(1) = a and X(NP) = b.

 9:  Y(IY,MNP) -- DOUBLE PRECISION array             Input/Output
     On entry: if INIT = 0, then Y need not be set.

     If INIT /= 0, then the array Y must contain an initial
     approximation to the solution such that Y(j,i) contains an
     approximation to
                 y (x ) , i=1,2,...,NP ; j=1,2,...,n.
                  j  i
     On exit: the approximate solution z (x ) satisfying (5) on
                                        j  i
     the final mesh, that is
             Y(j,i)=z (x ) , i=1,2,...,NP ; j=1,2,...,n,
                     j  i
     where NP is the number of points in the final mesh. If an
     error has occurred then Y contains the latest approximation
     to the solution. The remaining columns of Y are not used.

10:  IY -- INTEGER                                           Input
     On entry:
     the first dimension of the array Y as declared in the
     (sub)program from which D02RAF is called.
     Constraint: IY >= N.

11:  ABT(N) -- DOUBLE PRECISION array                       Output
     On exit: ABT(i), for i=1,2,...,n, holds the largest
     estimated error (in magnitude) of the ith component of the
     solution over all mesh points.

12:  FCN -- SUBROUTINE, supplied by the user.
                                                 External Procedure
     FCN must evaluate the functions f  (i.e., the derivatives
                                     i
     y' ) at a general point x for a given value of (epsilon),
```

```
  i
the continuation parameter (see Section 3).

Its specification is:

      SUBROUTINE FCN (X, EPS, Y, F, N)
      INTEGER          N
      DOUBLE PRECISION X, EPS, Y(N), F(N)

 1:  X -- DOUBLE PRECISION                                Input
     On entry: the value of the argument x.

 2:  EPS -- DOUBLE PRECISION                              Input
     On entry: the value of the continuation parameter,
     (epsilon). This is 1 if continuation is not being used.

 3:  Y(N) -- DOUBLE PRECISION array                       Input
     On entry: the value of the argument y , for
                                          i
     i=1,2,...,n.

 4:  F(N) -- DOUBLE PRECISION array                      Output
     On exit: the values of f , for i=1,2,...,n.
                             i

 5:  N -- INTEGER                                         Input
     On entry: the number of equations.
FCN must be declared as EXTERNAL in the (sub)program
from which D02RAF is called. Parameters denoted as
Input must not be changed by this procedure.

13:  G -- SUBROUTINE, supplied by the user.
                                      External Procedure
     G must evaluate the boundary conditions in equation (3) and
     place them in the array BC.

     Its specification is:

       SUBROUTINE G (EPS, YA, YB, BC, N)
       INTEGER          N
       DOUBLE PRECISION EPS, YA(N), YB(N), BC(N)

      1:  EPS -- DOUBLE PRECISION                         Input
          On entry: the value of the continuation parameter,
          (epsilon). This is 1 if continuation is not being used.

      2:  YA(N) -- DOUBLE PRECISION array                 Input
          On entry: the value y (a), for i=1,2,...,n.
                               i
```

```
 3:  YB(N) -- DOUBLE PRECISION array                    Input
     On entry: the value y (b), for i=1,2,...,n.
                          i

 4:  BC(N) -- DOUBLE PRECISION array                   Output
     On exit: the values g (y(a),y(b),(epsilon)), for
                          i
     i=1,2,...,n. These must be ordered as follows:
       (i)   first, the conditions involving only y(a) (see
             NUMBEG description above);

       (ii)  next, the NUMMIX coupled conditions involving
             both y(a) and y(b) (see NUMMIX description
             above); and,

       (iii) finally, the conditions involving only y(b) (N-
             NUMBEG-NUMMIX).

 5:  N -- INTEGER                                       Input
     On entry: the number of equations, n.
G must be declared as EXTERNAL in the (sub)program from
which D02RAF is called. Parameters denoted as Input
must not be changed by this procedure.

14:  IJAC -- INTEGER                                    Input
     On entry: indicates whether or not the user is supplying
     Jacobian evaluation routines. If IJAC /= 0 then the user
     must supply routines JACOBF and JACOBG and also, when
     continuation is used, routines JACEPS and JACGEP. If IJAC =
     0 numerical differentiation is used to calculate the
     Jacobian and the routines D02GAZ, D02GAY, D02GAZ and D02GAX
                                      ( ddf )
                                      (    i)
     JACOBF must evaluate the Jacobian ( ----) for i,j=1,2,...,n,
                                      ( ddy )
                                      (    j)
     given x and y , for j=1,2,...,n.
                 j

     Its specification is:

           SUBROUTINE JACOBF (X, EPS, Y, F, N)
           INTEGER          N
           DOUBLE PRECISION X, EPS, Y(N), F(N,N)

     1:  X -- DOUBLE PRECISION                          Input
         On entry: the value of the argument x.

     2:  EPS -- DOUBLE PRECISION                        Input
         On entry: the value of the continuation parameter
```

(epsilon). This is 1 if continuation is not being used.

3: Y(N) -- DOUBLE PRECISION array Input
On entry: the value of the argument y_i, for $i=1,2,\ldots,n$.

4: F(N,N) -- DOUBLE PRECISION array Output
On exit: F(i,j) must be set to the value of $\frac{\partial f_i}{\partial y_j}$, evaluated at the point (x,y), for i,j=1,2,...,n.

5: N -- INTEGER Input
On entry: the number of equations, n.

JACOBF must be declared as EXTERNAL in the (sub)program from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

16: JACOBG -- SUBROUTINE, supplied by the user.
External Procedure

JACOBG must evaluate the Jacobians $\left(\frac{\partial g_i}{\partial y_j}(a)\right)$ and $\left(\frac{\partial g_i}{\partial y_j}(b)\right)$.

The ordering of the rows of AJ and BJ must correspond to the ordering of the boundary conditions described in the specification of subroutine G above.

Its specification is:

```
SUBROUTINE JACOBG (EPS, YA, YB, AJ, BJ, N)
INTEGER          N
DOUBLE PRECISION EPS, YA(N), YB(N), AJ(N,N), BJ
1                (N,N)
```

1: EPS -- DOUBLE PRECISION Input
On entry: the value of the continuation parameter, (epsilon). This is 1 if continuation is not being used.

2: YA(N) -- DOUBLE PRECISION array Input
On entry: the value $y_i(a)$, for i=1,2,...,n.

3: YB(N) -- DOUBLE PRECISION array Input
On entry: the value $y_i(b)$, for i=1,2,...,n.

```
 4:  AJ(N,N) -- DOUBLE PRECISION array                Output
                                                      ddg
                                                         i
     On exit: AJ(i,j) must be set to the value        -------,
                                                      ddy (a)
                                                         j
     for i,j=1,2,...,n.

 5:  BJ(N,N) -- DOUBLE PRECISION array                Output
                                                      ddg
                                                         i
     On exit: BJ(i,j) must be set to the value        -------,
                                                      ddy (b)
                                                         j
     for i,j=1,2...,n.

 6:  N -- INTEGER                                      Input
     On entry: the number of equations, n.
JACOBG must be declared as EXTERNAL in the (sub)program
from which D02RAF is called. Parameters denoted as
Input must not be changed by this procedure.

17:  DELEPS -- DOUBLE PRECISION                     Input/Output
     On entry: DELEPS must be given a value which specifies
     whether continuation is required. If DELEPS <= 0.0 or DELEPS
     >= 1.0 then it is assumed that continuation is not required.
     If 0.0 < DELEPS < 1.0 then it is assumed that continuation
     is required unless DELEPS < square root of machine precision
     when an error exit is taken. DELEPS is used as the increment
     (epsilon) -(epsilon)  (see (4)) and the choice DELEPS = 0.1
              2          1
     is recommended. On exit: an overestimate of the increment
     (epsilon) -(epsilon)    (in fact the value of the increment
              p          p-1
     which would have been tried if the restriction (epsilon) =1
                                                             p
     had not been imposed). If continuation was not requested
     then DELEPS = 0.0.

     If continuation is not requested then the parameters JACEPS
     and JACGEP may be replaced by dummy actual parameters in the
     call to D02RAF. (D02GAZ and D02GAX respectively may be used
     as the dummy parameters.)

18:  JACEPS -- SUBROUTINE, supplied by the user.
                                              External Procedure
                                            ddf
                                               i
     JACEPS must evaluate the derivative  ----------- given x and
```

```
                                          dd(epsilon)
     y if continuation is being used.

     Its specification is:

           SUBROUTINE JACEPS (X, EPS, Y, F, N)
           INTEGER          N
           DOUBLE PRECISION X, EPS, Y(N), F(N)

      1:  X -- DOUBLE PRECISION                                Input
          On entry: the value of the argument x.

      2:  EPS -- DOUBLE PRECISION                              Input
          On entry: the value of the continuation parameter,
          (epsilon).

      3:  Y(N) -- DOUBLE PRECISION array                       Input
          On entry: the solution values y  at the point x, for
                                         i
          i=1,2,...,n.

      4:  F(N) -- DOUBLE PRECISION array                      Output
                                               ddf
                                                  i
          On exit: F(i) must contain the value  ----------- at
                                              dd(epsilon)
          the point (x,y), for i=1,2,...,n.

      5:  N -- INTEGER                                         Input
          On entry: the number of equations, n.
     JACEPS must be declared as EXTERNAL in the (sub)program
     from which D02RAF is called. Parameters denoted as
     Input must not be changed by this procedure.

     19:  JACGEP -- SUBROUTINE, supplied by the user.
                                               External Procedure
                                              ddg
                                                 i
          JACGEP must evaluate the derivatives  ----------- if
                                              dd(epsilon)
          continuation is being used.

          Its specification is:

                SUBROUTINE JACGEP (EPS, YA, YB, BCEP, N)
                INTEGER          N
                DOUBLE PRECISION EPS, YA(N), YB(N), BCEP(N)

           1:  EPS -- DOUBLE PRECISION                         Input
               On entry: the value of the continuation parameter,
```

```
          (epsilon).

       2: YA(N) -- DOUBLE PRECISION array                    Input
          On entry: the value of y (a), for i=1,2,...,n.
                                  i

       3: YB(N) -- DOUBLE PRECISION array                    Input
          On entry: the value of y (b), for i=1,2,...,n.
                                  i

       4: BCEP(N) -- DOUBLE PRECISION array                 Output
          On exit: BCEP(i) must contain the value of
              ddg
                 i
           -----------, for i=1,2,...,n.
           dd(epsilon)

       5: N -- INTEGER                                       Input
          On entry: the number of equations, n.
     JACGEP must be declared as EXTERNAL in the (sub)program
     from which D02RAF is called. Parameters denoted as
     Input must not be changed by this procedure.

20:  WORK(LWORK) -- DOUBLE PRECISION array              Workspace

21:  LWORK -- INTEGER                                       Input
     On entry:
     the dimension of the array WORK as declared in the
     (sub)program from which D02RAF is called.
                               2         2
     Constraint: LWORK>=MNP*(3N +6N+2)+4N +3N.

22:  IWORK(LIWORK) -- INTEGER array                     Workspace

23:  LIWORK -- INTEGER                                      Input
     On entry:
     the dimension of the array IWORK as declared in the
     (sub)program from which D02RAF is called.
     Constraints:
          LIWORK>=MNP*(2*N+1)+N, if IJAC /= 0,

                               2
          LIWORK>=MNP*(2*N+1)+N +4*N+2, if IJAC = 0.

24:  IFAIL -- INTEGER                                Input/Output
     For this routine, the normal use of IFAIL is extended to
     control the printing of error and warning messages as well
     as specifying hard or soft failure (see the Essential
     Introduction).
```

```
     Before entry, IFAIL must be set to a value with the decimal
     expansion cba, where each of the decimal digits c, b and a
     must have a value of 0 or 1.
     a=0   specifies hard failure, otherwise soft failure;

     b=0   suppresses error messages, otherwise error messages
           will be printed (see Section 6);

     c=0   suppresses warning messages, otherwise warning
           messages will be printed (see Section 6).
     The recommended value for inexperienced users is 110 (i.e.,
     hard failure with all messages printed).

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
     One or more of the parameters N, MNP, NP, NUMBEG, NUMMIX,
     TOL, DELEPS, LWORK or LIWORK has been incorrectly set, or X
     (1) >= X(NP) or the mesh points X(i) are not in strictly
     ascending order.

IFAIL= 2
     A finer mesh is required for the accuracy requested; that is
     MNP is not large enough. This error exit normally occurs
     when the problem being solved is difficult (for example,
     there is a boundary layer) and high accuracy is requested. A
     poor initial choice of mesh points will make this error exit
     more likely.

IFAIL= 3
     The Newton iteration has failed to converge. There are
     several possible causes for this error:
     (i)   faulty coding in one of the Jacobian calculation
           routines;

     (ii)  if IJAC = 0 then inaccurate Jacobians may have been
           calculated numerically (this is a very unlikely
           cause); or,

     (iii) a poor initial mesh or initial approximate solution
           has been selected either by the user or by default or
```

```
          there are not enough points in the initial mesh.
          Possibly, the user should try the continuation
          facility.

IFAIL= 4
     The Newton iteration has reached round-off error level. It
     could be however that the answer returned is satisfactory.
     The error is likely to occur if too high an accuracy is
     requested.

IFAIL= 5
     The Jacobian calculated by JACOBG (or the equivalent matrix
     calculated by numerical differentiation) is singular. This
     may occur due to faulty coding of JACOBG or, in some
     circumstances, to a zero initial choice of approximate
     solution (such as is chosen when INIT = 0).

IFAIL= 6
     There is no dependence on (epsilon) when continuation is
     being used. This can be due to faulty coding of JACEPS or
     JACGEP or, in some circumstances, to a zero initial choice
     of approximate solution (such as is chosen when INIT = 0).

IFAIL= 7
     DELEPS is required to be less than machine precision for
     continuation to proceed. It is likely that either the
     problem (3) has no solution for some value near the current
     value of (epsilon) (see the advisory print out from D02RAF)
     or that the problem is so difficult that even with
     continuation it is unlikely to be solved using this routine.
     If the latter cause is suspected then using more mesh points
     initially may help.

IFAIL= 8
     Indicates that a serious error has occurred in a call to
     D02RAF. Check all array subscripts and subroutine parameter
     lists in calls to D02RAF. Seek expert help.

IFAIL= 9
     Indicates that a serious error has occurred in a call to
     D02RAR. Check all array subscripts and subroutine parameter
     lists in calls to D02RAF. Seek expert help.

7. Accuracy

The solution returned by the routine will be accurate to the
user's tolerance as defined by the relation (5) except in extreme
circumstances. The final error estimate over the whole mesh for
each component is given in the array ABT. If too many points are
specified in the initial mesh, the solution may be more accurate
```

than requested and the error may not be approximately equidistributed.

8. Further Comments

There are too many factors present to quantify the timing. The time taken by the routine is negligible only on very simple problems.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation.

In the case where the user wishes to solve a sequence of similar problems, the use of the final mesh and solution from one case as the initial mesh is strongly recommended for the next.

9. Example

We solve the differential equation

$$y'''=-yy''-2(\text{epsilon})(1-y'^{\,2})$$

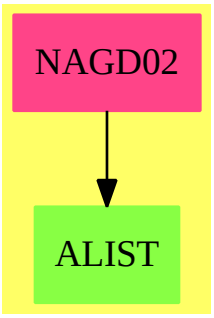
with (epsilon)=1 and boundary conditions

$$y(0)=y'(0)=0,\quad y'(10)=1$$

to an accuracy specified by TOL=1.0E-4. The continuation facility is used with the continuation parameter (epsilon) introduced as in the differential equation above and with DELEPS = 0.1 initially. (The continuation facility is not needed for this problem and is used here for illustration.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

15.10.1 NagOrdinaryDifferentialEquationsPackage (NAGD02)



Exports:

d02bbf	d02bhf	d02cjf	d02ejf	d02gaf
d02gbf	d02kef	d02kef	d02raf	

— package NAGD02 NagOrdinaryDifferentialEquationsPackage —

```
)abbrev package NAGD02 NagOrdinaryDifferentialEquationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Mon Jun 20 17:56:33 1994
++ Description:
++ This package uses the NAG Library to calculate the numerical solution of
++ ordinary differential equations. There are two main types of problem,
++ those in which all boundary conditions are specified at one point
++ (initial-value problems), and those in which the boundary
++ conditions are distributed between two or more points (boundary-
++ value problems and eigenvalue problems). Routines are available
++ for initial-value problems, two-point boundary-value problems and
++ Sturm-Liouville eigenvalue problems.

NagOrdinaryDifferentialEquationsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    d02bbf : (DoubleFloat,Integer,Integer,Integer,_
        DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
        Union(fn:FileName,fp:Asp7(FCN)),_
        Union(fn:FileName,fp:Asp8(OUTPUT))) -> Result
     ++ d02bbf(xend,m,n,irelab,x,y,tol,ifail,fcn,output)
     ++ integrates a system of first-order ordinary differential
     ++ equations over an interval with suitable initial conditions,
     ++ using a Runge-Kutta-Merson method, and returns the solution at
     ++ points specified by the user.
     ++ See \downlink{Manual Page}{manpageXXd02bbf}.
```

```
d02bhf : (DoubleFloat,Integer,Integer,DoubleFloat,_
    DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp9(G)),_
    Union(fn:FileName,fp:Asp7(FCN))) -> Result
 ++ d02bhf(xend,n,irelab,hmax,x,y,tol,ifail,g,fcn)
 ++ integrates a system of first-order ordinary differential
 ++ equations over an interval with suitable initial conditions,
 ++ using a Runge-Kutta-Merson method, until a user-specified
 ++ function of the solution is zero.
 ++ See \downlink{Manual Page}{manpageXXd02bhf}.
d02cjf : (DoubleFloat,Integer,Integer,DoubleFloat,_
    String,DoubleFloat,Matrix DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp9(G)),Union(fn:FileName,fp:Asp7(FCN)),_
    Union(fn:FileName,fp:Asp8(OUTPUT))) -> Result
 ++ d02cjf(xend,m,n,tol,relabs,x,y,ifail,g,fcn,output)
 ++ integrates a system of first-order ordinary differential
 ++ equations over a range with suitable initial conditions, using a
 ++ variable-order, variable-step Adams method until a user-specified
 ++ function, if supplied, of the solution is zero, and returns the
 ++ solution at points specified by the user, if desired.
 ++ See \downlink{Manual Page}{manpageXXd02cjf}.
d02ejf : (DoubleFloat,Integer,Integer,String,_
    Integer,DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp9(G)),Union(fn:FileName,fp:Asp7(FCN)),_
    Union(fn:FileName,fp:Asp31(PEDERV)),_
    Union(fn:FileName,fp:Asp8(OUTPUT))) -> Result
 ++ d02ejf(xend,m,n,relabs,iw,x,y,tol,ifail,g,fcn,pederv,output)
 ++ integrates a stiff system of first-order ordinary
 ++ differential equations over an interval with suitable initial
 ++ conditions, using a variable-order, variable-step method
 ++ implementing the Backward Differentiation Formulae (BDF), until a
 ++ user-specified function, if supplied, of the solution is zero,
 ++ and returns the solution at points specified by the user, if
 ++ desired.
 ++ See \downlink{Manual Page}{manpageXXd02ejf}.
d02gaf : (Matrix DoubleFloat,Matrix DoubleFloat,Integer,DoubleFloat,_
    DoubleFloat,DoubleFloat,Integer,Integer,Integer,Matrix DoubleFloat,_
    Integer,Integer,Union(fn:FileName,fp:Asp7(FCN))) -> Result
 ++ d02gaf(u,v,n,a,b,tol,mnp,lw,liw,x,np,ifail,fcn)
 ++ solves the two-point boundary-value problem with assigned
 ++ boundary values for a system of ordinary differential equations,
 ++ using a deferred correction technique and a Newton iteration.
 ++ See \downlink{Manual Page}{manpageXXd02gaf}.
d02gbf : (DoubleFloat,DoubleFloat,Integer,DoubleFloat,_
    Integer,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
    Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer,_
    Union(fn:FileName,fp:Asp77(FCNF)),_
    Union(fn:FileName,fp:Asp78(FCNG))) -> Result
 ++ d02gbf(a,b,n,tol,mnp,lw,liw,c,d,gam,x,np,ifail,fcnf,fcng)
 ++ solves a general linear two-point boundary value problem
```

```
    ++ for a system of ordinary differential equations using a deferred
    ++ correction technique.
    ++ See \downlink{Manual Page}{manpageXXd02gbf}.
   d02kef : (Matrix DoubleFloat,Integer,Integer,DoubleFloat,_
       Integer,Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,Integer,_
       Integer,Union(fn:FileName,fp:Asp10(COEFFN)),_
       Union(fn:FileName,fp:Asp80(BDYVAL))) -> Result
    ++ d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,
    ++        hmax,maxit,ifail,coeffn,bdyval)
    ++ finds a specified eigenvalue of a regular singular second-
    ++ order Sturm-Liouville system on a finite or infinite range, using
    ++ a Pruefer transformation and a shooting method. It also reports
    ++ values of the eigenfunction and its derivatives. Provision is
    ++ made for discontinuities in the coefficient functions or their
    ++ derivatives.
    ++ See \downlink{Manual Page}{manpageXXd02kef}.
    ++ ASP domains Asp12 and Asp33 are used to supply default
    ++ subroutines for the MONIT and REPORT arguments via their
    ++ \axiomOp{outputAsFortran} operation.
   d02kef : (Matrix DoubleFloat,Integer,Integer,DoubleFloat,_
       Integer,Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,Integer,_
       Integer,Union(fn:FileName,fp:Asp10(COEFFN)),_
       Union(fn:FileName,fp:Asp80(BDYVAL)),FileName,FileName) -> Result
    ++ d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,
    ++        hmax,maxit,ifail,coeffn,bdyval,monit,report)
    ++ finds a specified eigenvalue of a regular singular second-
    ++ order Sturm-Liouville system on a finite or infinite range, using
    ++ a Pruefer transformation and a shooting method. It also reports
    ++ values of the eigenfunction and its derivatives. Provision is
    ++ made for discontinuities in the coefficient functions or their
    ++ derivatives.
    ++ See \downlink{Manual Page}{manpageXXd02kef}.
    ++ Files \spad{monit} and \spad{report} will be used to define
    ++ the subroutines for the
    ++ MONIT and REPORT arguments.
    ++ See \downlink{Manual Page}{manpageXXd02gbf}.
   d02raf : (Integer,Integer,Integer,Integer,_
       DoubleFloat,Integer,Integer,Integer,Integer,Integer,Integer,_
       Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
       Union(fn:FileName,fp:Asp41(FCN,JACOBF,JACEPS)),_
       Union(fn:FileName,fp:Asp42(G,JACOBG,JACGEP))) -> Result
    ++ d02raf(n,mnp,numbeg,nummix,tol,init,iy,ijac,lwork,
    ++        liwork,np,x,y,deleps,ifail,fcn,g)
    ++ solves the two-point boundary-value problem with general
    ++ boundary conditions for a system of ordinary differential
    ++ equations, using a deferred correction technique and Newton
    ++ iteration.
    ++ See \downlink{Manual Page}{manpageXXd02raf}.
  Implementation ==> add
```

```
import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp7(FCN))
import Union(fn:FileName,fp:Asp8(OUTPUT))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(Matrix DoubleFloat)


d02bbf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_
    irelabArg:Integer,xArg:DoubleFloat,yArg:Matrix DoubleFloat,_
    tolArg:DoubleFloat,ifailArg:Integer,_
    fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_
    outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==
    pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
    if fcnArg case fn
      then outputAsFortran(fcnArg.fn)
      else outputAsFortran(fcnArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
    if outputArg case fn
      then outputAsFortran(outputArg.fn)
      else outputAsFortran(outputArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([fcnFilename, outputFilename]$Lisp,_
    "d02bbf",_
    ["xend"::S,"m"::S,"n"::S,"irelab"::S,"x"::S,"tol"::S,"ifail"::S,_
    "fcn"::S,"output"::S,"result"::S,"y"::S,"w"::S]$Lisp,_
    ["result"::S,"w"::S,"fcn"::S,"output"::S]$Lisp,_
    [["double"::S,"xend"::S,["result"::S,"m"::S,"n"::S]$Lisp_
    ,"x"::S,["y"::S,"n"::S]$Lisp,"tol"::S,["w"::S,"n"::S,7$Lisp]$Lisp,_
    "fcn"::S,"output"::S]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"irelab"::S,"ifail"::S]$Lisp]$Lisp,_
    ["result"::S,"x"::S,"y"::S,"tol"::S,"ifail"::S]$Lisp,_
    [([xendArg::Any,mArg::Any,nArg::Any,irelabArg::Any,xArg::Any,_
    tolArg::Any,ifailArg::Any,yArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d02bhf(xendArg:DoubleFloat,nArg:Integer,irelabArg:Integer,_
    hmaxArg:DoubleFloat,xArg:DoubleFloat,yArg:Matrix DoubleFloat,_
    tolArg:DoubleFloat,ifailArg:Integer,_
```

```
        gArg:Union(fn:FileName,fp:Asp9(G)),_
        fcnArg:Union(fn:FileName,fp:Asp7(FCN))): Result ==
        pushFortranOutputStack(gFilename := aspFilename "g")$FOP
        if gArg case fn
          then outputAsFortran(gArg.fn)
          else outputAsFortran(gArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
        if fcnArg case fn
          then outputAsFortran(fcnArg.fn)
          else outputAsFortran(fcnArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([gFilename,fcnFilename]$Lisp,_
        "d02bhf",_
        ["xend"::S,"n"::S,"irelab"::S,"hmax"::S,"x"::S_
        ,"tol"::S,"ifail"::S,"g"::S,"fcn"::S,"y"::S,"w"::S]$Lisp,_
        ["w"::S,"g"::S,"fcn"::S]$Lisp,_
        [["double"::S,"xend"::S,"hmax"::S,"x"::S,["y"::S,"n"::S]$Lisp_
        ,"tol"::S,["w"::S,"n"::S,7$Lisp]$Lisp,"g"::S,"fcn"::S]$Lisp_
        ,["integer"::S,"n"::S,"irelab"::S,"ifail"::S]$Lisp]$Lisp,_
        ["x"::S,"y"::S,"tol"::S,"ifail"::S]$Lisp,_
        [([xendArg::Any,nArg::Any,irelabArg::Any,hmaxArg::Any,_
        xArg::Any,tolArg::Any,ifailArg::Any,yArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    d02cjf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_
        tolArg:DoubleFloat,relabsArg:String,xArg:DoubleFloat,_
        yArg:Matrix DoubleFloat,ifailArg:Integer,_
        gArg:Union(fn:FileName,fp:Asp9(G)),_
        fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_
        outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==
        pushFortranOutputStack(gFilename := aspFilename "g")$FOP
        if gArg case fn
          then outputAsFortran(gArg.fn)
          else outputAsFortran(gArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
        if fcnArg case fn
          then outputAsFortran(fcnArg.fn)
          else outputAsFortran(fcnArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
        if outputArg case fn
          then outputAsFortran(outputArg.fn)
          else outputAsFortran(outputArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([gFilename,fcnFilename,outputFilename]$Lisp,_
        "d02cjf",_
        ["xend"::S,"m"::S,"n"::S,"tol"::S,"relabs"::S_
```

```
      ,"x"::S,"ifail"::S,"g"::S,"fcn"::S,"output"::S_
      ,"result"::S,"y"::S,"w"::S]$Lisp,_
      ["result"::S,"w"::S,"g"::S,"fcn"::S,"output"::S]$Lisp,_
      [["double"::S,"xend"::S,"tol"::S,["result"::S,"m"::S,"n"::S]$Lisp_
      ,"x"::S,["y"::S,"n"::S]$Lisp,["w"::S,["+"::S,_
      ["*"::S,21$Lisp,"n"::S]$Lisp,28$Lisp]$Lisp]$Lisp,"g"::S_
      ,"fcn"::S,"output"::S]$Lisp_
      ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
      ,["character"::S,"relabs"::S]$Lisp]$Lisp,_
      ["result"::S,"x"::S,"y"::S,"ifail"::S]$Lisp,_
      [([xendArg::Any,mArg::Any,nArg::Any,tolArg::Any,relabsArg::Any,_
      xArg::Any,ifailArg::Any,yArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result

  d02ejf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_
      relabsArg:String,iwArg:Integer,xArg:DoubleFloat,_
      yArg:Matrix DoubleFloat,tolArg:DoubleFloat,ifailArg:Integer,_
      gArg:Union(fn:FileName,fp:Asp9(G)),_
      fcnArg:Union(fn:FileName,fp:Asp7(FCN)),_
      pedervArg:Union(fn:FileName,fp:Asp31(PEDERV)),_
      outputArg:Union(fn:FileName,fp:Asp8(OUTPUT))): Result ==
      pushFortranOutputStack(gFilename := aspFilename "g")$FOP
      if gArg case fn
        then outputAsFortran(gArg.fn)
        else outputAsFortran(gArg.fp)
      popFortranOutputStack()$FOP
      pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
      if fcnArg case fn
        then outputAsFortran(fcnArg.fn)
        else outputAsFortran(fcnArg.fp)
      popFortranOutputStack()$FOP
      pushFortranOutputStack(pedervFilename := aspFilename "pederv")$FOP
      if pedervArg case fn
        then outputAsFortran(pedervArg.fn)
        else outputAsFortran(pedervArg.fp)
      popFortranOutputStack()$FOP
      pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
      if outputArg case fn
        then outputAsFortran(outputArg.fn)
        else outputAsFortran(outputArg.fp)
      popFortranOutputStack()$FOP
      [(invokeNagman([gFilename,fcnFilename,pedervFilename,_
      outputFilename]$Lisp,_
      "d02ejf",_
      ["xend"::S,"m"::S,"n"::S,"relabs"::S,"iw"::S_
      ,"x"::S,"tol"::S,"ifail"::S,"g"::S,"fcn"::S_
      ,"pederv"::S,"output"::S,"result"::S,"y"::S,"w"::S]$Lisp,_
      ["result"::S,"w"::S,"g"::S,"fcn"::S,"pederv"::S,"output"::S]$Lisp,_
      [["double"::S,"xend"::S,["result"::S,"m"::S,"n"::S]$Lisp_
```

```
    ,"x"::S,["y"::S,"n"::S]$Lisp,"tol"::S,["w"::S,"iw"::S]$Lisp,_
    "g"::S,"fcn"::S,"pederv"::S,"output"::S]$Lisp_
    ,["integer"::S,"m"::S,"n"::S,"iw"::S,"ifail"::S_
    ]$Lisp_
    ,["character"::S,"relabs"::S]$Lisp_
    ]$Lisp,_
    ["result"::S,"x"::S,"y"::S,"tol"::S,"ifail"::S]$Lisp,_
    [([xendArg::Any,mArg::Any,nArg::Any,relabsArg::Any,iwArg::Any,_
    xArg::Any,tolArg::Any,ifailArg::Any,yArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d02gaf(uArg:Matrix DoubleFloat,vArg:Matrix DoubleFloat,nArg:Integer,_
    aArg:DoubleFloat,bArg:DoubleFloat,tolArg:DoubleFloat,_
    mnpArg:Integer,lwArg:Integer,liwArg:Integer,_
    xArg:Matrix DoubleFloat,npArg:Integer,ifailArg:Integer,_
    fcnArg:Union(fn:FileName,fp:Asp7(FCN))): Result ==
    pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
    if fcnArg case fn
      then outputAsFortran(fcnArg.fn)
      else outputAsFortran(fcnArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([fcnFilename]$Lisp,_
    "d02gaf",_
    ["n"::S,"a"::S,"b"::S,"tol"::S,"mnp"::S_
    ,"lw"::S,"liw"::S,"np"::S,"ifail"::S,"fcn"::S_
    ,"u"::S,"v"::S,"y"::S,"x"::S,"w"::S_
    ,"iw"::S]$Lisp,_
    ["y"::S,"w"::S,"iw"::S,"fcn"::S]$Lisp,_
    [["double"::S,["u"::S,"n"::S,2$Lisp]$Lisp,["v"::S,"n"::S,2$Lisp]$Lisp_
    ,"a"::S,"b"::S,"tol"::S,["y"::S,"n"::S,"mnp"::S]$Lisp,_
    ["x"::S,"mnp"::S]$Lisp,["w"::S,"lw"::S]$Lisp_
    ,"fcn"::S]$Lisp_
    ,["integer"::S,"n"::S,"mnp"::S,"lw"::S,"liw"::S_
    ,"np"::S,"ifail"::S,["iw"::S,"liw"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["y"::S,"x"::S,"np"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,aArg::Any,bArg::Any,tolArg::Any,mnpArg::Any,_
    lwArg::Any,liwArg::Any,npArg::Any,ifailArg::Any,uArg::Any,_
    vArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d02gbf(aArg:DoubleFloat,bArg:DoubleFloat,nArg:Integer,_
    tolArg:DoubleFloat,mnpArg:Integer,lwArg:Integer,_
    liwArg:Integer,cArg:Matrix DoubleFloat,dArg:Matrix DoubleFloat,_
    gamArg:Matrix DoubleFloat,xArg:Matrix DoubleFloat,npArg:Integer,_
    ifailArg:Integer,fcnfArg:Union(fn:FileName,fp:Asp77(FCNF)),_
    fcngArg:Union(fn:FileName,fp:Asp78(FCNG))): Result ==
    pushFortranOutputStack(fcnfFilename := aspFilename "fcnf")$FOP
```

```
    if fcnfArg case fn
      then outputAsFortran(fcnfArg.fn)
      else outputAsFortran(fcnfArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(fcngFilename := aspFilename "fcng")$FOP
    if fcngArg case fn
      then outputAsFortran(fcngArg.fn)
      else outputAsFortran(fcngArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([fcnfFilename,fcngFilename]$Lisp,_
    "d02gbf",_
    ["a"::S,"b"::S,"n"::S,"tol"::S,"mnp"::S_
    ,"lw"::S,"liw"::S,"np"::S,"ifail"::S,"fcnf"::S_
    ,"fcng"::S,"y"::S,"c"::S,"d"::S,"gam"::S,"x"::S_
    ,"w"::S,"iw"::S]$Lisp,_
    ["y"::S,"w"::S,"iw"::S,"fcnf"::S,"fcng"::S]$Lisp,_
    [["double"::S,"a"::S,"b"::S,"tol"::S,["y"::S,"n"::S,"mnp"::S]$Lisp_
    ,["c"::S,"n"::S,"n"::S]$Lisp,["d"::S,"n"::S,"n"::S]$Lisp,_
    ["gam"::S,"n"::S]$Lisp,["x"::S,"mnp"::S]$Lisp_
    ,["w"::S,"lw"::S]$Lisp,"fcnf"::S,"fcng"::S]$Lisp_
    ,["integer"::S,"n"::S,"mnp"::S,"lw"::S,"liw"::S_
    ,"np"::S,"ifail"::S,["iw"::S,"liw"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["y"::S,"c"::S,"d"::S,"gam"::S,"x"::S,"np"::S,"ifail"::S]$Lisp,_
    [([aArg::Any,bArg::Any,nArg::Any,tolArg::Any,mnpArg::Any,lwArg::Any,_
    liwArg::Any,npArg::Any,ifailArg::Any,cArg::Any,dArg::Any,_
    gamArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d02kef(xpointArg:Matrix DoubleFloat,mArg:Integer,kArg:Integer,_
    tolArg:DoubleFloat,maxfunArg:Integer,matchArg:Integer,_
    elamArg:DoubleFloat,delamArg:DoubleFloat,hmaxArg:Matrix DoubleFloat,_
    maxitArg:Integer,ifailArg:Integer,_
    coeffnArg:Union(fn:FileName,fp:Asp10(COEFFN)),_
    bdyvalArg:Union(fn:FileName,fp:Asp80(BDYVAL))): Result ==
    pushFortranOutputStack(coeffnFilename := aspFilename "coeffn")$FOP
    if coeffnArg case fn
      then outputAsFortran(coeffnArg.fn)
      else outputAsFortran(coeffnArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(bdyvalFilename := aspFilename "bdyval")$FOP
    if bdyvalArg case fn
      then outputAsFortran(bdyvalArg.fn)
      else outputAsFortran(bdyvalArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
    outputAsFortran()$Asp12(MONIT)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(reportFilename := aspFilename "report")$FOP
```

```
        outputAsFortran()$Asp33(REPORT)
        popFortranOutputStack()$FOP
        [(invokeNagman([coeffnFilename,bdyvalFilename,monitFilename,_
        reportFilename]$Lisp,_
        "d02kef",_
        ["m"::S,"k"::S,"tol"::S,"maxfun"::S,"match"::S_
        ,"elam"::S,"delam"::S,"maxit"::S,"ifail"::S,"coeffn"::S_
        ,"bdyval"::S,"monit"::S,"report"::S,"xpoint"::S,"hmax"::S]$Lisp,_
        ["coeffn"::S,"bdyval"::S,"monit"::S,"report"::S]$Lisp,_
        [["double"::S,["xpoint"::S,"m"::S]$Lisp,"tol"::S_
        ,"elam"::S,"delam"::S,["hmax"::S,2$Lisp,"m"::S]$Lisp,_
        "coeffn"::S,"bdyval"::S,"monit"::S,"report"::S]$Lisp_
        ,["integer"::S,"m"::S,"k"::S,"maxfun"::S,"match"::S_
        ,"maxit"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["match"::S,"elam"::S,"delam"::S,"hmax"::S,"maxit"::S,_
        "ifail"::S]$Lisp,_
        [([mArg::Any,kArg::Any,tolArg::Any,maxfunArg::Any,matchArg::Any,_
        elamArg::Any,delamArg::Any,maxitArg::Any,ifailArg::Any,_
        xpointArg::Any,hmaxArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    d02kef(xpointArg:Matrix DoubleFloat,mArg:Integer,kArg:Integer,_
        tolArg:DoubleFloat,maxfunArg:Integer,matchArg:Integer,_
        elamArg:DoubleFloat,delamArg:DoubleFloat,hmaxArg:Matrix DoubleFloat,_
        maxitArg:Integer,ifailArg:Integer,_
        coeffnArg:Union(fn:FileName,fp:Asp10(COEFFN)),_
        bdyvalArg:Union(fn:FileName,fp:Asp80(BDYVAL)),_
        monitArg:FileName,reportArg:FileName): Result ==
        pushFortranOutputStack(coeffnFilename := aspFilename "coeffn")$FOP
        if coeffnArg case fn
          then outputAsFortran(coeffnArg.fn)
          else outputAsFortran(coeffnArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(bdyvalFilename := aspFilename "bdyval")$FOP
        if bdyvalArg case fn
          then outputAsFortran(bdyvalArg.fn)
          else outputAsFortran(bdyvalArg.fp)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
        outputAsFortran(monitArg)
        popFortranOutputStack()$FOP
        pushFortranOutputStack(reportFilename := aspFilename "report")$FOP
        outputAsFortran(reportArg)
        popFortranOutputStack()$FOP
        [(invokeNagman([coeffnFilename,bdyvalFilename,monitFilename,_
        reportFilename]$Lisp,_
        "d02kef",_
        ["m"::S,"k"::S,"tol"::S,"maxfun"::S,"match"::S_
```

```
      ,"elam"::S,"delam"::S,"maxit"::S,"ifail"::S,"coeffn"::S_
      ,"bdyval"::S,"monit"::S,"report"::S,"xpoint"::S,"hmax"::S]$Lisp,_
      ["coeffn"::S,"bdyval"::S,"monit"::S,"report"::S]$Lisp,_
      [["double"::S,["xpoint"::S,"m"::S]$Lisp,"tol"::S_
      ,"elam"::S,"delam"::S,["hmax"::S,2$Lisp,"m"::S]$Lisp,_
      "coeffn"::S,"bdyval"::S,"monit"::S,"report"::S]$Lisp_
      ,["integer"::S,"m"::S,"k"::S,"maxfun"::S,"match"::S_
      ,"maxit"::S,"ifail"::S]$Lisp_
      ]$Lisp,_
      ["match"::S,"elam"::S,"delam"::S,"hmax"::S,"maxit"::S,_
      "ifail"::S]$Lisp,_
      [([mArg::Any,kArg::Any,tolArg::Any,maxfunArg::Any,_
      matchArg::Any,elamArg::Any,delamArg::Any,maxitArg::Any,_
      ifailArg::Any,xpointArg::Any,hmaxArg::Any ])_
      @List Any]$Lisp)$Lisp)_
      pretend List (Record(key:Symbol,entry:Any))]$Result

  d02raf(nArg:Integer,mnpArg:Integer,numbegArg:Integer,_
      nummixArg:Integer,tolArg:DoubleFloat,initArg:Integer,_
      iyArg:Integer,ijacArg:Integer,lworkArg:Integer,_
      liworkArg:Integer,npArg:Integer,xArg:Matrix DoubleFloat,_
      yArg:Matrix DoubleFloat,delepsArg:DoubleFloat,ifailArg:Integer,_
      fcnArg:Union(fn:FileName,fp:Asp41(FCN,JACOBF,JACEPS)),_
      gArg:Union(fn:FileName,fp:Asp42(G,JACOBG,JACGEP))): Result ==
      pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
      if fcnArg case fn
        then outputAsFortran(fcnArg.fn)
        else outputAsFortran(fcnArg.fp)
      popFortranOutputStack()$FOP
      pushFortranOutputStack(gFilename := aspFilename "g")$FOP
      if gArg case fn
        then outputAsFortran(gArg.fn)
        else outputAsFortran(gArg.fp)
      popFortranOutputStack()$FOP
      [(invokeNagman([fcnFilename,gFilename]$Lisp,_
      "d02raf",_
      ["n"::S,"mnp"::S,"numbeg"::S,"nummix"::S,"tol"::S_
      ,"init"::S,"iy"::S,"ijac"::S,"lwork"::S,"liwork"::S_
      ,"np"::S,"deleps"::S,"ifail"::S,"fcn"::S,"g"::S_
      ,"abt"::S,"x"::S,"y"::S,"work"::S,"iwork"::S_
      ]$Lisp,_
      ["abt"::S,"work"::S,"iwork"::S,"fcn"::S,"g"::S]$Lisp,_
      [["double"::S,"tol"::S,["abt"::S,"n"::S]$Lisp_
      ,["x"::S,"mnp"::S]$Lisp,["y"::S,"iy"::S,"mnp"::S]$Lisp,_
      "deleps"::S,["work"::S,"lwork"::S]$Lisp,"fcn"::S,"g"::S]$Lisp_
      ,["integer"::S,"n"::S,"mnp"::S,"numbeg"::S_
      ,"nummix"::S,"init"::S,"iy"::S,"ijac"::S,"lwork"::S,"liwork"::S,_
      "np"::S,"ifail"::S,["iwork"::S,"liwork"::S]$Lisp]$Lisp_
      ]$Lisp,_
      ["abt"::S,"np"::S,"x"::S,"y"::S,"deleps"::S,"ifail"::S]$Lisp,_
```

```
        [([nArg::Any,mnpArg::Any,numbegArg::Any,nummixArg::Any,tolArg::Any,_
        initArg::Any,iyArg::Any,ijacArg::Any,lworkArg::Any,liworkArg::Any,_
        npArg::Any,delepsArg::Any,ifailArg::Any,xArg::Any,yArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGD02.dotabb** —

```
"NAGD02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGD02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGD02" -> "ALIST"
```

———-

15.11 package NAGD03 NagPartialDifferentialEquationsPackage

— **NagPartialDifferentialEquationsPackage.help** —

```
D03 -- Partial Differential Equations         Introduction -- D03
                          Chapter D03
                Partial Differential Equations

1. Scope of the Chapter

This chapter is concerned with the solution of partial
differential equations.

2. Background to the Problems

The definition of a partial differential equation problem
includes not only the equation itself but also the domain of
interest and appropriate subsidiary conditions. Indeed, partial
differential equations are usually classified as elliptic,
hyperbolic or parabolic according to the form of the equation and
the form of the subsidiary conditions which must be assigned to
produce a well-posed problem. Ultimately it is hoped that this
chapter will contain routines for the solution of equations of
each of these types together with automatic mesh generation
routines and other utility routines particular to the solution of
partial differential equations. The routines in this chapter will
```

often call upon routines from the Linear Algebra Chapter F04 -- Simultaneous Linear Equations.

The classification of partial differential equations is easily described in the case of linear equations of the second order in two independent variables, i.e., equations of the form

$$au_{xx} + 2bu_{xy} + cu_{yy} + du_x + eu_y + fu + g = 0, \quad (1)$$

where a, b, c, d, e, f and g are functions of x and y only. Equation (1) is called elliptic, hyperbolic or parabolic according as $ac-b^2$ is positive, negative or zero. Useful definitions of the concepts of elliptic, hyperbolic and parabolic character can also be given for differential equations in more than two independent variables, for systems and for nonlinear differential equations.

For elliptic equations, of which Laplace's equation

$$u_{xx} + u_{yy} = 0 \quad (2)$$

is the simplest example of second order, the subsidiary conditions take the form of boundary conditions, i.e., conditions which provide information about the solution at all points of a closed boundary. For example, if equation (2) holds in a plane domain D bounded by a contour C, a solution u may be sought subject to the condition

$$u = f \text{ on } C, \quad (3)$$

where f is a given function. The condition (3) is known as a Dirichlet boundary condition. Equally common is the Neumann boundary condition

$$u' = g \text{ on } C, \quad (4)$$

which is one form of a more general condition

$$u' + fu = g \text{ on } C, \quad (5)$$

where u' denotes the derivative of u normal to the contour C and f and g are given functions. Provided that f and g satisfy certain restrictions, condition (5) yields a well-posed boundary value problem for Laplace's equation. In the case of the Neumann problem, one further piece of information, e.g. the value of u at a particular point, is necessary for uniqueness of the solution. Boundary conditions similar to the above are applicable to more

general second order elliptic equations, whilst two such conditions are required for equations of fourth order.

For hyperbolic equations, the wave equation

$$u_{tt} - u_{xx} = 0 \tag{6}$$

is the simplest example of second order. It is equivalent to a first order system

$$u_t - v_x = 0, \quad v_t - u_x = 0. \tag{7}$$

The subsidiary conditions may take the form of initial conditions, i.e., conditions which provide information about the solution at points on a suitable open boundary. For example, if equation (6) is satisfied for t>0, a solution u may be sought such that

$$u(x,0)=f(x), \quad u_t(x,0)=g(x), \tag{8}$$

where f and g are given functions. This is an example of an initial value problem, sometimes known as Cauchy's problem.

For parabolic equations, of which the heat conduction equation

$$u_t - u_{xx} = 0 \tag{9}$$

is the simplest example, the subsidiary conditions always include some of initial type and may also include some of boundary type. For example, if equation (9) is satisfied for t>0 and 0<x<1, a solution u may be sought such that

$$u(x,0)=f(x), \quad 0<x<1, \tag{10}$$

and

$$u(0,t)=0, \quad u(1,t)=1, \quad t>0. \tag{11}$$

This is an example of a mixed initial/boundary value problem.

For all types of partial differential equations, finite difference methods (Mitchell and Griffiths [5]) and finite element methods (Wait and Mitchell [9]) are the most common means of solution and such methods obviously feature prominently either in this chapter or in the companion NAG Finite Element Library. Many of the utility routines in this chapter are concerned with

the solution of the large sparse systems of equations which arise from the finite difference and finite element methods.

Alternative methods of solution are often suitable for special classes of problems. For example, the method of characteristics is the most common for hyperbolic equations involving time and one space dimension (Smith [7]). The method of lines (see Mikhlin and Smolitsky [4]) may be used to reduce a parabolic equation to a (stiff) system of ordinary differential equations, which may be solved by means of routines from Chapter D02 -- Ordinary Differential Equations. Similarly, integral equation or boundary element methods (Jaswon and Symm [3]) are frequently used for elliptic equations. Typically, in the latter case, the solution of a boundary value problem is represented in terms of certain boundary functions by an integral expression which satisfies the differential equation throughout the relevant domain. The boundary functions are obtained by applying the given boundary conditions to this representation. Implementation of this method necessitates discretisation of only the boundary of the domain, the dimensionality of the problem thus being effectively reduced by one. The boundary conditions yield a full system of simultaneous equations, as opposed to the sparse systems yielded by the finite difference and finite element methods, but the full system is usually of much lower order. Solution of this system yields the boundary functions, from which the solution of the problem may be obtained, by quadrature, as and where required.

2.1. References

[1] Ames W F (1977) Nonlinear Partial Differential Equations in Engineering. Academic Press (2nd Edition).

[2] Berzins M (1990) Developments in the NAG Library Software for Parabolic Equations. Scientific Software Systems. (ed J C Mason and M G Cox) Chapman and Hall. 59--72.

[3] Jaswon M A and Symm G T (1977) Integral Equation Methods in Potential Theory and Elastostatics. Academic Press.

[4] Mikhlin S G and Smolitsky K L (1967) Approximate Methods for the Solution of Differential and Integral Equations. Elsevier.

[5] Mitchell A R and Griffiths D F (1980) The Finite Difference Method in Partial Differential Equations. Wiley.

[6] Richtmyer R D and Morton K W (1967) Difference Methods for Initial-value Problems. Interscience (2nd Edition).

[7] Smith G D (1985) Numerical Solution of Partial Differential

Equations: Finite Difference Methods. Oxford University Press (3rd Edition).

[8] Swarztrauber P N and Sweet R A (1979) Efficient Fortran Subprograms for the Solution of Separable Elliptic Partial Differential Equations. ACM Trans. Math. Softw. 5 352--364.

[9] Wait R and Mitchell A R (1985) Finite Element Analysis and Application. Wiley.

3. Recommendations on Choice and Use of Routines

The choice of routine will depend first of all upon the type of partial differential equation to be solved. At present no special allowances are made for problems with boundary singularities such as may arise at corners of domains or at points where boundary conditions change. For such problems results should be treated with caution.

Users may wish to construct their own partial differential equation solution software for problems not solvable by the routines described in Sections 3.1 to 3.4 below. In such cases users can employ appropriate routines from the Linear Algebra Chapters to solve the resulting linear systems; see Section 3.5 for further details.

3.1. Elliptic Equations

The routine D03EDF solves a system of seven-point difference equations in a rectangular grid (in two dimensions), using the multigrid iterative method. The equations are supplied by the user, and the seven-point form allows cross-derivative terms to be represented (see Mitchell and Griffiths [5]). The method is particularly efficient for large systems of equations with diagonal dominance.

The routine D03EEF discretises a second-order equation on a two-dimensional rectangular region using finite differences and a seven-point molecule. The routine allows for cross-derivative terms, Dirichlet, Neumann or mixed boundary conditions, and either central or upwind differences. The resulting seven-diagonal difference equations are in a form suitable for passing directly to the multigrid routine D03EDF, although other solution methods could easily be used.

The routine D03FAF, based on the routine HW3CRT from FISHPACK (Swarztrauber and Sweet [8]), solves the Helmholtz equation in a three-dimensional cuboidal region, with any combination of Dirichlet, Neumann or periodic boundary conditions. The method used is based on the fast Fourier transform algorithm, and is

likely to be particularly efficient on vector-processing machines.

3.2. Hyperbolic Equations

There are no routines available yet for the solution of these equations.

3.3. Parabolic Equations

There are no routines available yet for the solution of these equations.

But problems in two space dimensions plus time may be treated as a succession of elliptic equations [1], [6] using appropriate D03E- routines or one may use codes from the NAG Finite Element Library.

3.4. Utility Routines

There are no utility routines available yet, but routines are available in the Linear Algebra Chapters for the direct and iterative solution of linear equations. Here we point to some of the routines that may be of use in solving the linear systems that arise from finite difference or finite element approximations to partial differential equation solutions. Chapters F01 and F04 should be consulted for further information and for the routine documents. Decision trees for the solution of linear systems are given in Section 3.5 of the F04 Chapter Introduction.

The following routines allow the direct solution of symmetric positive-definite systems:

Band	F04ACF
Variable band (skyline)	F01MCF and F04MCF
Tridiagonal	F04FAF
Sparse	F01MAF* and F04MAF

(* the parameter DROPTL should be set to zero for F01MAF to give a direct solution)

and the following routines allow the iterative solution of symmetric positive-definite systems:

Sparse (incomplete F01MAF and F04MBF

```
    Cholesky)

    Sparse (conjugate  F04MBF
    gradient)

The latter routine above allows the user to supply a pre-
conditioner and also allows the solution of indefinite symmetric
systems.

The following routines allow the direct solution of unsymmetric
systems:

    Band               F01LBF and F04LDF

    Almost block-      F01LHF and F04LHF
    diagonal

    Tridiagonal        F01LEF and F04LEF or F04EAF

    Sparse             F01BRF (and F01BSF) and F04AXF

and the following routine allows the iterative solution of
unsymmetric systems:

    Sparse             F04QAF

The above routine allows the user to supply a pre-conditioner and
also allows the solution of least-squares systems.

3.5. Index

Elliptic equations
     equations on rectangular grid (seven-point 2-D        D03EDF
     molecule)
     discretisation on rectangular grid (seven-point 2-D   D03EEF
     molecule)
     Helmholtz's equation in three dimensions              D03FAF



D03 -- Partial Differential Equations            Contents -- D03
Chapter D03

Partial Differential Equations

D03EDF  Elliptic PDE, solution of finite difference equations by
        a multigrid technique

D03EEF  Discretize a 2nd order elliptic PDE on a rectangle
```

```
D03FAF  Elliptic PDE, Helmholtz equation, 3-D Cartesian co-
        ordinates

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D03 -- Partial Differential Equations                        D03EDF
        D03EDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

D03EDF solves seven-diagonal systems of linear equations which
arise from the discretization of an elliptic partial differential
equation on a rectangular region. This routine uses a multigrid
technique.

2. Specification

       SUBROUTINE D03EDF (NGX, NGY, LDA, A, RHS, UB, MAXIT, ACC,
      1                   US, U, IOUT, NUMIT, IFAIL)
       INTEGER          NGX, NGY, LDA, MAXIT, IOUT, NUMIT, IFAIL
       DOUBLE PRECISION A(LDA,7), RHS(LDA), UB(NGX*NGY), ACC, US
      1                 (LDA), U(LDA)

3. Description

D03EDF solves, by multigrid iteration, the seven-point scheme

 6            7
A   u       +A   u
 i,j i-1,j+1  i,j i,j+1

  3           4        5
+A   u      +A   u   +A   u
  i,j i-1,j   i,j ij   i,j i+1,j

              1          2
            +A   u     +A   u        =f  ,
              i,j i,j-1  i,j i+1,j-1  ij

                                     i=1,2,...,n ; j=1,2,...,n ,
                                               x             y

which arises from the discretization of an elliptic partial
differential equation of the form
```

```
(alpha)(x,y)U  +(beta)(x,y)U  +(gamma)(x,y)U  +(delta)(x,y)U
             xx             xy              yy              x

+(epsilon)(x,y)U +(phi)(x,y)U=(psi)(x,y)
                y
```

and its boundary conditions, defined on a rectangular region. This we write in matrix form as

$$Au=f$$

The algorithm is described in separate reports by Wesseling [2], [3] and McCarthy [1].

Systems of linear equations, matching the seven-point stencil defined above, are solved by a multigrid iteration. An initial estimate of the solution must be provided by the user. A zero guess may be supplied if no better approximation is available.

A 'smoother' based on incomplete Crout decomposition is used to eliminate the high frequency components of the error. A restriction operator is then used to map the system on to a sequence of coarser grids. The errors are then smoothed and prolongated (mapped onto successively finer grids). When the finest cycle is reached, the approximation to the solution is corrected. The cycle is repeated for MAXIT iterations or until the required accuracy, ACC, is reached.

D03EDF will automatically determine the number l of possible coarse grids, 'levels' of the multigrid scheme, for a particular problem. In other words, D03EDF determines the maximum integer l so that n_x and n_y can be expressed in the form

$$n_x = m_2^{l-1} + 1, \quad n_y = n_2^{l-1} + 1, \quad \text{with } m \geq 2 \text{ and } n \geq 2.$$

It should be noted that the rate of convergence improves significantly with the number of levels used (see McCarthy [1]), so that n_x and n_y should be carefully chosen so that $n_x - 1$ and $n_y - 1$ have factors of the form 2^l, with l as large as possible. For good convergence the integer l should be at least 2.

D03EDF has been found to be robust in application, but being an iterative method the problem of divergence can arise. For a

strictly diagonally dominant matrix A

$$|A_{ij}| > \sum_{k=1,\,k\neq 4}^{4} |A_{ik}|$$

no such problem is foreseen. The diagonal dominance of A is not a necessary condition, but should this condition be strongly violated then divergence may occur. The quickest test is to try the routine.

4. References

[1] McCarthy G J (1983) Investigation into the Multigrid Code MGD1. Report AERE-R 10889. Harwell.

[2] Wesseling P (1982) MGD1 - A Robust and Efficient Multigrid Method. Multigrid Methods. Lecture Notes in Mathematics. 960 Springer-Verlag. 614--630.

[3] Wesseling P (1982) Theoretical Aspects of a Multigrid Method. SIAM J. Sci. Statist. Comput. 3 387--407.

5. Parameters

1: NGX -- INTEGER Input
On entry: the number of interior grid points in the x-direction, n_x. NGX-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGX >= 3.

2: NGY -- INTEGER Input
On entry: the number of interior grid points in the y-direction, n_y. NGY-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGY >= 3.

3: LDA -- INTEGER Input
On entry: the first dimension of the array A as declared in the (sub)program from which D03EDF is called, which must also be a lower bound for the dimensions of the arrays RHS, US and U. It is always sufficient to set LDA>=(4*(NGX+1)*(NGY+1))/3, but slightly smaller values may be permitted, depending on the values of NGX and NGY. If on entry, LDA is too small, an error message gives the minimum permitted value. (LDA must be large enough to allow space for the coarse-grid approximations).

4: A(LDA,7) -- DOUBLE PRECISION array Input/Output
On entry: A(i+(j-1)*NGX,k) must be set to A_{ij}^{k}, for i = 1,2,...,NGX; j = 1,2,...,NGY and k = 1,2,...,7. On exit: A is overwritten.

5: RHS(LDA) -- DOUBLE PRECISION array Input/Output
On entry: RHS(i+(j-1)*NGX) must be set to f_{ij}, for i = 1,2,...,NGX; j = 1,2,...,NGY. On exit: the first NGX*NGY elements are unchanged and the rest of the array is used as workspace.

6: UB(NGX*NGY) -- DOUBLE PRECISION array Input/Output
On entry: UB(i+(j-1)*NGX) must be set to the initial estimate for the solution u_{ij}. On exit: the corresponding component of the residual r=f-Au.

7: MAXIT -- INTEGER Input
On entry: the maximum permitted number of multigrid iterations. If MAXIT = 0, no multigrid iterations are performed, but the coarse-grid approximations and incomplete Crout decompositions are computed, and may be output if IOUT is set accordingly. Constraint: MAXIT >= 0.

8: ACC -- DOUBLE PRECISION Input
On entry: the required tolerance for convergence of the residual 2-norm:

$$||r||_2 = \sqrt{\sum_{k=1}^{NGX*NGY} (r_k)^2}$$

where r=f-Au and u is the computed solution. Note that the norm is not scaled by the number of equations. The routine will stop after fewer than MAXIT iterations if the residual 2-norm is less than the specified tolerance. (If MAXIT > 0, at least one iteration is always performed.)

If on entry ACC = 0.0, then the machine precision is used as a default value for the tolerance; if ACC > 0.0, but ACC is less than the machine precision, then the routine will stop when the residual 2-norm is less than the machine precision and IFAIL will be set to 4. Constraint: ACC >= 0.0.

9: US(LDA) -- DOUBLE PRECISION array Output

```
     On exit: the residual 2-norm, stored in element US(1).

10:  U(LDA) -- DOUBLE PRECISION array                        Output
     On exit: the computed solution u   is returned in U(i+(j-
                                     ij
     1)*NGX), for i = 1,2,...,NGX; j = 1,2,...,NGY.

11:  IOUT -- INTEGER                                          Input
     On entry: controls the output of printed information to the
     advisory message unit as returned by X04ABF:
     IOUT = 0
           No output.

     IOUT = 1
           The solution u   , for i = 1,2,...,NGX; j = 1,2,...
                        ij
           ,NGY.

     IOUT = 2
           The residual 2-norm after each iteration, with the
           reduction factor over the previous iteration.

     IOUT = 3
           As for IOUT = 1 and IOUT = 2.

     IOUT = 4
           As for IOUT = 3, plus the final residual (as returned
           in UB).

     IOUT = 5
           As for IOUT = 4, plus the initial elements of A and
           RHS.

     IOUT = 6
           As for IOUT = 5, plus the Galerkin coarse grid
           approximations.

     IOUT = 7
           As for IOUT = 6, plus the incomplete Crout
           decompositions.

     IOUT = 8
           As for IOUT = 7, plus the residual after each
           iteration.
     The elements A(p,k), the Galerkin coarse grid approximations
     and the incomplete Crout decompositions are output in the
     format:
          Y-index = j

          X-index = i A(p,1) A(p,2) A(p,3) A(p,4) A(p,5) A(p,6)
```

```
          A(p,7)

          where p=i+(j-1)*NGX, i = 1,2,...,NGX and j = 1,2,...
          ,NGY.
     The vectors U(p), UB(p), RHS(p) are output in matrix form
     with NGY rows and NGX columns. Where NGX > 10, the NGX
     values for a given j-value are produced in rows of 10.
     Values of IOUT > 4 may yield considerable amounts of output.
     Constraint: 0 <= IOUT <= 8.

12:  NUMIT -- INTEGER                                        Output
     On exit: the number of iterations performed.

13:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry NGX < 3,

     or       NGY < 3,

     or       LDA is too small,

     or       ACC < 0.0,

     or       MAXIT < 0,

     or       IOUT < 0,

     or       IOUT > 8.

IFAIL= 2
     MAXIT iterations have been performed with the residual 2-
     norm decreasing at each iteration but the residual 2-norm
     has not been reduced to less than the specified tolerance
     (see ACC). Examine the progress of the iteration by setting
     IOUT >= 2.
```

IFAIL= 3
     As for IFAIL = 2, except that at one or more iterations the
     residual 2-norm did not decrease. It is likely that the
     method fails to converge for the given matrix A.

IFAIL= 4
     On entry ACC is less than the machine precision. The routine
     terminated because the residual norm is less than the
     machine precision.

7. Accuracy

See ACC (Section 5).

8. Further Comments

The rate of convergence of this routine is strongly dependent
upon the number of levels, l, in the multigrid scheme, and thus
the choice of NGX and NGY is very important. The user is advised
to experiment with different values of NGX and NGY to see the
effect they have on the rate of convergence; for example, using a
                         6
value such as NGX = 65 (=2 +1) followed by NGX = 64 (for which l
= 1).

9. Example

The program solves the elliptic partial differential equation

$$U_{xx} - (\alpha)U_{xy} + U_{yy} = -4, \quad (\alpha)=1.7$$

on the unit square 0<=x,y<=1, with boundary conditions

$$U=0 \text{ on } \begin{cases} x=0, & (0<=y<=1) \\ y=0, & (0<=x<=1) \\ y=1, & (0<=x<=1) \end{cases} \quad U=1 \text{ on } x=1, \quad 0<=y<=1.$$

For the equation to be elliptic, (alpha) must be less than 2.

The equation is discretized on a square grid with mesh spacing h
in both directions using the following approximations:

Please see figure in printed Reference Manual

$$U_{xx} \tilde{=} \frac{1}{2}(U_E - 2U_O + U_W)$$

$$h$$

$$U_{yy} \tilde{=} \frac{1}{h^2}(U_N - 2U_O + U_S)$$

$$U_{xy} \tilde{=} \frac{1}{2h}(U_{NW} - U_N + U_E - 2U_O + U_W - U_S + U_{SE}).$$

Thus the following equations are solved:

$$\frac{1}{2}\alpha u_{i-1,j+1} + \left(1 - \frac{1}{2}\alpha\right)u_{i,j+1}$$

$$+\left(1 - \frac{1}{2}\alpha\right)u_{i-1,j} + (-4 + \alpha)u_{ij} + \left(1 - \frac{1}{2}\alpha\right)u_{i+1,j}$$

$$+\left(1 - \frac{1}{2}\alpha\right)u_{i,j-1} + \frac{1}{2}\alpha u_{i+1,j-1}$$

$$= -4h^2$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

D03 -- Partial Differential Equations D03EEF
D03EEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D03EEF discretizes a second order elliptic partial differential equation (PDE) on a rectangular region.

2. Specification

```
      SUBROUTINE D03EEF (XMIN, XMAX, YMIN, YMAX, PDEF, BNDY,
     1                   NGX, NGY, LDA, A, RHS, SCHEME, IFAIL)
      INTEGER          NGX, NGY, LDA, IFAIL
      DOUBLE PRECISION XMIN, XMAX, YMIN, YMAX, A(LDA,7), RHS(LDA)
      CHARACTER*1      SCHEME
      EXTERNAL         PDEF, BNDY
```

3. Description

D03EEF discretizes a second order linear elliptic partial differential equation of the form

$$(\alpha)(x,y)\frac{d^2U}{dx^2}+(\beta)(x,y)\frac{d^2U}{dxdy}+(\gamma)(x,y)\frac{d^2U}{dy^2}+(\delta)(x,y)\frac{dU}{dx}+(\epsilon)(x,y)\frac{dU}{dy}+(\phi)(x,y)U=(\psi)(x,y) \quad (1)$$

on a rectangular region

$$x_A \le x \le x_B$$

$$y_A \le y \le y_B$$

subject to boundary conditions of the form

$$a(x,y)U+b(x,y)\frac{dU}{dn}=c(x,y)$$

where $\frac{dU}{dn}$ denotes the outward pointing normal derivative on the boundary. Equation (1) is said to be elliptic if

$$4(\alpha)(x,y)(\gamma)(x,y) \ge ((\beta)(x,y))^2$$

for all points in the rectangular region. The linear equations produced are in a form suitable for passing directly to the

multigrid routine D03EDF.

The equation is discretized on a rectangular grid, with n_x grid points in the x-direction and n_y grid points in the y-direction. The grid spacing used is therefore

$$h_x = (x_B - x_A)/(n_x - 1)$$

$$h_y = (y_B - y_A)/(n_y - 1)$$

and the co-ordinates of the grid points (x_i, y_j) are

$$x_i = x_A + (i-1)h_x, \quad i = 1,2,\ldots,n_x,$$

$$y_j = y_A + (j-1)h_y, \quad j = 1,2,\ldots,n_y.$$

At each grid point (x_i, y_j) six neighbouring grid points are used to approximate the partial differential equation, so that the equation is discretized on the following seven-point stencil:

Please see figure in printed Reference Manual

For convenience the approximation u_{ij} to the exact solution $U(x_i, y_j)$ is denoted by u_0, and the neighbouring approximations are labelled according to points of the compass as shown. Where numerical labels for the seven points are required, these are also shown above.

The following approximations are used for the second derivatives:

$$\frac{\partial^2 U}{\partial x^2} \tilde{=} \frac{1}{h_x^2}(u_E - 2u_0 + u_W)$$

$$\frac{\partial^2}{\partial}$$

```
dd U   1
----~= --(u -2u +u )
  2     2  N   O  S
dd y   h
        y

   2
 dd U    1
------~= -----(u -u  +u -2u +u -u  +u ).
ddxddy 2h h   N  NW  E   O  W  SE  S
         x y
```

Two possible schemes may be used to approximate the first derivatives:

Central Differences

```
ddU     1
---~=  ---(u -u )
ddx    2h   W  E
         x

ddU     1
---~=  ---(u -u )
ddy    2h   N  S
         y
```

Upwind Differences

```
ddU    1
---~=  --(u -u )if (delta)(x,y)>0
ddx    h   E  O
        x

ddU    1
---~=  --(u -u )if (delta)(x,y)<0
ddx    h   O  W
        x

ddU    1
---~=  --(u -u )if (epsilon)(x,y)>0
ddy    h   N  O
        y

ddU    1
---~=  --(u -u )if (epsilon)(x,y)<0.
ddy    h   O  S
        y
```

Central differences are more accurate than upwind differences,

but upwind differences may lead to a more diagonally dominant
matrix for those problems where the coefficients of the first
derivatives are significantly larger than the coefficients of the
second derivatives.

The approximations used for the first derivatives may be written
in a more compact form as follows:

$$\frac{\widetilde{dU}}{dx} = \frac{1}{2h_x}\left((k-1)u - 2k\,u + (k+1)u\right)_{W \quad x\,O \quad x \quad E}$$

$$\frac{\widetilde{dU}}{dy} = \frac{1}{2h_y}\left((k-1)u - 2k\,u + (k+1)u\right)_{S \quad y\,O \quad y \quad N}$$

where k_x =sign (delta) and k_y =sign (epsilon) for upwind
differences, and $k_x = k_y = 0$ for central differences.

At all points in the rectangular domain, including the boundary,
the coefficients in the partial differential equation are
evaluated by calling the user-supplied subroutine PDEF, and
applying the approximations. This leads to a seven-diagonal
system of linear equations of the form:

$$A^{6}_{ij}\,u_{i-1,j+1} + A^{7}_{ij}\,u_{i,j+1} + A^{3}_{ij}\,u_{i-1,j} + A^{4}_{ij}\,u_{ij} + A^{5}_{ij}\,u_{i+1,j} + A^{1}_{ij}\,u_{i,j-1} + A^{2}_{ij}\,u_{i+1,j-1} = f_{ij}, \quad i=1,2,\ldots,n_x;\ j=1,2,\ldots,n_y,$$

where the coefficients are given by

$$A^{1}_{ij} = (\beta)(x_i, y_j)\frac{1}{2h_x h_y} + (\gamma)(x_i, y_j)\frac{1}{h_y} + (\varepsilon)(x_i, y_j)\frac{1}{2h_x}(k_x - 1)$$

$$A^{2}_{ij} = -(\beta)(x_i, y_j)\frac{1}{2h_x h_y}$$

```
 3                1                  1                   1
A  =(alpha)(x ,y ) --+(beta)(x ,y ) -----+(delta)(x ,y ) ---(k -1)
 ij          i  j   2         i  j  2h h           i  j  2h   x
                   h                  x y                  x
                    x

 4                 2                   1                   2
A  =-(alpha)(x ,y ) -- -(beta)(x ,y ) ---- -(gamma)(x ,y ) --
 ij           i  j   2          i  j  h h            i  j   2
                    h                  x y                 h
                     x                                      y

                    k                   k
                     y                   y
    -(delta)(x ,y ) -- -(epsilon)(x ,y ) -- -(phi)(x ,y )
              i  j  h              i  j  h          i  j
                     x                   y

 5                1                  1                   1
A  =(alpha)(x ,y ) --+(beta)(x ,y ) -----+(delta)(x ,y ) ---(k +1)
 ij          i  j   2         i  j  2h h           i  j  2h   x
                   h                  x y                  x
                    x

 6                  1
A  =-(beta)(x ,y ) -----
 ij           i  j  2h h
                     x y

 7                1                  1                   1
A  =(beta)(x ,y )-----+(gamma)(x ,y )--+(epsilon)(x ,y )---(k +1)
 ij          i  j 2h h           i  j  2             i j 2h   y
                   x y                h                    y
                                       y

f  =(psi)(x ,y )
 ij        i  j
```

These equations then have to be modified to take account of the boundary conditions. These may be Dirichlet (where the solution is given), Neumann (where the derivative of the solution is given), or mixed (where a linear combination of solution and derivative is given).

If the boundary conditions are Dirichlet, there are an infinity of possible equations which may be applied:

$$(mu)u_{ij} = (mu)f_{ij} \quad , (mu)/=0. \tag{2}$$

If D03EDF is used to solve the discretized equations, it turns out that the choice of (mu) can have a dramatic effect on the rate of convergence, and the obvious choice (mu)=1 is not the best. Some choices may even cause the multigrid method to fail altogether. In practice it has been found that a value of the same order as the other diagonal elements of the matrix is best, and the following value has been found to work well in practice:

```
                    ( { 2   2 }  4 )
(mu)=min  (-{ --+ --},A  ).
       ij( {  2   2}  ij)
         ( { h   h }    )
         ( {  x   y}    )
```

If the boundary conditions are either mixed or Neumann (i.e., B /= 0 on return from the user-supplied subroutine BNDY), then one of the points in the seven-point stencil lies outside the domain. In this case the normal derivative in the boundary conditions is used to eliminate the 'fictitious' point, u :
outside

```
ddU   1
---~= --(u       -u      ).                  (3)
ddn   2h  outside  inside
```

It should be noted that if the boundary conditions are Neumann and (phi)(x,y)==0, then there is no unique solution. The routine returns with IFAIL = 5 in this case, and the seven-diagonal matrix is singular.

The four corners are treated separately. The user-supplied subroutine BNDY is called twice, once along each of the edges meeting at the corner. If both boundary conditions at this point are Dirichlet and the prescribed solution values agree, then this value is used in an equation of the form (2). If the prescribed solution is discontinuous at the corner, then the average of the two values is used. If one boundary condition is Dirichlet and the other is mixed, then the value prescribed by the Dirichlet condition is used in an equation of the form given above. Finally, if both conditions are mixed or Neumann, then two ' fictitious' points are eliminated using two equations of the form (3).

It is possible that equations for which the solution is known at all points on the boundary, have coefficients which are not defined on the boundary. Since this routine calls the user-supplied subroutine PDEF at all points in the domain, including boundary points, arithmetic errors may occur in the user's routine PDEF which this routine cannot trap. If the user has an

equation with Dirichlet boundary conditions (i.e., B = 0 at all points on the boundary), but with PDE coefficients which are singular on the boundary, then D03EDF could be called directly only using interior grid points with the user's own discretization.

After the equations have been set up as described above, they are checked for diagonal dominance. That is to say,

$$|A_{ij}| > \sum_{\substack{k=1 \\ k \neq 4}}^{4} |A_{ij}|, \quad i=1,2,\ldots,n \ ; \ j=1,2,\ldots,n \ .$$
$$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x \qquad\qquad y$$

If this condition is not satisfied then the routine returns with IFAIL = 6. The multigrid routine D03EDF may still converge in this case, but if the coefficients of the first derivatives in the partial differential equation are large compared with the coefficients of the second derivative, the user should consider using upwind differences (SCHEME = 'U').

Since this routine is designed primarily for use with D03EDF, this document should be read in conjunction with the document for that routine.

4. References

[1] Wesseling P (1982) MGD1 - A Robust and Efficient Multigrid Method. Multigrid Methods. Lecture Notes in Mathematics. 960 Springer-Verlag. 614--630.

5. Parameters

1: XMIN -- DOUBLE PRECISION Input

2: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper x co-ordinates of the rectangular region respectively, x_A and x_B. Constraint: XMIN < XMAX.

3: YMIN -- DOUBLE PRECISION Input

4: YMAX -- DOUBLE PRECISION Input
On entry: the lower and upper y co-ordinates of the rectangular region respectively, y_A and y_B. Constraint: YMIN < YMAX.

5: PDEF -- SUBROUTINE, supplied by the user.

External Procedure

PDEF must evaluate the functions (alpha)(x,y), (beta)(x,y), (gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y) and (psi)(x,y) which define the equation at a general point (x,y).

Its specification is:

```
      SUBROUTINE PDEF (X, Y, ALPHA, BETA, GAMMA,
     1                 DELTA, EPSLON, PHI, PSI)
      DOUBLE PRECISION X, Y, ALPHA, BETA, GAMMA, DELTA,
     1                 EPSLON, PHI, PSI
```

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input
On entry: the x and y co-ordinates of the point at which the coefficients of the partial differential equation are to be evaluated. 8

3: ALPHA -- DOUBLE PRECISION Output

4: BETA -- DOUBLE PRECISION Output

5: GAMMA -- DOUBLE PRECISION Output

6: DELTA -- DOUBLE PRECISION Output

7: EPSLON -- DOUBLE PRECISION Output

8: PHI -- DOUBLE PRECISION Output

9: PSI -- DOUBLE PRECISION Output
On exit: ALPHA, BETA, GAMMA, DELTA, EPSLON, PHI and PSI must be set to the values of (alpha)(x,y), (beta)(x,y), (gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y) and (psi)(x,y) respectively at the point specified by X and Y.

PDEF must be declared as EXTERNAL in the (sub)program from which D03EEF is called. Parameters denoted as Input must not be changed by this procedure.

6: BNDY -- SUBROUTINE, supplied by the user.
External Procedure

BNDY must evaluate the functions a(x,y), b(x,y), and c(x,y) involved in the boundary conditions.

Its specification is:

```
      SUBROUTINE BNDY (X, Y, A, B, C, IBND)
```

```
      INTEGER          IBND
      DOUBLE PRECISION X, Y, A, B, C

 1:  X -- DOUBLE PRECISION                                  Input

 2:  Y -- DOUBLE PRECISION                                  Input
     On entry: the x and y co-ordinates of the point at
     which the boundary conditions are to be evaluated.

 3:  A -- DOUBLE PRECISION                                 Output

 4:  B -- DOUBLE PRECISION                                 Output

 5:  C -- DOUBLE PRECISION                                 Output
     On exit: A, B and C must be set to the values of the
     functions appearing in the boundary conditions.

 6:  IBND -- INTEGER                                        Input
     On entry: specifies on which boundary the point (X,Y)
     lies. IBND = 0, 1, 2 or 3 according as the point lies
     on the bottom, right, top or left boundary.
    BNDY must be declared as EXTERNAL in the (sub)program
    from which D03EEF is called. Parameters denoted as
    Input must not be changed by this procedure.

 7:  NGX -- INTEGER                                         Input

 8:  NGY -- INTEGER                                         Input
    On entry: the number of interior grid points in the x- and y
    -directions respectively, n  and n . If the seven-diagonal
                               x      y
    equations are to be solved by D03EDF, then NGX-1 and NGY-1
    should preferably be divisible by as high a power of 2 as
    possible. Constraint: NGX >= 3, NGY >= 3.

 9:  LDA -- INTEGER                                         Input
    On entry:
    the first dimension of the array A as declared in the
    (sub)program from which D03EEF is called.
    Constraint: if only the seven-diagonal equations are
    required, then LDA >= NGX*NGY. If a call to this routine is
    to be followed by a call to D03EDF to solve the seven-
    diagonal linear equations, LDA >= (4*(NGX+1)*(NGY+1))/3.

    Note: this routine only checks the former condition. D03EDF,
    if called, will check the latter condition.

10:  A(LDA,7) -- DOUBLE PRECISION array                    Output
    On exit: A(i,j), for i=1,2,...,NGX*NGY; j = 1,2,...,7,
    contains the seven-diagonal linear equations produced by the
```

```
     discretization described above. If LDA > NGX*NGY, the
     remaining elements are not referenced by the routine, but if
     LDA >= (4*(NGX+1)*(NGY+1))/3 then the array A can be passed
     directly to D03EDF, where these elements are used as
     workspace.

11:  RHS(LDA) -- DOUBLE PRECISION array                    Output
     On exit: the first NGX*NGY elements contain the right-hand
     sides of the seven-diagonal linear equations produced by the
     discretization described above. If LDA > NGX*NGY, the
     remaining elements are not referenced by the routine, but if
     LDA >= (4*(NGY+1)*(NGY+1))/3 then the array RHS can be
     passed directly to D03EDF, where these elements are used as
     workspace.

12:  SCHEME -- CHARACTER*1                                 Input
     On entry: the type of approximation to be used for the first
     derivatives which occur in the partial differential
     equation.

     If SCHEME = 'C', then central differences are used.

     If SCHEME = 'U', then upwind differences are used.
     Constraint: SCHEME = 'C' or 'U'.

     Note: generally speaking, if at least one of the
     coefficients multiplying the first derivatives (DELTA or
     EPSLON as returned by PDEF) are large compared with the
     coefficients multiplying the second derivatives, then upwind
     differences may be more appropriate. Upwind differences are
     less accurate than central differences, but may result in
     more rapid convergence for strongly convective equations.
     The easiest test is to try both schemes.

13:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:
```

```
If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry XMIN >= XMAX,

     or       YMIN >= YMAX,

     or       NGX < 3,

     or       NGY < 3,

     or       LDA < NGX*NGY,

     or       SCHEME is not one of 'C' or 'U'.

IFAIL= 2
     At some point on the boundary there is a derivative in the
     boundary conditions (B /= 0 on return from a BNDY) and there
                                                          2
                                                        dd U
     is a non-zero coefficient of the mixed derivative  ------
                                                       ddxddy
     (BETA /= 0 on return from PDEF).

IFAIL= 3
     A null boundary has been specified, i.e., at some point both
     A and B are zero on return from a call to BNDY.

IFAIL= 4
                                                           2
     The equation is not elliptic, i.e., 4*ALPHA*GAMMA<BETA  after
     a call to PDEF. The discretization has been completed, but
     the convergence of D03EDF cannot be guaranteed.

IFAIL= 5
     The boundary conditions are purely Neumann (only the
     derivative is specified) and there is, in general, no unique
     solution.

IFAIL= 6
     The equations were not diagonally dominant. (See Section 3).

7. Accuracy

Not applicable.

8. Further Comments
```

```
If this routine is used as a pre-processor to the multigrid
routine D03EDF it should be noted that the rate of convergence of
that routine is strongly dependent upon the number of levels in
the multigrid scheme, and thus the choice of NGX and NGY is very
important.

9. Example

The program solves the elliptic partial differential equation

                  2     2
                dd U  dd U   { ddU  ddU}
                ----+ ----+50{ ---+ ---}=f(x,y)
                   2     2   { ddx  ddy}
                ddx   ddy

on the unit square 0<=x, y<=1, with boundary conditions

      ddU
      --- given on x=0 and y=0,
      ddn

     U given on x=1 and y=1.

The function f(x,y) and the exact form of the boundary conditions
are derived from the exact solution U(x,y)=sinxsiny.

The equation is first solved using central differences. Since the
coefficients of the first derivatives are large, the linear
equations are not diagonally dominated, and convergence is slow.
The equation is solved a second time with upwind differences,
showing that convergence is more rapid, but the solution is less
accurate.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D03 -- Partial Differential Equations                        D03FAF
        D03FAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose
```

D03FAF solves the Helmholtz equation in Cartesian co-ordinates in three dimensions using the standard seven-point finite difference approximation. This routine is designed to be particularly efficient on vector processors.

2. Specification

```
      SUBROUTINE D03FAF (XS, XF, L, LBDCND, BDXS, BDXF, YS, YF,
     1                   M, MBDCND, BDYS, BDYF, ZS, ZF, N,
     2                   NBDCND, BDZS, BDZF, LAMBDA, LDIMF,
     3                   MDIMF, F, PERTRB, W, LWRK, IFAIL)
      INTEGER           L, LBDCND, M, MBDCND, N, NBDCND, LDIMF,
     1                  MDIMF, LWRK, IFAIL
      DOUBLE PRECISION XS, XF, BDXS(MDIMF,N+1), BDXF(MDIMF,N+1),
     1                  YS, YF, BDYS(LDIMF,N+1), BDYF(LDIMF,N+1),
     2                  ZS, ZF, BDZS(LDIMF,M+1), BDZF(LDIMF,M+1),
     3                  LAMBDA, F(LDIMF,MDIMF,N+1), PERTRB, W
     4                  (LWRK)
```

3. Description

D03FAF solves the three-dimensional Helmholtz equation in cartesian co-ordinates:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}+(\lambda)u=f(x,y,z)$$

This subroutine forms the system of linear equations resulting from the standard seven-point finite difference equations, and then solves the system using a method based on the fast Fourier transform (FFT) described by Swarztrauber [1]. This subroutine is based on the routine HW3CRT from FISHPACK (see Swarztrauber and Sweet [2]).

More precisely, the routine replaces all the second derivatives by second-order central difference approximations, resulting in a block tridiagonal system of linear equations. The equations are modified to allow for the prescribed boundary conditions. Either the solution or the derivative of the solution may be specified on any of the boundaries, or the solution may be specified to be periodic in any of the three dimensions. By taking the discrete Fourier transform in the x- and y-directions, the equations are reduced to sets of tridiagonal systems of equations. The Fourier transforms required are computed using the multiple FFT routines found in Chapter C06 of the NAG Fortran Library.

4. References

[1] Swarztrauber P N (1984) Fast Poisson Solvers. Studies in Numerical Analysis. (ed G H Golub) Mathematical Association of America.

[2] Swarztrauber P N and Sweet R A (1979) Efficient Fortran Subprograms for the Solution of Separable Elliptic Partial Differential Equations. ACM Trans. Math. Softw. 5 352--364.

5. Parameters

1: XS -- DOUBLE PRECISION Input
On entry: the lower bound of the range of x, i.e., XS <= x <= XF. Constraint: XS < XF.

2: XF -- DOUBLE PRECISION Input
On entry: the upper bound of the range of x, i.e., XS <= x <= XF. Constraint: XS < XF.

3: L -- INTEGER Input
On entry: the number of panels into which the interval (XS,XF) is subdivided. Hence, there will be L+1 grid points in the x-direction given by x =XS+(i-1)*(delta)x, for
i
i=1,2,...,L+1, where (delta)x=(XF-XS)/L is the panel width. Constraint: L >= 5.

4: LBDCND -- INTEGER Input
On entry: indicates the type of boundary conditions at x = XS and x = XF.
LBDCND = 0
if the solution is periodic in x, i.e., u(XS,y,z)=u(XF,y,z).

LBDCND = 1
if the solution is specified at x = XS and x = XF.

LBDCND = 2
if the solution is specified at x = XS and the derivative of the solution with respect to x is specified at x = XF.

LBDCND = 3
if the derivative of the solution with respect to x is specified at x = XS and x = XF.

LBDCND = 4
if the derivative of the solution with respect to x is specified at x = XS and the solution is specified at x = XF.

```
      Constraint: 0 <= LBDCND <= 4.

 5:  BDXS(MDIMF,N+1) -- DOUBLE PRECISION array              Input
     On entry: the values of the derivative of the solution with
     respect to x at x = XS. When LBDCND = 3 or 4, BDXS
     (j,k)=u (XS,y ,z ), for j=1,2,...,M+1; k=1,2,...,N+1.
             x     j  k

     When LBDCND has any other value, BDXS is not referenced.

 6:  BDXF(MDIMF,N+1) -- DOUBLE PRECISION array              Input
     On entry: the values of the derivative of the solution with
     respect to x at x = XF. When LBDCND = 2 or 3, BDXF(j,k) =
     u (XF,y ,z ), for j=1,2,...,M+1; k=1,2,...,N+1.
      x     i  k

     When LBDCND has any other value, BDXF is not referenced.

 7:  YS -- DOUBLE PRECISION                                 Input
     On entry: the lower bound of the range of y, i.e., YS <= y
     <= YF. Constraint: YS < YF.

 8:  YF -- DOUBLE PRECISION                                 Input
     On entry: the upper bound of the range of y, i.e., YS <= y
     <= YF. Constraint: YS < YF.

 9:  M -- INTEGER                                           Input
     On entry: the number of panels into which the interval
     (YS,YF) is subdivided. Hence, there will be M+1 grid points
     in the y-direction given by y  = YS + (j-1)*(delta)y for
                                  j
     j=1,2,...,M+1, where (delta)y=(YF-YS)/M is the panel width.
     Constraint: M >= 5.

10:  MBDCND -- INTEGER                                      Input
     On entry: indicates the type of boundary conditions at y =
     YS and y = YF.
     MBDCND = 0
           if the solution is periodic in y, i.e.,
           u(x,YF,z)=u(x,YS,z).

     MBDCND = 1
           if the solution is specified at y = YS and y = YF.

     MBDCND = 2
           if the solution is specified at y = YS and the
           derivative of the solution with respect to y is
           specified at y = YF.

     MBDCND = 3
```

```
            if the derivative of the solution with respect to y is
            specified at y = YS and y = YF.

      MBDCND = 4
            if the derivative of the solution with respect to y is
            specified at y = YS and the solution is specified at y
            = YF.
       Constraint: 0 <= MBDCND <= 4.

11:   BDYS(LDIMF,N+1) -- DOUBLE PRECISION array              Input
      On entry: the values of the derivative of the solution with
      respect to y at y = YS. When MBDCND = 3 or 4, BDYS
      (i,k)=u (x ,y ,z ), for i=1,2,...,L+1; k=1,2,...,N+1.
             y  i  s  k

      When MBDCND has any other value, BDYS is not referenced.

12:   BDYF(LDIMF,N+1) -- DOUBLE PRECISION array              Input
      On entry: the values of the derivative of the solution with
      respect to y at y = YF. When MBDCND = 2 or 3, BDYF
      (i,k)=u (x ,YF,z ), for i=1,2,...,L+1; k=1,2,...,N+1.
             y  i     k

      When MBDCND has any other value, BDYF is not referenced.

13:   ZS -- DOUBLE PRECISION                                 Input
      On entry: the lower bound of the range of z, i.e., ZS <= z
      <= ZF. Constraint: ZS < ZF.

14:   ZF -- DOUBLE PRECISION                                 Input
      On entry: the upper bound of the range of z, i.e., ZS <= z
      <= ZF. Constraint: ZS < ZF.

15:   N -- INTEGER                                          Input
      On entry: the number of panels into which the interval
      (ZS,ZF) is subdivided. Hence, there will be N+1 grid points
      in the z-direction given by z =ZS+(k-1)*(delta)z, for
                                    k
      k=1,2,...,N+1, where (delta)z=(ZF-ZS)/N is the panel width.
      Constraint: N >= 5.

16:   NBDCND -- INTEGER                                      Input
      On entry: specifies the type of boundary conditions at z =
      ZS and z = ZF.
      NBDCND = 0
            if the solution is periodic in z, i.e.,
            u(x,y,ZF)=u(x,y,ZS).

      NBDCND = 1
            if the solution is specified at z = ZS and z = ZF.
```

```
      NBDCND = 2
            if the solution is specified at z = ZS and the
            derivative of the solution with respect to z is
            specified at z = ZF.

      NBDCND = 3
            if the derivative of the solution with respect to z is
            specified at z = ZS and z = ZF.

      NBDCND = 4
            if the derivative of the solution with respect to z is
            specified at z = ZS and the solution is specified at z
            = ZF.
       Constraint: 0 <= NBDCND <= 4.

17:   BDZS(LDIMF,M+1) -- DOUBLE PRECISION array              Input
      On entry: the values of the derivative of the solution with
      respect to z at z = ZS. When NBDCND = 3 or 4, BDZS
      (i,j)=u (x ,y ,ZS)=u(x,y,z), for i=1,2,...,L+1;
             z  i  j
      j=1,2,...,M+1.

      When NBDCND has any other value, BDZS is not referenced.

18:   BDZF(LDIMF,M+1) -- DOUBLE PRECISION array              Input
      On entry: the values of the derivative of the solution with
      respect to z at z = ZF. When NBDCND = 2 or 3, BDZF
      (i,j)=u (x ,y ,ZF)=u(x,y,z), for i=1,2,...,L+1;
             z  i  j
      j=1,2,...,M+1.

      When NBDCND has any other value, BDZF is not referenced.

19:   LAMBDA -- DOUBLE PRECISION                            Input
      On entry: the constant (lambda) in the Helmholtz equation.
      For certain positive values of (lambda) a solution to the
      differential equation may not exist, and close to these
      values the solution of the discretized problem will be
      extremely ill-conditioned. If (lambda)>0, then D03FAF will
      set IFAIL to 3, but will still attempt to find a solution.
      However, since in general the values of (lambda) for which
      no solution exists cannot be predicted a priori, the user is
      advised to treat any results computed with (lambda)>0 with
      great caution.

20:   LDIMF -- INTEGER                                      Input
      On entry:
      the first dimension of the arrays F, BDYS, BDYF, BDZS and
      BDZF as declared in the (sub)program from which D03FAF is
```

```
     called.
     Constraint: LDIMF >= L + 1.

21:  MDIMF -- INTEGER                                          Input
     On entry: the second dimension of the array F and
     the first dimension of the arrays BDXS and BDXF as declared
     in the (sub)program from which D03FAF is called.
     Constraint: MDIMF >= M + 1.

22:  F(LDIMF,MDIMF,N+1) -- DOUBLE PRECISION array    Input/Output
     On entry: the values of the right-side of the Helmholtz
     equation and boundary values (if any).

     F(i,j,k)=f(x ,y ,z ) i=2,3,...,L, j=2,3,...,M and k
                i  j  k
     =2,3,...,N.

     On the boundaries F is defined by
     LBDCND    F(1,j,k)  F(L+1,j,k)

     0         f(XS,y ,z )f(XS,y ,z )
                     j  k      j  k

     1         u(XS,y ,z )u(XF,y ,z )
                     j  k      j  k

     2         u(XS,y ,z )f(XF,y ,z )  j=1,2,...,M+1
                     j  k      j  k

     3         f(XS,y ,z )f(XF,y ,z )  k=1,2,...,N+1
                     j  k      j  k

     4         f(XS,y ,z )u(XF,y ,z )
                     j  k      j  k



     MBDCND    F(i,1,k)  F(
                         i,M+1,k)

     0         f(x ,YS,z )f(x ,YS,z )
                  i     k   i     k

     1         u(x ,YS,z )u(x ,YF,z )
                  i     k   i     k

     2         u(x ,YS,z )f(x ,YF,z )  i=1,2,...,L+1
                  i     k   i     k

     3         f(x ,YS,z )f(x ,YF,z )  k=1,2,...,N+1
```

```
                 i     k   i     k

4          f(x ,YS,z )u(x ,YF,z )
              i     k   i     k



NBDCND     F(i,j,1)  F(i,j,N+1
                     )

0          f(x ,y ,ZS)f(x ,y ,ZS)
              i  j      i  j

1          u(x ,y ,ZS)u(x ,y ,ZF)
              i  j      i  j

2          u(x ,y ,ZS)f(x ,y ,ZF)  i=1,2,...,L+1
              i  j      i  j

3          f(x ,y ,ZS)f(x ,y ,ZF)  j=1,2,...,M+1
              i  j      i  j

4          f(x ,y ,ZS)u(x ,y ,ZF)
              i  j      i  j
Note: if the table calls for both the solution u and the
right-hand side f on a boundary, then the solution must be
specified. On exit: F contains the solution u(i,j,k) of the
finite difference approximation for the grid point (
x ,y ,z ) for i=1,2,...,L+1, j=1,2,...,M+1 and k=1,2,...,N+1.
 i  j  k
```

23: PERTRB -- DOUBLE PRECISION Output

On exit: PERTRB = 0, unless a solution to Poisson's equation ((lambda)=0) is required with a combination of periodic or derivative boundary conditions (LBDCND, MBDCND and NBDCND = 0 or 3). In this case a solution may not exist. PERTRB is a constant, calculated and subtracted from the array F, which ensures that a solution exists. D03FAF then computes this solution, which is a least-squares solution to the original approximation. This solution is not unique and is unnormalised. The value of PERTRB should be small compared to the right-hand side F, otherwise a solution has been obtained to an essentially different problem. This comparison should always be made to insure that a meaningful solution has been obtained.

24: W(LWRK) -- DOUBLE PRECISION array Workspace

25: LWRK -- INTEGER Input

On entry:

```
     the dimension of the array W as declared in the (sub)program
     from which D03FAF is called.
     LRWK>=2*(N+1)*max(L,M)+3*L+3*M+4*N+6 is an upper bound on
     the required size of W. If LWRK is too small, the routine
     exits with IFAIL = 2, and if on entry IFAIL = 0 or IFAIL = -
     1, a message is output giving the exact value of LWRK
     required to solve the current problem.

26:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. Users who are
     unfamiliar with this parameter should refer to the Essential
     Introduction for details.

     On exit: IFAIL = 0 unless the routine detects an error or
     gives a warning (see Section 6).

     For this routine, because the values of output parameters
     may be useful even if IFAIL /=0 on exit, users are
     recommended to set IFAIL to -1 before entry. It is then
     essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry XS >= XF,

     or       L < 5,

     or       LBDCND < 0,

     or       LBDCND > 4,

     or       YS >= YF,

     or       M < 5,

     or       MBDCND < 0,

     or       MBDCND > 4,

     or       ZS >= ZF,

     or       N < 5,

     or       NBDCND < 0,
```

```
     or       NBDCND > 4,

     or       LDIMF < L + 1 > 0,

     or       MDIMF < M + 1.

IFAIL= 2
     On entry LWRK is too small.

IFAIL= 3
     On entry (lambda) > 0.

7. Accuracy

None.

8. Further Comments

The execution time is roughly proportional to
L*M*N*(log L+log M+5), but also depends on input parameters
          2     2
LBDCND and MBDCND.

9. Example

The example solves the Helmholz equation

                2     2     2
               dd u  dd u  dd u
               ----+ ----+ ----+(lambda)u=f(x,y,z)
                  2     2     2
               ddx   ddy   ddz

                                      (pi)
for (x,y,z)is in [0,1]*[0,2(pi)]*[0, ----] where (lambda)=-2, and
                                       2
f(x,y,z) is derived from the exact solution

                                 4
                      u(x,y,z)=x sin(y)cos(z).

The equation is subject to the following boundary conditions,
again derived from the exact solution given above.

     u(0,y,z) and u(1,y,z) are prescribed (i.e., LBDCND = 1).

     u(x,0,z)=u(x,2(pi),z) (i.e., MBDCND = 0).

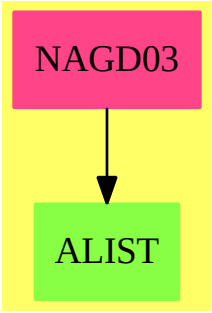
                           (pi)
```

```
u(x,y,0) and u (x,y, ----) are prescribed (i.e. NBDCND = 2).
                x       2

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

———-

15.11.1 NagPartialDifferentialEquationsPackage (NAGD03)



Exports:
d03edf d03eef d03faf

— package NAGD03 NagPartialDifferentialEquationsPackage —

```
)abbrev package NAGD03 NagPartialDifferentialEquationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:51 1994
++ Description:
++ This package uses the NAG Library to solve partial
++ differential equations.

NagPartialDifferentialEquationsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    d03edf : (Integer,Integer,Integer,Integer,_
        DoubleFloat,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
     ++ d03edf(ngx,ngy,lda,maxit,acc,iout,a,rhs,ub,ifail)
```

```
    ++ solves seven-diagonal systems of linear equations which
    ++ arise from the discretization of an elliptic partial differential
    ++ equation on a rectangular region. This routine uses a multigrid
    ++ technique.
    ++ See \downlink{Manual Page}{manpageXXd03edf}.
  d03eef : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
      Integer,Integer,Integer,String,Integer,_
      Union(fn:FileName,fp:Asp73(PDEF)),_
      Union(fn:FileName,fp:Asp74(BNDY))) -> Result
    ++ d03eef(xmin,xmax,ymin,ymax,ngx,ngy,lda,scheme,ifail,pdef,bndy)
    ++ discretizes a second order elliptic partial differential
    ++ equation (PDE) on a rectangular region.
    ++ See \downlink{Manual Page}{manpageXXd03eef}.
  d03faf : (DoubleFloat,DoubleFloat,Integer,Integer,_
      Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,DoubleFloat,_
      Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,_
      DoubleFloat,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
      DoubleFloat,Integer,Integer,Integer,_
      ThreeDimensionalMatrix DoubleFloat,Integer) -> Result
    ++ d03faf(xs,xf,l,lbdcnd,bdxs,bdxf,ys,yf,m,mbdcnd,bdys,bdyf,zs,
    ++        zf,n,nbdcnd,bdzs,bdzf,lambda,ldimf,mdimf,lwrk,f,ifail)
    ++ solves the Helmholtz equation in Cartesian co-ordinates in
    ++ three dimensions using the standard seven-point finite difference
    ++ approximation. This routine is designed to be particularly
    ++ efficient on vector processors.
    ++ See \downlink{Manual Page}{manpageXXd03faf}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
  import AnyFunctions1(Integer)
  import AnyFunctions1(String)
  import AnyFunctions1(DoubleFloat)
  import AnyFunctions1(Matrix DoubleFloat)
  import AnyFunctions1(ThreeDimensionalMatrix DoubleFloat)
  import FortranPackage
  import Union(fn:FileName,fp:Asp73(PDEF))
  import Union(fn:FileName,fp:Asp74(BNDY))

  d03edf(ngxArg:Integer,ngyArg:Integer,ldaArg:Integer,_
      maxitArg:Integer,accArg:DoubleFloat,ioutArg:Integer,_
      aArg:Matrix DoubleFloat,rhsArg:Matrix DoubleFloat,_
      ubArg:Matrix DoubleFloat,_
      ifailArg:Integer): Result ==
```

```
    [(invokeNagman(NIL$Lisp,_
    "d03edf",_
    ["ngx"::S,"ngy"::S,"lda"::S,"maxit"::S,"acc"::S,"iout"::S,"numit"::S,_
    "ifail"::S,"us"::S,"u"::S,"a"::S,"rhs"::S,"ub"::S_
    ]$Lisp,_
    ["us"::S,"u"::S,"numit"::S]$Lisp,_
    [["double"::S,"acc"::S,["us"::S,"lda"::S]$Lisp_
    ,["u"::S,"lda"::S]$Lisp,["a"::S,"lda"::S,7$Lisp]$Lisp,_
    ["rhs"::S,"lda"::S]$Lisp,_
    ["ub"::S,["*"::S,"ngx"::S,"ngy"::S]$Lisp]$Lisp]$Lisp_
    ,["integer"::S,"ngx"::S,"ngy"::S,"lda"::S,"maxit"::S_
    ,"iout"::S,"numit"::S,"ifail"::S]$Lisp]$Lisp,_
    ["us"::S,"u"::S,"numit"::S,"a"::S,"rhs"::S,"ub"::S,"ifail"::S]$Lisp,_
    [([ngxArg::Any,ngyArg::Any,ldaArg::Any,maxitArg::Any,accArg::Any,_
    ioutArg::Any,ifailArg::Any,aArg::Any,rhsArg::Any,ubArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

d03eef(xminArg:DoubleFloat,xmaxArg:DoubleFloat,yminArg:DoubleFloat,_
    ymaxArg:DoubleFloat,ngxArg:Integer,ngyArg:Integer,_
    ldaArg:Integer,schemeArg:String,ifailArg:Integer,_
    pdefArg:Union(fn:FileName,fp:Asp73(PDEF)),bndyArg:Union(fn:FileName,_
    fp:Asp74(BNDY))): Result ==
    pushFortranOutputStack(pdefFilename := aspFilename "pdef")$FOP
    if pdefArg case fn
      then outputAsFortran(pdefArg.fn)
      else outputAsFortran(pdefArg.fp)
    popFortranOutputStack()$FOP
    pushFortranOutputStack(bndyFilename := aspFilename "bndy")$FOP
    if bndyArg case fn
      then outputAsFortran(bndyArg.fn)
      else outputAsFortran(bndyArg.fp)
    popFortranOutputStack()$FOP
    [(invokeNagman([pdefFilename,bndyFilename]$Lisp,_
    "d03eef",_
    ["xmin"::S,"xmax"::S,"ymin"::S,"ymax"::S,"ngx"::S_
    ,"ngy"::S,"lda"::S,"scheme"::S,"ifail"::S,"pdef"::S_
    ,"bndy"::S,"a"::S,"rhs"::S]$Lisp,_
    ["a"::S,"rhs"::S,"pdef"::S,"bndy"::S]$Lisp,_
    [["double"::S,"xmin"::S,"xmax"::S,"ymin"::S,"ymax"::S,_
    ["a"::S,"lda"::S,7$Lisp]$Lisp,_
    ["rhs"::S,"lda"::S]$Lisp,"pdef"::S,"bndy"::S]$Lisp_
    ,["integer"::S,"ngx"::S,"ngy"::S,"lda"::S,"ifail"::S]$Lisp_
    ,["character"::S,"scheme"::S]$Lisp]$Lisp,_
    ["a"::S,"rhs"::S,"ifail"::S]$Lisp,_
    [([xminArg::Any,xmaxArg::Any,yminArg::Any,ymaxArg::Any,ngxArg::Any,_
    ngyArg::Any,ldaArg::Any,schemeArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result
```

```
    d03faf(xsArg:DoubleFloat,xfArg:DoubleFloat,lArg:Integer,_
        lbdcndArg:Integer,bdxsArg:Matrix DoubleFloat,_
        bdxfArg:Matrix DoubleFloat,_
        ysArg:DoubleFloat,yfArg:DoubleFloat,mArg:Integer,_
        mbdcndArg:Integer,bdysArg:Matrix DoubleFloat,_
        bdyfArg:Matrix DoubleFloat,_
        zsArg:DoubleFloat,zfArg:DoubleFloat,nArg:Integer,_
        nbdcndArg:Integer,bdzsArg:Matrix DoubleFloat,_
        bdzfArg:Matrix DoubleFloat,_
        lambdaArg:DoubleFloat,ldimfArg:Integer,mdimfArg:Integer,_
        lwrkArg:Integer,fArg:ThreeDimensionalMatrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "d03faf",_
        ["xs"::S,"xf"::S,"l"::S,"lbdcnd"::S,"ys"::S_
        ,"yf"::S,"m"::S,"mbdcnd"::S,"zs"::S,"zf"::S_
        ,"n"::S,"nbdcnd"::S,"lambda"::S,"ldimf"::S,"mdimf"::S_
        ,"lwrk"::S,"pertrb"::S,"ifail"::S,"bdxs"::S,"bdxf"::S,"bdys"::S,_
        "bdyf"::S,"bdzs"::S_
        ,"bdzf"::S,"f"::S,"w"::S]$Lisp,_
        ["pertrb"::S,"w"::S]$Lisp,_
        [["double"::S,"xs"::S,"xf"::S,["bdxs"::S,"mdimf"::S,_
        ["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp_
        ,["bdxf"::S,"mdimf"::S,["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp,"ys"::S,_
        "yf"::S,["bdys"::S,"ldimf"::S,["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp_
        ,["bdyf"::S,"ldimf"::S,["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp,"zs"::S_
        ,"zf"::S,["bdzs"::S,"ldimf"::S,["+"::S,"m"::S,1$Lisp]$Lisp]$Lisp,_
        ["bdzf"::S,"ldimf"::S,["+"::S,"m"::S,1$Lisp]$Lisp]$Lisp_
        ,"lambda"::S,"pertrb"::S,["f"::S,"ldimf"::S,"mdimf"::S,_
        ["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp,["w"::S,"lwrk"::S]$Lisp]$Lisp_
        ,["integer"::S,"l"::S,"lbdcnd"::S,"m"::S,"mbdcnd"::S_
        ,"n"::S,"nbdcnd"::S,"ldimf"::S,"mdimf"::S,"lwrk"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["pertrb"::S,"f"::S,"ifail"::S]$Lisp,_
        [([xsArg::Any,xfArg::Any,lArg::Any,lbdcndArg::Any,ysArg::Any,_
        yfArg::Any,mArg::Any,mbdcndArg::Any,zsArg::Any,zfArg::Any,_
        nArg::Any,nbdcndArg::Any,lambdaArg::Any,ldimfArg::Any,mdimfArg::Any,_
        lwrkArg::Any,ifailArg::Any,bdxsArg::Any,bdxfArg::Any,bdysArg::Any,_
        bdyfArg::Any,bdzsArg::Any,bdzfArg::Any,fArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGD03.dotabb** —

```
"NAGD03" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGD03"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
```

```
"NAGD03" -> "ALIST"
```

———-

15.12 package NAGC02 NagPolynomialRootsPackage

— **NagPolynomialRootsPackage.help** —

```
C02(3NAG)          Foundation Library (12/10/92)          C02(3NAG)

     C02 -- Zeros of Polynomials                    Introduction -- C02
                                Chapter C02
                           Zeros of Polynomials

     1. Scope of the Chapter

     This chapter is concerned with computing the zeros of a
     polynomial with real or complex coefficients.

     2. Background to the Problems

     Let f(z) be a polynomial of degree n with complex coefficients
     a :
      i

                          n    n-1    n-2
                 f(z)==a z +a z   +a z   +...+a   z+a ,  a /=0.
                        0    1      2          n-1   n    0

     A complex number z  is called a zero of f(z) (or equivalently a
                       1
     root of the equation f(z)=0), if:

                                  f(z )=0.
                                     1

     If z  is a zero, then f(z) can be divided by a factor (z-z ):
         1                                                     1

                             f(z)=(z-z )f (z)                          (1)
                                      1  1

     where f (z) is a polynomial of degree n-1. By the Fundamental
            1
     Theorem of Algebra, a polynomial f(z) always has a zero, and so
     the process of dividing out factors (z-z ) can be continued until
```

we have a complete factorization of f(z)

$$f(z)==a_0(z-z_1)(z-z_2)\ldots(z-z_n).$$

Here the complex numbers $z_1, z_2, \ldots, z_n$ are the zeros of f(z); they may not all be distinct, so it is sometimes more convenient to write:

$$f(z)==a_0(z-z_1)^{m_1}(z-z_2)^{m_2}\ldots(z-z_k)^{m_k},\quad k<=n,$$

with distinct zeros $z_1, z_2, \ldots, z_k$ and multiplicities $m_i >=1$. If $m_i =1$, z_i is called a single zero, if $m_i >1$, z_i is called a multiple or repeated zero; a multiple zero is also a zero of the derivative of f(z).

If the coefficients of f(z) are all real, then the zeros of f(z) are either real or else occur as pairs of conjugate complex numbers x+iy and x-iy. A pair of complex conjugate zeros are the zeros of a quadratic factor of f(z), (z^2+rz+s), with real coefficients r and s.

Mathematicians are accustomed to thinking of polynomials as pleasantly simple functions to work with. However the problem of numerically computing the zeros of an arbitrary polynomial is far from simple. A great variety of algorithms have been proposed, of which a number have been widely used in practice; for a fairly comprehensive survey, see Householder [1]. All general algorithms are iterative. Most converge to one zero at a time; the corresponding factor can then be divided out as in equation (1) above - this process is called deflation or, loosely, dividing out the zero - and the algorithm can be applied again to the polynomial $f_1(z)$. A pair of complex conjugate zeros can be divided out together - this corresponds to dividing f(z) by a quadratic factor.

Whatever the theoretical basis of the algorithm, a number of practical problems arise: for a thorough discussion of some of them see Peters and Wilkinson [2] and Wilkinson [3]. The most elementary point is that, even if z_1 is mathematically an exact

zero of f(z), because of the fundamental limitations of computer arithmetic the computed value of f(z) will not necessarily be
1
exactly 0.0. In practice there is usually a small region of values of z about the exact zero at which the computed value of f(z) becomes swamped by rounding errors. Moreover in many algorithms this inaccuracy in the computed value of f(z) results in a similar inaccuracy in the computed step from one iterate to the next. This limits the precision with which any zero can be computed. Deflation is another potential cause of trouble, since, in the notation of equation (1), the computed coefficients of f (z) will not be completely accurate, especially if z is not an
1 1
exact zero of f(z); so the zeros of the computed f (z) will
1
deviate from the zeros of f(z).

A zero is called ill-conditioned if it is sensitive to small changes in the coefficients of the polynomial. An ill-conditioned zero is likewise sensitive to the computational inaccuracies just mentioned. Conversely a zero is called well-conditioned if it is comparatively insensitive to such perturbations. Roughly speaking a zero which is well separated from other zeros is well-conditioned, while zeros which are close together are ill-conditioned, but in talking about 'closeness' the decisive factor is not the absolute distance between neighbouring zeros but their ratio: if the ratio is close to 1 the zeros are ill-conditioned. In particular, multiple zeros are ill-conditioned. A multiple zero is usually split into a cluster of zeros by perturbations in the polynomial or computational inaccuracies.

2.1. References

[1] Householder A S (1970) The Numerical Treatment of a Single Nonlinear Equation. McGraw-Hill.

[2] Peters G and Wilkinson J H (1971) Practical Problems Arising in the Solution of Polynomial Equations. J. Inst. Maths Applics. 8 16--35.

[3] Wilkinson J H (1963) Rounding Errors in Algebraic Processes, Chapter 2. HMSO.

3. Recommendations on Choice and Use of Routines

3.1. Discussion

Two routines are available: C02AFF for polynomials with complex coefficients and C02AGF for polynomials with real coefficients.

```
C02AFF and C02AGF both use a variant of Laguerre's Method due to
BT Smith to calculate each zero until the degree of the deflated
polynomial is less than 3, whereupon the remaining zeros are
obtained using the 'standard' closed formulae for a quadratic or
linear equation.

The accuracy of the roots will depend on how ill-conditioned they
are. Peters and Wilkinson [2] describe techniques for estimating
the errors in the zeros after they have been computed.

3.2. Index

 Zeros of a complex polynomial                              C02AFF
 Zeros of a real polynomial                                 C02AGF


C02 -- Zeros of Polynomials                      Contents -- C02
Chapter C02

Zeros of Polynomials

C02AFF  All zeros of complex polynomial, modified Laguerre method

C02AGF  All zeros of real polynomial, modified Laguerre method
```

%%

```
C02AFF(3NAG)      Foundation Library (12/10/92)      C02AFF(3NAG)

C02 -- Zeros of Polynomials                                C02AFF
        C02AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C02AFF finds all the roots of a complex polynomial equation,
using a variant of Laguerre's Method.

2. Specification

       SUBROUTINE C02AFF (A, N, SCALE, Z, W, IFAIL)
       INTEGER          N, IFAIL
       DOUBLE PRECISION A(2,N+1), Z(2,N), W(4*(N+1))
       LOGICAL          SCALE

3. Description
```

The routine attempts to find all the roots of the nth degree complex polynomial equation

$$P(z)=a_0 z^n + a_1 z^{n-1} + a_2 z^{n-2} + \ldots + a_{n-1} z + a_n = 0.$$

The roots are located using a modified form of Laguerre's Method, originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

$$L(z_k) = z_{k+1} - z_k = \frac{-n P(z_k)}{P'(z_k) \pm \sqrt{H(z_k)}},$$

where $H(z_k)=(n-1)[(n-1)(P'(z_k))^2 - nP(z_k)P''(z_k)]$, and z_0 is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at z_k, viz. $|L(z_k)|$, is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots. The routine generates a sequence of iterates $z_1, z_2, z_3, \ldots$, such that $|P(z_{k+1})| < |P(z_k)|$ and ensures that $z_{k+1} + L(z_{k+1})$ 'roughly' lies inside a circular region of radius $|F|$ about z_k known to contain a zero of $P(z)$; that is, $|L(z_{k+1})| \leq |F|$, where F denotes the Fejer bound (see Marden [1]) at the point z_k. Following Smith [2], F is taken to be min(B,1.445*n*R), where B is an upper bound for the magnitude of the smallest zero given by

$$B = 1.0001 * \min\left(\sqrt[n]{L(z_k)}, |r_1|, \left|\frac{a_n}{a_0}\right|^{1/n}\right),$$

r_1 is the zero X of smaller magnitude of the quadratic equation

$$2(P''(z_k)/(2*n*(n-1)))X^2 + 2(P'(z_k)/n)X + P(z_k) = 0$$

and the Cauchy lower bound R for the smallest zero is computed (using Newton's Method) as the positive root of the polynomial equation

$$|a_0|z^n + |a_1|z^{n-1} + |a_2|z^{n-2} + \ldots + |a_{n-1}|z - |a_n| = 0.$$

Starting from the origin, successive iterates are generated according to the rule $z_{k+1} = z_k + L(z_k)$ for k = 1,2,3,... and $L(z_k)$ is 'adjusted' so that $|P(z_{k+1})| < |P(z_k)|$ and $|L(z_{k+1})| <= |F|$. The iterative procedure terminates if $P(z_{k+1})$ is smaller in absolute value than the bound on the rounding error in $P(z_{k+1})$ and the current iterate $z = z_{k+1}$ is taken to be a zero of P(z). The deflated polynomial $\tilde{P}(z) = P(z)/(z - z_p)$ of degree n-1 is then formed, and the above procedure is repeated on the deflated polynomial until n<3, whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear (n = 1) or quadratic (n = 2) equation.

To obtain the roots of a quadratic polynomial, C02AHF(*) can be used.

4. References

[1] Marden M (1966) Geometry of Polynomials. Mathematical Surveys. 3 Am. Math. Soc., Providence, RI.

[2] Smith B T (1967) ZERPOL: A Zero Finding Algorithm for Polynomials Using Laguerre's Method. Technical Report. Department of Computer Science, University of Toronto, Canada.

[3] Wilkinson J H (1965) The Algebraic Eigenvalue Problem. Clarendon Press.

5. Parameters

```
 1:  A(2,N+1) -- DOUBLE PRECISION array                      Input
     On entry: if A is declared with bounds (2,0:N), then A(1,i)
     and A(2,i) must contain the real and imaginary parts of a
                                                                 i
                                n-i
     (i.e., the coefficient of z   ), for i=0,1,...,n.
     Constraint: A(1,0) /= 0.0 or A(2,0) /= 0.0.

 2:  N -- INTEGER                                            Input
     On entry: the degree of the polynomial, n. Constraint: N >=
     1.

 3:  SCALE -- LOGICAL                                        Input
     On entry: indicates whether or not the polynomial is to be
     scaled. See Section 8 for advice on when it may be
     preferable to set SCALE = .FALSE. and for a description of
     the scaling strategy. Suggested value: SCALE = .TRUE..

 4:  Z(2,N) -- DOUBLE PRECISION array                       Output
     On exit: the real and imaginary parts of the roots are
     stored in Z(1,i) and Z(2,i) respectively, for i=1,2,...,n.

 5:  W(4*(N+1)) -- DOUBLE PRECISION array                Workspace

 6:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry A(1,0) = 0.0 and A(2,0) = 0.0,

     or       N < 1.

IFAIL= 2
     The iterative procedure has failed to converge. This error
     is very unlikely to occur. If it does, please contact NAG
     immediately, as some basic assumption for the arithmetic has
     been violated. See also Section 8.
```

IFAIL= 3
     Either overflow or underflow prevents the evaluation of P(z)
     near some of its zeros. This error is very unlikely to
     occur. If it does, please contact NAG immediately. See also
     Section 8.

7. Accuracy

All roots are evaluated as accurately as possible, but because of
the inherent nature of the problem complete accuracy cannot be
guaranteed.

8. Further Comments

If SCALE = .TRUE., then a scaling factor for the coefficients is
chosen as a power of the base B of the machine so that the
                                                        EMAX-P
largest coefficient in magnitude approaches THRESH = B   .
Users should note that no scaling is performed if the largest
coefficient in magnitude exceeds THRESH, even if SCALE = .TRUE..
(For definition of B, EMAX and P see the Chapter Introduction
X02.)

However, with SCALE = .TRUE., overflow may be encountered when
the input coefficients a ,a ,a ,...,a vary widely in magnitude,
                       0 1 2   n
                                          (4*P)
particularly on those machines for which B   overflows. In
such cases, SCALE should be set to .FALSE. and the coefficients
scaled so that the largest coefficient in magnitude does not
        (EMAX-2*P)
exceed B   .

Even so, the scaling strategy used in C02AFF is sometimes
insufficient to avoid overflow and/or underflow conditions. In
such cases, the user is recommended to scale the independent
variable (z) so that the disparity between the largest and
smallest coefficient in magnitude is reduced. That is, use the
routine to locate the zeros of the polynomial d*P(cz) for some
suitable values of c and d. For example, if the original
                     -100   100 20   -10
polynomial was P(z)=2   i+2   z , then choosing c=2   and
   100   20
d=2   , for instance, would yield the scaled polynomial i+z ,
which is well-behaved relative to overflow and underflow and has
                10
zeros which are 2   times those of P(z).

If the routine fails with IFAIL = 2 or 3, then the real and
imaginary parts of any roots obtained before the failure occurred

```
are stored in Z in the reverse order in which they were found.
Let n  denote the number of roots found before the failure
     R
occurred. Then Z(1,n) and Z(2,n) contain the real and imaginary
parts of the 1st root found, Z(1,n-1) and Z(2,n-1) contain the
real and imaginary parts of the 2nd root found, ..., Z(1,n ) and
                                                             R
Z(2,n ) contain the real and imaginary parts of the n th root
     R                                               R
found. After the failure has occurred, the remaining 2*(n-n )
                                                       R
elements of Z contain a large negative number (equal to


-1/(X02AMF().\/2)).

9. Example

                                        5    4    3    2
To find the roots of the polynomial a z +a z +a z +a z +a z+a =0,
                                     0    1    2    3    4   5
where a =(5.0+6.0i), a =(30.0+20.0i), a =-(0.2+6.0i),
       0             1                 2
a =(50.0+100000.0i), a =-(2.0-40.0i) and a =(10.0+1.0i).
 3                   4                   5

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
C02AGF(3NAG)      Foundation Library (12/10/92)      C02AGF(3NAG)

C02 -- Zeros of Polynomials                                 C02AGF
        C02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C02AGF finds all the roots of a real polynomial equation, using a
variant of Laguerre's Method.

2. Specification

       SUBROUTINE C02AGF (A, N, SCALE, Z, W, IFAIL)
```

```
      INTEGER          N, IFAIL
      DOUBLE PRECISION A(N+1), Z(2,N), W(2*(N+1))
      LOGICAL          SCALE
```

3. Description

The routine attempts to find all the roots of the nth degree real polynomial equation

$$P(z)=a_0 z^n + a_1 z^{n-1} + a_2 z^{n-2} + \ldots + a_{n-1} z + a_n = 0.$$

The roots are located using a modified form of Laguerre's Method, originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

$$L(z_k) = z_{k+1} - z_k = \frac{-n P(z_k)}{P'(z_k) \pm \sqrt{H(z_k)}},$$

where $H(z_k)=(n-1)[(n-1)(P'(z_k))^2 - nP(z_k)P''(z_k)]$, and z_0 is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at z_k, viz. $|L(z_k)|$, is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots. The routine generates a sequence of iterates $z_1, z_2, z_3, \ldots$, such that $|P(z_{k+1})|<|P(z_k)|$ and ensures that $z_{k+1} + L(z_{k+1})$ 'roughly' lies inside a circular region of radius $|F|$ about z_k known to contain a zero of $P(z)$; that is, $|L(z_{k+1})|<=|F|$, where F denotes the Fejer bound (see Marden [1]) at the point z_k. Following Smith [2], F is taken to be min(B,1.445*n*R), where B is an upper bound for the magnitude of the smallest zero given by

$$1/n$$

$$B=1.0001*\min(\backslash/n*L(z\ ),|r\ |,|a\ /a\ |\quad),$$
$$\qquad\qquad\qquad\qquad\qquad\quad k\quad 1\quad n\ \ 0$$

r_1 is the zero X of smaller magnitude of the quadratic equation

$$2(P''(z_k)/(2*n*(n-1)))X^2+2(P'(z_k)/n)X+P(z_k)=0$$

and the Cauchy lower bound R for the smallest zero is computed (using Newton's Method) as the positive root of the polynomial equation

$$|a_0|z^n+|a_1|z^{n-1}+|a_2|z^{n-2}+\ldots+|a_{n-1}|z-|a_n|=0.$$

Starting from the origin, successive iterates are generated according to the rule $z_{k+1}=z_k+L(z_k)$ for k=1,2,3,... and $L(z_k)$ is

iterative procedure terminates if $P(z_{k+1})$ is smaller in absolute value than the bound on the rounding error in $P(z_{k+1})$ and the current iterate $z_k=z_{p\ k-1}$ is taken to be a zero of P(z) (as is its

conjugate z_p if z_p is complex). The deflated polynomial

$\tilde{P}(z)=P(z)/(z-z_p)$ of degree n-1 if z_p is real

$\tilde{P}(z)=P(z)/((z-z_p)(z-z_{\bar{p}}))$ of degree n-2 if z_p is complex) is then formed, and the above procedure is repeated on the deflated polynomial until n<3, whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear (n = 1) or quadratic (n = 2) equation.

To obtain the roots of a quadratic polynomial, C02AJF(*) can be used.

4. References

[1] Marden M (1966) Geometry of Polynomials. Mathematical Surveys. 3 Am. Math. Soc., Providence, RI.

[2] Smith B T (1967) ZERPOL: A Zero Finding Algorithm for Polynomials Using Laguerre's Method. Technical Report. Department of Computer Science, University of Toronto, Canada.

[3] Wilkinson J H (1965) The Algebraic Eigenvalue Problem. Clarendon Press.

5. Parameters

1: A(N+1) -- DOUBLE PRECISION array Input
On entry: if A is declared with bounds (0:N), then A(i) must contain a_i (i.e., the coefficient of z^{n-i}), for i=0,1,...,n. Constraint: A(0) /= 0.0.

2: N -- INTEGER Input
On entry: the degree of the polynomial, n. Constraint: N >= 1.

3: SCALE -- LOGICAL Input
On entry: indicates whether or not the polynomial is to be scaled. See Section 8 for advice on when it may be preferable to set SCALE = .FALSE. and for a description of the scaling strategy. Suggested value: SCALE = .TRUE..

4: Z(2,N) -- DOUBLE PRECISION array Output
On exit: the real and imaginary parts of the roots are stored in Z(1,i) and Z(2,i) respectively, for i=1,2,...,n. Complex conjugate pairs of roots are stored in consecutive pairs of elements of Z; that is, Z(1,i+1) = Z(1,i) and Z(2,i+1)=-Z(2,i).

5: W(2*(N+1)) -- DOUBLE PRECISION array Workspace

6: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

```
IFAIL= 1
     On entry A(0) = 0.0,

     or       N < 1.

IFAIL= 2
     The iterative procedure has failed to converge. This error
     is very unlikely to occur. If it does, please contact NAG
     immediately, as some basic assumption for the arithmetic has
     been violated. See also Section 8.

IFAIL= 3
     Either overflow or underflow prevents the evaluation of P(z)
     near some of its zeros. This error is very unlikely to
     occur. If it does, please contact NAG immediately. See also
     Section 8.

7. Accuracy

All roots are evaluated as accurately as possible, but because of
the inherent nature of the problem complete accuracy cannot be
guaranteed.

8. Further Comments

If SCALE = .TRUE., then a scaling factor for the coefficients is
chosen as a power of the base B of the machine so that the
                                                        EMAX-P
largest coefficient in magnitude approaches THRESH = B      .
Users should note that no scaling is performed if the largest
coefficient in magnitude exceeds THRESH, even if SCALE = .TRUE..
(For definition of B, EMAX and P see the Chapter Introduction
X02.)

However, with SCALE = .TRUE., overflow may be encountered when
the input coefficients a ,a ,a ,...,a  vary widely in magnitude,
                       0  1  2      n
                                           (4*P)
particularly on those machines for which B      overflows. In
such cases, SCALE should be set to .FALSE. and the coefficients
scaled so that the largest coefficient in magnitude does not
        (EMAX-2*P)
exceed B          .

Even so, the scaling strategy used in C02AGF is sometimes
insufficient to avoid overflow and/or underflow conditions. In
such cases, the user is recommended to scale the independent
variable (z) so that the disparity between the largest and
smallest coefficient in magnitude is reduced. That is, use the
```

routine to locate the zeros of the polynomial d*P(cz) for some
suitable values of c and d. For example, if the original
polynomial was $P(z)=2^{-100}+2^{100}z^{20}+2^{-10}$, then choosing $c=2^{100}$ and
$d=2^{20}$, for instance, would yield the scaled polynomial $1+z^{20}$,
which is well-behaved relative to overflow and underflow and has
zeros which are 2^{10} times those of P(z).

If the routine fails with IFAIL = 2 or 3, then the real and
imaginary parts of any roots obtained before the failure occurred
are stored in Z in the reverse order in which they were found.
Let n_R denote the number of roots found before the failure
occurred. Then Z(1,n) and Z(2,n) contain the real and imaginary
parts of the 1st root found, Z(1,n-1) and Z(2,n-1) contain the
real and imaginary parts of the 2nd root found, ..., Z(1,n_R) and
Z(2,n_R) contain the real and imaginary parts of the n_R th root
found. After the failure has occurred, the remaining $2*(n-n_R)$
elements of Z contain a large negative number (equal to

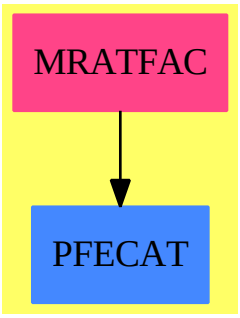
-1/(X02AMF().\/2)).

9. Example

To find the roots of the 5th degree polynomial
$z^5+2z^4+3z^3+4z^2+5z+6=0$.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

15.12.1 NagPolynomialRootsPackage (NAGC02)



Exports:
c02aff c02agf

— package NAGC02 NagPolynomialRootsPackage —

```
)abbrev package NAGC02 NagPolynomialRootsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:27 1994
++ Description:
++ This package uses the NAG Library to compute the zeros of a
++ polynomial with real or complex coefficients.

NagPolynomialRootsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    c02aff : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
     ++ c02aff(a,n,scale,ifail)
     ++ finds all the roots of a complex polynomial equation,
     ++ using a variant of Laguerre's Method.
     ++ See \downlink{Manual Page}{manpageXXc02aff}.
    c02agf : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
     ++ c02agf(a,n,scale,ifail)
     ++ finds all the roots of a real polynomial equation, using a
     ++ variant of Laguerre's Method.
     ++ See \downlink{Manual Page}{manpageXXc02agf}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Matrix DoubleFloat
    import Any
    import Record
```

```
    import Integer
    import Boolean
    import NAGLinkSupportPackage
    import AnyFunctions1(Matrix DoubleFloat)
    import AnyFunctions1(Integer)
    import AnyFunctions1(Boolean)

    c02aff(aArg:Matrix DoubleFloat,nArg:Integer,scaleArg:Boolean,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c02aff",_
        ["n"::S,"scale"::S,"ifail"::S,"a"::S,"z"::S,"w"::S]$Lisp,_
        ["z"::S,"w"::S]$Lisp,_
        [["double"::S,["a"::S,2$Lisp,["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp_
        ,["z"::S,2$Lisp,"n"::S]$Lisp,["w"::S,["*"::S,_
        ["+"::S,"n"::S,1$Lisp]$Lisp,4$Lisp]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
        ,["logical"::S,"scale"::S]$Lisp_
        ]$Lisp,_
        ["z"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,scaleArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c02agf(aArg:Matrix DoubleFloat,nArg:Integer,scaleArg:Boolean,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c02agf",_
        ["n"::S,"scale"::S,"ifail"::S,"a"::S,"z"::S,"w"::S]$Lisp,_
        ["z"::S,"w"::S]$Lisp,_
        [["double"::S,["a"::S,["+"::S,"n"::S,1$Lisp]$Lisp]$Lisp_
        ,["z"::S,2$Lisp,"n"::S]$Lisp,["w"::S,["*"::S,_
        ["+"::S,"n"::S,1$Lisp]$Lisp,2$Lisp]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
        ,["logical"::S,"scale"::S]$Lisp_
        ]$Lisp,_
        ["z"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,scaleArg::Any,ifailArg::Any,aArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— NAGC02.dotabb —

```
"NAGC02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGC02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGC02" -> "ALIST"
```

———-

15.13 package NAGC05 NagRootFindingPackage

— NagRootFindingPackage.help —

```
C05(3NAG)          Foundation Library (12/10/92)          C05(3NAG)

     C05 -- Roots of One or More Transcendental Equations
                                                  Introduction -- C05
                              Chapter C05
            Roots of One or More Transcendental Equations

     1. Scope of the Chapter

     This chapter is concerned with the calculation of real zeros of
     continuous real functions of one or more variables. (Complex
     equations must be expressed in terms of the equivalent larger
     system of real equations.)

     2. Background to the Problems

     The chapter divides naturally into two parts.

     2.1. A Single Equation

     The first deals with the real zeros of a real function of a
     single variable f(x).

     At present, there is only one routine with a simple calling
     sequence. This routine assumes that the user can determine an
     initial interval [a,b] within which the desired zero lies, that
     is f(a)*f(b)<0, and outside which all other zeros lie. The
     routine then systematically subdivides the interval to produce a
     final interval containing the zero. This final interval has a
     length bounded by the user's specified error requirements; the
     end of the interval where the function has smallest magnitude is
     returned as the zero. This routine is guaranteed to converge to a
     simple zero of the function. (Here we define a simple zero as a
     zero corresponding to a sign-change of the function.) The
     algorithm used is due to Bus and Dekker.

     2.2. Systems of Equations

     The routines in the second part of this chapter are designed to
```

solve a set of nonlinear equations in n unknowns

$$f_i(x)=0, \quad i=1,2,\ldots,n, \quad x=(x_1,x_2,\ldots,x_n)^T \qquad (1)$$

where T stands for transpose.

It is assumed that the functions are continuous and differentiable so that the matrix of first partial derivatives of the functions, the Jacobian matrix $J_{ij}(x)=df_i/dx_j$ evaluated at the point x, exists, though it may not be possible to calculate it directly.

The functions f_i must be independent, otherwise there will be an infinity of solutions and the methods will fail. However, even when the functions are independent the solutions may not be unique. Since the methods are iterative, an initial guess at the solution has to be supplied, and the solution located will usually be the one closest to this initial guess.

2.3. References

[1] Gill P E and Murray W (1976) Algorithms for the Solution of the Nonlinear Least-squares Problem. NAC 71 National Physical Laboratory.

[2] More J J, Garbow B S and Hillstrom K E (1974) User Guide for Minpack-1. ANL-80-74 Argonne National Laboratory.

[3] Ortega J M and Rheinboldt W C (1970) Iterative Solution of Nonlinear Equations in Several Variables. Academic Press.

[4] Rabinowitz P (1970) Numerical Methods for Nonlinear Algebraic Equations. Gordon and Breach.

3. Recommendations on Choice and Use of Routines

3.1. Zeros of Functions of One Variable

There is only one routine (C05ADF) for solving a single nonlinear equation. This routine is designed for solving problems where the function f(x) whose zero is to be calculated, can be coded as a user-supplied routine.

C05ADF may only be used when the user can supply an interval [a,b] containing the zero, that is f(a)*f(b)<0.

3.2. Solution of Sets of Nonlinear Equations

The solution of a set of nonlinear equations

$$f_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1, 2, \ldots, n \tag{2}$$

can be regarded as a special case of the problem of finding a minimum of a sum of squares

$$s(x) = \sum_{i=1}^{m} [f_i(x_1, x_2, \ldots, x_n)]^2 \quad (m \geq n). \tag{3}$$

So the routines in Chapter E04 of the Library are relevant as well as the special nonlinear equations routines.

There are two routines (C05NBF and C05PBF) for solving a set of nonlinear equations. These routines require the f_i (and possibly their derivatives) to be calculated in user-supplied routines. These should be set up carefully so the Library routines can work as efficiently as possible.

The main decision which has to be made by the user is whether to supply the derivatives $\frac{\partial f_i}{\partial x_j}$. It is advisable to do so if possible, since the results obtained by algorithms which use derivatives are generally more reliable than those obtained by algorithms which do not use derivatives.

C05PBF requires the user to provide the derivatives, whilst C05NBF does not. C05NBF and C05PBF are easy-to-use routines. A routine, C05ZAF, is provided for use in conjunction with C05PBF to check the user-provided derivatives for consistency with the functions themselves. The user is strongly advised to make use of this routine whenever C05PBF is used.

Firstly, the calculation of the functions and their derivatives should be ordered so that cancellation errors are avoided. This is particularly important in a routine that uses these quantities to build up estimates of higher derivatives.

Secondly, scaling of the variables has a considerable effect on

the efficiency of a routine. The problem should be designed so that the elements of x are of similar magnitude. The same comment applies to the functions, all the f_i should be of comparable size.

The accuracy is usually determined by the accuracy parameters of the routines, but the following points may be useful:

(i) Greater accuracy in the solution may be requested by choosing smaller input values for the accuracy parameters. However, if unreasonable accuracy is demanded, rounding errors may become important and cause a failure.

(ii) Some idea of the accuracies of the x_i may be obtained by monitoring the progress of the routine to see how many figures remain unchanged during the last few iterations.

(iii) An approximation to the error in the solution x, given by e where e is the solution to the set of linear equations

$$J(x)e=-f(x)$$

where $f(x)=(f_1(x),f_2(x),\ldots,f_n(x))^T$ (see Chapter F04).

(iv) If the functions $f_i(x)$ are changed by small amounts $(\text{epsilon})_i$, for i=1,2,...,n, then the corresponding change in the solution x is given approximately by (sigma), where (sigma) is the solution of the set of linear equations

$$J(x)(\text{sigma})=-(\text{epsilon}), \text{ (see Chapter F04).}$$

Thus one can estimate the sensitivity of x to any uncertainties in the specification of $f_i(x)$, for i=1,2,...,n.

3.3. Index

Zeros of functions of one variable:

Bus and Dekker algorithm	C05ADF

Zeros of functions of several variables:

easy-to-use	C05NBF
easy-to-use, derivatives required	C05PBF

Checking Routine:

```
     Checks user-supplied Jacobian                          C05ZAF


     C05 -- Roots of One or More Transcendental Equations
                                                    Contents -- C05
     Chapter C05

     Roots of One or More Transcendental Equations

     C05ADF  Zero of continuous function in given interval, Bus and
             Dekker algorithm

     C05NBF  Solution of system of nonlinear equations using function
             values only

     C05PBF  Solution of system of nonlinear equations using 1st
             derivatives

     C05ZAF  Check user's routine for calculating 1st derivatives

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C05ADF(3NAG)      Foundation Library (12/10/92)      C05ADF(3NAG)

     C05 -- Roots of One or More Transcendental Equations       C05ADF
             C05ADF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     C05ADF locates a zero of a continuous function in a given
     interval by a combination of the methods of linear interpolation,
     extrapolation and bisection.

     2. Specification

            SUBROUTINE C05ADF (A, B, EPS, ETA, F, X, IFAIL)
            INTEGER          IFAIL
            DOUBLE PRECISION A, B, EPS, ETA, F, X
            EXTERNAL         F

     3. Description

     The routine attempts to obtain an approximation to a simple zero
     of the function f(x) given an initial interval [a,b] such that
     f(a)*f(b)<=0. The zero is found by calls to C05AZF(*) whose
```

```
specification should be consulted for details of the method used.

The approximation x to the zero (alpha) is determined so that one
or both of the following criteria are satisfied:

     (i) |x-(alpha)|<EPS,

     (ii) |f(x)|<ETA.

4. References

None.

5. Parameters

 1:  A -- DOUBLE PRECISION                                    Input
     On entry: the lower bound of the interval, a.

 2:  B -- DOUBLE PRECISION                                    Input
     On entry: the upper bound of the interval, b. Constraint: B
     /= A.

 3:  EPS -- DOUBLE PRECISION                                  Input
     On entry: the absolute tolerance to which the zero is
     required (see Section 3). Constraint: EPS > 0.0.

 4:  ETA -- DOUBLE PRECISION                                  Input
     On entry: a value such that if |f(x)|<ETA, x is accepted as
     the zero. ETA may be specified as 0.0 (see Section 7).

 5:  F -- DOUBLE PRECISION FUNCTION, supplied by the user.
                                        External Procedure
     F must evaluate the function f whose zero is to be
     determined.

     Its specification is:

            DOUBLE PRECISION FUNCTION F (XX)
            DOUBLE PRECISION XX

      1:  XX -- DOUBLE PRECISION                              Input
          On entry: the point at which the function must be
          evaluated.
     F must be declared as EXTERNAL in the (sub)program from
     which C05ADF is called. Parameters denoted as Input
     must not be changed by this procedure.

 6:  X -- DOUBLE PRECISION                                   Output
     On exit: the approximation to the zero.
```

```
 7:  IFAIL -- INTEGER                                    Input/Output
     Before entry, IFAIL must be assigned a value. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     Unless the routine detects an error (see Section 6), IFAIL
     contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry EPS <= 0.0,

     or       A = B,

     or       F(A)*F(B)>0.0.

IFAIL= 2
     Too much accuracy has been requested in the computation,
     that is, EPS is too small for the computer being used. The
     final value of X is an accurate approximation to the zero.

IFAIL= 3
     A change in sign of f(x) has been determined as occurring
     near the point defined by the final value of X. However,
     there is some evidence that this sign-change corresponds to
     a pole of f(x).

IFAIL= 4
     Indicates that a serious error has occurred in C05AZF(*).
     Check all routine calls. Seek expert help.

7. Accuracy

This depends on the value of EPS and ETA. If full machine
accuracy is required, they may be set very small, resulting in an
error exit with IFAIL = 2, although this may involve more
iterations than a lesser accuracy. The user is recommended to set
ETA = 0.0 and to use EPS to control the accuracy, unless he has
considerable knowledge of the size of f(x) for values of x near
the zero.

8. Further Comments

The time taken by the routine depends primarily on the time spent
evaluating F (see Section 5).

If it is important to determine an interval of length less than
```

```
EPS containing the zero, or if the function F is expensive to
evaluate and the number of calls to F is to be restricted, then
use of C05AZF(*) is recommended. Use of C05AZF(*) is also
recommended when the structure of the problem to be solved does
not permit a simple function F to be written: the reverse
communication facilities of C05AZF(*) are more flexible than the
direct communication of F required by C05ADF.

9. Example

                                                        -x
The example program below calculates the zero of e  -x within the
interval [0,1] to approximately 5 decimal places.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
C05NBF(3NAG)      Foundation Library (12/10/92)      C05NBF(3NAG)

C05 -- Roots of One or More Transcendental Equations       C05NBF
       C05NBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C05NBF is an easy-to-use routine to find a solution of a system
of nonlinear equations by a modification of the Powell hybrid
method.

2. Specification

      SUBROUTINE C05NBF (FCN, N, X, FVEC, XTOL, WA, LWA, IFAIL)
      INTEGER          N, LWA, IFAIL
      DOUBLE PRECISION X(N), FVEC(N), XTOL, WA(LWA)
      EXTERNAL         FCN

3. Description

The system of equations is defined as:

                f (x ,x ,...,x )=0, for i=1,2,...,n.
                 i  1  2      n
```

C05NBF is based upon the MINPACK routine HYBRD1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is approximated by forward differences, but these are not used again until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].

4. References

[1] More J J, Garbow B S and Hillstrom K E User Guide for MINPACK-1. Technical Report ANL-80-74. Argonne National Laboratory.

[2] Powell M J D (1970) A Hybrid Method for Nonlinear Algebraic Equations. Numerical Methods for Nonlinear Algebraic Equations. (ed P Rabinowitz) Gordon and Breach.

5. Parameters

1: FCN -- SUBROUTINE, supplied by the user. External Procedure
FCN must return the values of the functions f_i at a point x.

Its specification is:

```
SUBROUTINE FCN (N, X, FVEC, IFLAG)
INTEGER          N, IFLAG
DOUBLE PRECISION X(N), FVEC(N)
```

1: N -- INTEGER Input
On entry: the number of equations, n.

2: X(N) -- DOUBLE PRECISION array Input
On entry: the components of the point x at which the functions must be evaluated.

3: FVEC(N) -- DOUBLE PRECISION array Output
On exit: the function values $f_i(x)$ (unless IFLAG is set to a negative value by FCN).

4: IFLAG -- INTEGER Input/Output
On entry: IFLAG > 0. On exit: in general, IFLAG should not be reset by FCN. If, however, the user wishes to terminate execution (perhaps because some illegal point

```
          X has been reached), then IFLAG should be set to a
          negative integer. This value will be returned through
          IFAIL.
     FCN must be declared as EXTERNAL in the (sub)program
     from which C05NBF is called. Parameters denoted as
     Input must not be changed by this procedure.

 2:  N -- INTEGER                                            Input
     On entry: the number of equations, n. Constraint: N > 0.

 3:  X(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: an initial guess at the solution vector. On
     exit: the final estimate of the solution vector.

 4:  FVEC(N) -- DOUBLE PRECISION array                      Output
     On exit: the function values at the final point, X.

 5:  XTOL -- DOUBLE PRECISION                                Input
     On entry: the accuracy in X to which the solution is
     required. Suggested value: the square root of the machine
     precision. Constraint: XTOL >= 0.0.

 6:  WA(LWA) -- DOUBLE PRECISION array                   Workspace

 7:  LWA -- INTEGER                                          Input
     On entry: the dimension of the array WA. Constraint:
     LWA>=N*(3*N+13)/2.

 8:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL< 0
     The user has set IFLAG negative in FCN. The value of IFAIL
     will be the same as the user's setting of IFLAG.

IFAIL= 1
     On entry N <= 0,
```

```
     or       XTOL < 0.0,

     or       LWA<N*(3*N+13)/2.

IFAIL= 2
     There have been at least 200*(N+1) evaluations of FCN.
     Consider restarting the calculation from the final point
     held in X.

IFAIL= 3
     No further improvement in the approximate solution X is
     possible; XTOL is too small.

IFAIL= 4
     The iteration is not making good progress. This failure exit
     may indicate that the system does not have a zero, or that
     the solution is very close to the origin (see Section 7).
     Otherwise, rerunning C05NBF from a different starting point
     may avoid the region of difficulty.

7. Accuracy

   ^
If x is the true solution, C05NBF tries to ensure that

                          ^             ^
                   ||x-x||<=XTOL*||x||.

                                              -k
If this condition is satisfied with XTOL=10   , then the larger
components of x have k significant decimal digits. There is a
danger that the smaller components of x may have large relative
errors, but the fast rate of convergence of C05NBF usually avoids
this possibility.

If XTOL is less than machine precision, and the above test is
satisfied with the machine precision in place of XTOL, then the
routine exits with IFAIL = 3.

Note: this convergence test is based purely on relative error,
and may not indicate convergence if the solution is very close to
the origin.

The test assumes that the functions are reasonably well behaved.
If this condition is not satisfied, then C05NBF may incorrectly
indicate convergence. The validity of the answer can be checked,
for example, by rerunning C05NBF with a tighter tolerance.

8. Further Comments
```

The time required by C05NBF to solve a given problem depends on n , the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by C05NBF to process each call of FCN is about $11.5*n^2$. Unless FCN can be evaluated quickly, the timing of C05NBF will be strongly influenced by the time spent in FCN.

Ideally the problem should be scaled so that at the solution the function values are of comparable magnitude.

9. Example

To determine the values $x_1,\ldots, x_9$ which satisfy the tridiagonal equations:

$$(3-2x_1)x_1 - 2x_2 = -1$$

$$-x_{i-1} + (3-2x_i)x_i - 2x_{i+1} = -1, \quad i=2,3,\ldots,8$$

$$-x_8 + (3-2x_9)x_9 = -1.$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

C05PBF(3NAG) Foundation Library (12/10/92) C05PBF(3NAG)

C05 -- Roots of One or More Transcendental Equations C05PBF
C05PBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C05PBF is an easy-to-use routine to find a solution of a system of nonlinear equations by a modification of the Powell hybrid method. The user must provide the Jacobian.

2. Specification

```
      SUBROUTINE C05PBF (FCN, N, X, FVEC, FJAC, LDFJAC, XTOL,
     1                   WA, LWA, IFAIL)
      INTEGER          N, LDFJAC, LWA, IFAIL
      DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N), XTOL, WA
     1                 (LWA)
      EXTERNAL         FCN
```

3. Description

The system of equations is defined as:

$$f_i(x_1, x_2, \ldots, x_n) = 0, \quad i = 1, 2, \ldots, n.$$

C05PBF is based upon the MINPACK routine HYBRJ1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is calculated, but it is not recalculated until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].

4. References

[1] More J J, Garbow B S and Hillstrom K E User Guide for MINPACK-1. Technical Report ANL-80-74. Argonne National Laboratory.

[2] Powell M J D (1970) A Hybrid Method for Nonlinear Algebraic Equations. Numerical Methods for Nonlinear Algebraic Equations. (ed P Rabinowitz) Gordon and Breach.

5. Parameters

1: FCN -- SUBROUTINE, supplied by the user. External Procedure

Depending upon the value of IFLAG, FCN must either return the values of the functions f_i at a point x or return the Jacobian at x.

Its specification is:

```
      SUBROUTINE FCN (N, X, FVEC, FJAC, LDFJAC, IFLAG)
      INTEGER          N, LDFJAC, IFLAG
      DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N)
```

1: N -- INTEGER Input

```
        On entry: the number of equations, n.

     2: X(N) -- DOUBLE PRECISION array                      Input
        On entry: the components of the point x at which the
        functions or the Jacobian must be evaluated.

     3: FVEC(N) -- DOUBLE PRECISION array                  Output
        On exit: if IFLAG = 1 on entry, FVEC must contain the
        function values f (x) (unless IFLAG is set to a
                         i
        negative value by FCN). If IFLAG = 2 on entry, FVEC
        must not be changed.

     4: FJAC(LDFJAC,N) -- DOUBLE PRECISION array           Output
        On exit: if IFLAG = 2 on entry, FJAC(i,j) must contain
                     ddf
                        i
        the value of ---- at the point x, for i,j=1,2,...,n
                     ddx
                        j
        (unless IFLAG is set to a negative value by FCN).

        If IFLAG = 1 on entry, FJAC must not be changed.

     5: LDFJAC -- INTEGER                                   Input
        On entry: the first dimension of FJAC.

     6: IFLAG -- INTEGER                              Input/Output
        On entry: IFLAG = 1 or 2:
             if IFLAG = 1, FVEC is to be updated;

             if IFLAG = 2, FJAC is to be updated.
        On exit: in general, IFLAG should not be reset by FCN.
        If, however, the user wishes to terminate execution
        (perhaps because some illegal point x has been reached)
        then IFLAG should be set to a negative integer. This
        value will be returned through IFAIL.
    FCN must be declared as EXTERNAL in the (sub)program
    from which C05PBF is called. Parameters denoted as
    Input must not be changed by this procedure.

2:  N -- INTEGER                                            Input
    On entry: the number of equations, n. Constraint: N > 0.

3:  X(N) -- DOUBLE PRECISION array                    Input/Output
    On entry: an initial guess at the solution vector. On
    exit: the final estimate of the solution vector.

4:  FVEC(N) -- DOUBLE PRECISION array                       Output
    On exit: the function values at the final point, X.
```

```
 5:  FJAC(LDFJAC,N) -- DOUBLE PRECISION array              Output
     On exit: the orthogonal matrix Q produced by the QR
     factorization of the final approximate Jacobian.

 6:  LDFJAC -- INTEGER                                     Input
     On entry:
     the first dimension of the array FJAC as declared in the
     (sub)program from which C05PBF is called.
     Constraint: LDFJAC >= N.

 7:  XTOL -- DOUBLE PRECISION                              Input
     On entry: the accuracy in X to which the solution is
     required. Suggested value: the square root of the machine
     precision. Constraint: XTOL >= 0.0.

 8:  WA(LWA) -- DOUBLE PRECISION array                   Workspace

 9:  LWA -- INTEGER                                        Input
     On entry: the dimension of the array WA. Constraint:
     LWA>=N*(N+13)/2.

10:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL< 0
     A negative value of IFAIL indicates an exit from C05PBF
     because the user has set IFLAG negative in FCN. The value of
     IFAIL will be the same as the user's setting of IFLAG.

IFAIL= 1
     On entry N <= 0,

     or       LDFJAC < N,

     or       XTOL < 0.0,

     or       LWA<N*(N+13)/2.
```

```
IFAIL= 2
     There have been 100*(N+1) evaluations of the functions.
     Consider restarting the calculation from the final point
     held in X.

IFAIL= 3
     No further improvement in the approximate solution X is
     possible; XTOL is too small.

IFAIL= 4
     The iteration is not making good progress. This failure exit
     may indicate that the system does not have a zero or that
     the solution is very close to the origin (see Section 7).
     Otherwise, rerunning C05PBF from a different starting point
     may avoid the region of difficulty.

7. Accuracy

   ^
If x is the true solution, C05PBF tries to ensure that

                        ^              ^
                  ||x-x|| <=XTOL*||x|| .
                         2            2

                                          -k
If this condition is satisfied with XTOL=10  , then the larger
components of x have k significant decimal digits. There is a
danger that the smaller components of x may have large relative
errors, but the fast rate of convergence of C05PBF usually avoids
the possibility.

If XTOL is less than machine precision and the above test is
satisfied with the machine precision in place of XTOL, then the
routine exits with IFAIL = 3.

Note: this convergence test is based purely on relative error,
and may not indicate convergence if the solution is very close to
the origin.

The test assumes that the functions and Jacobian are coded
consistently and that the functions are reasonably well behaved.
If these conditions are not satisfied then C05PBF may incorrectly
indicate convergence. The coding of the Jacobian can be checked
using C05ZAF. If the Jacobian is coded correctly, then the
validity of the answer can be checked by rerunning C05PBF with a
tighter tolerance.

8. Further Comments
```

The time required by C05PBF to solve a given problem depends on n , the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by C05PBF is about $11.5*n^2$ to process each evaluation of the functions and about $1.3*n^3$ to process each evaluation of the Jacobian. Unless FCN can be evaluated quickly, the timing of C05PBF will be strongly influenced by the time spent in FCN.

Ideally the problem should be scaled so that, at the solution, the function values are of comparable magnitude.

9. Example

To determine the values $x_1, \ldots, x_9$ which satisfy the tridiagonal equations:

$$(3-2x_1)x_1 - 2x_2 = -1$$

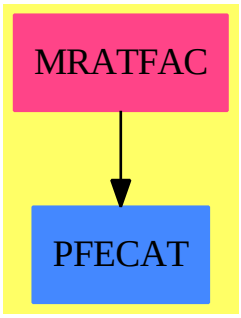
$$-x_{i-1} + (3-2x_i)x_i - 2x_{i+1} = -1, \quad i=2,3,\ldots,8.$$

$$-x_8 + (3-2x_9)x_9 = -1.$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

———-

15.13.1 NagRootFindingPackage (NAGC05)



Exports:
c05adf c05nbf c05pbf

— package NAGC05 NagRootFindingPackage —

```
)abbrev package NAGC05 NagRootFindingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:28 1994
++ Description:
++ This package uses the NAG Library to calculate real zeros of
++ continuous real functions of one or more variables. (Complex
++ equations must be expressed in terms of the equivalent larger
++ system of real equations.)

NagRootFindingPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    c05adf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
        Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
     ++ c05adf(a,b,eps,eta,ifail,f)
     ++ locates a zero of a continuous function in a given
     ++ interval by a combination of the methods of linear interpolation,
     ++ extrapolation and bisection.
     ++ See \downlink{Manual Page}{manpageXXc05adf}.
    c05nbf : (Integer,Integer,Matrix DoubleFloat,DoubleFloat,_
        Integer,Union(fn:FileName,fp:Asp6(FCN))) -> Result
     ++ c05nbf(n,lwa,x,xtol,ifail,fcn)
     ++ is an easy-to-use routine to find a solution of a system
     ++ of nonlinear equations by a modification of the Powell hybrid
     ++ method.
     ++ See \downlink{Manual Page}{manpageXXc05nbf}.
    c05pbf : (Integer,Integer,Integer,Matrix DoubleFloat,_
```

```
      DoubleFloat,Integer,Union(fn:FileName,fp:Asp35(FCN))) -> Result
    ++ c05pbf(n,ldfjac,lwa,x,xtol,ifail,fcn)
    ++ is an easy-to-use routine to find a solution of a system
    ++ of nonlinear equations by a modification of the Powell hybrid
    ++ method. The user must provide the Jacobian.
    ++ See \downlink{Manual Page}{manpageXXc05pbf}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Any
    import Record
    import Integer
    import Matrix DoubleFloat
    import Boolean
    import NAGLinkSupportPackage
    import FortranPackage
    import Union(fn:FileName,fp:Asp1(F))
    import AnyFunctions1(DoubleFloat)
    import AnyFunctions1(Matrix DoubleFloat)
    import AnyFunctions1(Integer)

    c05adf(aArg:DoubleFloat,bArg:DoubleFloat,epsArg:DoubleFloat,_
        etaArg:DoubleFloat,ifailArg:Integer,_
        fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
        pushFortranOutputStack(fFilename := aspFilename "f")$FOP
        if fArg case fn
          then outputAsFortran(fArg.fn)
          else outputAsFortran(fArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([fFilename]$Lisp,_
        "c05adf",_
        ["a"::S,"b"::S,"eps"::S,"eta"::S,"x"::S_
        ,"ifail"::S,"f"::S]$Lisp,_
        ["x"::S,"f"::S]$Lisp,_
        [["double"::S,"a"::S,"b"::S,"eps"::S,"eta"::S_
        ,"x"::S,"f"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"ifail"::S]$Lisp,_
        [([aArg::Any,bArg::Any,epsArg::Any,etaArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c05nbf(nArg:Integer,lwaArg:Integer,xArg:Matrix DoubleFloat,_
        xtolArg:DoubleFloat,ifailArg:Integer,_
        fcnArg:Union(fn:FileName,fp:Asp6(FCN))): Result ==
        pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
        if fcnArg case fn
          then outputAsFortran(fcnArg.fn)
```

```
          else outputAsFortran(fcnArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([fcnFilename]$Lisp,_
        "c05nbf",_
        ["n"::S,"lwa"::S,"xtol"::S,"ifail"::S,"fcn"::S_
        ,"fvec"::S,"x"::S,"wa"::S]$Lisp,_
        ["fvec"::S,"wa"::S,"fcn"::S]$Lisp,_
        [["double"::S,["fvec"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp_
        ,"xtol"::S,["wa"::S,"lwa"::S]$Lisp,"fcn"::S]$Lisp_
        ,["integer"::S,"n"::S,"lwa"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["fvec"::S,"x"::S,"xtol"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,lwaArg::Any,xtolArg::Any,ifailArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c05pbf(nArg:Integer,ldfjacArg:Integer,lwaArg:Integer,_
        xArg:Matrix DoubleFloat,xtolArg:DoubleFloat,ifailArg:Integer,_
        fcnArg:Union(fn:FileName,fp:Asp35(FCN))): Result ==
        pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
        if fcnArg case fn
          then outputAsFortran(fcnArg.fn)
          else outputAsFortran(fcnArg.fp)
        popFortranOutputStack()$FOP
        [(invokeNagman([fcnFilename]$Lisp,_
        "c05pbf",_
        ["n"::S,"ldfjac"::S,"lwa"::S,"xtol"::S,"ifail"::S_
        ,"fcn"::S,"fvec"::S,"fjac"::S,"x"::S,"wa"::S]$Lisp,_
        ["fvec"::S,"fjac"::S,"wa"::S,"fcn"::S]$Lisp,_
        [["double"::S,["fvec"::S,"n"::S]$Lisp,_
        ["fjac"::S,"ldfjac"::S,"n"::S]$Lisp_
        ,["x"::S,"n"::S]$Lisp,"xtol"::S,["wa"::S,"lwa"::S]$Lisp,"fcn"::S]$Lisp_
        ,["integer"::S,"n"::S,"ldfjac"::S,"lwa"::S_
        ,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["fvec"::S,"fjac"::S,"x"::S,"xtol"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,ldfjacArg::Any,lwaArg::Any,xtolArg::Any,_
        ifailArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGC05.dotabb** —

```
"NAGC05" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGC05"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGC05" -> "ALIST"
```

———-

15.14 package NAGC06 NagSeriesSummationPackage

— NagSeriesSummationPackage.help —

```
C06(3NAG)          Foundation Library (12/10/92)          C06(3NAG)


C06 -- Summation of Series                      Introduction -- C06
                          Chapter C06
                      Summation of Series

1. Scope of the Chapter

This chapter is concerned with calculating the discrete Fourier
transform of a sequence of real or complex data values, and
applying it to calculate convolutions and correlations.

2. Background to the Problems

2.1. Discrete Fourier Transforms

2.1.1.  Complex transforms

Most of the routines in this chapter calculate the finite
discrete Fourier transform  (DFT) of a sequence of n complex
numbers z , for j=0,1,...,n-1. The transform is defined by:
         j

                          n-1
                ^    1   --      (   2(pi)jk)
                z = --- >  z exp(-i -------)                  (1)
                 k      --  j   (      n   )
                    \/n j=0

for k=0,1,...,n-1. Note that equation (1) makes sense for all
                                   ^
integral k and with this extension z  is periodic with period n,
                                   k
     ^  ^                          ^   ^
i.e. z =z    , and in particular z  =z   .
      k  k+-n                     -k  n-k
```

```
                          ^                                   ^
If we write z =x +iy  and z =a +ib , then the definition of z
            j  j   j      k  k   k                           k
may be written in terms of sines and cosines as:

                n-1
             1  -- (     ( 2(pi)jk)      ( 2(pi)jk))
          a = --- >  (x cos( -------)+y sin( -------))
           k      -- ( j   (    n   )  j   (    n   ))
              \/n j=0

                n-1
             1  -- (     ( 2(pi)jk)      ( 2(pi)jk))
          b = --- >  (y cos( -------)-x sin( -------)).
           k      -- ( j   (    n   )  j   (    n   ))
              \/n j=0

The original data values z  may conversely be recovered from the
                         j
          ^
transform z  by an  inverse discrete Fourier transform:
           k


                          n-1
                       1  -- ^    (   2(pi)jk)
                    z = --- >  z exp(+i -------)                  (2)
                     j      --  k   (      n   )
                         \/n k=0

for j=0,1,...,n-1. If we take the complex conjugate of (2), we


                                                          ^
find that the sequence z  is the DFT of the sequence z . Hence
                        j                             k
                                  ^
the inverse DFT of the sequence z  may be obtained by: taking the
                                 k
                             ^
complex conjugates of the z ; performing a DFT; and taking the
                           k
complex conjugates of the result.

Notes: definitions of the discrete Fourier transform vary.
Sometimes (2) is used as the definition of the DFT, and (1) as


the definition of the inverse. Also the scale-factor of 1/\/n may
be omitted in the definition of the DFT, and replaced by 1/n in
```

the definition of the inverse.

2.1.2. Real transforms

If the original sequence is purely real valued, i.e. z =x , then
                                                      j j

$$\hat{z}_k = a_k + ib_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp\left(-i\frac{2\pi jk}{n}\right)$$

and $\hat{z}_{n-k}$ is the complex conjugate of $\hat{z}_k$. Thus the DFT of a real sequence is a particular type of complex sequence, called a Hermitian sequence, or half-complex or conjugate symmetric with the properties:

$$a_{n-k} = a_k \quad b_{n-k} = -b_k \quad b_0 = 0 \text{ and, if n is even, } b_{n/2} = 0.$$

Thus a Hermitian sequence of n complex data values can be represented by only n, rather than 2n, independent real values. This can obviously lead to economies in storage, the following scheme being used in this chapter: the real parts a_k for $0 \le k \le n/2$ are stored in normal order in the first n/2+1 locations of an array X of length n; the corresponding non-zero imaginary parts are stored in reverse order in the remaining locations of X. In other words, if X is declared with bounds (0:n-1) in the user's (sub)program, the real and imaginary parts of $\hat{z}_k$ are stored as follows:

	if n=2s	if n=2s-1
X(0)	a_0	a_0
X(1)	a_1	a_1
X(2)	a_2	a_2
.	.	.
.	.	.

```
    .          .          .

    X(s-1)     a          a
                s-1        s-1

    X(s)       a          b
                s          s-1

    X(s+1)     b          b
                s-1        s-2

    .          .          .

    .          .          .

    .          .          .

    X(n-2)     b          b
                2          2

    X(n-1)     b          b
                1          1
```

$$\text{Hence} \quad x_j = \frac{1}{n}\left(a_0 + 2\sum_{k=0}^{n/2-1}\left(a_k \cos\left(\frac{2\pi jk}{n}\right) - b_k \sin\left(\frac{2\pi jk}{n}\right)\right) + a_{n/2}\right)$$

where $a_{n/2} = 0$ if n is odd.

2.1.3. Fourier integral transforms

The usual application of the discrete Fourier transform is that of obtaining an approximation of the Fourier integral transform

$$F(s) = \int_{-\infty}^{+\infty} f(t)\exp(-i2\pi st)\,dt$$

when f(t) is negligible outside some region (0,c). Dividing the region into n equal intervals we have

$$\sum_{}^{n-1}$$

$$\tilde{F}(s) = \frac{c}{n} \sum_{j=0}^{n-1} f_j \exp(-i2\pi sjc/n)$$

and so

$$\tilde{F}_k = \frac{c}{n} \sum_{j=0}^{n-1} f_j \exp(-i2\pi jk/n)$$

for k=0,1,...,n-1, where $f_j = f(jc/n)$ and $\tilde{F}_k = F(k/c)$.

Hence the discrete Fourier transform gives an approximation to the Fourier integral transform in the region s=0 to s=n/c.

If the function f(t) is defined over some more general interval (a,b), then the integral transform can still be approximated by the discrete transform provided a shift is applied to move the point a to the origin.

2.1.4. Convolutions and correlations

One of the most important applications of the discrete Fourier transform is to the computation of the discrete convolution or correlation of two vectors x and y defined (as in Brigham [1]) by:

$$\text{convolution: } z_k = \sum_{j=0}^{n-1} x_j y_{k-j}$$

$$\text{correlation: } w_k = \sum_{j=0}^{n-1} x_j y_{k+j}$$

(Here x and y are assumed to be periodic with period n.)

Under certain circumstances (see Brigham [1]) these can be used as approximations to the convolution or correlation integrals defined by:

$$+\infty$$

$$z(s) = \int_{-\infty}^{} x(t)y(s-t)dt$$

and

$$w(s) = \int_{-\infty}^{+\infty} x(t)y(s+t)dt, \quad -\infty<s<+\infty.$$

For more general advice on the use of Fourier transforms, see Hamming [2]; more detailed information on the fast Fourier transform algorithm can be found in Van Loan [3] and Brigham [1].

2.2. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

[2] Hamming R W (1962) Numerical Methods for Scientists and Engineers. McGraw-Hill.

[3] Van Loan C (1992) Computational Frameworks for the Fast Fourier Transform. SIAM Philadelphia.

3. Recommendations on Choice and Use of Routines

3.1. One-dimensional Fourier Transforms

The choice of routine is determined first of all by whether the data values constitute a real, Hermitian or general complex sequence. It is wasteful of time and storage to use an inappropriate routine.

Two groups, each of three routines, are provided

	Group 1	Group 2
Real sequences	C06EAF	C06FPF
Hermitian sequences	C06EBF	C06FQF
General complex sequences	C06ECF	C06FRF

Group 1 routines each compute a single transform of length n, without requiring any extra working storage. The Group 1 routines

impose some restrictions on the value of n, namely that no prime factor of n may exceed 19 and the total number of prime factors (including repetitions) may not exceed 20 (though the latter
                                                    6
restriction only becomes relevant when n>10).

Group 2 routines are designed to perform several transforms in a single call, all with the same value of n. They do however require more working storage. Even on scalar processors, they may be somewhat faster than repeated calls to Group 1 routines because of reduced overheads and because they pre-compute and store the required values of trigonometric functions. Group 2 routines impose no practical restrictions on the value of n; however the fast Fourier transform algorithm ceases to be 'fast' if applied to values of n which cannot be expressed as a product of small prime factors. All the above routines are particularly efficient if the only prime factors of n are 2, 3 or 5.

If extensive use is to be made of these routines, users who are concerned about efficiency are advised to conduct their own timing tests.

To compute inverse discrete Fourier transforms the above routines should be used in conjunction with the utility routines C06GBF, C06GCF and C06GQF which form the complex conjugate of a Hermitian or general sequence of complex data values.

3.2. Multi-dimensional Fourier Transforms

C06FUF computes a 2-dimensional discrete Fourier transform of a 2-dimensional sequence of complex data values. This is defined by

```
                n -1 n -1
                 1    2          (   2(pi)j k )   (   2(pi)j k )
^         1     --   --          (         1 1)   (         2 2)
z     = ------- >    >   z    exp(-i ---------)exp(-i ---------).
                --   --          (      n     )   (      n     )
 k k     /n n   j =0 j =0 j j    (       1    )   (       2    )
  1 2  \/  1 2  1    2    1 2
```

3.3. Convolution and Correlation

C06EKF computes either the discrete convolution or the discrete correlation of two real vectors.

3.4. Index

Complex conjugate,
     complex sequence C06GCF
     Hermitian sequence C06GBF

```
     multiple Hermitian sequences                          C06GQF
Complex sequence from Hermitian sequences                  C06GSF
Convolution or Correlation
     real vectors                                          C06EKF
Discrete Fourier Transform
     two-dimensional
          complex sequence                                 C06FUF
     one-dimensional, multiple transforms
          complex sequence                                 C06FRF
          Hermitian sequence                               C06FQF
          real sequence                                    C06FPF
     one-dimensional, single transforms
          complex sequence                                 C06ECF
          Hermitian sequence                               C06EBF
          real sequence                                    C06EAF
```


C06 -- Summation of Series Contents -- C06
Chapter C06

Summation of Series

C06EAF Single 1-D real discrete Fourier transform, no extra workspace

C06EBF Single 1-D Hermitian discrete Fourier transform, no extra workspace

C06ECF Single 1-D complex discrete Fourier transform, no extra workspace

C06EKF Circular convolution or correlation of two real vectors, no extra workspace

C06FPF Multiple 1-D real discrete Fourier transforms

C06FQF Multiple 1-D Hermitian discrete Fourier transforms

C06FRF Multiple 1-D complex discrete Fourier transforms

C06FUF 2-D complex discrete Fourier transform

C06GBF Complex conjugate of Hermitian sequence

C06GCF Complex conjugate of complex sequence

C06GQF Complex conjugate of multiple Hermitian sequences

C06GSF Convert Hermitian sequences to general complex sequences

%%%

C06EAF(3NAG) Foundation Library (12/10/92) C06EAF(3NAG)

C06 -- Summation of Series C06EAF
C06EAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06EAF calculates the discrete Fourier transform of a sequence of n real data values. (No extra workspace required.)

2. Specification

```
SUBROUTINE C06EAF (X, N, IFAIL)
INTEGER          N, IFAIL
DOUBLE PRECISION X(N)
```

3. Description

Given a sequence of n real data values x_j, for $j=0,1,\ldots,n-1$, this routine calculates their discrete Fourier transform defined by:

$$\hat{z}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp\left(-i\,\frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1.$$

(Note the scale factor of $\frac{1}{\sqrt{n}}$ in this definition.) The transformed values $\hat{z}_k$ are complex, but they form a Hermitian sequence (i.e., $\hat{z}_{n-k}$ is the complex conjugate of $\hat{z}_k$), so they are completely determined by n real numbers (see also the Chapter Introduction).

To compute the inverse discrete Fourier transform defined by:

$$\hat{w}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp\left(+i\,\frac{2\pi jk}{n}\right),$$

this routine should be followed by a call of C06GBF to form the complex conjugates of the $\hat{z}_k$.

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of n (see Section 5).

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: if X is declared with bounds (0:N-1) in the (sub) program from which C06EAF is called, then X(j) must contain x_j, for j=0,1,...,n-1. On exit: the discrete Fourier transform stored in Hermitian form. If the components of the transform $\hat{z}_k$ are written as $a_k + ib_k$, and if X is declared with bounds (0:N-1) in the (sub)program from which C06EAF is called, then for 0<=k<=n/2, a_k is contained in X(k), and for 1<=k<=(n-1)/2, b_k is contained in X(n-k). (See also Section 2.1.2 of the Chapter Introduction, and the Example Program.)

2: N -- INTEGER Input
On entry: the number of data values, n. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential

```
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     At least one of the prime factors of N is greater than 19.

IFAIL= 2
     N has more than 20 prime factors.

IFAIL= 3
     N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
n*logn, but also depends on the factorization of n. The routine
is somewhat faster than average if the only prime factors of n
are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has
several unpaired prime factors, i.e., if the 'square-free' part
of n has several factors. For such values of n, routine C06FAF(*)
(which requires an additional n elements of workspace) is
considerably faster.

9. Example

This program reads in a sequence of real data values, and prints
their discrete Fourier transform (as computed by C06EAF), after
expanding it from Hermitian form into a full complex sequence.

It then performs an inverse transform using C06GBF and C06EBF,
and prints the sequence so obtained alongside the original data
values.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

C06EBF(3NAG) Foundation Library (12/10/92) C06EBF(3NAG)

C06 -- Summation of Series C06EBF
C06EBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06EBF calculates the discrete Fourier transform of a Hermitian sequence of n complex data values. (No extra workspace required.)

2. Specification

```
SUBROUTINE C06EBF (X, N, IFAIL)
INTEGER          N, IFAIL
DOUBLE PRECISION X(N)
```

3. Description

Given a Hermitian sequence of n complex data values z_j (i.e., a sequence such that z_0 is real and z_{n-j} is the complex conjugate of z_j, for j=1,2,...,n-1) this routine calculates their discrete Fourier transform defined by:

$$\hat{x}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \times \exp\left(-i\,\frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1.$$

(Note the scale factor of $\frac{1}{\sqrt{n}}$ in this definition.) The transformed values $\hat{x}_k$ are purely real (see also the the Chapter Introduction).

To compute the inverse discrete Fourier transform defined by:

$$\hat{y}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(+i\,\frac{2\pi jk}{n}\right),$$

this routine should be preceded by a call of C06GBF to form the complex conjugates of the z_j.

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of n (see Section 5).

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: the sequence to be transformed stored in Hermitian form. If the data values z_j are written as $x_j + iy_j$, and if X is declared with bounds (0:N-1) in the subroutine from which C06EBF is called, then for $0 \le j \le n/2$, x_j is contained in X(j), and for $1 \le j \le (n-1)/2$, y_j is contained in X(n-j). (See also Section 2.1.2 of the Chapter Introduction and the Example Program.) On exit: the components of the discrete Fourier transform $\hat{x}_k$. If X is declared with bounds (0:N-1) in the (sub)program from which C06EBF is called, then $\hat{x}_k$ is stored in X(k), for k=0,1,...,n-1.

2: N -- INTEGER Input
On entry: the number of data values, n. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not

```
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     At least one of the prime factors of N is greater than 19.

IFAIL= 2
     N has more than 20 prime factors.

IFAIL= 3
     N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
n*logn, but also depends on the factorization of n. The routine
is somewhat faster than average if the only prime factors of n
are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has
several unpaired prime factors, i.e., if the 'square-free' part
of n has several factors. For such values of n, routine C06FBF(*)
(which requires an additional n elements of workspace) is
considerably faster.

9. Example

This program reads in a sequence of real data values which is
assumed to be a Hermitian sequence of complex data values stored
in Hermitian form. The input sequence is expanded into a full
complex sequence and printed alongside the original sequence. The
discrete Fourier transform (as computed by C06EBF) is printed
out.

The program then performs an inverse transform using C06EAF and
C06GBF, and prints the sequence so obtained alongside the
original data values.
```

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
C06ECF(3NAG)      Foundation Library (12/10/92)      C06ECF(3NAG)

C06 -- Summation of Series                                    C06ECF
        C06ECF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C06ECF calculates the discrete Fourier transform of a sequence of
n complex data values. (No extra workspace required.)

2. Specification

       SUBROUTINE C06ECF (X, Y, N, IFAIL)
       INTEGER          N, IFAIL
       DOUBLE PRECISION X(N), Y(N)

3. Description

Given a sequence of n complex data values z , for j=0,1,...,n-1,
                                           j
this routine calculates their discrete Fourier transform defined
by:

                 n-1
          ^    1  --       (   2(pi)jk)
          z = --- >  z *exp(-i -------), k=0,1,...,n-1.
           k      --  j    (      n   )
              \/n j=0

                           1
(Note the scale factor of  --- in this definition.)


                          \/n

To compute the inverse discrete Fourier transform defined by:

                         n-1
```

$$\hat{w}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{} z_j \exp\left(+i \frac{2(\pi)jk}{n}\right),$$

this routine should be preceded and followed by calls of C06GCF to form the complex conjugates of the z_j and the $\hat{z}_k$.

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of n (see Section 5).

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: if X is declared with bounds (0:N-1) in the (sub) program from which C06ECF is called, then X(j) must contain x_j, the real part of z_j, for j=0,1,...,n-1. On exit: the real parts a_k of the components of the discrete Fourier transform. If X is declared with bounds (0:N-1) in the (sub) program from which C06ECF is called, then a_k is contained in X(k), for k=0,1,...,n-1.

2: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: if Y is declared with bounds (0:N-1) in the (sub) program from which C06ECF is called, then Y(j) must contain y_j, the imaginary part of z_j, for j=0,1,...,n-1. On exit: the imaginary parts b_k of the components of the discrete Fourier transform. If Y is declared with bounds (0:N-1) in the (sub)program from which C06ECF is called, then b_k is contained in Y(k), for k=0,1,...,n-1.

3: N -- INTEGER Input
On entry: the number of data values, n. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

```
 4:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     At least one of the prime factors of N is greater than 19.

IFAIL= 2
     N has more than 20 prime factors.

IFAIL= 3
     N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
n*logn, but also depends on the factorization of n. The routine
is somewhat faster than average if the only prime factors of n
are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has
several unpaired prime factors, i.e., if the 'square-free' part
of n has several factors. For such values of n, routine C06FCF(*)
(which requires an additional n real elements of workspace) is
considerably faster.

9. Example

This program reads in a sequence of complex data values and
prints their discrete Fourier transform.

It then performs an inverse transform using C06GCF and C06ECF,
and prints the sequence so obtained alongside the original data
values.
```

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
C06EKF(3NAG)      Foundation Library (12/10/92)      C06EKF(3NAG)

C06 -- Summation of Series                                    C06EKF
        C06EKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C06EKF calculates the circular convolution or correlation of two
real vectors of period n. No extra workspace is required.

2. Specification

       SUBROUTINE C06EKF (JOB, X, Y, N, IFAIL)
       INTEGER          JOB, N, IFAIL
       DOUBLE PRECISION X(N), Y(N)

3. Description

This routine computes:

if JOB =1, the discrete convolution of x and y, defined by:

                          n-1        n-1
                          --         --
                    z = >  x y   = >  x   y ;
                     k  --  j k-j  --  k-j j
                          j=0        j=0

if JOB =2, the discrete correlation of x and y defined by:

                              n-1
                              --
                         w = >  x y   .
                          k  --  j k+j
                              j=0

Here x and y are real vectors, assumed to be periodic, with
period n, i.e., x =x    =x     =...; z and w are then also
                j  j+-n  j+-2n
```

periodic with period n.

Note: this usage of the terms 'convolution' and 'correlation' is taken from Brigham [1]. The term 'convolution' is sometimes used to denote both these computations.

If x, y, z and w are the discrete Fourier transforms of these sequences,

$$\text{i.e., } \hat{x}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp\left(-i\frac{2\pi jk}{n}\right), \text{ etc,}$$

$$\text{then } z_k = \sqrt{n}\, \hat{x}_k \hat{y}_k$$

$$\text{and } w_k = \sqrt{n}\, \hat{x}_k \bar{\hat{y}}_k$$

(the bar denoting complex conjugate).

This routine calls the same auxiliary routines as C06EAF and C06EBF to compute discrete Fourier transforms, and there are some restrictions on the value of n.

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

5. Parameters

1: JOB -- INTEGER Input

On entry: the computation to be performed:

$$\text{if JOB = 1, } z_k = \sum_{j=0}^{n-1} x_j y_{k-j} \quad \text{(convolution);}$$

$$\text{if JOB = 2, } w_k = \sum_{j=0}^{n-1} x_j \bar{y}_{k-j} \quad \text{(correlation).}$$

$$\sum_{j=0}^{k} j\,k+j$$

Constraint: JOB = 1 or 2.

2: X(N) -- DOUBLE PRECISION array Input/Output
On entry: the elements of one period of the vector x. If X is declared with bounds (0:N-1) in the (sub)program from which C06EKF is called, then X(j) must contain x_j, for j=0,1,...,n-1. On exit: the corresponding elements of the discrete convolution or correlation.

3: Y(N) -- DOUBLE PRECISION array Input/Output
On entry: the elements of one period of the vector y. If Y is declared with bounds (0:N-1) in the (sub)program from which C06EKF is called, then Y(j) must contain y_j, for j=0,1,...,n-1. On exit: the discrete Fourier transform of the convolution or correlation returned in the array X; the transform is stored in Hermitian form, exactly as described in the document C06EAF.

4: N -- INTEGER Input
On entry: the number of values, n, in one period of the vectors X and Y. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

5: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of N is greater than 19.

IFAIL= 2
N has more than 20 prime factors.

IFAIL= 3
N <= 1.

IFAIL= 4

```
     JOB /= 1 or 2.

7. Accuracy

The results should be accurate to within a small multiple of the
machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to
n*logn, but also depends on the factorization of n. The routine
is faster than average if the only prime factors are 2, 3 or 5;
and fastest of all if n is a power of 2.

The routine is particularly slow if n has several unpaired prime
factors, i.e., if the 'square free' part of n has several
factors. For such values of n, routine C06FKF(*) is considerably
faster (but requires an additional workspace of n elements).

9. Example

This program reads in the elements of one period of two real
vectors x and y and prints their discrete convolution and
correlation (as computed by C06EKF). In realistic computations
the number of data values would be much larger.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
C06FPF(3NAG)      Foundation Library (12/10/92)      C06FPF(3NAG)

C06 -- Summation of Series                                  C06FPF
        C06FPF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

C06FPF computes the discrete Fourier transforms of m sequences,
each containing n real data values. This routine is designed to
be particularly efficient on vector processors.

2. Specification
```

```
      SUBROUTINE C06FPF (M, N, X, INIT, TRIG, WORK, IFAIL)
      INTEGER          M, N, IFAIL
      DOUBLE PRECISION X(M*N), TRIG(2*N), WORK(M*N)
      CHARACTER*1      INIT
```

3. Description

Given m sequences of n real data values x_j^p, for j=0,1,...,n-1; p=1,2,...,m, this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

$$\hat{z}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j^p \times \exp\left(-i\,\frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1;\ p=1,2,\ldots,m.$$

(Note the scale factor $\frac{1}{\sqrt{n}}$ in this definition.)

The transformed values $\hat{z}_k^p$ are complex, but for each value of p the $\hat{z}_k^p$ form a Hermitian sequence (i.e., $\hat{z}_{n-k}^p$ is the complex conjugate of $\hat{z}_k^p$), so they are completely determined by mn real numbers (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term:

$$\hat{z}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j^p \times \exp\left(+i\,\frac{2\pi jk}{n}\right).$$

To compute this form, this routine should be followed by a call to C06GQF to form the complex conjugates of the $\hat{z}_k^p$.

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special coding is provided for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as M, the number of transforms to be computed in parallel, increases.

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

[2] Temperton C (1983) Fast Mixed-Radix Real Fourier Transforms. J. Comput. Phys. 52 340--350.

5. Parameters

1: M -- INTEGER Input
On entry: the number of sequences to be transformed, m.
Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of real values in each sequence, n.
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the data must be stored in X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array. In other words, if the data values of the pth sequence to be transformed are denoted by x_j^p, for j=0,1,...,n-1, then the mn elements of the array X must contain the values

$$x_0^1, x_0^2, \ldots, x_0^m, x_1^1, x_1^2, \ldots, x_1^m, \ldots, x_{n-1}^1, x_{n-1}^2, \ldots, x_{n-1}^m .$$

On exit: the m discrete Fourier transforms stored as if in a two-dimensional array of dimension (1:M,0:N-1). Each of the m transforms is stored in a row of the array in Hermitian form, overwriting the corresponding original sequence. If the n components of the discrete Fourier transform $\hat{z}_k^p$ are written as $a_k^p + ib_k^p$, then for 0<=k<=n/2, a_k^p is contained in X(p,k), and for 1<=k<=(n-1)/2, b_k^p is contained in X(p,n-k). (See also Section 2.1.2 of the

```
    Chapter Introduction.)

 4: INIT -- CHARACTER*1                                          Input
    On entry: if the trigonometric coefficients required to
    compute the transforms are to be calculated by the routine
    and stored in the array TRIG, then INIT must be set equal to
    'I' (Initial call).

    If INIT contains 'S' (Subsequent call), then the routine
    assumes that trigonometric coefficients for the specified
    value of n are supplied in the array TRIG, having been
    calculated in a previous call to one of C06FPF, C06FQF or
    C06FRF.

    If INIT contains 'R' (Restart then the routine assumes that
    trigonometric coefficients for the particular value of n are
    supplied in the array TRIG, but does not check that C06FPF,
    C06FQF or C06FRF have previously been called. This option
    allows the TRIG array to be stored in an external file, read
    in and re-used without the need for a call with INIT equal
    to 'I'. The routine carries out a simple test to check that
    the current value of n is consistent with the array TRIG.
    Constraint: INIT = 'I', 'S' or 'R'.

 5: TRIG(2*N) -- DOUBLE PRECISION array              Input/Output
    On entry: if INIT = 'S' or 'R', TRIG must contain the
    required coefficients calculated in a previous call of the
    routine. Otherwise TRIG need not be set. On exit: TRIG
    contains the required coefficients (computed by the routine
    if INIT = 'I').

 6: WORK(M*N) -- DOUBLE PRECISION array                 Workspace

 7: IFAIL -- INTEGER                                 Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).


6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
```

```
          On entry M < 1.

     IFAIL= 2
          N < 1.

     IFAIL= 3
          INIT is not one of 'I', 'S' or 'R'.

     IFAIL= 4
          INIT = 'S', but none of C06FPF, C06FQF or C06FRF has
          previously been called.

     IFAIL= 5
          INIT = 'S' or 'R', but the array TRIG and the current value
          of N are inconsistent.

     7. Accuracy

     Some indication of accuracy can be obtained by performing a
     subsequent inverse transform and comparing the results with the
     original sequence (in exact arithmetic they would be identical).

     8. Further Comments

     The time taken by the routine is approximately proportional to
     nm*logn, but also depends on the factors of n. The routine is
     fastest if the only prime factors of n are 2, 3 and 5, and is
     particularly slow if n is a large prime, or has large prime
     factors.

     9. Example

     This program reads in sequences of real data values and prints
     their discrete Fourier transforms (as computed by C06FPF). The
     Fourier transforms are expanded into full complex form using
     C06GSF and printed. Inverse transforms are then calculated by
     calling C06GQF followed by C06FQF showing that the original
     sequences are restored.

     The example program is not reproduced here. The source code for
     all example programs is distributed with the NAG Foundation
     Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C06FQF(3NAG)      Foundation Library (12/10/92)      C06FQF(3NAG)

     C06 -- Summation of Series                                C06FQF
            C06FQF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06FQF computes the discrete Fourier transforms of m Hermitian sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.

2. Specification

```
      SUBROUTINE C06FQF (M, N, X, INIT, TRIG, WORK, IFAIL)
      INTEGER          M, N, IFAIL
      DOUBLE PRECISION X(M*N), TRIG(2*N), WORK(M*N)
      CHARACTER*1      INIT
```

3. Description

Given m Hermitian sequences of n complex data values z_j^p, for $j=0,1,\ldots,n-1$; $p=1,2,\ldots,m$, this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

$$\hat{x}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j^p \times \exp\left(-i\,\frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1;\ p=1,2,\ldots,m.$$

(Note the scale factor $\frac{1}{\sqrt{n}}$ in this definition.)

The transformed values are purely real (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term

$$\hat{x}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j^p \times \exp\left(+i\,\frac{2\pi jk}{n}\right).$$

To compute this form, this routine should be preceded by a call
to C06GQF to form the complex conjugates of the $\hat{z}_j^p$.

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special code is included for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as m, the number of transforms to be computed in parallel, increases.

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

[2] Temperton C (1983) Fast Mixed-Radix Real Fourier Transforms. J. Comput. Phys. 52 340--350.

5. Parameters

1: M -- INTEGER Input
On entry: the number of sequences to be transformed, m.
Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of data values in each sequence, n.
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the data must be stored in X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array in Hermitian form.
If the n data values z_j^p are written as $x_j^p + iy_j^p$, then for $0 \le j \le n/2$, x_j^p is contained in X(p,j), and for $1 \le j \le (n-1)/2$, y_j^p is contained in X(p,n-j). (See also Section 2.1.2 of the Chapter Introduction.) On exit: the components of the m discrete Fourier transforms, stored as if in a two-dimensional array of dimension (1:M,0:N-1). Each of the m transforms is stored as a row of the array, overwriting the corresponding original sequence. If the n components of the

discrete Fourier transform are denoted by $\hat{x}_k^p$, for $k=0,1,\ldots,n-1$, then the mn elements of the array X contain the values

$$\hat{x}_0^1\ \hat{x}_0^2\ \ldots\ \hat{x}_0^m\ \ \hat{x}_1^1\ \hat{x}_1^2\ \ldots\ \hat{x}_1^m\ \ \ldots\ \ \hat{x}_{n-1}^1\ \hat{x}_{n-1}^2\ \ldots\ \hat{x}_{n-1}^m .$$

4: INIT -- CHARACTER*1 Input

On entry: if the trigonometric coefficients required to compute the transforms are to be calculated by the routine and stored in the array TRIG, then INIT must be set equal to 'I' (Initial call).

If INIT contains 'S' (Subsequent call), then the routine assumes that trigonometric coefficients for the specified value of n are supplied in the array TRIG, having been calculated in a previous call to one of C06FPF, C06FQF or C06FRF.

If INIT contains 'R' (Restart), then the routine assumes that trigonometric coefficients for the particular value of N are supplied in the array TRIG, but does not check that C06FPF, C06FQF or C06FRF have previously been called. This option allows the TRIG array to be stored in an external file, read in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is compatible with the array TRIG. Constraint: INIT = 'I', 'S' or 'R'.

5: TRIG(2*N) -- DOUBLE PRECISION array Input/Output

On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

6: WORK(M*N) -- DOUBLE PRECISION array Workspace

7: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

```
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 1.

IFAIL= 2
     On entry N < 1.

IFAIL= 3
     On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
     On entry INIT = 'S', but none of C06FPF, C06FQF and C06FRF
     has previously been called.

IFAIL= 5
     On entry INIT = 'S' or 'R', but the array TRIG and the
     current value of n are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
nm*logn, but also depends on the factors of n. The routine is
fastest if the only prime factors of n are 2, 3 and 5, and is
particularly slow if n is a large prime, or has large prime
factors.

9. Example

This program reads in sequences of real data values which are
assumed to be Hermitian sequences of complex data stored in
Hermitian form. The sequences are expanded into full complex form
using C06GSF and printed. The discrete Fourier transforms are
then computed (using C06FQF) and printed out. Inverse transforms
are then calculated by calling C06FPF followed by C06GQF showing
that the original sequences are restored.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

C06FRF(3NAG) Foundation Library (12/10/92) C06FRF(3NAG)

C06 -- Summation of Series C06FRF
C06FRF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06FRF computes the discrete Fourier transforms of m sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.

2. Specification

```
SUBROUTINE C06FRF (M, N, X, Y, INIT, TRIG, WORK, IFAIL)
INTEGER          M, N, IFAIL
DOUBLE PRECISION X(M*N), Y(M*N), TRIG(2*N), WORK(2*M*N)
CHARACTER*1      INIT
```

3. Description

Given m sequences of n complex data values z_j^p, for $j=0,1,\ldots,n-1$; $p=1,2,\ldots,m$, this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

$$\hat{z}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j^p \exp\left(-i\frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1;\ p=1,2,\ldots,m.$$

(Note the scale factor $\frac{1}{\sqrt{n}}$ in this definition.)

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term

$$\hat{z}_k^p = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j^p \left(\frac{2\pi jk}{n}\right)$$

```
                z = --- >  z *exp(+i -------).
                 k      --  j    (      n   )
                    \/n j=0
```

To compute this form, this routine should be preceded and followed by a call of C06GCF to form the complex conjugates of z_j^p and the $\hat{z}_k^p$.

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special code is provided for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as m, the number of transforms to be computed in parallel, increases.

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

[2] Temperton C (1983) Self-sorting Mixed-radix Fast Fourier Transforms. J. Comput. Phys. 52 1--23.

5. Parameters

1: M -- INTEGER Input
On entry: the number of sequences to be transformed, m.
Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of complex values in each sequence, n.
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output

4: Y(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the real and imaginary parts of the complex data must be stored in X and Y respectively as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of each array. In other words, if the real parts of the pth sequence to be transformed are denoted by x_j^p, for j=0,1,...,n-1, then the mn elements of the array X must contain the values

$$x_0^1, x_0^2, \ldots, x_0^m, x_1^1, x_1^2, \ldots, x_1^m, \ldots, x_{n-1}^1, x_{n-1}^2, \ldots, x_{n-1}^m .$$

```
     0  0      0   1  1      1       n-1  n-1      n-1
On exit: X and Y are overwritten by the real and imaginary
parts of the complex transforms.

5:  INIT -- CHARACTER*1                                        Input
On entry: if the trigonometric coefficients required to
compute the transforms are to be calculated by the routine
and stored in the array TRIG, then INIT must be set equal to
'I' (Initial call).

If INIT contains 'S' (Subsequent call), then the routine
assumes that trigonometric coefficients for the specified
value of n are supplied in the array TRIG, having been
calculated in a previous call to one of C06FPF, C06FQF or
C06FRF.

If INIT contains 'R' (Restart) then the routine assumes that
trigonometric coefficients for the particular value of n are
supplied in the array TRIG, but does not check that C06FPF,
C06FQF or C06FRF have previously been called. This option
allows the TRIG array to be stored in an external file, read
in and re-used without the need for a call with INIT equal
to 'I'. The routine carries out a simple test to check that
the current value of n is compatible with the array TRIG.
Constraint: INIT = 'I', 'S' or 'R'.

6:  TRIG(2*N) -- DOUBLE PRECISION array                Input/Output
On entry: if INIT = 'S' or 'R', TRIG must contain the
required coefficients calculated in a previous call of the
routine. Otherwise TRIG need not be set. On exit: TRIG
contains the required coefficients (computed by the routine
if INIT = 'I').

7:  WORK(2*M*N) -- DOUBLE PRECISION array                  Workspace

8:  IFAIL -- INTEGER                                       Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).
```

```
IFAIL= 1
     On entry M < 1.

IFAIL= 2
     On entry N < 1.

IFAIL= 3
     On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
     On entry INIT = 'S', but none of C06FPF, C06FQF and C06FRF
     has previously been called.

IFAIL= 5
     On entry INIT = 'S' or 'R', but the array TRIG and the
     current value of n are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
nm*logn, but also depends on the factors of n. The routine is
fastest if the only prime factors of n are 2, 3 and 5, and is
particularly slow if n is a large prime, or has large prime
factors.

9. Example

This program reads in sequences of complex data values and prints
their discrete Fourier transforms (as computed by C06FRF).
Inverse transforms are then calculated using C06GCF and C06FRF
and printed out, showing that the original sequences are
restored.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
C06FUF(3NAG)      Foundation Library (12/10/92)      C06FUF(3NAG)

C06 -- Summation of Series                                  C06FUF
       C06FUF -- NAG Foundation Library Routine Document
```

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06FUF computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values. This routine is designed to be particularly efficient on vector processors.

2. Specification

```
      SUBROUTINE C06FUF (M, N, X, Y, INIT, TRIGM, TRIGN, WORK,
     1                   IFAIL)
      INTEGER           M, N, IFAIL
      DOUBLE PRECISION X(M*N), Y(M*N), TRIGM(2*M), TRIGN(2*N),
     1                 WORK(2*M*N)
      CHARACTER*1       INIT
```

3. Description

This routine computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values $z_{j_1 j_2}$, where $j_1 = 0,1,\ldots,m-1$, $j_2 = 0,1,\ldots,n-1$.

The discrete Fourier transform is here defined by:

$$\hat{z}_{k_1 k_2} = \frac{1}{\sqrt{mn}} \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} z_{j_1 j_2} \exp\left(-2\pi i \left(\frac{j_1 k_1}{m} + \frac{j_2 k_2}{n}\right)\right),$$

where $k_1 = 0,1,\ldots,m-1$, $k_2 = 0,1,\ldots,n-1$.

(Note the scale factor of $\frac{1}{\sqrt{mn}}$ in this definition.)

To compute the inverse discrete Fourier transform, defined with $\exp(+2\pi i(\ldots))$ in the above formula instead of $\exp(-2\pi i(\ldots))$, this routine should be preceded and followed by

calls of C06GCF to form the complex conjugates of the data values and the transform.

This routine calls C06FRF to perform multiple one-dimensional discrete Fourier transforms by the fast Fourier transform (FFT) algorithm in Brigham [1]. It is designed to be particularly efficient on vector processors.

4. References

[1] Brigham E O (1973) The Fast Fourier Transform. Prentice-Hall.

[2] Temperton C (1983) Self-sorting Mixed-radix Fast Fourier Transforms. J. Comput. Phys. 52 1--23.

5. Parameters

1: M -- INTEGER Input
On entry: the number of rows, m, of the arrays X and Y.
Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of columns, n, of the arrays X and Y.
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output

4: Y(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the real and imaginary parts of the complex data values must be stored in arrays X and Y respectively. If X and Y are regarded as two-dimensional arrays of dimension (0:M-1,0:N-1), then X(j ,j) and Y(j ,j) must contain the
1 2 1 2
real and imaginary parts of $z_{j_1 j_2}$. On exit: the real and imaginary parts respectively of the corresponding elements of the computed transform.

5: INIT -- CHARACTER*1 Input
On entry: if the trigonometric coefficients required to compute the transforms are to be calculated by the routine and stored in the arrays TRIGM and TRIGN, then INIT must be set equal to 'I', (Initial call).

If INIT contains 'S', (Subsequent call), then the routine assumes that trigonometric coefficients for the specified values of m and n are supplied in the arrays TRIGM and TRIGN, having been calculated in a previous call to the

```
     routine.

     If INIT contains 'R', (Restart), then the routine assumes
     that trigonometric coefficients for the particular values of
     m and n are supplied in the arrays TRIGM and TRIGN, but does
     not check that the routine has previously been called. This
     option allows the TRIGM and TRIGN arrays to be stored in an
     external file, read in and re-used without the need for a
     call with INIT equal to 'I'. The routine carries out a
     simple test to check that the current values of m and n are
     compatible with the arrays TRIGM and TRIGN. Constraint: INIT
     = 'I', 'S' or 'R'.

 6:  TRIGM(2*M) -- DOUBLE PRECISION array            Input/Output

 7:  TRIGN(2*N) -- DOUBLE PRECISION array            Input/Output
     On entry: if INIT = 'S' or 'R',TRIGM and TRIGN must contain
     the required coefficients calculated in a previous call of
     the routine. Otherwise TRIGM and TRIGN need not be set.

     If m=n the same array may be supplied for TRIGM and TRIGN.
     On exit: TRIGM and TRIGN contain the required coefficients
     (computed by the routine if INIT = 'I').

 8:  WORK(2*M*N) -- DOUBLE PRECISION array              Workspace

 9:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 1.

IFAIL= 2
     On entry N < 1.

IFAIL= 3
     On entry INIT is not one of 'I', 'S' or 'R'.
```

```
IFAIL= 4
     On entry INIT = 'S', but C06FUF has not previously been
     called.

IFAIL= 5
     On entry INIT = 'S' or 'R', but at least one of the arrays
     TRIGM and TRIGN is inconsistent with the current value of M
     or N.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the
original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to
mn*log(mn), but also depends on the factorization of the
individual dimensions m and n. The routine is somewhat faster
than average if their only prime factors are 2, 3 or 5; and
fastest of all if they are powers of 2.

9. Example

This program reads in a bivariate sequence of complex data values
and prints the two-dimensional Fourier transform. It then
performs an inverse transform and prints the sequence so
obtained, which may be compared to the original data values.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
C06GBF(3NAG)      Foundation Library (12/10/92)      C06GBF(3NAG)

     C06 -- Summation of Series                              C06GBF
             C06GBF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     C06GBF forms the complex conjugate of a Hermitian sequence of n
     data values.
```

2. Specification

```
SUBROUTINE C06GBF (X, N, IFAIL)
INTEGER          N, IFAIL
DOUBLE PRECISION X(N)
```

3. Description

This is a utility routine for use in conjunction with C06EAF, C06EBF, C06FAF(*) or C06FBF(*) to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: if the data values z_j are written as $x_j + iy_j$ and if X is declared with bounds (0:N-1) in the (sub)program from which C06GBF is called, then for $0 \le j \le n/2$, X(j) must contain x_j (=x_j), while for $n/2 < j \le n-1$, X(j) must contain $-y_{n-j}$ (=y_{n-j}). In other words, X must contain the Hermitian sequence in Hermitian form. (See also Section 2.1.2 of the Chapter Introduction). On exit: the imaginary parts y_j are negated. The real parts x_j are not referenced.

2: N -- INTEGER Input
On entry: the number of data values, n. Constraint: N >= 1.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1

```
     N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example

This program reads in a sequence of real data values, calls
C06EAF followed by C06GBF to compute their inverse discrete
Fourier transform, and prints this after expanding it from
Hermitian form into a full complex sequence.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
C06GCF(3NAG)      Foundation Library (12/10/92)      C06GCF(3NAG)

     C06 -- Summation of Series                              C06GCF
            C06GCF -- NAG Foundation Library Routine Document

     Note: Before using this routine, please read the Users' Note for
     your implementation to check implementation-dependent details.
     The symbol (*) after a NAG routine name denotes a routine that is
     not included in the Foundation Library.

     1. Purpose

     C06GCF forms the complex conjugate of a sequence of n data
     values.

     2. Specification

            SUBROUTINE C06GCF (Y, N, IFAIL)
            INTEGER          N, IFAIL
            DOUBLE PRECISION Y(N)

     3. Description

     This is a utility routine for use in conjunction with C06ECF or
     C06FCF(*) to calculate inverse discrete Fourier transforms (see
     the Chapter Introduction).
```

```
4. References

None.

5. Parameters

 1:  Y(N) -- DOUBLE PRECISION array                  Input/Output
     On entry: if Y is declared with bounds (0:N-1) in the (sub)
     program which C06GCF is called, then Y(j) must contain the
     imaginary part of the jth data value, for 0<=j<=n-1. On
     exit: these values are negated.

 2:  N -- INTEGER                                            Input
     On entry: the number of data values, n. Constraint: N >= 1.

 3:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example

This program reads in a sequence of complex data values and
prints their inverse discrete Fourier transform as computed by
calling C06GCF, followed by C06ECF and C06GCF again.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

C06GQF(3NAG) Foundation Library (12/10/92) C06GQF(3NAG)

C06 -- Summation of Series C06GQF
C06GQF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06GQF forms the complex conjugates of m Hermitian sequences, each containing n data values.

2. Specification

```
SUBROUTINE C06GQF (M, N, X, IFAIL)
INTEGER          M, N, IFAIL
DOUBLE PRECISION X(M*N)
```

3. Description

This is a utility routine for use in conjunction with C06FPF and C06FQF to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: M -- INTEGER Input
On entry: the number of Hermitian sequences to be conjugated, m. Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of data values in each Hermitian sequence, n. Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the data must be stored in array X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array in Hermitian form. If the n data values z_j^p are written as $x_j^p + iy_j^p$, then for $0 \le j \le n/2$, x_j^p is contained in X(p,j), and for 1<=j<=(n-

```
                    j
          p
     1)/2, y  is contained in X(p,n-j). (See also Section 2.1.2
            j
     of the Chapter Introduction.) On exit: the imaginary parts
      p                               p
     y  are negated. The real parts x  are not referenced.
      j                               j

 4:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 1.

IFAIL= 2
     On entry N < 1.

7. Accuracy

Exact.

8. Further Comments

None.

9. Example

This program reads in sequences of real data values which are
assumed to be Hermitian sequences of complex data stored in
Hermitian form. The sequences are expanded into full complex form
using C06GSF and printed. The sequences are then conjugated
(using C06GQF) and the conjugated sequences are expanded into
complex form using C06GSF and printed out.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

     C06GSF(3NAG)      Foundation Library (12/10/92)      C06GSF(3NAG)

          C06 -- Summation of Series                                C06GSF
                  C06GSF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          C06GSF takes m Hermitian sequences, each containing n data
          values, and forms the real and imaginary parts of the m
          corresponding complex sequences.

          2. Specification

                 SUBROUTINE C06GSF (M, N, X, U, V, IFAIL)
                 INTEGER          M, N, IFAIL
                 DOUBLE PRECISION X(M*N), U(M*N), V(M*N)

          3. Description

          This is a utility routine for use in conjunction with C06FPF and
          C06FQF (see the Chapter Introduction).

          4. References

          None.

          5. Parameters

           1:  M -- INTEGER                                          Input
               On entry: the number of Hermitian sequences, m, to be
               converted into complex form. Constraint: M >= 1.

           2:  N -- INTEGER                                          Input
               On entry: the number of data values, n, in each sequence.
               Constraint: N >= 1.

           3:  X(M,N) -- DOUBLE PRECISION array                      Input
               On entry: the data must be stored in X as if in a two-
               dimensional array of dimension (1:M,0:N-1); each of the m
               sequences is stored in a row of the array in Hermitian form.
                                    p                p   p
               If the n data values z  are written as x +iy , then for
```

```
                          j                 j   j
              p
0<=j<=n/2, x  is contained in X(p,j), and for 1<=j<=(n-1)/2,
              j
 p
y  is contained in X(p,n-j). (See also Section 2.1.2 of the
 j
Chapter Introduction.)

 4:  U(M,N) -- DOUBLE PRECISION array                        Output

 5:  V(M,N) -- DOUBLE PRECISION array                        Output
     On exit: the real and imaginary parts of the m sequences of
     length n, are stored in U and V respectively, as if in two-
     dimensional arrays of dimension (1:M,0:N-1); each of the m
     sequences is stored as if in a row of each array. In other
     words, if the real parts of the pth sequence are denoted by
      p
     x , for j=0,1,...,n-1 then the mn elements of the array U
      j
     contain the values
           1  2      m   1  2      m       1    2        m
          x ,x ,...,x , x ,x ,...,x ,..., x   ,x   ,...,x
           0  0      0   1  1      1       n-1  n-1      n-1

 6:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 1.

IFAIL= 2
     On entry N < 1.

7. Accuracy

Exact.
```

```
8. Further Comments

None.

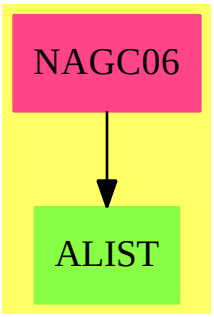
9. Example

This program reads in sequences of real data values which are
assumed to be Hermitian sequences of complex data stored in
Hermitian form. The sequences are then expanded into full complex
form using C06GSF and printed.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

———-

15.14.1 NagSeriesSummationPackage (NAGC06)



Exports:

c06eaf	c06ebf	c06ecf	c06ekf	c06fpf
c06fqf	c06frf	c06fuf	c06gbf	c06gcf
c06gqf	c06gsf			

— package NAGC06 NagSeriesSummationPackage —

```
)abbrev package NAGC06 NagSeriesSummationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:30 1994
++ Description:
++ This package uses the NAG Library to calculate the discrete Fourier
++ transform of a sequence of real or complex data values, and
++ applies it to calculate convolutions and correlations.
```

```
NagSeriesSummationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    c06eaf : (Integer,Matrix DoubleFloat,Integer) -> Result
     ++ c06eaf(n,x,ifail)
     ++ calculates the discrete Fourier transform of a sequence of
     ++ n real data values. (No extra workspace required.)
     ++ See \downlink{Manual Page}{manpageXXc06eaf}.
    c06ebf : (Integer,Matrix DoubleFloat,Integer) -> Result
     ++ c06ebf(n,x,ifail)
     ++ calculates the discrete Fourier transform of a Hermitian
     ++ sequence of n complex data values. (No extra workspace required.)
     ++ See \downlink{Manual Page}{manpageXXc06ebf}.
    c06ecf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
     ++ c06ecf(n,x,y,ifail)
     ++ calculates the discrete Fourier transform of a sequence of
     ++ n complex data values. (No extra workspace required.)
     ++ See \downlink{Manual Page}{manpageXXc06ecf}.
    c06ekf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
        Integer) -> Result
     ++ c06ekf(job,n,x,y,ifail)
     ++ calculates the circular convolution of two
     ++ real vectors of period n. No extra workspace is required.
     ++ See \downlink{Manual Page}{manpageXXc06ekf}.
    c06fpf : (Integer,Integer,String,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
     ++ c06fpf(m,n,init,x,trig,ifail)
     ++ computes the discrete Fourier transforms of m sequences,
     ++ each containing n real data values. This routine is designed to
     ++ be particularly efficient on vector processors.
     ++ See \downlink{Manual Page}{manpageXXc06fpf}.
    c06fqf : (Integer,Integer,String,Matrix DoubleFloat,_
        Matrix DoubleFloat,Integer) -> Result
     ++ c06fqf(m,n,init,x,trig,ifail)
     ++ computes the discrete Fourier transforms of m Hermitian
     ++ sequences, each containing n complex data values. This routine is
     ++ designed to be particularly efficient on vector processors.
     ++ See \downlink{Manual Page}{manpageXXc06fqf}.
    c06frf : (Integer,Integer,String,Matrix DoubleFloat,_
        Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
     ++ c06frf(m,n,init,x,y,trig,ifail)
     ++ computes the discrete Fourier transforms of m sequences,
     ++ each containing n complex data values. This routine is designed
     ++ to be particularly efficient on vector processors.
     ++ See \downlink{Manual Page}{manpageXXc06frf}.
    c06fuf : (Integer,Integer,String,Matrix DoubleFloat,_
        Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
```

```
      Integer) -> Result
    ++ c06fuf(m,n,init,x,y,trigm,trign,ifail)
    ++ computes the two-dimensional discrete Fourier transform of
    ++ a bivariate sequence of complex data values. This routine is
    ++ designed to be particularly efficient on vector processors.
    ++ See \downlink{Manual Page}{manpageXXc06fuf}.
  c06gbf : (Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06gbf(n,x,ifail)
    ++ forms the complex conjugate of n
    ++ data values.
    ++ See \downlink{Manual Page}{manpageXXc06gbf}.
  c06gcf : (Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06gcf(n,y,ifail)
    ++ forms the complex conjugate of a sequence of n data
    ++ values.
    ++ See \downlink{Manual Page}{manpageXXc06gcf}.
  c06gqf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06gqf(m,n,x,ifail)
    ++ forms the complex conjugates,
    ++ each containing n data values.
    ++ See \downlink{Manual Page}{manpageXXc06gqf}.
  c06gsf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06gsf(m,n,x,ifail)
    ++ takes m Hermitian sequences, each containing n data
    ++ values, and forms the real and imaginary parts of the m
    ++ corresponding complex sequences.
    ++ See \downlink{Manual Page}{manpageXXc06gsf}.
Implementation ==> add

  import Lisp
  import DoubleFloat
  import Any
  import Record
  import Integer
  import Matrix DoubleFloat
  import Boolean
  import NAGLinkSupportPackage
  import AnyFunctions1(Integer)
  import AnyFunctions1(String)
  import AnyFunctions1(Matrix DoubleFloat)

  c06eaf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
      [(invokeNagman(NIL$Lisp,_
      "c06eaf",_
      ["n"::S,"ifail"::S,"x"::S]$Lisp,_
      []$Lisp,_
      [["double"::S,["x"::S,"n"::S]$Lisp]$Lisp_
      ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
      ]$Lisp,_
      ["x"::S,"ifail"::S]$Lisp,_
```

```
    [([nArg::Any,ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result


c06ebf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "c06ebf",_
    ["n"::S,"ifail"::S,"x"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,["x"::S,"n"::S]$Lisp]$Lisp_
    ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["x"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,ifailArg::Any,xArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

c06ecf(nArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "c06ecf",_
    ["n"::S,"ifail"::S,"x"::S,"y"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,["x"::S,"n"::S]$Lisp,["y"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["x"::S,"y"::S,"ifail"::S]$Lisp,_
    [([nArg::Any,ifailArg::Any,xArg::Any,yArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

c06ekf(jobArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
    yArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "c06ekf",_
    ["job"::S,"n"::S,"ifail"::S,"x"::S,"y"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,["x"::S,"n"::S]$Lisp,["y"::S,"n"::S]$Lisp_
    ]$Lisp_
    ,["integer"::S,"job"::S,"n"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["x"::S,"y"::S,"ifail"::S]$Lisp,_
    [([jobArg::Any,nArg::Any,ifailArg::Any,xArg::Any,yArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

c06fpf(mArg:Integer,nArg:Integer,initArg:String,_
    xArg:Matrix DoubleFloat,trigArg:Matrix DoubleFloat,_
```

```
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06fpf",_
        ["m"::S,"n"::S,"init"::S,"ifail"::S,"x"::S,"trig"::S,"work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ,["trig"::S,["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,_
        ["work"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
        ,["character"::S,"init"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"trig"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
        xArg::Any,trigArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06fqf(mArg:Integer,nArg:Integer,initArg:String,_
        xArg:Matrix DoubleFloat,trigArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06fqf",_
        ["m"::S,"n"::S,"init"::S,"ifail"::S,"x"::S,"trig"::S,"work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ,["trig"::S,["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,_
        ["work"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
        ,["character"::S,"init"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"trig"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
        trigArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06frf(mArg:Integer,nArg:Integer,initArg:String,_
        xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        trigArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06frf",_
        ["m"::S,"n"::S,"init"::S,"ifail"::S,"x"::S,"y"::S,"trig"::S,_
        "work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ,["y"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,["trig"::S,_
        ["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,["work"::S,_
        ["*"::S,["*"::S,2$Lisp,"m"::S]$Lisp,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
```

```
        ,["character"::S,"init"::S]$Lisp]$Lisp,_
        ["x"::S,"y"::S,"trig"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
        xArg::Any,yArg::Any,trigArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06fuf(mArg:Integer,nArg:Integer,initArg:String,_
        xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
        trigmArg:Matrix DoubleFloat,_
        trignArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06fuf",_
        ["m"::S,"n"::S,"init"::S,"ifail"::S,"x"::S,"y"::S,"trigm"::S,_
        "trign"::S,"work"::S]$Lisp,_
        ["work"::S]$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ,["y"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,["trigm"::S,_
        ["*"::S,2$Lisp,"m"::S]$Lisp]$Lisp,["trign"::S,_
        ["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp_
        ,["work"::S,["*"::S,["*"::S,2$Lisp,"m"::S]$Lisp,_
        "n"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
        ,["character"::S,"init"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"y"::S,"trigm"::S,"trign"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
        yArg::Any,trigmArg::Any,trignArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06gbf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06gbf",_
        ["n"::S,"ifail"::S,"x"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,["x"::S,"n"::S]$Lisp]$Lisp_
        ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,ifailArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06gcf(nArg:Integer,yArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06gcf",_
        ["n"::S,"ifail"::S,"y"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,["y"::S,"n"::S]$Lisp]$Lisp_
```

```
        ,["integer"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["y"::S,"ifail"::S]$Lisp,_
        [([nArg::Any,ifailArg::Any,yArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06gqf(mArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06gqf",_
        ["m"::S,"n"::S,"ifail"::S,"x"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["x"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,ifailArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    c06gsf(mArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "c06gsf",_
        ["m"::S,"n"::S,"ifail"::S,"x"::S,"u"::S,"v"::S]$Lisp,_
        ["u"::S,"v"::S]$Lisp,_
        [["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp_
        ,["u"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,_
        ["v"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["u"::S,"v"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,nArg::Any,ifailArg::Any,xArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

— **NAGC06.dotabb** —

```
"NAGC06" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGC06"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGC06" -> "ALIST"
```

———-

15.15 package NAGS NagSpecialFunctionsPackage

— NagSpecialFunctionsPackage.help —

```
S -- Approximations of Special Functions        Introduction -- S
                            Chapter S
                Approximations of Special Functions

1. Scope of the Chapter

This chapter is concerned with the provision of some commonly
occurring physical and mathematical functions.

2. Background to the Problems

The majority of the routines in this chapter approximate real-
valued functions of a single real argument, and the techniques
involved are described in Section 2.1. In addition the chapter
contains routines for elliptic integrals (see Section 2.2),
Bessel and Airy functions of a complex argument (see Section 2.3)
, and exponential of a complex argument.

2.1. Functions of a Single Real Argument

Most of the routines for functions of a single real argument have
been based on truncated Chebyshev expansions. This method of
approximation was adopted as a compromise between the conflicting
requirements of efficiency and ease of implementation on many
different machine ranges. For details of the reasons behind this
choice and the production and testing procedures followed in
constructing this chapter see Schonfelder [7].

Basically if the function to be approximated is f(x), then for
xis in  [a,b] an approximation of the form


                              --'
                   f(x)=g(x) >  C T (t)
                              --  r r
                              r=0

            --'
is used, ( >  denotes, according to the usual convention, a
            --
summation in which the first term is halved), where g(x) is some
suitable auxiliary function which extracts any singularities,
asymptotes and, if possible, zeros of the function in the range
in question and t=t(x) is a mapping of the general range [a,b] to
```

the specific range [-1,+1] required by the Chebyshev polynomials, $T_r(t)$. For a detailed description of the properties of the Chebyshev polynomials see Clenshaw [5] and Fox and Parker [6].

The essential property of these polynomials for the purposes of function approximation is that $T_n(t)$ oscillates between +-1 and it takes its extreme values n+1 times in the interval [-1,+1]. Therefore, provided the coefficients C_r decrease in magnitude sufficiently rapidly the error made by truncating the Chebyshev expansion after n terms is approximately given by

$$E(t)\tilde{=}C_n T_n(t)$$

That is the error oscillates between +-C_n and takes its extreme value n+1 times in the interval in question. Now this is just the condition that the approximation be a mini-max representation, one which minimizes the maximum error. By suitable choice of the interval, [a,b], the auxiliary function, g(x), and the mapping of the independent variable, t(x), it is almost always possible to obtain a Chebyshev expansion with rapid convergence and hence truncations that provide near mini-max polynomial approximations to the required function. The difference between the true mini-max polynomial and the truncated Chebyshev expansion is seldom sufficiently great to be of significance.

The evaluation of the Chebyshev expansions follows one of two methods. The first and most efficient, and hence most commonly used, works with the equivalent simple polynomial. The second method, which is used on the few occasions when the first method proves to be unstable, is based directly on the truncated Chebyshev series and uses backward recursion to evaluate the sum. For the first method, a suitably truncated Chebyshev expansion (truncation is chosen so that the error is less than the machine precision) is converted to the equivalent simple polynomial. That is we evaluate the set of coefficients b_r such that

$$y(t)= \sum_{r=0}^{n-1} b_r t^r = \sum_{r=0}^{n-1}{}' C_r T_r(t).$$

The polynomial can then be evaluated by the efficient Horner's

method of nested multiplications,

$$y(t)=(b_0+t(b_1+t(b_2+\ldots t(b_{n-2}+tb_{n-1}))))\ldots).$$

This method of evaluation results in efficient routines but for some expansions there is considerable loss of accuracy due to cancellation effects. In these cases the second method is used. It is well known that if

$$b_{n-1}=C_{n-1}$$

$$b_{n-2}=2tb_{n-1}+C_{n-2}$$

$$b_j=2tb_{j+1}-b_{j+2}+C_j,\quad j=n-3,n-4,\ldots,0$$

then

$$\sum_{r=0}^{\infty} C_r T_r(t)=-(b_0-b_2)$$

and this is always stable. This method is most efficiently implemented by using three variables cyclically and explicitly constructing the recursion.

That is,

$$\begin{aligned}
(\alpha) &= C_{n-1}\\
(\beta) &= 2t(\alpha)+C_{n-2}\\
(\gamma) &= 2t(\beta)-(\alpha)+C_{n-3}\\
(\alpha) &= 2t(\gamma)-(\beta)+C_{n-4}\\
(\beta) &= 2t(\alpha)-(\gamma)+C_{n-5}\\
&\ldots\\
&\ldots\\
(\alpha) &= 2t(\gamma)-(\beta)+C_2 \quad \text{(say)}\\
(\beta) &= 2t(\alpha)-(\gamma)+C_1\\
y(t) &= t(\beta)-(\alpha)+\tfrac{1}{2}C_0
\end{aligned}$$

2 0

The auxiliary functions used are normally functions compounded of simple polynomial (usually linear) factors extracting zeros, and the primary compiler-provided functions, sin, cos, ln, exp, sqrt, which extract singularities and/or asymptotes or in some cases basic oscillatory behaviour, leaving a smooth well-behaved function to be approximated by the Chebyshev expansion which can therefore be rapidly convergent.

The mappings of [a,b] to [-1,+1] used, range from simple linear mappings to the case when b is infinite and considerable improvement in convergence can be obtained by use of a bilinear form of mapping. Another common form of mapping is used when the function is even, that is it involves only even powers in its expansion. In this case an approximation over the whole interval

$$[-a,a] \text{ can be provided using a mapping } t=2\left(\frac{x}{a}\right)^2-1.$$

This embodies the evenness property but the expansion in t involves all powers and hence removes the necessity of working with an expansion with half its coefficients zero.

For many of the routines an analysis of the error in principle is given, viz, if E and (nabla) are the absolute errors in function and argument and (epsilon) and (delta) are the corresponding relative errors, then

$$E\tilde{}=|f'(x)|(\nabla)$$

$$E\tilde{}=|xf'(x)|(\delta)$$

$$\varepsilon\tilde{}=\left|\frac{xf'(x)}{f(x)}\right|(\delta)$$

If we ignore errors that arise in the argument of the function by propagation of data errors etc and consider only those errors that result from the fact that a real number is being represented in the computer in floating-point form with finite precision, then (delta) is bounded and this bound is independent of the magnitude of x; e.g. on an 11-digit machine

$$|(\delta)|\leq 10^{-11}.$$

(This of course implies that the absolute error (nabla)=x(delta) is also bounded but the bound is now dependent on x). However because of this the last two relations above are probably of more interest. If possible the relative error propagation is

discussed; that is the behaviour of the error amplification factor |xf'(x)/f(x)| is described, but in some cases, such as near zeros of the function which cannot be extracted explicitly, absolute error in the result is the quantity of significance and here the factor |xf'(x)| is described. In general, testing of the functions has shown that their error behaviour follows fairly well these theoretical error behaviours. In regions, where the error amplification factors are less than or of the order of one, the errors are slightly larger than the above predictions. The errors are here limited largely by the finite precision of arithmetic in the machine but (epsilon) is normally no more than a few times greater than the bound on (delta). In regions where the amplification factors are large, order of ten or greater, the theoretical analysis gives a good measure of the accuracy obtainable.

It should be noted that the definitions and notations used for the functions in this chapter are all taken from Abramowitz and Stegun [1]. Users are strongly recommended to consult this book for details before using the routines in this chapter.

2.2. Approximations to Elliptic Integrals

The functions provided here are symmetrised variants of the classic elliptic integrals. These alternative definitions have been suggested by Carlson (see [2], [3] and [4]) and he also developed the basic algorithms used in this chapter.

The standard integral of the first kind is represented by

$$R_F(x,y,z) = \frac{1}{2}\int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}$$

where x,y,z>=0 and at most one may be equal to zero.

The normalisation factor, $\frac{1}{2}$, is chosen so as to make

$$R_F(x,x,x)=1/\sqrt{x}.$$

If any two of the variables are equal, R_F degenerates into the

second function

$$R_C(x,y)=R_F(x,y,y)=\frac{1}{2}\int_0^{\infty}\frac{dt}{\sqrt{t+x}(t+y)}$$

where the argument restrictions are now x>=0 and y/=0.

This function is related to the logarithm or inverse hyperbolic functions if 0<y<x, and to the inverse circular functions if 0<=x<=y.

The integrals of the second kind are defined by

$$R_D(x,y,z)=\frac{3}{2}\int_0^{\infty}\frac{dt}{(t+x)(t+y)(t+z)^3}$$

with z>0, x>=0 and y>=0 but only one of x or y may be zero.

The function is a degenerate special case of the integral of the third kind

$$R_J(x,y,z,(\rho))=\frac{3}{2}\int_0^{\infty}\frac{dt}{(t+x)(t+y)(t+z)(t+\rho)}$$

with (rho)/=0, x,y,z>=0 with at most one equality holding. Thus $R_D(x,y,z)=R_J(x,y,z,z)$. The normalisation of both these functions is chosen so that

$$R_D(x,x,x)=R_J(x,x,x,x)=1/(x\sqrt{x})$$

The algorithms used for all these functions are based on duplication theorems. These allow a recursion system to be established which constructs a new set of arguments from the old using a combination of arithmetic and geometric means. The value of the function at the original arguments can then be simply

related to the value at the new arguments. These recursive reductions are used until the arguments differ from the mean by an amount small enough for a Taylor series about the mean to give sufficient accuracy when retaining terms of order less than six. Each step of the recurrences reduces the difference from the mean by a factor of four, and as the truncation error is of order six, the truncation error goes like $(4096)^{-n}$, where n is the number of iterations.

The above forms can be related to the more traditional canonical forms (see Abramowitz and Stegun [1], 17.2).

If we write $q=\cos^2(\phi)$, $r=1-m\sin^2(\phi)$, $s=1+n\sin^2(\phi)$, where $0<(\phi)\leq \frac{1}{2}\pi$, we have: the elliptic integral of the first kind:

$$F((\phi)|m) = \int_0^{\sin(\phi)} (1-t^2)^{-1/2} (1-mt^2)^{1-/2} dt = \sin(\phi).R_F(q,r,1);$$

the elliptic integral of the second kind:

$$E((\phi)|m) = \int_0^{\sin(\phi)} (1-t^2)^{-1/2} (1-mt^2)^{-1/2} dt = \sin(\phi).R_F(q,r,1) - \frac{1}{3}m\sin^3(\phi).R_D(q,r,1)$$

the elliptic integral of the third kind:

$$(Pi)(n;(\phi)|m) = \int_0^{\sin(\phi)} (1-t^2)^{-1/2} (1-mt^2)^{-1/2} (1+nt^2)^{2\,-1} dt = \sin(\phi).R_F(q,r,1) - \frac{1}{3}n\sin^3(\phi).R_J(q,r,1,s)$$

Also the complete elliptic integral of the first kind:

```
      (pi)/2
      /             2       -1/2
K(m)= |     (1-m.sin (theta))   d(theta)=R (0,1-m,1);
      /                                   F
      0
```

the complete elliptic integral of the second kind:

```
      (pi)/2
      /         2        1/2                    1
E(m)= | (1-m.sin (theta))  d(theta)=R (0,1-m,1)- -mR (0,1-m,1).
      /                               F          3  D
      0
```

2.3. Bessel and Airy Functions of a Complex Argument

The routines for Bessel and Airy functions of a real argument are based on Chebyshev expansions, as described in Section 2.1. The routines for functions of a complex argument, however, use different methods. These routines relate all functions to the modified Bessel functions I (z) and K (z) computed in the
                        (nu)   (nu)
right-half complex plane, including their analytic continuations. I and K are computed by different methods according to
 (nu)   (nu)
the values of zand (nu). The methods include power series, asymptotic expansions and Wronskian evaluations. The relations between functions are based on well known formulae (see Abramowitz and Stegun [1]).

2.4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Carlson B C (1977) Special Functions of Applied Mathematics. Academic Press.

[3] Carlson B C (1965) On Computing Elliptic Integrals and Functions. J Math Phys. 44 36--51.

[4] Carlson B C (1977) Elliptic Integrals of the First Kind. SIAM J Math Anal. 8 231--242.

[5] Clenshaw C W (1962) Mathematical Tables. Chebyshev Series for Mathematical Functions. HMSO.

[6] Fox L and Parker I B (1968) Chebyshev Polynomials in

Numerical Analysis. Oxford University Press.

[7] Schonfelder J L (1976) The Production of Special Function Routines for a Multi-Machine Library. Software Practice and Experience. 6(1)

3. Recommendations on Choice and Use of Routines

3.1. Elliptic Integrals

IMPORTANT ADVICE: users who encounter elliptic integrals in the course of their work are strongly recommended to look at transforming their analysis directly to one of the Carlson forms, rather than the traditional canonical Legendre forms. In general, the extra symmetry of the Carlson forms is likely to simplify the analysis, and these symmetric forms are much more stable to calculate.

The routine S21BAF for R_C is largely included as an auxiliary to the other routines for elliptic integrals. This integral essentially calculates elementary functions, e.g.

$$\ln x = (x-1).R_C\left(\frac{(1+x)^2}{4}, x\right), x>0;$$

$$\arcsin x = x.R_C\left(1-x^2, 1\right), |x|\le 1;$$

$$\operatorname{arcsinh} x = x.R_C\left(1+x^2, 1\right), \text{ etc}$$

In general this method of calculating these elementary functions is not recommended as there are usually much more efficient specific routines available in the Library. However, S21BAF may be used, for example, to compute lnx/(x-1) when x is close to 1, without the loss of significant figures that occurs when lnx and x-1 are computed separately.

3.2. Bessel and Airy Functions

For computing the Bessel functions $J_{(nu)}(x)$, $Y_{(nu)}(x)$, $I_{(nu)}(x)$ and $K_{(nu)}(x)$ where x is real and (nu)=0 or 1, special routines are provided, which are much faster than the more general

routines that allow a complex argument and arbitrary real (nu)>=0
functions and their derivatives Ai(x), Bi(x), Ai'(x), Bi'(x) for
a real argument which are much faster than the routines for
complex arguments.

3.3. Index

Airy function, Ai, real argument S17AGF
Airy function, Ai', real argument S17AJF
Airy function, Ai or Ai', complex argument, optionally scaled S17DGF
Airy function, Bi, real argument S17AHF
Airy function, Bi', real argument S17AKF
Airy function, Bi or Bi', complex argument, optionally scaled S17DHF
Bessel function, J , real argument S17AEF
0
Bessel function, J , real argument S17AFF
1
Bessel function, J , complex argument, optionally scaled S17DEF
(nu)
Bessel function, Y , real argument S17ACF
0
Bessel function, Y , real argument S17ADF
1
Bessel function, Y , complex argument, optionally scaled S17DCF
(nu)
Complement of the Error function S15ADF
Cosine Integral S13ACF
Elliptic integral, symmetrised, degenerate of 1st kind, R S21BAF
C
Elliptic integral, symmetrised, of 1st kind, R S21BBF
F
Elliptic integral, symmetrised, of 2nd kind, R S21BCF
D
Elliptic integral, symmetrised, of 3rd kind, R S21BDF
J
Erf, real argument S15AEF
Erfc, real argument S15ADF
Error function S15AEF
Exponential, complex S01EAF
Exponential Integral S13AAF
Fresnel Integral, C S20ADF
Fresnel Integral, S S20ACF
Gamma function S14AAF
Gamma function, incomplete S14BAF
Generalized Factorial function S14AAF

                (1)   (2)
Hankel function H   or H   , complex argument,   S17DLF
                (nu)   (nu)
optionally scaled
Incomplete Gamma function   S14BAF
Jacobian elliptic functions, sn, cn, dn   S21CAF
Kelvin function, bei x   S19ABF
Kelvin function, ber x   S19AAF
Kelvin function, kei x   S19ADF
Kelvin function, ker x   S19ACF
Logarithm of Gamma function   S14ABF
Modified Bessel function, I , real argument   S18AEF
                           0
Modified Bessel function, I , real argument   S18AFF
                           1
Modified Bessel function, I   , complex argument,   S18DEF
                           (nu)
optionally scaled
Modified Bessel function, K , real argument   S18ACF
                           0
Modified Bessel function, K , real argument   S18ADF
                           1
Modified Bessel function, K   , complex argument,   S18DCF
                           (nu)
optionally scaled
Sine integral   S13ADF

Chapter S

Approximations of Special Functions

                              z
S01EAF Complex exponential, e

S13AAF Exponential integral E (x)
                              1

S13ACF Cosine integral Ci(x)

S13ADF Sine integral Si(x)

S14AAF Gamma function

S14ABF Log Gamma function

S14BAF Incomplete gamma functions P(a,x) and Q(a,x)

S15ADF Complement of error function erfc x

```
S15AEF  Error function erf x

S17ACF  Bessel function Y (x)
                         0

S17ADF  Bessel function Y (x)
                         1

S17AEF  Bessel function J (x)
                         0

S17AFF  Bessel function J (x)
                         1

S17AGF  Airy function Ai(x)

S17AHF  Airy function Bi(x)

S17AJF  Airy function Ai'(x)

S17AKF  Airy function Bi'(x)

S17DCF  Bessel functions Y      (z), real a>=0, complex z,
                          (nu)+a
        (nu)=0,1,2,...

S17DEF  Bessel functions J      (z), real a>=0, complex z,
                          (nu)+a
        (nu)=0,1,2,...

S17DGF  Airy functions Ai(z) and Ai'(z), complex z

S17DHF  Airy functions Bi(z) and Bi'(z), complex z

                          (j)
S17DLF  Hankel functions H      (z), j=1,2, real a>=0, complex z,
                          (nu)+a
        (nu)=0,1,2,...

S18ACF  Modified Bessel function K (x)
                                  0

S18ADF  Modified Bessel function K (x)
                                  1

S18AEF  Modified Bessel function I (x)
                                  0

S18AFF  Modified Bessel function I (x)
```

```
                                    1

S18DCF  Modified Bessel functions K      (z), real a>=0, complex
                                     (nu)+a
        z, (nu)=0,1,2,...

S18DEF  Modified Bessel functions I      (z), real a>=0, complex
                                     (nu)+a
        z, (nu)=0,1,2,...

S19AAF  Kelvin function ber x

S19ABF  Kelvin function bei x

S19ACF  Kelvin function ker x

S19ADF  Kelvin function kei x

S20ACF  Fresnel integral S(x)

S20ADF  Fresnel integral C(x)

S21BAF  Degenerate symmetrised elliptic integral of 1st kind
        R (x,y)
         C

S21BBF  Symmetrised elliptic integral of 1st kind R (x,y,z)
                                                    F

S21BCF  Symmetrised elliptic integral of 2nd kind R (x,y,z)
                                                    D

S21BDF  Symmetrised elliptic integral of 3rd kind R (x,y,z,r)
                                                    J
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S01 -- Approximations of Special Functions                 S01EAF
        S01EAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

                                              z
S01EAF evaluates the exponential function e , for complex z.

2. Specification
```

```
COMPLEX(KIND(1.0D0)) FUNCTION S01EAF (Z, IFAIL)
INTEGER                               IFAIL
COMPLEX(KIND(1.0D0))                  Z
```

3. Description

This routine evaluates the exponential function e^z, taking care to avoid machine overflow, and giving a warning if the result cannot be computed to more than half precision. The function is evaluated as $e^z = e^x(\cos y + i \sin y)$, where x and y are the real and imaginary parts respectively of z.

Since $\cos y$ and $\sin y$ are less than or equal to 1 in magnitude, it is possible that e^x may overflow although $e^x \cos y$ or $e^x \sin y$ does not. In this case the alternative formula $\text{sign}(\cos y) e^{x+\ln|\cos y|}$ is used for the real part of the result, and $\text{sign}(\sin y) e^{x+\ln|\sin y|}$ for the imaginary part. If either part of the result still overflows, a warning is returned through parameter IFAIL.

If Im z is too large, precision may be lost in the evaluation of $\sin y$ and $\cos y$. Again, a warning is returned through IFAIL.

4. References

None.

5. Parameters

1: Z -- COMPLEX(KIND(1.0D0)) Input
On entry: the argument z of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

```
IFAIL= 1
     The real part of the result overflows, and is set to the
     largest safe number with the correct sign. The imaginary
     part of the result is meaningful.

IFAIL= 2
     The imaginary part of the result overflows, and is set to
     the largest safe number with the correct sign. The real part
     of the result is meaningful.

IFAIL= 3
     Both real and imaginary parts of the result overflow, and
     are set to the largest safe number with the correct signs.

IFAIL= 4
     The computed result is accurate to less than half precision,
     due to the size of Im z.

IFAIL= 5
     The computed result has no precision, due to the size of Im
     z, and is set to zero.

7. Accuracy

Accuracy is limited in general only by the accuracy of the
Fortran intrinsic functions in the computation of siny, cosy and
 x
e , where x=Re z, y=Im z. As y gets larger, precision will
probably be lost due to argument reduction in the evaluation of
the sine and cosine functions, until the warning error IFAIL = 4


occurs when y gets larger than \/1/(epsilon), where (epsilon) is
the machine precision. Note that on some machines, the intrinsic
functions SIN and COS will not operate on arguments larger than


about \/1/(epsilon), and so IFAIL can never return as 4.

In the comparatively rare event that the result is computed by
                        x+ln|cosy|                x+ln|siny|
the formulae sign(cosy)e           and sign(siny)e          , a
further small loss of accuracy may be expected due to rounding
errors in the logarithmic function.

8. Further Comments

None.

9. Example
```

```
          The example program reads values of the argument z from a file,
          evaluates the function at each value of z and prints the results.

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          S13 -- Approximations of Special Functions                S13AAF
                 S13AAF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          S13AAF returns the value of the exponential integral E (x), via
                                                                1
          the routine name.

          2. Specification

                 DOUBLE PRECISION FUNCTION S13AAF (X, IFAIL)
                 INTEGER          IFAIL
                 DOUBLE PRECISION X

          3. Description

          The routine calculates an approximate value for

                                    infty
                                    /     e-u
                           E (x)= |     -  du , x>0.
                            1     /     u
                                  x

          For 0<x<=4, the approximation is based on the Chebyshev expansion

                                  --                          1
                   E (x)=y(t)-lnx= > 'a T (t)-lnx , where t= -x-1.
                    1             --  r r                    2
                                  r

          For x>4,

                                  -x       -x
```

$$E_1(x) = \frac{e^x}{x} y(t) = \frac{e^x}{x} \sum_r{}' a_r T_r(t),$$

$$\text{where } t = -1.0 + 14.5/(x+3.25) = \frac{11.25-x}{3.25+x}.$$

In both cases, -1<=t<=+1.

To guard against producing underflows, if $x > x_{hi}$ the result is set directly to zero. For the value x_{hi} see the Users' Note for your implementation.

4. References

[1] Abramowitz M and Stegun I A (1965) Handbook of Mathematical Functions. Dover Publications. Ch. 26.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

2: IFAIL -- INTEGER Input/Output
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The routine has been called with an argument less than or equal to zero for which the function is not defined. The result returned is zero.

7. Accuracy

If (delta) and (epsilon) are the relative errors in argument and result respectively, then in principle,

```
                                  |   -x         |
                                  |  e           |
                        |(epsilon)|~=| -----*(delta)|
                                  | E (x)        |
                                  |  1           |
```

so the relative error in the argument is amplified in the result
by at least a factor $e^{-x}/E_1(x)$. The equality should hold if (delta) is greater than the machine precision ((delta) due to data errors etc) but if (delta) is simply a result of round-off in the machine representation, it is possible that an extra figure may be lost in internal calculation and round-off.

The behaviour of this amplification factor is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

It should be noted that, for small x, the amplification factor tends to zero and eventually the error in the result will be limited by machine precision.

For large x,

(epsilon)~x(delta)=(Delta),

the absolute error in the argument.

8. Further Comments

None.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

S13 -- Approximations of Special Functions S13ACF
S13ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for

your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S13ACF returns the value of the cosine integral

$$Ci(x)=(\gamma)+\ln x+\int_0^x \frac{\cos u-1}{u}\,du, \quad x>0$$

via the routine name, where (γ) denotes Euler's constant.

2. Specification

```
      DOUBLE PRECISION FUNCTION S13ACF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for Ci(x).

For $0<x<=16$ it is based on the Chebyshev expansion

$$Ci(x)=\ln x+\sum_{r=0}{}' a_r T_r(t), \quad t=2\left(\frac{x}{16}\right)^2-1.$$

For $16<x<x_{hi}$ where the value of x_{hi} is given in the Users' Note
for your implementation,

$$Ci(x)=\frac{f(x)\sin x}{x}-\frac{g(x)\cos x}{x^2}$$

$$\text{where } f(x)=\sum_{r=0}{}' f_r T_r(t) \text{ and } g(x)=\sum_{r=0}{}' g_r T_r(t), \quad t=2\left(\frac{16}{x}\right)^2-1.$$

For $x>=x_{hi}$, Ci(x)=0 to within the accuracy possible (see Section
7).

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The routine has been called with an argument less than or equal to zero for which the function is not defined. The result returned is zero.

7. Accuracy

If E and (epsilon) are the absolute and relative errors in the result and (delta) is the relative error in the argument then in principle these are related by

$$|E| \simeq |(\mathrm{delta})\cos x| \text{ and } |(\mathrm{epsilon})| \simeq \left|\frac{(\mathrm{delta})\cos x}{\mathrm{Ci}(x)}\right|.$$

That is accuracy will be limited by machine precision near the origin and near the zeros of cosx, but near the zeros of Ci(x) only absolute accuracy can be maintained.

The behaviour of this amplification is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

sinx

```
For large values of x, Ci(x)~ ---- therefore
                                x
(epsilon)~(delta)xcotx and since (delta) is limited by the finite
precision of the machine it becomes impossible to return results
which have any relative accuracy. That is, when x>=1/(delta) we
have that |Ci(x)|<=1/x~E and hence is not significantly different
from zero.

Hence x   is chosen such that for values of x>=x  , Ci(x) in
       hi                                      hi
principle would have values less than the machine precision and
so is essentially zero.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S13 -- Approximations of Special Functions                S13ADF
        S13ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S13ADF returns the value of the sine integral

                                x
                                / sinu
                      Si(x)= | ----du,
                                /  u
                                0

via the routine name.

2. Specification
```

```
      DOUBLE PRECISION FUNCTION S13ADF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for Si(x).

For |x|<=16.0 it is based on the Chebyshev expansion

$$Si(x) = x \sum_{r=0}^{\prime\prime} a_r T_r(t), \quad t = 2\left(\frac{x}{16}\right)^2 - 1.$$

For $16 < |x| < x_{hi}$, where x_{hi} is an implementation dependent number,

$$Si(x) = \mathrm{sign}(x)\left\{\frac{\pi}{2} - \frac{f(x)\cos x}{x} - \frac{g(x)\sin x}{x^2}\right\}$$

where $f(x) = \sum_{r=0}^{\prime\prime} f_r T_r(t)$ and $g(x) = \sum_{r=0}^{\prime\prime} g_r T_r(t)$, $t = 2\left(\frac{16}{x}\right)^2 - 1$.

For $|x| \geq x_{hi}$, $Si(x) = -\frac{1}{2}(\pi)\mathrm{sign}x$ to within machine precision.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

```
6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL
has been included for consistency with other routines in this
chapter.

7. Accuracy

If (delta) and (epsilon) are the relative errors in the argument
and result, respectively, then in principle

                                | (delta)sinx|
                 |(epsilon)|~=| -----------|.
                                |    Si(x)   |

The equality may hold if (delta) is greater than the machine
precision ((delta) due to data errors etc) but if (delta) is
simply due to round-off in the machine representation, then since
the factor relating (delta) to (epsilon) is always less than one,
the accuracy will be limited by machine precision.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S14 -- Approximations of Special Functions                 S14AAF
        S14AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S14AAF returns the value of the Gamma function (Gamma)(x), via
the routine name.
```

2. Specification

```
DOUBLE PRECISION FUNCTION S14AAF (X, IFAIL)
INTEGER          IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Gamma function (Gamma)(x). The routine is based on the Chebyshev expansion:

$$(\text{Gamma})(1+u) = \sum_{r=0}^{--'} a_r T_r(t), \quad \text{where } 0 \le u < 1,\ t = 2u-1,$$

and uses the property (Gamma)(1+x)=x(Gamma)(x). If x=N+1+u where N is integral and 0<=u<1 then it follows that:

for N>0 $(\text{Gamma})(x) = (x-1)(x-2)\ldots(x-N)(\text{Gamma})(1+u)$,

for N=0 $(\text{Gamma})(x) = (\text{Gamma})(1+u)$,

for N<0 $$(\text{Gamma})(x) = \frac{(\text{Gamma})(1+u)}{x(x+1)(x+2)\ldots(x-N-1)}.$$

There are four possible failures for this routine:

(i) if x is too large, there is a danger of overflow since (Gamma)(x) could become too large to be represented in the machine;

(ii) if x is too large and negative, there is a danger of underflow;

(iii) if x is equal to a negative integer, (Gamma)(x) would overflow since it has poles at such points;

(iv) if x is too near zero, there is again the danger of overflow on some machines. For small x, $(\text{Gamma})(x) \tilde{=} \frac{1}{x}$, and on some machines there exists a range of non-zero but small values of x for which 1/x is larger than the greatest representable value.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X must not be a negative integer.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The argument is too large. On soft failure the routine returns the approximate value of (Gamma)(x) at the nearest valid argument.

IFAIL= 2
The argument is too large and negative. On soft failure the routine returns zero.

IFAIL= 3
The argument is too close to zero. On soft failure the routine returns the approximate value of (Gamma)(x) at the nearest valid argument.

IFAIL= 4
The argument is a negative integer, at which value (Gamma)(x) is infinite. On soft failure the routine returns a large positive value.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and the result respectively. If (delta) is somewhat larger than the machine precision (i.e., is due to data errors etc), then (epsilon) and (delta) are approximately related by:

(epsilon)~=|x(Psi)(x)|(delta)

(provided (epsilon) is also greater than the representation

error). Here (Psi)(x) is the digamma function $\frac{(\text{Gamma})'(x)}{(\text{Gamma})(x)}$.

Figure 1 shows the behaviour of the error amplification factor |x(Psi)(x)|.

Figure 1
Please see figure in printed Reference Manual

If (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

There is clearly a severe, but unavoidable, loss of accuracy for arguments close to the poles of (Gamma)(x) at negative integers. However relative accuracy is preserved near the pole at x=0 right up to the point of failure arising from the danger of overflow.

Also accuracy will necessarily be lost as x becomes large since in this region

(epsilon)~=(delta)xlnx.

However since (Gamma)(x) increases rapidly with x, the routine must fail due to the danger of overflow before this loss of accuracy is too great. (e.g. for x=20, the amplification factor ~= 60.)

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

S14 -- Approximations of Special Functions S14ABF
S14ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details.

```
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S14ABF returns a value for the logarithm of the Gamma function,
ln(Gamma)(x), via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S14ABF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to ln(Gamma)(x). It is
based on two Chebyshev expansions.

For 0<x<=x     , ln(Gamma)(x)=-lnx to within machine accuracy.
          small

For x     <x<=15.0, the recursive relation
     small
(Gamma)(1+x)=x(Gamma)(x) is used to reduce the calculation to one
involving (Gamma)(1+u), 0<=u<1 which is evaluated as:

                              --'
                 (Gamma)(1+u)= >   a T (t),  t=2u-1.
                              --   r r
                              r=0

Once (Gamma)(x) has been calculated, the required result is
produced by taking the logarithm.

For 15.0<x<=x   ,
             big

                                  1        1
              ln(Gamma)(x)=(x- -)lnx-x+ -ln2(pi)+y(x)/x
                                  2        2
                                     2
              --'              ( 15)
where y(x)= >   b T (t), t=2( --) -1.
              --   r r        ( x )
              r=0

For x   <x<=x     the term y(x)/x is negligible and so its
     big     vbig
calculation is omitted.
```

```
For x>x     there is a danger of setting overflow so the routine
       vbig
must fail.

For x<=0.0 the function is not defined and the routine fails.

Note: x      is calculated so that if x<x     , (Gamma)(x)=1/x to
       small                             small
within machine accuracy. x    is calculated so that if x>x   ,
                         big                               big

                                 1        1
              ln(Gamma)(x)=(x- -)lnx-x+ -ln2(pi)
                                 2        2

to within machine accuracy. x     is calculated so that
                             vbig
ln(Gamma)(x    ) is close to the value returned by X02ALF(*).
          vbig

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x of the function. Constraint: X > 0.
     0.

 2:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     X <= 0.0, the function is undefined. On soft failure, the
     routine returns zero.

IFAIL= 2
     X is too large, the function would overflow. On soft
```

failure, the routine returns the value of the function at
the largest permissible argument.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument
and result respectively, and E be the absolute error in the
result.

If (delta) is somewhat larger than the relative machine
precision, then

$$E\tilde{}=|x*(\mathrm{Psi})(x)|(\mathrm{delta}) \text{ and } (\mathrm{epsilon})\tilde{}=\left|\frac{x*(\mathrm{Psi})(x)}{\ln(\mathrm{Gamma})(x)}\right|(\mathrm{delta})$$

where (Psi)(x) is the digamma function $\frac{(\mathrm{Gamma})'(x)}{(\mathrm{Gamma})(x)}$. Figure 1 and
Figure 2 show the behaviour of these error amplification factors.

Figure 1
Please see figure in printed Reference Manual

Figure 2
Please see figure in printed Reference Manual

These show that relative error can be controlled, since except
near x=1 or 2 relative error is attenuated by the function or at
least is not greatly amplified.

$$\text{For large x, } (\mathrm{epsilon})\tilde{}=\left(1+\frac{1}{\ln x}\right)(\mathrm{delta}) \text{ and for small x,}$$

$$(\mathrm{epsilon})\tilde{}=\frac{1}{\ln x}(\mathrm{delta}).$$

The function ln(Gamma)(x) has zeros at x=1 and 2 and hence
relative accuracy is not maintainable near those points. However
absolute accuracy can still be provided near those zeros as is
shown above.

If however, (delta) is of the order of the machine precision,
then rounding errors in the routine's internal arithmetic may
result in errors which are slightly larger than those predicted
by the equalities. It should be noted that even in areas where
strong attenuation of errors is predicted the relative precision
is bounded by the effective machine precision.

```
          8. Further Comments

          For details of the time taken by the routine see the Users' Note
          for your implementation.

          9. Example

          The example program reads values of the argument x from a file,
          evaluates the function at each value of x and prints the results.

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          S14 -- Approximations of Special Functions                S14BAF
                  S14BAF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          S14BAF computes values for the incomplete gamma functions P(a,x)
          and Q(a,x).

          2. Specification

                 SUBROUTINE S14BAF (A, X, TOL, P, Q, IFAIL)
                 INTEGER          IFAIL
                 DOUBLE PRECISION A, X, TOL, P, Q

          3. Description

          This subroutine evaluates the incomplete gamma functions in the
          normalised form

                                          x
                                1         /  a-1 -t
                     P(a,x)= ---------- | t   e  dt,
                             (Gamma)(a) /
                                        0

                                       infty
                                1       /     a-1 -t
                     Q(a,x)= ---------- |    t   e  dt,
```

```
                        (Gamma)(a) /
                                   x
```

with x>=0 and a>0, to a user-specified accuracy. With this normalisation, P(a,x)+Q(a,x)=1.

Several methods are used to evaluate the functions depending on the arguments a and x, the methods including Taylor expansion for P(a,x), Legendre's continued fraction for Q(a,x), and power series for Q(a,x). When both a and x are large, and a~=x, the uniform asymptotic expansion of Temme [3] is employed for greater efficiency - specifically, this expansion is used when a>=20 and 0.7a<=x<=1.4a.

Once either of P or Q is computed, the other is obtained by subtraction from 1. In order to avoid loss of relative precision in this subtraction, the smaller of P and Q is computed first.

This routine is derived from subroutine GAM in Gautschi [2].

4. References

[1] Gautschi W (1979) A Computational Procedure for Incomplete Gamma Functions. ACM Trans. Math. Softw. 5 466--481.

[2] Gautschi W (1979) Algorithm 542: Incomplete Gamma Functions. ACM Trans. Math. Softw. 5 482--489.

[3] Temme N M (1987) On the Computation of the Incomplete Gamma Functions for Large Values of the Parameters. Algorithms for Approximation. (ed J C Mason and M G Cox) Oxford University Press.

5. Parameters

1: A -- DOUBLE PRECISION Input
On entry: the argument a of the functions. Constraint: A > 0.0.

2: X -- DOUBLE PRECISION Input
On entry: the argument x of the functions. Constraint: X >= 0.0.

3: TOL -- DOUBLE PRECISION Input
On entry: the relative accuracy required by the user in the results. If S14BAF is entered with TOL greater than 1.0 or less than machine precision, then the value of machine precision is used instead.

4: P -- DOUBLE PRECISION Output

```
 5:  Q -- DOUBLE PRECISION                                  Output
     On exit: the values of the functions P(a,x) and Q(a,x)
     respectively.

 6:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry A <= 0.0.

IFAIL= 2
     On entry X < 0.0.

IFAIL= 3
     Convergence of the Taylor series or Legendre continued
     fraction fails within 600 iterations. This error is
     extremely unlikely to occur; if it does, contact NAG.

7. Accuracy

There are rare occasions when the relative accuracy attained is
somewhat less than that specified by parameter TOL. However, the
error should never exceed more than one or two decimal places.
Note also that there is a limit of 18 decimal places on the
achievable accuracy, because constants in the routine are given
to this precision.

8. Further Comments

The time taken for a call of S14BAF depends on the precision
requested through TOL, and also varies slightly with the input
arguments a and x.

9. Example

The example program reads values of the argument a and x from a
file, evaluates the function and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
```

Library software and should be available on-line.

%%

S15 -- Approximations of Special Functions S15ADF
S15ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S15ADF returns the value of the complementary error function, erfcx, via the routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S15ADF (X, IFAIL)
INTEGER          IFAIL
DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for the complement of the error function

$$\operatorname{erfc} x = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-u^2} du = 1 - \operatorname{erf} x.$$

For x>=0, it is based on the Chebyshev expansion

$$\operatorname{erfc} x = e^{-x^2} y(x),$$

$$\text{where } y(x) = \sum_{r=0}^{\prime\prime} a_r T_r(t) \text{ and } t=(x-3.75)/(x+3.75),\ -1 \le t \le +1.$$

For $x \ge x_{hi}$, where there is a danger of setting underflow, the result is returned as zero.

```
                   2
                 -x
For x<0, erfc x=2-e   y(|x|).

For x<x   <0, the result is returned as 2.0 which is correct to
       low
within machine precision. The values of x   and x    are given in
                                         hi      low
the Users' Note for your implementation.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x of the function.

 2:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL
has been included for consistency with other routines in this
chapter.

7. Accuracy

If (delta) and (epsilon) are relative errors in the argument and
result, respectively, then in principle

                           |         2          |
                           |       -x           |
                           |    2xe             |
             |(epsilon)|~=| ------------(delta)|.
                           |                    |

                           | \/(pi)erfc x       |

That is, the relative error in the argument, x, is amplified by a
               2
             -x
```

$$\text{factor } \frac{2xe}{\sqrt{\pi}\,\mathrm{erfc}\,x} \text{ in the result.}$$

The behaviour of this factor is shown in Figure 1.

Figure 1

Please see figure in printed Reference Manual

It should be noted that near x=0 this factor behaves as $\frac{2x}{\sqrt{\pi}}$ and hence the accuracy is largely determined by the machine precision. Also for large negative x, where the factor is $\tilde{} \frac{xe^{-x^2}}{\sqrt{\pi}}$, accuracy is mainly limited by machine precision.

However, for large positive x, the factor becomes $\tilde{}2x^2$ and to an extent relative accuracy is necessarily lost. The absolute accuracy E is given by

$$E\tilde{} = \frac{2xe^{-x^2}}{\sqrt{\pi}}(\mathrm{delta})$$

so absolute accuracy is guaranteed for all x.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file,

```
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S15 -- Approximations of Special Functions                 S15AEF
        S15AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S15AEF returns the value of the error function erfx, via the
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S15AEF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

Evaluates the error function,

                          2
                       x -t
                 2     /
     erf x =  ------ |e   dt.
                     /
              \/(pi) 0

For |x|<=2,
                 --'                    1 2
      erf x = x >   a T (t), where t= -x -1.
                 --   r r               2
                 r=0

For 2<|x|<x   ,
           hi
                    {        2              }
                    {      -x               }
                    {     e       --         }           x-7
      erf x = signx{1- ---------- >  'b T (t)}, where t= ---.
```

```
                  {             --   r r   }           x+3
                  {   |x|\/(pi) r=0        }
```

For $|x| \geq x_{hi}$,

$$\operatorname{erf} x = \operatorname{signx}.$$

x_{hi} is the value above which erf x=+-1 within machine precision.
Its value is given in the Users' Note for your implementation.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

On a machine with approximately 11 significant figures the routine agrees with available tables to 10 figures and consistency checking with S15ADF of the relation

$$\operatorname{erf} x + \operatorname{erfc} x = 1.0$$

shows errors in at worst the 11th figure.

8. Further Comments

None.

9. Example

```
The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                  S17ACF
        S17ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17ACF returns the value of the Bessel Function Y (x), via the
                                                  0
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17ACF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Bessel Function of
the second kind Y (x).
                0

Note: Y (x) is undefined for x<=0 and the routine will fail for
       0
such arguments.

The routine is based on four Chebyshev expansions:

For 0<x<=8,
                                                          2
            2      --'          --'                  ( x)
    Y (x)= ----lnx >   a T (t)+ >   b T (t), with t=2( -) -1,
     0     (pi)    --   r r     --   r r             ( 8)
                   r=0          r=0

For x>8,
```

$$Y_0(x) = \sqrt{\frac{2}{\pi x}}\left\{P_0(x)\sin\left(x-\frac{\pi}{4}\right)+Q_0(x)\cos\left(x-\frac{\pi}{4}\right)\right\}$$

$$\text{where } P_0(x) = \sum_{r=0}^{8} c_r T_r(t),$$

$$\text{and } Q_0(x) = \frac{8}{x}\sum_{r=0}^{8} d_r T_r(t), \text{ with } t = 2\left(\frac{8}{x}\right)^2 - 1.$$

For x near zero, $Y_0(x) \tilde{=} \frac{2}{\pi}\left(\ln\left(\frac{x}{2}\right)+\gamma\right)$, where γ denotes Euler's constant. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For very large x, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $Y_0(x)$; only the amplitude, $\sqrt{\frac{2}{\pi x}}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision: the routine will fail if $x \tilde{>} 1/\text{machine precision}$ (see the Users' Note for your implementation for details).

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Clenshaw C W (1962) Mathematical Tables. Chebyshev Series for Mathematical Functions. HMSO.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

```
 2:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     X is too large. On soft failure the routine returns the
     amplitude of the Y  oscillation, \/2/(pi)x.
                      0

IFAIL= 2
     X <= 0.0, Y  is undefined. On soft failure the routine
                0
     returns zero.

7. Accuracy

Let (delta) be the relative error in the argument and E be the
absolute error in the result. (Since Y (x) oscillates about zero,
                                       0
absolute error and not relative error is significant, except for
very small x.)

If (delta) is somewhat larger than the machine representation
error (e.g. if (delta) is due to data errors etc), then E and
(delta) are approximately related by

                       E~=|xY (x)|(delta)
                             1

(provided E is also within machine bounds). Figure 1 displays the
behaviour of the amplification factor |xY (x)|.
                                        1


                           Figure 1
            Please see figure in printed Reference Manual

However, if (delta) is of the same order as the machine
representation errors, then rounding errors could make E slightly
larger than the above relation predicts.
```

For very small x, the errors are essentially independent of
(delta) and the routine should provide relative accuracy bounded
by the machine precision.

For very large x, the above relation ceases to apply. In this

region, $Y_0(x) \tilde{=} \sqrt{\frac{2}{(pi)x}}\sin\left(x - \frac{(pi)}{4}\right)$. The amplitude
can be calculated with reasonable accuracy for all x, but
$\sin\left(x - \frac{(pi)}{4}\right)$ cannot. If $x - \frac{(pi)}{4}$ is written as $2N(pi)+(theta)$
where N is an integer and $0 \le (theta) < 2(pi)$, then $\sin\left(x - \frac{(pi)}{4}\right)$ is
determined by (theta) only. If $x > \tilde{} (delta)^{-1}$, (theta) cannot be
determined with any accuracy at all. Thus if x is greater than,
or of the order of the inverse of machine precision, it is
impossible to calculate the phase of $Y_0(x)$ and the routine must
fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%

S17 -- Approximations of Special Functions S17ADF
S17ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

```
S17ADF returns the value of the Bessel Function Y (x), via the
                                                  1
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17ADF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Bessel Function of
the second kind Y (x).
                1

Note: Y (x) is undefined for x<=0 and the routine will fail for
       1
such arguments.

The routine is based on four Chebyshev expansions:

For 0<x<=8,

             2      x --'           2     x --'
      Y (x)= ----lnx - >   a T (t)- ----x+ - >   b T (t) , with t
       1     (pi)    8 --   r r     (pi)   8 --   r r
                       r=0                   r=0
                 2
               ( x)
             =2( -) -1;
               ( 8)


For x>8,


                  /   2  {        (    (pi))         (    (pi))}
           Y (x)=  /  -----{P (x)sin(x-3 ----)+Q (x)cos(x-3 ----)}
            1    \/   (pi)x{ 1      (     4  )  1      (     4  )}

                     --'
                     >
           where P (x)=   c T (t),
                  1     --   r r
                        r=0
                                              2
                     8 --'                  ( 8)
           and Q (x)= - >   d T (t), with t=2( -) -1.
                1     x --   r r             ( x)
```

r=0

2
For x near zero, Y (x)~=- -----. This approximation is used when
1 (pi)x
x is sufficiently small for the result to be correct to machine precision. For extremely small x, there is a danger of overflow
2
in calculating - ----- and for such arguments the routine will
(pi)x
fail.

For very large x, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the

/ 2
phase of oscillation of Y (x), only the amplitude, / -----,
1 \/ (pi)x
can be determined and this is returned on soft failure. The range for which this occurs is roughly related to machine precision; the routine will fail if x>~1/machine precision (see the Users' Note for your implementation for details).

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Clenshaw C W (1962) Mathematical Tables. Chebyshev Series for Mathematical Functions. HMSO.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large. On soft failure the routine returns the

amplitude of the Y_1 oscillation, $\sqrt{\frac{2}{\pi x}}$.

IFAIL= 2
$X \leq 0.0$, Y_1 is undefined. On soft failure the routine returns zero.

IFAIL= 3
X is too close to zero, there is a danger of overflow. On soft failure, the routine returns the value of $Y_1(x)$ at the smallest valid argument.

7. Accuracy

Let (delta) be the relative error in the argument and E be the absolute error in the result. (Since $Y_1(x)$ oscillates about zero, absolute error and not relative error is significant, except for very small x.)

If (delta) is somewhat larger than the machine precision (e.g. if (delta) is due to data errors etc), then E and (delta) are approximately related by:

$$E \tilde{=} |xY_0(x) - Y_1(x)| (\delta)$$

(provided E is also within machine bounds). Figure 1 displays the behaviour of the amplification factor $|xY_0(x) - Y_1(x)|$.

Figure 1
Please see figure in printed Reference Manual

However, if (delta) is of the same order as machine precision, then rounding errors could make E slightly larger than the above relation predicts.

For very small x, absolute error becomes large, but the relative error in the result is of the same order as (delta).

For very large x, the above relation ceases to apply. In this
region, $Y_1(x) \tilde{=} \frac{2}{x}\sin\left(x - \frac{3\pi}{4}\right)$. The amplitude $\frac{2}{x}$ can be
calculated with reasonable accuracy for all x, but $\sin\left(x - \frac{3\pi}{4}\right)$
cannot. If $x - \frac{3\pi}{4}$ is written as $2N\pi+\theta$ where N is an
integer and $0 \le \theta < 2\pi$, then $\sin\left(x - \frac{3\pi}{4}\right)$ is determined by
θ only. If $x > \delta^{-1}$, θ cannot be determined with
any accuracy at all. Thus if x is greater than, or of the order
of, the inverse of the machine precision, it is impossible to
calculate the phase of $Y_1(x)$ and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%

S17 -- Approximations of Special Functions S17AEF
S17AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17AEF returns the value of the Bessel Function $J_0(x)$, via the
routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S17AEF (X, IFAIL)
INTEGER          IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Bessel Function of the first kind $J_0(x)$.

Note: $J_0(-x)=J_0(x)$, so the approximation need only consider $x\geq 0$.

The routine is based on three Chebyshev expansions:

For $0<x\leq 8$,

$$J_0(x) = \sum_{r=0}^{\prime} a_r T_r(t), \quad \text{with } t = 2\left(\frac{x}{8}\right)^2 - 1.$$

For $x>8$,

$$J_0(x) = \sqrt{\frac{2}{\pi x}}\left\{P_0(x)\cos\left(x - \frac{\pi}{4}\right) - Q_0(x)\sin\left(x - \frac{\pi}{4}\right)\right\}$$

$$\text{where } P_0(x) = \sum_{r=0}^{\prime} b_r T_r(t),$$

$$\text{and } Q_0(x) = \frac{8}{x}\sum_{r=0}^{\prime} c_r T_r(t), \quad \text{with } t = 2\left(\frac{8}{x}\right)^2 - 1.$$

For x near zero, $J_0(x)\tilde{}=1$. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For very large x, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the

phase of oscillation of $J_0(x)$; only the amplitude, $\sqrt{\frac{2}{(\pi)|x|}}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if |x|>~1/machine precision (see the Users' Note for your implementation).

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Clenshaw C W (1962) Mathematical Tables. Chebyshev Series for Mathematical Functions. HMSO.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large. On soft failure the routine returns the

amplitude of the J_0 oscillation, $\sqrt{\frac{2}{(\pi)|x|}}$.

7. Accuracy

Let (delta) be the relative error in the argument and E be the absolute error in the result. (Since $J_0(x)$ oscillates about zero, absolute error and not relative error is significant.)

If (delta) is somewhat larger than the machine precision (e.g. if (delta) is due to data errors etc), then E and (delta) are

approximately related by:

$$E \tilde{=} |xJ_1(x)|(\text{delta})$$

(provided E is also within machine bounds). Figure 1 displays the behaviour of the amplification factor $|xJ_1(x)|$.

Figure 1
Please see figure in printed Reference Manual

However, if (delta) is of the same order as machine precision, then rounding errors could make E slightly larger than the above relation predicts.

For very large x, the above relation ceases to apply. In this region, $J_0(x) \tilde{=} \sqrt{\frac{2}{\pi|x|}}\cos\left(x - \frac{\pi}{4}\right)$. The amplitude $\sqrt{\frac{2}{\pi|x|}}$ can be calculated with reasonable accuracy for all x but $\cos\left(x - \frac{\pi}{4}\right)$ cannot. If $x - \frac{\pi}{4}$ is written as $2N\pi+\theta$ where N is an integer and $0 \le \theta < 2\pi$, then $\cos\left(x - \frac{\pi}{4}\right)$ is determined by θ only. If $x > \tilde{} (\text{delta})^{-1}$, θ cannot be determined with any accuracy at all. Thus if x is greater than, or of the order of, the inverse of the machine precision, it is impossible to calculate the phase of $J_0(x)$ and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                 S17AFF
        S17AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17AFF returns the value of the Bessel Function J (x), via the
                                                 1
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17AFF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Bessel Function of
the first kind J (x).
               1

Note: J (-x)=-J (x), so the approximation need only consider x>=0
       1       1

The routine is based on three Chebyshev expansions:

For 0<x<=8,                                 2
            x --'                    ( x)
      J (x)= - >   a T (t) , with t=2( -) -1.
       1     8 --   r r              ( 8)
               r=0

For x>8,


               /  2   {        (   3(pi))         (   3(pi))}
      J (x)=  /  ----x{P (x)cos(x- -----)-Q (x)sin(x- -----)}
       1    \/   (pi) { 1      (     4  )  1      (     4  )}
```

$$\text{where} P_1(x) = \sum_{r=0}^{} b_r T_r(t),$$

$$\text{and } Q_1(x) = \frac{8}{x}\sum_{r=0}^{} c_r T_r(t), \text{ with } t = 2\left(\frac{8}{x}\right)^2 - 1.$$

For x near zero, $J_1(x) \tilde{=} \frac{x}{2}$. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For very large x, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $J_1(x)$; only the amplitude, $\sqrt{\frac{2}{\pi|x|}}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if $|x| \tilde{>} 1/\text{machine precision}$ (see the Users' Note for your implementation for details).

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Clenshaw C W (1962) Mathematical Tables. Chebyshev Series for Mathematical Functions. HMSO.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large. On soft failure the routine returns the

$$\text{amplitude of the } J_1 \text{ oscillation, } \sqrt{\frac{2}{\pi|x|}}.$$

7. Accuracy

Let (delta) be the relative error in the argument and E be the absolute error in the result. (Since $J_1(x)$ oscillates about zero, absolute error and not relative error is significant.)

If (delta) is somewhat larger than machine precision (e.g. if (delta) is due to data errors etc), then E and (delta) are approximately related by:

$$E \tilde{=} |xJ_0(x)|(\text{delta})$$

(provided E is also within machine bounds). Figure 1 displays the behaviour of the amplification factor $|xJ_0(x)|$.

Figure 1
Please see figure in printed Reference Manual

However, if (delta) is of the same order as machine precision, then rounding errors could make E slightly larger than the above relation predicts.

For very large x, the above relation ceases to apply. In this region, $J_1(x) \tilde{=} \sqrt{\frac{2}{\pi|x|}}\cos\left(x - \frac{3\pi}{4}\right)$. The amplitude $\sqrt{\frac{2}{\pi|x|}}$ can be calculated with reasonable accuracy for all x

```
       (   3(pi))              3(pi)
but cos(x- -----) cannot. If x- ----- is written as
       (     4  )               4
2N(pi)+(theta) where N is an integer and 0<=(theta)<2(pi), then
   (   3(pi))                                          -1
cos(x- -----) is determined by (theta) only. If x>~(delta)  ,
   (     4  )
(theta) cannot be determined with any accuracy at all. Thus if x
is greater than, or of the order of, machine precision, it is
impossible to calculate the phase of J (x) and the routine must
                                       1
fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
S17 -- Approximations of Special Functions                 S17AGF
        S17AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17AGF returns a value for the Airy function, Ai(x), via the
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17AGF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description
```

```
This routine evaluates an approximation to the Airy function,
Ai(x). It is based on a number of Chebyshev expansions:

For x<-5,

                          a(t)sinz-b(t)cosz
                  Ai(x)= -----------------
                                 1/4
                             (-x)



         (pi)  2  /  3
where z= ----+ -\/ -x , and a(t) and b(t) are expansions in the
          4    3
                 3
             ( 5)
variable t=-2( -) -1.
             ( x)

For -5<=x<=0,

                      Ai(x)=f(t)-xg(t),
                                     3
                                  ( x)
where f and g are expansions in t=-2( -) -1.
                                  ( 5)

For 0<x<4.5,

                             -3x/2
                      Ai(x)=e     y(t),

where y is an expansion in t=4x/9-1.

For 4.5<=x<9,

                             -5x/2
                      Ai(x)=e     u(t),

where u is an expansion in t=4x/9-3.

For x>=9,

                               -z
                              e  v(t)
                       Ai(x)= -------,
                                1/4
                               x
```

```
        2  / 3                              ( 18)
where z= -\/ x  and v is an expansion in t=2( --)-1.
        3                                   ( z )
```

For |x|< the machine precision, the result is set directly to Ai(0). This both saves time and guards against underflow in intermediate calculations.

For large negative arguments, it becomes impossible to calculate the phase of the oscillatory function with any precision and so

$$\text{the routine must fail. This occurs if } x < -\left(\frac{3}{2(\text{epsilon})}\right)^{2/3}, \text{ where}$$

(epsilon) is the machine precision.

For large positive arguments, where Ai decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential-like and here relative error is appropriate. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

$$E\tilde{}=|xAi'(x)|(\text{delta}),\quad (\text{epsilon})\tilde{}=\left|\frac{xAi'(x)}{Ai(x)}\right|(\text{delta}).$$

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, errors are strongly damped by the function and hence will be bounded by the machine precision.

For moderate negative x, the error behaviour is oscillatory but the amplitude of the error grows like

$$\text{amplitude}\left(\frac{E}{(\text{delta})}\right)\tilde{}\ \frac{|x|^{5/4}}{\sqrt{\pi}}.$$

However the phase error will be growing roughly like $\frac{2}{3}\sqrt{|x|^3}$ and hence all accuracy will be lost for large negative arguments due to the impossibility of calculating sin and cos to any

$$\text{accuracy if}\quad \frac{2}{3}\sqrt{|x|^3} > \frac{1}{(\text{delta})}.$$

For large positive arguments, the relative error amplification is considerable:

$$(\text{epsilon})\ \ /\ 3$$

```
                                  ---------~\/ x .
                                   (delta)
```

This means a loss of roughly two decimal places accuracy for arguments in the region of 20. However very large arguments are not possible due to the danger of setting underflow and so the errors are limited in practice.

8. Further Comments

None.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%

S17 -- Approximations of Special Functions S17AHF
S17AHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AHF returns a value of the Airy function, Bi(x), via the routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S17AHF (X, IFAIL)
INTEGER          IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Airy function Bi(x). It is based on a number of Chebyshev expansions.

For x<-5,

```
                              a(t)cosz+b(t)sinz
```

$$Bi(x)= \frac{}{(-x)^{1/4}},$$

$$\text{where } z= \frac{(pi)}{4}+\frac{2}{3}\sqrt{-x}^{3} \text{ and a(t) and b(t) are expansions in the variable } t=-2\left(\frac{-5}{x}\right)-1.$$

For -5<=x<=0,

$$Bi(x)=\sqrt{3}(f(t)+xg(t)),$$

where f and g are expansions in $t=-2\left(\frac{-5}{x}\right)^{3}-1$.

For 0<x<4.5,

$$Bi(x)=e^{11x/8}y(t),$$

where y is an expansion in t=4x/9-1.

For 4.5<=x<=9,

$$Bi(x)=e^{5x/2}v(t),$$

where v is an expansion in t=4x/9-3.

For x>=9,

$$Bi(x)= \frac{e^{z}u(t)}{x^{1/4}},$$

$$\text{where } z= -\frac{2}{3}\sqrt{x}^{3} \text{ and u is an expansion in } t=2\left(\frac{--}{18}\right)-1.$$

```
        3                                  ( z )

For |x|< the machine precision, the result is set directly to
Bi(0). This both saves time and avoids possible intermediate
underflows.

For large negative arguments, it becomes impossible to calculate
the phase of the oscillating function with any accuracy so the
                                                    2/3
                                      (     3     )
routine must fail. This occurs if x<-( ----------)   , where
                                      ( 2(epsilon))
(epsilon) is the machine precision.

For large positive arguments, there is a danger of causing
overflow since Bi grows in an essentially exponential manner, so
the routine must fail.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                  Input
     On entry: the argument x of the function.

 2:  IFAIL -- INTEGER                                Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     X is too large and positive. On soft failure, the routine
     returns zero.

IFAIL= 2
     X is too large and negative. On soft failure, the routine
     returns zero.

7. Accuracy
```

For negative arguments the function is oscillatory and hence
absolute error is the appropriate measure. In the positive region
the function is essentially exponential-like and here relative
error is appropriate. The absolute error, E, and the relative
error, (epsilon), are related in principle to the relative error
in the argument, (delta), by

$$E\tilde{}=|xBi'(x)|(\text{delta}),\quad (\text{epsilon})\tilde{}=\left|\frac{xBi'(x)}{Bi(x)}\right|(\text{delta}).$$

In practice, approximate equality is the best that can be
expected. When (delta), (epsilon) or E is of the order of the
machine precision, the errors in the result will be somewhat
larger.

For small x, errors are strongly damped and hence will be bounded
essentially by the machine precision.

For moderate to large negative x, the error behaviour is clearly
oscillatory but the amplitude of the error grows like amplitude

$$\left(\frac{E}{(\text{delta})}\right)\tilde{}\ \frac{|x|^{5/4}}{\sqrt{\pi}}.$$

However the phase error will be growing roughly as $\frac{2}{3}\sqrt{|x|^3}$ and
hence all accuracy will be lost for large negative arguments.

This is due to the impossibility of calculating sin and cos to

$$\text{any accuracy if } \frac{2}{3}\sqrt{|x|^3} > \frac{1}{(\text{delta})}.$$

For large positive arguments, the relative error amplification is
considerable:

$$\frac{(\text{epsilon})}{(\text{delta})}\tilde{}\sqrt{x^3}.$$

```
This means a loss of roughly two decimal places accuracy for
arguments in the region of 20. However very large arguments are
not possible due to the danger of causing overflow and errors are
therefore limited in practice.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S17 -- Approximations of Special Functions                S17AJF
        S17AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17AJF returns a value of the derivative of the Airy function
Ai(x), via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17AJF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the derivative of the
Airy function Ai(x). It is based on a number of Chebyshev
expansions.

For x<-5,

                       4   [           b(t)     ]
              Ai'(x)=\/-x[a(t)cosz+ ------sinz],
```

```
                              [          (zeta)    ]


         (pi)                 2  /  3
where z= -----+(zeta), (zeta)= -\/ -x  and a(t) and b(t) are
          4                   3
                               3
                            ( 5)
expansions in variable t=-2( -) -1.
                            ( x)

For -5<=x<=0,

                                2
                        Ai'(x)=x f(t)-g(t),
                                          3
                                       ( x)
where f and g are expansions in t=-2( -) -1.
                                       ( 5)

For 0<x<4.5,

                                -11x/8
                        Ai'(x)=e      y(t),

                                  ( x)
where y(t) is an expansion in t=4( -)-1.
                                  ( 9)

For 4.5<=x<9,

                                -5x/2
                        Ai'(x)=e     v(t),

                                  ( x)
where v(t) is an expansion in t=4( -)-3.
                                  ( 9)

For x>=9,

                               4    -z
                        Ai'(x)=\/-xe  u(t),


         2  / 3                                  ( 18)
where z= -\/ x  and u(t) is an expansion in t=2( --)-1.
         3                                       ( z )

For |x|< the square of the machine precision, the result is set
```

directly to Ai'(0). This both saves time and avoids possible intermediate underflows.

For large negative arguments, it becomes impossible to calculate a result for the oscillating function with any accuracy and so

$$\text{the routine must fail. This occurs for } x < -\left(\frac{\pi}{\varepsilon}\right)^{4/7},$$

where (epsilon) is the machine precision.

For large positive arguments, where Ai' decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region

the function is essentially exponential in character and here relative error is needed. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

$$E \tilde{=} |x\ \mathrm{Ai}(x)|^2 (\mathrm{delta}) \qquad (\mathrm{epsilon}) \tilde{=} \left| \frac{x\ \mathrm{Ai}(x)^2}{\mathrm{Ai}'(x)} \right| (\mathrm{delta}).$$

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, positive or negative, errors are strongly attenuated by the function and hence will be roughly bounded by the machine precision.

For moderate to large negative x, the error, like the function, is oscillatory; however the amplitude of the error grows like

$$\frac{|x|^{7/4}}{\sqrt{\pi}}.$$

Therefore it becomes impossible to calculate the function with

$$\text{any accuracy if } |x|^{7/4} > \frac{\sqrt{\pi}}{(\mathrm{delta})}.$$

For large positive x, the relative error amplification is considerable:

$$\frac{(\mathrm{epsilon})}{(\mathrm{delta})} \tilde{=} \sqrt{x^3}.$$

However, very large arguments are not possible due to the danger of underflow. Thus in practice error amplification is limited.

8. Further Comments

```
None.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                  S17AKF
        S17AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17AKF returns a value for the derivative of the Airy function
Bi(x), via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S17AKF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine calculates an approximate value for the derivative
of the Airy function Bi(x). It is based on a number of Chebyshev
expansions.

For x<-5,

                        4   [            b(t)     ]
               Bi'(x)=\/-x[-a(t)sinz+ ------cosz],
                           [           (zeta)    ]



         (pi)                2  /  3
where z= ----+(zeta),(zeta)= -\/ -x  and a(t) and b(t) are
          4                  3
                                3
```

```
                                ( 5)
expansions in the variable t=-2( -) -1.
                                ( x)

For -5<=x<=0,

                                 2
                    Bi'(x)=\/3(x f(t)+g(t)),
                                       3
                                   ( x)
where f and g are expansions in t=-2( -) -1.
                                   ( 5)

For 0<x<4.5,

                                3x/2
                        Bi'(x)=e    y(t),

where y(t) is an expansion in t=4x/9-1.

For 4.5<=x<9,

                               21x/8
                      Bi'(x)=e     u(t),

where u(t) is an expansion in t=4x/9-3.

For x>=9,

                               4   z
                        Bi'(x)=\/xe v(t),



        2  / 3                                  ( 18)
where z= -\/ x  and v(t) is an expansion in t=2( --)-1.
        3                                       ( z )

For |x|< the square of the machine precision, the result is set
directly to Bi'(0). This saves time and avoids possible
underflows in calculation.

For large negative arguments, it becomes impossible to calculate
a result for the oscillating function with any accuracy so the
                                                    4/7
                                      (          )
                                      (  \/(pi)  )
routine must fail. This occurs for x<-( ---------)   , where
                                      ( (epsilon))
(epsilon) is the machine precision.
```

For large positive arguments, where Bi' grows in an essentially exponential manner, there is a danger of overflow so the routine must fail.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large and positive. On soft failure the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is appropriate. In the positive region the function has essentially exponential behaviour and hence relative error is needed. The absolute error, E, and the relative error (epsilon), are related in principle to the relative error in the argument (delta), by

$$E \approx |x\,\mathrm{Bi}(x)|\,(\mathrm{delta}) \qquad (\mathrm{epsilon}) \approx \left|\frac{x\,\mathrm{Bi}(x)}{\mathrm{Bi}'(x)}\right|(\mathrm{delta}).$$

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the

machine precision, the errors in the result will be somewhat larger.

For small x, positive or negative, errors are strongly attenuated by the function and hence will effectively be bounded by the machine precision.

For moderate to large negative x, the error is, like the function, oscillatory. However, the amplitude of the absolute

$$\text{error grows like } \frac{|x|^{7/4}}{\sqrt{\pi}}.$$

Therefore it becomes impossible to calculate the function with any accuracy if

$$|x|^{7/4} > \frac{\sqrt{\pi}}{\text{(delta)}}.$$

For large positive x, the relative error amplification is

$$\text{considerable: } \frac{\text{(epsilon)}}{\text{(delta)}} \tilde{\sqrt{x}}\,.$$

However, very large arguments are not possible due to the danger of overflow. Thus in practice the actual amplification that occurs is limited.

8. Further Comments

None.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

S17 -- Approximations of Special Functions S17DCF
S17DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details.

```
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17DCF returns a sequence of values for the Bessel functions
Y      (z) for complex z, non-negative (nu) and n=0,1,...,N-1,
 (nu)+n
with an option for exponential scaling.

2. Specification

       SUBROUTINE S17DCF (FNU, Z, N, SCALE, CY, NZ, CWRK, IFAIL)
       INTEGER              N, NZ, IFAIL
       DOUBLE PRECISION     FNU
       COMPLEX(KIND(1.0D0)) Z, CY(N), CWRK(N)
       CHARACTER*1          SCALE

3. Description

This subroutine evaluates a sequence of values for the Bessel
function Y    (z), where z is complex, -(pi) < arg z <= (pi), and
          (nu)
(nu) is the real, non-negative order. The N-member sequence is
generated for orders (nu), (nu)+1,...,(nu)+N-1. Optionally, the
                                    -|Im z|
sequence is scaled by the factor e       .

Note: although the routine may not be called with (nu) less than
zero, for negative orders the formula
Y     (z)=Y    (z)cos((pi)(nu))+J    (z)sin((pi)(nu)) may be used
 -(nu)     (nu)                 (nu)
(for the Bessel function J    (z), see S17DEF).
                          (nu)

The routine is derived from the routine CBESY in Amos [2]. It is
                                  (1)      (2)
                                 H    (z)-H    (z)
                                  (nu)     (nu)            (1)
based on the relation Y    (z)= -----------------, where H    (z)
                       (nu)            2i                 (nu)
     (2)
and H    (z) are the Hankel functions of the first and second
     (nu)
kinds respectively (see S17DLF).

When N is greater than 1, extra values of Y    (z) are computed
                                           (nu)
using recurrence relations.
```

```
For very large |z| or ((nu)+N-1), argument reduction will cause
total loss of accuracy, and so no computation is performed. For
slightly smaller |z| or ((nu)+N-1), the computation is performed
but results are accurate to less than half of machine precision.
If |z| is very small, near the machine underflow threshold, or
((nu)+N-1) is too large, there is a risk of overflow and so no
computation is performed. In all the above cases, a warning is
given by the routine.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

[2]   Amos D E (1986) Algorithm 644: A Portable Package for Bessel
      Functions of a Complex Argument and Nonnegative Order. ACM
      Trans. Math. Softw. 12 265--273.

5. Parameters

 1:  FNU -- DOUBLE PRECISION                                  Input
     On entry: the order, (nu), of the first member of the
     sequence of functions. Constraint: FNU >= 0.0.

 2:  Z -- COMPLEX(KIND(1.0D0))                                Input
     On entry: the argument, z, of the functions. Constraint: Z
     /= (0.0, 0.0).

 3:  N -- INTEGER                                             Input
     On entry: the number, N, of members required in the sequence
     Y    (z), Y      (z),...,Y        (z). Constraint: N >= 1.
      (nu)      (nu)+1         (nu)+N-1

 4:  SCALE -- CHARACTER*1                                     Input
     On entry: the scaling option.

     If SCALE = 'U', the results are returned unscaled.

     If SCALE = 'S', the results are returned scaled by the
             -|Imz|
     factor e      . Constraint: SCALE = 'U' or 'S'.

 5:  CY(N) -- COMPLEX(KIND(1.0D)) array                       Output
     On exit: the N required function values: CY(i) contains
     Y        (z), for i=1,2,...,N.
      (nu)+i-1

 6:  NZ -- INTEGER                                            Output
     On exit: the number of components of CY that are set to zero
     due to underflow. The positions of such components in the
```

```
     array CY are arbitrary.

 7:  CWRK(N) -- COMPLEX(KIND(1.0D)) array                Workspace

 8:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry FNU < 0.0,

     or       Z = (0.0, 0.0),

     or       N < 1,

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because ABS(Z) is less than a machine-dependent
     threshold value (given in the Users' Note for your
     implementation).

IFAIL= 3
     No computation has been performed due to the likelihood of
     overflow, because FNU + N - 1 is too large - how large
     depends on Z as well as the overflow threshold of the
     machine.

IFAIL= 4
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the results returned by S17DCF are accurate to less
     than half of machine precision. This error exit may occur if
     either ABS(Z) or FNU + N - 1 is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation).

IFAIL= 5
```

No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S17DCF would be lost. This error exit may occur if either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 6

No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DCF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DCF, the actual number of correct digits is limited, in general, by p-s, where s~max(1,|log |z||,|log (nu)|) represents the number of digits
          10   10
lost due to the argument reduction. Thus the larger the values of |z| and (nu), the less the precision in the result. If S17DCF is called with N>1, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DCF with different base values of (nu) and different N, the computed values may not agree exactly. Empirical tests with modest values of (nu) and z have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S17DCF is approximately proportional to the value of N, plus a constant. In general it is much cheaper to call S17DCF with N greater than 1, rather than to make N separate calls to S17DCF.

Paradoxically, for some values of z and (nu), it is cheaper to call S17DCF with a larger value of N than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different N, and the costs in each region may differ greatly.

```
Note that if the function required is Y (x) or Y (x), i.e., (nu)
                                       0        1
= 0.0 or 1.0, where x is real and positive, and only a single
unscaled function value is required, then it may be much cheaper
to call S17ACF or S17ADF respectively.

9. Example

The example program prints a caption and then proceeds to read
sets of data from the input data stream. The first datum is a
value for the order FNU, the second is a complex value for the
argument, Z, and the third is a value for the parameter SCALE.
The program calls the routine with N = 2 to evaluate the function
for orders FNU and FNU + 1, and it prints the results. The
process is repeated until the end of the input data stream is
encountered.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                S17DEF
        S17DEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17DEF returns a sequence of values for the Bessel functions
J      (z) for complex z, non-negative (nu) and n=0,1,...,N-1,
 (nu)+n
with an option for exponential scaling.

2. Specification

       SUBROUTINE S17DEF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
       INTEGER              N, NZ, IFAIL
       DOUBLE PRECISION     FNU
       COMPLEX(KIND(1.0D0)) Z, CY(N)
       CHARACTER*1          SCALE

3. Description

This subroutine evaluates a sequence of values for the Bessel
```

function J (z), where z is complex, -(pi) < arg z <= (pi), and
(nu)
(nu) is the real, non-negative order. The N-member sequence is
generated for orders (nu), (nu)+1,...,(nu)+N-1. Optionally, the
-|Im z|
sequence is scaled by the factor e .

Note: although the routine may not be called with (nu) less than
zero, for negative orders the formula
J (z)=J (z)cos((pi)(nu))-Y (z)sin((pi)(nu)) may be used
-(nu) (nu) (nu)
(for the Bessel function Y (z), see S17DCF).
(nu)

The routine is derived from the routine CBESJ in Amos [2]. It is
(nu)(pi)i/2
based on the relations J (z)=e I (-iz), Im z>=0.0
(nu) (nu)
-(nu)(pi)i/2
and J (z)=e I (iz), Im z<0.0.
(nu) (nu)

The Bessel function I (z) is computed using a variety of
(nu)
techniques depending on the region under consideration.

When N is greater than 1, extra values of J (z) are computed
(nu)
using recurrence relations.

For very large |z| or ((nu)+N-1), argument reduction will cause
total loss of accuracy, and so no computation is performed. For
slightly smaller |z| or ((nu)+N-1), the computation is performed
but results are accurate to less than half of machine precision.
If Im z is large, there is a risk of overflow and so no
computation is performed. In all the above cases, a warning is
given by the routine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical
Functions. Dover Publications.

[2] Amos D E (1986) Algorithm 644: A Portable Package for Bessel
Functions of a Complex Argument and Nonnegative Order. ACM
Trans. Math. Softw. 12 265--273.

5. Parameters

1: FNU -- DOUBLE PRECISION Input

```
     On entry: the order, (nu), of the first member of the
     sequence of functions. Constraint: FNU >= 0.0.

 2:  Z -- COMPLEX(KIND(1.0D0))                                  Input
     On entry: the argument z of the functions.

 3:  N -- INTEGER                                               Input
     On entry: the number, N, of members required in the sequence
     J    (z),J      (z),...,J        (z). Constraint: N >= 1.
      (nu)     (nu)+1          (nu)+N-1

 4:  SCALE -- CHARACTER*1                                       Input
     On entry: the scaling option.
          If SCALE = 'U', the results are returned unscaled.

          If SCALE = 'S', the results are returned scaled by the
                  -|Imz|
          factor e      .
     Constraint: SCALE = 'U' or 'S'.

 5:  CY(N) -- COMPLEX(KIND(1.0D)) array                         Output
     On exit: the N required function values: CY(i) contains
     J        (z), for i=1,2,...,N.
      (nu)+i-1

 6:  NZ -- INTEGER                                              Output
     On exit: the number of components of CY that are set to zero
     due to underflow. If NZ > 0, then elements CY(N-NZ+1), CY(N-
     NZ+2),...,CY(N) are set to zero.

 7:  IFAIL -- INTEGER                                   Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry FNU < 0.0,

     or       N < 1,
```

```
     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because Im Z is larger than a machine-dependent
     threshold value (given in the Users' Note for your
     implementation). This error exit can only occur when SCALE =
     'U'.

IFAIL= 3
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the results returned by S17DEF are accurate to less
     than half of machine precision. This error exit may occur if
     either ABS(Z) or FNU + N - 1 is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation).

IFAIL= 4
     No computation has been performed because the errors due to
     argument reduction in elementary functions mean that all
     precision in results returned by S17DEF would be lost. This
     error exit may occur when either ABS(Z) or FNU + N - 1 is
     greater than a machine-dependent threshold value (given in
     the Users' Note for your implementation).

IFAIL= 5
     No results are returned because the algorithm termination
     condition has not been met. This may occur because the
     parameters supplied to S17DEF would have caused overflow or
     underflow.

7. Accuracy

All constants in subroutine S17DEF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is
limited by p=min(t,18). Because of errors in argument reduction
when computing elementary functions inside S17DEF, the actual
number of correct digits is limited, in general, by p-s, where
s~max(1,|log  |(|z|),|log  (nu)|) represents the number of digits
           10           10
lost due to the argument reduction. Thus the larger the values of
|z| and (nu), the less the precision in the result. If S17DEF is
called with N>1, then computation of function values via
recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are
computed by calls to S17DEF with different base values of (nu)
```

```
and different N, the computed values may not agree exactly.
Empirical tests with modest values of (nu) and z have shown that
the discrepancy is limited to the least significant 3-4 digits of
precision.

8. Further Comments

The time taken by the routine for a call of S17DEF is
approximately proportional to the value of N, plus a constant. In
general it is much cheaper to call S17DEF with N greater than 1,
rather than to make N separate calls to S17DEF.

Paradoxically, for some values of z and (nu), it is cheaper to
call S17DEF with a larger value of N than is required, and then
discard the extra function values returned. However, it is not
possible to state the precise circumstances in which this is
likely to occur. It is due to the fact that the base value used
to start recurrence may be calculated in different regions for
different N, and the costs in each region may differ greatly.

Note that if the function required is J (x) or J (x), i.e., (nu)
                                      0        1
= 0.0 or 1.0, where x is real and positive, and only a single
unscaled function value is required, then it may be much cheaper
to call S17AEF or S17AFF respectively.

9. Example

The example program prints a caption and then proceeds to read
sets of data from the input data stream. The first datum is a
value for the order FNU, the second is a complex value for the
argument, Z, and the third is a value for the parameter SCALE.

The program calls the routine with N = 2 to evaluate the function
for orders FNU and FNU + 1, and it prints the results. The
process is repeated until the end of the input data stream is
encountered.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                S17DGF
       S17DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
```

not included in the Foundation Library.

1. Purpose

S17DGF returns the value of the Airy function Ai(z) or its derivative Ai'(z) for complex z, with an option for exponential scaling.

2. Specification

```
SUBROUTINE S17DGF (DERIV, Z, SCALE, AI, NZ, IFAIL)
INTEGER              NZ, IFAIL
COMPLEX(KIND(1.0D0)) Z, AI
CHARACTER*1          DERIV, SCALE
```

3. Description

This subroutine returns a value for the Airy function Ai(z) or its derivative Ai'(z), where z is complex, -(pi) < arg z <= (pi).

Optionally, the value is scaled by the factor $e^{2z\sqrt{z}/3}$.

The routine is derived from the routine CAIRY in Amos [2]. It is based on the relations

$$Ai(z)= \frac{\sqrt{zK_{1/3}(w)}}{(pi)\sqrt{3}}, \text{ and } Ai'(z)= \frac{-zK_{2/3}(w)}{(pi)/3},$$

where $K_{(nu)}$ is the modified Bessel function and $w=2z\sqrt{z}/3$.

For very large |z|, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller |z|, the computation is performed but results are accurate to less than half of machine precision. If Re w is too large, and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Amos D E (1986) Algorithm 644: A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order. ACM Trans. Math. Softw. 12 265--273.

5. Parameters

1: DERIV -- CHARACTER*1 Input
On entry: specifies whether the function or its derivative is required.
If DERIV = 'F', Ai(z) is returned.

If DERIV = 'D', Ai'(z) is returned.
Constraint: DERIV = 'F' or 'D'.

2: Z -- COMPLEX(KIND(1.0D0)) Input
On entry: the argument z of the function.

3: SCALE -- CHARACTER*1 Input
On entry: the scaling option.

If SCALE = 'U', the result is returned unscaled.

If SCALE = 'S', the result is returned scaled by the factor

$$e^{2z\sqrt{z}/3}$$

. Constraint: SCALE = 'U' or 'S'.

4: AI -- COMPLEX(KIND(1.0D0)) Output
On exit: the required function or derivative value.

5: NZ -- INTEGER Output
On exit: NZ indicates whether or not AI is set to zero due to underflow. This can only occur when SCALE = 'U'.

If NZ = 0, AI is not set to zero.

If NZ = 1, AI is set to zero.

6: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

```
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry DERIV /= 'F' or 'D'.

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of


     overflow, because Re w is too large, where w=2Z\/Z/3 -- how
     large depends on Z and the overflow threshold of the
     machine. This error exit can only occur when SCALE = 'U'.

IFAIL= 3
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the result returned by S17DGF is accurate to less than
     half of machine precision. This error exit may occur if ABS
     (Z) is greater than a machine-dependent threshold value
     (given in the Users' Note for your implementation).

IFAIL= 4
     No computation has been performed because the errors due to
     argument reduction in elementary functions mean that all
     precision in the result returned by S17DGF would be lost.
     This error exit may occur if ABS(Z) is greater than a
     machine-dependent threshold value (given in the Users' Note
     for your implementation).

IFAIL= 5
     No result is returned because the algorithm termination
     condition has not been met. This may occur because the
     parameters supplied to S17DGF would have caused overflow or
     underflow.

7. Accuracy

All constants in subroutine S17DGF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is
limited by p=min(t,18). Because of errors in argument reduction
when computing elementary functions inside S17DGF, the actual
number of correct digits is limited, in general, by p-s, where
s~max(1,|log  |z||) represents the number of digits lost due to
```

```
            10
the argument reduction. Thus the larger the value of |z|, the
less the precision in the result.

Empirical tests with modest values of z, checking relations
between Airy functions Ai(z), Ai'(z), Bi(z) and Bi'(z), have
shown errors limited to the least significant 3-4 digits of
precision.

8. Further Comments

Note that if the function is required to operate on a real
argument only, then it may be much cheaper to call S17AGF or
S17AJF.

9. Example

The example program prints a caption and then proceeds to read
sets of data from the input data stream. The first datum is a
value for the parameter DERIV, the second is a complex value for
the argument, Z, and the third is a value for the parameter
SCALE. The program calls the routine and prints the results. The
process is repeated until the end of the input data stream is
encountered.


The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions                S17DHF
        S17DHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17DHF returns the value of the Airy function Bi(z) or its
derivative Bi'(z) for complex z, with an option for exponential
scaling.

2. Specification

       SUBROUTINE S17DHF (DERIV, Z, SCALE, BI, IFAIL)
       INTEGER              IFAIL
```

```
      COMPLEX(KIND(1.0D0)) Z, BI
      CHARACTER*1          DERIV, SCALE
```

3. Description

This subroutine returns a value for the Airy function Bi(z) or its derivative Bi'(z), where z is complex, -(pi) < argz <= (pi).

Optionally, the value is scaled by the factor $e^{|\mathrm{Re}\,(2z\sqrt{z}/3)|}$.

The routine is derived from the routine CBIRY in Amos [2]. It is based on the relations $Bi(z)= \frac{\sqrt{z}}{3}(I_{-1/3}(w)+I_{1/3}(w))$, and $Bi'(z)= \frac{z}{\sqrt{3}}(I_{-2/3}(w)+I_{2/3}(w))$, where $I_{(nu)}$ is the modified Bessel function and $w=2z\sqrt{z}/3$.

For very large |z|, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller |z|, the computation is performed but results are accurate to less than half of machine precision. If Re z is too large, and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Hammersley J M and Handscomb D C (1967) Monte-Carlo Methods. Methuen.

5. Parameters

1: DERIV -- CHARACTER*1 Input
On entry: specifies whether the function or its derivative is required.
If DERIV = 'F', Bi(z) is returned.

```
          If DERIV = 'D', Bi'(z) is returned.
     Constraint: DERIV = 'F' or 'D'.

 2:  Z -- COMPLEX(KIND(1.0D0))                                  Input
     On entry: the argument z of the function.

 3:  SCALE -- CHARACTER*1                                       Input
     On entry: the scaling option.
          If SCALE = 'U', the result is returned unscaled.

          If SCALE = 'S', the result is returned scaled by the


                  |Re(2z\/z/3)|
          factor e             .
     Constraint: SCALE = 'U' or 'S'.

 4:  BI -- COMPLEX(KIND(1.0D0))                                 Output
     On exit: the required function or derivative value.

 5:  IFAIL -- INTEGER                                       Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry DERIV /= 'F' or 'D'.

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because real(Z) is too large - how large depends
     on the overflow threshold of the machine. This error exit
     can only occur when SCALE = 'U'.

IFAIL= 3
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the result returned by S17DHF is accurate to less than
```

half of machine precision. This error exit may occur if ABS (Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in the result returned by S17DHF would be lost. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No result is returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DHF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DHF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DHF, the actual number of correct digits is limited, in general, by p-s, where s~max(1,|log |z||) represents the number of digits lost due to
            10
the argument reduction. Thus the larger the value of |z|, the less the precision in the result.

Empirical tests with modest values of z, checking relations between Airy functions Ai(z), Ai'(z), Bi(z) and Bi'(z), have shown errors limited to the least significant 3-4 digits of precision.

8. Further Comments

Note that if the function is required to operate on a real argument only, then it may be much cheaper to call S17AHF or S17AKF.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the parameter DERIV, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine and prints the results. The

```
process is repeated until the end of the input data stream is
encountered.


The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S17 -- Approximations of Special Functions                S17DLF
        S17DLF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S17DLF returns a sequence of values for the Hankel functions
 (1)          (2)
H      (z) or H      (z) for complex z, non-negative (nu) and
 (nu)+n       (nu)+n
n=0,1,...,N-1, with an option for exponential scaling.

2. Specification

       SUBROUTINE S17DLF (M, FNU, Z, N, SCALE, CY, NZ, IFAIL)
       INTEGER              M, N, NZ, IFAIL
       DOUBLE PRECISION     FNU
       COMPLEX(KIND(1.0D0)) Z, CY(N)
       CHARACTER*1          SCALE

3. Description

This subroutine evaluates a sequence of values for the Hankel
          (1)         (2)
function H    (z) or H    (z), where z is complex, -(pi) < argz
          (nu)        (nu)
<= (pi), and (nu) is the real, non-negative order. The N-member
sequence is generated for orders (nu), (nu)+1,...,(nu)+N-1.
                                                      -iz
Optionally, the sequence is scaled by the factor e    if the
             (1)                       iz
function is H    (z) or by the factor e   if the function is
             (nu)
 (2)
H    (z).
 (nu)
```

Note: although the routine may not be called with (nu) less than zero, for negative orders the formulae
 (1)   (nu)(pi)i (1)   (2)   -(nu)(pi)i (2)
H   (z)=e   H   (z), and H   (z)=e   H   (z)
 -(nu)   (nu)   -(nu)   (nu)
may be used.

The routine is derived from the routine CBESH in Amos [2]. It is based on the relation

$$H^{(m)}_{(nu)}(z) = -e^{1 -p(nu)}\, K^{-p}_{(nu)}(ze^{\ }),$$

where $p = i\,\frac{(pi)}{2}$ if $m=1$ and $p=-i\,\frac{(pi)}{2}$ if $m=2$, and the Bessel function $K^{(nu)}_{\ }(z)$ is computed in the right half-plane only. Continuation of $K^{(nu)}_{\ }(z)$ to the left half-plane is computed in terms of the Bessel function $I^{(nu)}_{\ }(z)$. These functions are evaluated using a variety of different techniques, depending on the region under consideration.

When N is greater than 1, extra values of $H^{(m)}_{(nu)}(z)$ are computed using recurrence relations.

For very large |z| or ((nu)+N-1), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller |z| or ((nu)+N-1), the computation is performed but results are accurate to less than half of machine precision. If |z| is very small, near the machine underflow threshold, or ((nu)+N-1) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Amos D E (1986) Algorithm 644: A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order. ACM Trans. Math. Softw. 12 265--273.

5. Parameters

1: M -- INTEGER Input
On entry: the kind of functions required.
(1)
If M = 1, the functions are H (z).
(nu)

(2)
If M = 2, the functions are H (z).
(nu)
Constraint: M = 1 or 2.

2: FNU -- DOUBLE PRECISION Input
On entry: the order, (nu), of the first member of the sequence of functions. Constraint: FNU >= 0.0.

3: Z -- COMPLEX(KIND(1.0D0)) Input
On entry: the argument z of the functions. Constraint: Z /= (0.0, 0.0).

4: N -- INTEGER Input
On entry: the number, N, of members required in the sequence
(M) (M) (M)
H , H ,..., H . Constraint: N >= 1.
(nu) (nu)+1 (nu)+N-1

5: SCALE -- CHARACTER*1 Input
On entry: the scaling option.
If SCALE = 'U', the results are returned unscaled.

If SCALE = 'S', the results are returned scaled by the
-iz iz
factor e when M = 1, or by the factor e when M = 2.
Constraint: SCALE = 'U' or 'S'.

6: CY(N) -- COMPLEX(KIND(1.0D)) array Output
On exit: the N required function values: CY(i) contains
(M)
H , for i=1,2,...,N.
(nu)+i-1

7: NZ -- INTEGER Output
On exit: the number of components of CY that are set to zero due to underflow. If NZ > 0, then if Imz>0.0 and M = 1, or Imz<0.0 and M = 2, elements CY(1), CY(2),...,CY(NZ) are set to zero. In the complementary half-planes, NZ simply states the number of underflows, and not which elements they are.

```
 8:  IFAIL -- INTEGER                                    Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M /= 1 and M /= 2,

     or       FNU < 0.0,

     or       Z = (0.0, 0.0),

     or       N < 1,

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because ABS(Z) is less than a machine-dependent
     threshold value (given in the Users' Note for your
     implementation).

IFAIL= 3
     No computation has been performed due to the likelihood of
     overflow, because FNU + N - 1 is too large - how large
     depends on Z and the overflow threshold of the machine.

IFAIL= 4
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the results returned by S17DLF are accurate to less
     than half of machine precision. This error exit may occur if
     either ABS(Z) or FNU + N - 1 is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation).

IFAIL= 5
     No computation has been performed because the errors due to
     argument reduction in elementary functions mean that all
     precision in results returned by S17DLF would be lost. This
```

```
     error exit may occur when either of ABS(Z) or FNU + N - 1 is
     greater than a machine-dependent threshold value (given in
     the Users' Note for your implementation).

IFAIL= 6
     No results are returned because the algorithm termination
     condition has not been met. This may occur because the
     parameters supplied to S17DLF would have caused overflow or
     underflow.

7. Accuracy

All constants in subroutine S17DLF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is
limited by p=min(t,18). Because of errors in argument reduction
when computing elementary functions inside S17DLF, the actual
number of correct digits is limited, in general, by p-s, where
s~max(1,|log  |z||,|log  (nu)|) represents the number of digits
            10         10
lost due to the argument reduction. Thus the larger the values of
|z| and (nu), the less the precision in the result. If S17DLF is
called with N>1, then computation of function values via
recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are
computed by calls to S17DLF with different base values of (nu)
and different N, the computed values may not agree exactly.
Empirical tests with modest values of (nu) and z have shown that
the discrepancy is limited to the least significant 3-4 digits of
precision.

8. Further Comments

The time taken by the routine for a call of S17DLF is
approximately proportional to the value of N, plus a constant. In
general it is much cheaper to call S17DLF with N greater than 1,
rather than to make N separate calls to S17DLF.

Paradoxically, for some values of z and (nu), it is cheaper to
call S17DLF with a larger value of N than is required, and then
discard the extra function values returned. However, it is not
possible to state the precise circumstances in which this is
likely to occur. It is due to the fact that the base value used
to start recurrence may be calculated in different regions for
different N, and the costs in each region may differ greatly.

9. Example
```

```
          The example program prints a caption and then proceeds to read
          sets of data from the input data stream. The first datum is a
          value for the kind of function, M, the second is a value for the
          order FNU, the third is a complex value for the argument, Z, and
          the fourth is a value for the parameter SCALE. The program calls
          the routine with N = 2 to evaluate the function for orders FNU
          and FNU + 1, and it prints the results. The process is repeated
          until the end of the input data stream is encountered.

          The example program is not reproduced here. The source code for
          all example programs is distributed with the NAG Foundation
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          S18 -- Approximations of Special Functions                 S18ACF
                  S18ACF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          S18ACF returns the value of the modified Bessel Function K (x),
                                                                    0
          via the routine name.

          2. Specification

                 DOUBLE PRECISION FUNCTION S18ACF (X, IFAIL)
                 INTEGER          IFAIL
                 DOUBLE PRECISION X

          3. Description

          This routine evaluates an approximation to the modified Bessel
          Function of the second kind K (x).
                                       0

          Note: K (x) is undefined for x<=0 and the routine will fail for
                 0
          such arguments.

          The routine is based on five Chebyshev expansions:

          For 0<x<=1,

                           --'          --'                    2
```

```
      K (x)=-lnx >   a T (t)+ >   b T (t) , where t=2x -1;
       0         --   r r     --   r r
                 r=0          r=0
```

For 1<x<=2,

```
                    -x --'
                K (x)=e   >   c T (t) , where t=2x-3;
                 0        --   r r
                          r=0
```

For 2<x<=4,

```
                    -x --'
                K (x)=e   >   d T (t) , where t=x-3;
                 0        --   r r
                          r=0
```

For x>4,

```
                     -x
                    e   --'                      9-x
                K (x)= --- >   e T (t) , where t= ---.
                 0         --   r r              1+x
                       \/x r=0
```

For x near zero, $K_0(x) \tilde{=} -(\gamma) - \ln(x/2)$, where (γ) denotes Euler's constant. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For large x, where there is a danger of underflow due to the smallness of K_0, the result is set exactly to zero.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential

```
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     X <= 0.0, K  is undefined. On soft failure the routine
               0
     returns zero.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument
and result respectively.

If (delta) is somewhat larger than the machine precision (i.e.,
if (delta) is due to data errors etc), then (epsilon) and (delta)
are approximately related by:

                              | xK (x)|
                              |   1   |
                   (epsilon)~=| ------|(delta).
                              | K (x) |
                              |  0    |

Figure 1 shows the behaviour of the error amplification factor

                             | xK (x)|
                             |   1   |
                             | ------|.
                             | K (x) |
                             |  0    |

                            Figure 1
          Please see figure in printed Reference Manual

However, if (delta) is of the same order as machine precision,
then rounding errors could make (epsilon) slightly larger than
the above relation predicts.

                                                          |  1 |
For small x, the amplification factor is approximately    | ---|,
                                                          | lnx|
which implies strong attenuation of the error, but in general
(epsilon) can never be less than the machine precision.
```

```
For large x, (epsilon)~=x(delta) and we have strong amplification
of the relative error. Eventually K , which is asymptotically
                                   0
          -x
         e
given by  ---, becomes so small that it cannot be calculated


         \/x
without underflow and hence the routine will return zero. Note
that for large x the errors will be dominated by those of the
Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the appropriate
the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
S18 -- Approximations of Special Functions                S18ADF
        S18ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S18ADF returns the value of the modified Bessel Function K (x),
                                                          1
via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S18ADF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description
```

```
This routine evaluates an approximation to the modified Bessel
Function of the second kind K (x).
                             1

Note: K (x) is undefined for x<=0 and the routine will fail for
       1
such arguments.

The routine is based on five Chebyshev expansions:

For 0<x<=1,

           1      --'           --'                     2
    K (x)= -+xlnx >   a T (t)-x >   b T (t) , where t=2x -1;
     1     x      --   r r      --   r r
                  r=0           r=0

For 1<x<=2,

                      -x --'
              K (x)=e   >   c T (t) , where t=2x-3;
               1        --   r r
                        r=0

For 2<x<=4,

                      -x --'
              K (x)=e   >   d T (t) , where t=x-3;
               1        --   r r
                        r=0

For x>4,

                      -x
                     e   --'                    9-x
             K (x)= --- >   e T (t) , where t= ---.
              1         --   r r               1+x
                    \/x r=0

                         1
For x near zero, K (x)~= -. This approximation is used when x is
                 1       x
sufficiently small for the result to be correct to machine
precision. For very small x on some machines, it is impossible to
           1
calculate  - without overflow and the routine must fail.
           x

For large x, where there is a danger of underflow due to the
```

smallness of K , the result is set exactly to zero.
              1

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X > 0.0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X <= 0.0, K is undefined. On soft failure the routine
           1
returns zero.

IFAIL= 2
X is too small, there is a danger of overflow. On soft failure the routine returns approximately the largest representable value.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

$$(\text{epsilon}) \approx \left| \frac{x K_0(x) - K_1(x)}{K_1(x)} \right| (\text{delta}).$$

Figure 1 shows the behaviour of the error amplification factor

$$\left| \frac{xK_0(x) - K_1(x)}{K_1(x)} \right|.$$

Figure 1
Please see figure in printed Reference Manual

However if (delta) is of the same order as the machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, (epsilon)~=(delta) and there is no amplification of errors.

For large x, (epsilon)~=x(delta) and we have strong amplification of the relative error. Eventually K_1, which is asymptotically given by $\frac{e^{-x}}{\sqrt{x}}$, becomes so small that it cannot be calculated without underflow and hence the routine will return zero. Note that for large x the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

S18 -- Approximations of Special Functions S18AEF
S18AEF -- NAG Foundation Library Routine Document

```
Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S18AEF returns the value of the modified Bessel Function I (x),
                                                          0
via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S18AEF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the modified Bessel
Function of the first kind I (x).
                            0

Note: I (-x)=I (x), so the approximation need only consider x>=0.
       0      0

The routine is based on three Chebyshev expansions:

For 0<x<=4,

                    x --'                    ( x)
             I (x)=e  >   a T (t) , where t=2( -)-1;
              0       --   r r              ( 4)
                      r=0

For 4<x<=12,

                     x --'                    x-8
              I (x)=e  >   b T (t) , where t= ---;
               0       --   r r                4
                       r=0

For x>12,

                    x
                   e   --'                   ( 12)
           I (x)= --- >   c T (t) , where t=2( --)-1.
            0         --   r r              ( x )
                     \/x r=0
```

For small x, $I_0(x) \tilde{=} 1$. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For large x, the routine must fail because of the danger of overflow in calculating e^x.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large. On soft failure the routine returns the approximate value of $I_0(x)$ at the nearest valid argument.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

$$|\text{epsilon}| \tilde{=} \left| \frac{x I_1(x)}{I_0(x)} \right| |\text{delta}|.$$

```
                              |  0    |
```

Figure 1 shows the behaviour of the error amplification factor

```
                             | xI (x)|
                             |   1   |
                             | ------|.
                             | I (x) |
                             |  0    |
```

Figure 1
Please see figure in printed Reference Manual

However if (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x the amplification factor is approximately $\frac{x^2}{2}$, which implies strong attenuation of the error, but in general (epsilon) can never be less than the machine precision.

For large x, (epsilon)~=x(delta) and we have strong amplification of errors. However the routine must fail for quite moderate values of x, because $I_0(x)$ would overflow; hence in practice the loss of accuracy for large x is not excessive. Note that for large x the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%

S18 -- Approximations of Special Functions S18AFF
S18AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18AFF returns a value for the modified Bessel Function $I_1(x)$, via the routine name.

2. Specification

```
      DOUBLE PRECISION FUNCTION S18AFF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the modified Bessel Function of the first kind $I_1(x)$.

Note: $I_1(-x)=-I_1(x)$, so the approximation need only consider $x\geq 0$

The routine is based on three Chebyshev expansions:

For $0<x\leq 4$,

$$I_1(x)=x\sum_{r=0}^{\prime\prime} a_r T_r(t), \text{ where } t=2\left(\frac{x}{4}\right)^2-1;$$

For $4<x\leq 12$,

$$I_1(x)=e^x\sum_{r=0}^{\prime\prime} b_r T_r(t), \text{ where } t=\frac{x-8}{4};$$

For $x>12$,

$$I_1(x)=\frac{e^x}{\sqrt{x}}\sum_{r=0}^{\prime\prime} c_r T_r(t), \text{ where } t=2\left(\frac{12}{x}\right)-1.$$

For small x, I (x)~=x. This approximation is used when x is
              1
sufficiently small for the result to be correct to machine
precision.

For large x, the routine must fail because I (x) cannot be
                                            1
represented without overflow.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical
Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large. On soft failure the routine returns the approximate value of I (x) at the nearest valid argument.
                                                 1

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

$$|\text{epsilon}| \approx \left| \frac{x I_0(x) - I_1(x)}{I_1(x)} \right| |\text{delta}|.$$

```
                              |     1       |
```

Figure 1 shows the behaviour of the error amplification factor

```
                            | xI (x)-I (x)|
                            |   0     1   |
                            | ------------|,
                            |    I (x)    |
                            |     1       |
```

Figure 1
Please see figure in printed Reference Manual

However if (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, (epsilon)~=(delta) and there is no amplification of errors.

For large x, (epsilon)~=x(delta) and we have strong amplification of errors. However the routine must fail for quite moderate values of x because I (x) would overflow; hence in practice the
                      1
loss of accuracy for large x is not excessive. Note that for large x, the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%

S18 -- Approximations of Special Functions S18DCF
S18DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is

```
not included in the Foundation Library.

1. Purpose

S18DCF returns a sequence of values for the modified Bessel
functions K      (z) for complex z, non-negative (nu) and
           (nu)+n
n=0,1,...,N-1, with an option for exponential scaling.

2. Specification

       SUBROUTINE S18DCF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
       INTEGER              N, NZ, IFAIL
       DOUBLE PRECISION     FNU
       COMPLEX(KIND(1.0D0)) Z, CY(N)
       CHARACTER*1          SCALE

3. Description

This subroutine evaluates a sequence of values for the modified
Bessel function K    (z), where z is complex, -(pi) < arg z <=
                (nu)
(pi), and (nu) is the real, non-negative order. The N-member
sequence is generated for orders (nu), (nu)+1,...,(nu)+N-1.
                                                        z
Optionally, the sequence is scaled by the factor e .

The routine is derived from the routine CBESK in Amos [2].

Note: although the routine may not be called with (nu) less than
zero, for negative orders the formula K     (z)=K    (z) may be
                                       -(nu)     (nu)
used.

When N is greater than 1, extra values of K    (z) are computed
                                           (nu)
using recurrence relations.

For very large |z| or ((nu)+N-1), argument reduction will cause
total loss of accuracy, and so no computation is performed. For
slightly smaller |z| or ((nu)+N-1), the computation is performed
but results are accurate to less than half of machine precision.
If |z| is very small, near the machine underflow threshold, or
((nu)+N-1) is too large, there is a risk of overflow and so no
computation is performed. In all the above cases, a warning is
given by the routine.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
```

Functions. Dover Publications.

[2] Amos D E (1986) Algorithm 644: A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order. ACM Trans. Math. Softw. 12 265--273.

5. Parameters

1: FNU -- DOUBLE PRECISION Input
On entry: the order, (nu), of the first member of the sequence of functions. Constraint: FNU >= 0.0.

2: Z -- COMPLEX(KIND(1.0D0)) Input
On entry: the argument z of the functions. Constraint: Z /= (0.0, 0.0).

3: N -- INTEGER Input
On entry: the number, N, of members required in the sequence $K_{(nu)}(z), K_{(nu)+1}(z),\ldots, K_{(nu)+N-1}(z)$. Constraint: N >= 1.

4: SCALE -- CHARACTER*1 Input
On entry: the scaling option.

If SCALE = 'U', the results are returned unscaled.

If SCALE = 'S', the results are returned scaled by the factor e^{z}. Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D)) array Output
On exit: the N required function values: CY(i) contains $K_{(nu)+i-1}(z)$, for i=1,2,...,N.

6: NZ -- INTEGER Output
On exit: the number of components of CY that are set to zero due to underflow. If NZ > 0 and Rez>=0.0, elements CY(1),CY(2),...,CY(NZ) are set to zero. If Rez<0.0, NZ simply states the number of underflows, and not which elements they are.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

```
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry FNU < 0.0,

     or       Z = (0.0, 0.0),

     or       N < 1,

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because ABS(Z) is less than a machine-dependent
     threshold value (given in the Users' Note for your
     implementation).

IFAIL= 3
     No computation has been performed due to the likelihood of
     overflow, because FNU + N - 1 is too large - how large
     depends on Z and the overflow threshold of the machine.

IFAIL= 4
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the results returned by S18DCF are accurate to less
     than half of machine precision. This error exit may occur if
     either ABS(Z) or FNU + N - 1 is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation).

IFAIL= 5
     No computation has been performed because the errors due to
     argument reduction in elementary functions mean that all
     precision in results returned by S18DCF would be lost. This
     error exit may occur when either ABS(Z) or FNU + N - 1 is
     greater than a machine-dependent threshold value (given in
     the Users' Note for your implementation).

IFAIL= 6
     No results are returned because the algorithm termination
     condition has not been met. This may occur because the
     parameters supplied to S18DCF would have caused overflow or
     underflow.
```

7. Accuracy

All constants in subroutine S18DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S18DCF, the actual number of correct digits is limited, in general, by p-s, where s~max(1,|log |z||,|log (nu)|) represents the number of digits
           10   10
lost due to the argument reduction. Thus the larger the values of |z| and (nu), the less the precision in the result. If S18DCF is called with N>1, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S18DCF with different base values of (nu) and different N, the computed values may not agree exactly. Empirical tests with modest values of (nu) and z have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S18DCF is approximately proportional to the value of N, plus a constant. In general it is much cheaper to call S18DCF with N greater than 1, rather than to make N separate calls to S18DCF.

Paradoxically, for some values of z and (nu), it is cheaper to call S18DCF with a larger value of N than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different N, and the costs in each region may differ greatly.

Note that if the function required is K (x) or K (x), i.e.,
                                      0   1
(nu)=0.0 or 1.0, where x is real and positive, and only a single function value is required, then it may be much cheaper to call S18ACF, S18ADF, S18CCF(*) or S18CDF(*), depending on whether a scaled result is required or not.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a

```
value for the order FNU, the second is a complex value for the
argument, Z, and the third is a value for the parameter SCALE.
The program calls the routine with N = 2 to evaluate the function
for orders FNU and FNU + 1, and it prints the results. The
process is repeated until the end of the input data stream is
encountered.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S18 -- Approximations of Special Functions                  S18DEF
        S18DEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S18DEF returns a sequence of values for the modified Bessel
functions I      (z) for complex z, non-negative (nu) and
           (nu)+n
n=0,1,...,N-1, with an option for exponential scaling.

2. Specification

       SUBROUTINE S18DEF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
       INTEGER              N, NZ, IFAIL
       DOUBLE PRECISION     FNU
       COMPLEX(KIND(1.0D0)) Z, CY(N)
       CHARACTER*1          SCALE

3. Description

This subroutine evaluates a sequence of values for the modified
Bessel function I    (z), where z is complex, -(pi) < argz <=
                (nu)
(pi), and (nu) is the real, non-negative order. The N-member
sequence is generated for orders (nu), (nu)+1,...,(nu)+N-1.
                                                      -|Rez|
Optionally, the sequence is scaled by the factor e      .

The routine is derived from the routine CBESI in Amos [2].

Note: although the routine may not be called with (nu) less than
zero, for negative orders the formula
```

$$I_{-(nu)}(z) = I_{(nu)}(z) + \frac{2}{(pi)}\sin((pi)(nu))K_{(nu)}(z)$$

may be used (for the Bessel function $K_{(nu)}(z)$, see S18DCF).

When N is greater than 1, extra values of $I_{(nu)}(z)$ are computed using recurrence relations.

For very large |z| or ((nu)+N-1), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller |z| or ((nu)+N-1), the computation is performed but results are accurate to less than half of machine precision. If Re(z) is too large and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Amos D E (1986) Algorithm 644: A Portable Package for Bessel Functions of a Complex Argument and Nonnegative Order. ACM Trans. Math. Softw. 12 265--273.

5. Parameters

1: FNU -- DOUBLE PRECISION Input
On entry: the order, (nu), of the first member of the sequence of functions. Constraint: FNU >= 0.0.

2: Z -- COMPLEX(KIND(1.0D0)) Input
On entry: the argument z of the functions.

3: N -- INTEGER Input
On entry: the number, N, of members required in the sequence $I_{(nu)}(z), I_{(nu)+1}(z), \ldots, I_{(nu)+N-1}(z)$. Constraint: N >= 1.

4: SCALE -- CHARACTER*1 Input
On entry: the scaling option.
If SCALE = 'U', the results are returned unscaled.

If SCALE = 'S', the results are returned scaled by the factor $e^{-|Rez|}$.

Constraint: SCALE = 'U' or 'S'.

```
 5:  CY(N) -- COMPLEX(KIND(1.0D)) array                    Output
     On exit: the N required function values: CY(i) contains
     I        (z), for i=1,2,...,N.
      (nu)+i-1

 6:  NZ -- INTEGER                                         Output
     On exit: the number of components of CY that are set to zero
     due to underflow.

     If NZ > 0, then elements CY(N-NZ+1),CY(N-NZ+2),...,CY(N) are
     set to zero.

 7:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry FNU < 0.0,

     or       N < 1,

     or       SCALE /= 'U' or 'S'.

IFAIL= 2
     No computation has been performed due to the likelihood of
     overflow, because real(Z) is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation). This error exit can only occur when
     SCALE = 'U'.

IFAIL= 3
     The computation has been performed, but the errors due to
     argument reduction in elementary functions make it likely
     that the results returned by S18DEF are accurate to less
     than half of machine precision. This error exit may occur
     when either ABS(Z) or FNU + N - 1 is greater than a machine-
     dependent threshold value (given in the Users' Note for
     your implementation).
```

```
IFAIL= 4
     No computation has been performed because the errors due to
     argument reduction in elementary functions mean that all
     precision in results returned by S18DEF would be lost. This
     error exit may occur when either ABS(Z) or FNU + N - 1 is
     greater than a machine-dependent threshold value (given in
     the Users' Note for your implementation).

IFAIL= 5
     No results are returned because the algorithm termination
     condition has not been met. This may occur because the
     parameters supplied to S18DEF would have caused overflow or
     underflow.

7. Accuracy

All constants in subroutine S18DEF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is
limited by p=min(t,18). Because of errors in argument reduction
when computing elementary functions inside S18DEF, the actual
number of correct digits is limited, in general, by p-s, where
s~max(1,|log  |z||,|log  (nu)|) represents the number of digits
            10         10
lost due to the argument reduction. Thus the larger the values of
|z| and (nu), the less the precision in the result. If S18DEF is
called with N>1, then computation of function values via
recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are
computed by calls to S18DEF with different base values of (nu)
and different N, the computed values may not agree exactly.
Empirical tests with modest values of (nu) and z have shown that
the discrepancy is limited to the least significant 3-4 digits of
precision.

8. Further Comments

The time taken by the routine for a call of S18DEF is
approximately proportional to the value of N, plus a constant. In
general it is much cheaper to call S18DEF with N greater than 1,
rather than to make N separate calls to S18DEF.

Paradoxically, for some values of z and (nu), it is cheaper to
call S18DEF with a larger value of N than is required, and then
discard the extra function values returned. However, it is not
possible to state the precise circumstances in which this is
likely to occur. It is due to the fact that the base value used
to start recurrence may be calculated in different regions for
```

```
different N, and the costs in each region may differ greatly.

Note that if the function required is I (x) or I (x), i.e.,
                                       0        1
(nu)=0.0 or 1.0, where x is real and positive, and only a single
function value is required, then it may be much cheaper to call
S18AEF, S18AFF, S18CEF(*) or S18CFF(*), depending on whether a
scaled result is required or not.

9. Example

The example program prints a caption and then proceeds to read
sets of data from the input data stream. The first datum is a
value for the order FNU, the second is a complex value for the
argument, Z, and the third is a value for the parameter SCALE.
The program calls the routine with N = 2 to evaluate the function
for orders FNU and FNU + 1, and it prints the results. The
process is repeated until the end of the input data stream is
encountered.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S19 -- Approximations of Special Functions                 S19AAF
        S19AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S19AAF returns a value for the Kelvin function ber x via the
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S19AAF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Kelvin function
berx.
```

Note: ber(-x)=berx, so the approximation need only consider
x>=0.0.

The routine is based on several Chebyshev expansions:

For 0<=x<=5,

$$\text{berx} = \sum_{r=0}^{--'} a_r T_r(t) \text{ with } t = 2\left(\frac{x}{5}\right)^4 - 1;$$

For x>5,

$$\text{berx} = \frac{e^{x/\sqrt{2}}}{\sqrt{2\pi x}}\left[\left(1 + \frac{1}{-a(t)}\right)\cos(\alpha) + \frac{1}{-b(t)}\sin(\alpha)\right]$$

$$+ \frac{e^{-x/\sqrt{2}}}{\sqrt{2\pi x}}\left[\left(1 + \frac{1}{-c(t)}\right)\sin(\beta) + \frac{1}{-d(t)}\cos(\beta)\right]$$

$$\text{where } (\alpha) = \frac{x}{\sqrt{2}} - \frac{\pi}{8}, \quad (\beta) = \frac{x}{\sqrt{2}} + \frac{\pi}{8},$$

and a(t), b(t), c(t), and d(t) are expansions in the variable

$$t = \frac{10}{x} - 1.$$

When x is sufficiently close to zero, the result is set directly
to ber 0=1.0.

For large x, there is a danger of the result being totally
inaccurate, as the error amplification factor grows in an
essentially exponential manner; therefore the routine must fail.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical

```
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x of the function.

 2:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry ABS(X) is too large for an accurate result to be
     returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than
the relative error is important. Let E be the absolute error in
the result and (delta) be the relative error in the argument. If
(delta) is somewhat larger than the machine precision, then we
have:

                    |  x              |
                E~=| ---(ber x+bei x)|(delta)
                    |        1     1  |
                    | \/2             |

(provided E is within machine bounds).

For small x the error amplification is insignificant and thus the
absolute error is effectively bounded by the machine precision.

For medium and large x, the error behaviour is oscillatory and


                             /   x   x/\/2
its amplitude grows like    /  -----e    . Therefore it is not
                          \/   2(pi)

possible to calculate the function with any accuracy when
```

```
    x/\/2  /2(pi)
\/xe     > -------. Note that this value of x is much smaller than
           (delta)

the minimum value of x for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S19 -- Approximations of Special Functions                 S19ABF
        S19ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S19ABF returns a value for the Kelvin function bei x via the
routine name.

2. Specification

      DOUBLE PRECISION FUNCTION S19ABF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Kelvin function
beix.

Note: bei(-x)=beix, so the approximation need only consider
x>=0.0.
```

The routine is based on several Chebyshev expansions:

For 0<=x<=5,

$$\text{bei } x = \frac{x^2}{4} \sum_{r=0}^{\infty} {}' a_r T_r(t), \quad \text{with } t = 2\left(\frac{x}{5}\right)^4 - 1;$$

For x>5,

$$\text{bei } x = \frac{e^{x/\sqrt{2}}}{\sqrt{2\pi x}}\left[\left(1 - \frac{1}{a(t)}\right)\sin(\alpha) - \frac{b(t)}{x}\cos(\alpha)\right]$$

$$+ \frac{e^{x/\sqrt{2}}}{\sqrt{2\pi x}}\left[\left(1 - \frac{1}{c(t)}\right)\cos(\beta) - \frac{d(t)}{x}\sin(\beta)\right]$$

$$\text{where } \alpha = \frac{x}{\sqrt{2}} - \frac{\pi}{8}, \beta = \frac{x}{\sqrt{2}} + \frac{\pi}{8},$$

and a(t), b(t), c(t), and d(t) are expansions in the variable

$$t = \frac{10}{x} - 1.$$

When x is sufficiently close to zero, the result is computed as

$\text{bei } x = \frac{x^2}{4}$. If this result would underflow, the result returned is beix=0.0.

For large x, there is a danger of the result being totally inaccurate, as the error amplification factor grows in an essentially exponential manner; therefore the routine must fail.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry ABS(X) is too large for an accurate result to be returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than the relative error is important. Let E be the absolute error in the function, and (delta) be the relative error in the argument. If (delta) is somewhat larger than the machine precision, then we have:

$$\tilde{E} = \left| \frac{x}{\sqrt{2}} (-\mathrm{ber}\, x + \mathrm{bei}\, x) \right| (\mathrm{delta})$$

(provided E is within machine bounds).

For small x the error amplification is insignificant and thus the absolute error is effectively bounded by the machine precision.

For medium and large x, the error behaviour is oscillatory and its amplitude grows like $\sqrt{\frac{e^{x/\sqrt{2}}}{2\pi}}$. Therefore it is

```
impossible to calculate the functions with any accuracy when


    x/\/2 /2(pi)
\/xe     > -------. Note that this value of x is much smaller than
           (delta)

the minimum value of x for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users' Note
for your implementation.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S19 -- Approximations of Special Functions                 S19ACF
        S19ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S19ACF returns a value for the Kelvin function ker x, via the
routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S19ACF (X, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Kelvin function
ker x.

Note: for x<0 the function is undefined and at x=0 it is infinite
```

```
so we need only consider x>0.

The routine is based on several Chebyshev expansions:

For 0<x<=1,

                                (pi) 2
                ker x=-f(t)logx+ ----x g(t)+y(t)
                                 16

                                                                    4
where f(t), g(t) and y(t) are expansions in the variable t=2x -1;

For 1<x<=3,

                                (  11 )
                          ker x=exp(- --x)q(t)
                                (  16 )

where q(t) is an expansion in the variable t=x-2;

For x>3,


                      -x/\/2
              / (pi)       [(   1    )           1             ]
   ker x=   /  ----e       [(1+ -c(t))cos(beta)- -d(t)sin(beta)]
          \/    2x         [(   x    )           x             ]


                 x   (pi)
where (beta)= ---+ ----, and c(t) and d(t) are expansions in the
                   8
              \/2
             6
variable t= --1.
             x

When x is sufficiently close to zero, the result is computed as

                                                    2
                                ( x) (      3 2) x
                ker x=-(gamma)-log( -)+((pi)- -x ) --
                                ( 2) (      8  ) 16

and when x is even closer to zero, simply as
                    ( x)
ker x=-(gamma)-log( -).
                    ( 2)
```

```
                                                    / (pi) -x/\/2
For large x, ker x is asymptotically given by   /  ----e      and
                                              \/    2x

this becomes so small that it cannot be computed without
underflow and the routine fails.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x of the function. Constraint: X > 0.

 2:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry X is too large, the result underflows. On soft
     failure, the routine returns zero.

IFAIL= 2
     On entry X <= 0, the function is undefined. On soft failure
     the routine returns zero.

7. Accuracy

Let E be the absolute error in the result, (epsilon) be the
relative error in the result and (delta) be the relative error in
the argument. If (delta) is somewhat larger than the machine
precision, then we have:

                   |  x               |
                E~=| ---(ker  x+kei x)|(delta),
                   |         1     1  |
```

```
                  | \/2             |

                      |     ker  x+kei x|
                      |  x      1     1 |
          (epsilon)~=| --- ------------|(delta).
                      |        ker x    |
                      | \/2             |
```

For very small x, the relative error amplification factor is approximately given by $\frac{1}{|\log x|}$, which implies a strong attenuation of relative error. However, (epsilon) in general cannot be less than the machine precision.

For small x, errors are damped by the function and hence are limited by the machine precision.

For medium and large x, the error behaviour, like the function itself, is oscillatory, and hence only the absolute accuracy for the function can be maintained. For this range of x, the

$$\text{amplitude of the absolute error decays like } \sqrt{\frac{\pi}{2}}\, e^{-x/\sqrt{2}}$$

which implies a strong attenuation of error. Eventually, ker x,

$$\text{which asymptotically behaves like } \sqrt{\frac{\pi}{2x}}\, e^{-x/\sqrt{2}}, \text{ becomes so}$$

small that it cannot be calculated without causing underflow, and the routine returns zero. Note that for large x the errors are dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

Underflow may occur for a few values of x close to the zeros of ker x, below the limit which causes a failure with IFAIL = 1.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation

```
          Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

          S19 -- Approximations of Special Functions                 S19ADF
                  S19ADF -- NAG Foundation Library Routine Document

          Note: Before using this routine, please read the Users' Note for
          your implementation to check implementation-dependent details.
          The symbol (*) after a NAG routine name denotes a routine that is
          not included in the Foundation Library.

          1. Purpose

          S19ADF returns a value for the Kelvin function keix via the
          routine name.

          2. Specification

                DOUBLE PRECISION FUNCTION S19ADF (X, IFAIL)
                INTEGER          IFAIL
                DOUBLE PRECISION X

          3. Description

          This routine evaluates an approximation to the Kelvin function
          keix.

          Note: for x<0 the function is undefined, so we need only consider
          x>=0.

          The routine is based on several Chebyshev expansions:

          For 0<=x<=1,

                                          2
                                 (pi)     x
                        keix=- ----f(t)+ --[-g(t)logx+v(t)]
                                  4       4

                                                                        4
          where f(t), g(t) and v(t) are expansions in the variable t=2x -1;

          For 1<x<=3,

                                        (  9 )
                                 keix=exp(- -x)u(t)
                                        (  8 )

          where u(t) is an expansion in the variable t=x-2;
```

For x>3,

$$keix = \sqrt{\frac{\pi}{2x}}\, e^{-x/\sqrt{2}}\left[\left(1+\frac{1}{x}\right)c(t)\sin(\beta) + \left(\frac{1}{x}\right)d(t)\cos(\beta)\right]$$

$$\text{where } \beta = \frac{x}{\sqrt{2}} + \frac{\pi}{8}, \text{ and } c(t) \text{ and } d(t) \text{ are expansions in the variable } t = \frac{6}{x}\sqrt{2}.$$

For x<0, the function is undefined, and hence the routine fails and returns zero.

When x is sufficiently close to zero, the result is computed as

$$keix = -\frac{\pi}{4}\left(1-\gamma-\log\left(\frac{x}{2}\right)\right)\frac{x^2}{4}$$

and when x is even closer to zero simply as

$$keix = -\frac{\pi}{4}.$$

For large x, keix is asymptotically given by $\sqrt{\frac{\pi}{2x}}\, e^{-x/\sqrt{2}}$ and this becomes so small that it cannot be computed without underflow and the routine fails.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function. Constraint: X >= 0.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry X is too large, the result underflows. On soft failure, the routine returns zero.

IFAIL= 2
On entry X < 0, the function is undefined. On soft failure the routine returns zero.

7. Accuracy

Let E be the absolute error in the result, and (delta) be the relative error in the argument. If (delta) is somewhat larger than the machine representation error, then we have:

```
   |  x                |
E~=| ---(-ker  x+kei x)|(delta).
   |          1     1  |
   | \/2               |
```

For small x, errors are attenuated by the function and hence are limited by the machine precision.

For medium and large x, the error behaviour, like the function itself, is oscillatory and hence only absolute accuracy of the function can be maintained. For this range of x, the amplitude of

```
                                / (pi)x -x/\/2
the absolute error decays like  /  -----e      , which implies a
                              \/     2
```

strong attenuation of error. Eventually, keix, which is

```
                                / (pi) -x/\/2
asymptotically given by   /  ----e      , becomes so small that it
                        \/    2x

cannot be calculated without causing underflow and therefore the
routine returns zero. Note that for large x, the errors are
dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

Underflow may occur for a few values of x close to the zeros of
keix, below the limit which causes a failure with IFAIL = 1.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
S20 -- Approximations of Special Functions                S20ACF
        S20ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S20ACF returns a value for the Fresnel Integral S(x), via the
routine name.

2. Specification

      DOUBLE PRECISION FUNCTION S20ACF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Fresnel Integral

                           x
                           /   ( (pi) 2)
                    S(x)= |sin( ----t )dt.
```

$$\int_0^{\left(\frac{2}{x}\right)}$$

Note: S(x)=-S(-x), so the approximation need only consider x>=0.0

The routine is based on three Chebyshev expansions:

For 0<x<=3,

$$S(x)=x\sum_{r=0}^{3}{}' a_r T_r(t), \text{ with } t=2\left(\frac{x}{3}\right)^4-1;$$

For x>3,

$$S(x)=\frac{1}{2}\frac{f(x)}{x}\cos\left(\frac{\pi}{2}x^2\right)-\frac{g(x)}{x^3}\sin\left(\frac{\pi}{2}x^2\right),$$

$$\text{where } f(x)=\sum_{r=0}^{3}{}' b_r T_r(t),$$

$$\text{and } g(x)=\sum_{r=0}^{3}{}' c_r T_r(t), \text{ with } t=2\left(\frac{3}{x}\right)^4-1.$$

For small x, $S(x)\tilde{}=\frac{\pi}{6}x^3$. This approximation is used when x is sufficiently small for the result to be correct to machine precision. For very small x, this approximation would underflow; the result is then set exactly to zero.

For large x, $f(x)\tilde{}=\frac{1}{\pi}$ and $g(x)\tilde{}=\frac{1}{\pi^2}$. Therefore for moderately large x, when $\frac{1}{2^3\pi^3x^3}$ is negligible compared with $\frac{1}{2}$, the second term in the approximation for x>3 may be dropped. For very large x, when $\frac{1}{\pi x}$ becomes negligible, $S(x)\tilde{}=\frac{1}{2}$. However

there will be considerable difficulties in calculating
$\cos\left(\frac{(\pi)}{2}x\right)$ accurately before this final limiting value can be used. Since $\cos\left(\frac{(\pi)}{2}x\right)$ is periodic, its value is essentially determined by the fractional part of x^2. If $x = N+(\theta)$ where N is an integer and $0 \le (\theta) < 1$, then $\cos\left(\frac{(\pi)}{2}x\right)$ depends on (θ) and on N modulo 4. By exploiting this fact, it is possible to retain significance in the calculation of $\cos\left(\frac{(\pi)}{2}x\right)$ either all the way to the very large x limit, or at least until the integer part of $\frac{x}{2}$ is equal to the maximum integer allowed on the machine.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

5. Parameters

1: X -- DOUBLE PRECISION Input
On entry: the argument x of the function.

2: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

Let δ and ε be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e.,
if (delta) is due to data errors etc), then (epsilon) and (delta)
are approximately related by:

$$(\text{epsilon}) \tilde{=} \left| \frac{x \sin\left(\frac{(\text{pi})}{2} x\right)}{S(x)} \right| (\text{delta}).$$

Figure 1 shows the behaviour of the error amplification factor

$$\left| \frac{x \sin\left(\frac{(\text{pi})}{2} x\right)}{S(x)} \right|.$$

Figure 1
Please see figure in printed Reference Manual

However if (delta) is of the same order as the machine precision,
then rounding errors could make (epsilon) slightly larger than
the above relation predicts.

For small x, (epsilon)~=3(delta) and hence there is only moderate
amplification of relative error. Of course for very small x where
the correct result would underflow and exact zero is returned,
relative error-control is lost.

For moderately large values of x,

$$|(\text{epsilon})| \tilde{=} \left| 2x \sin\left(\frac{(\text{pi})}{2} x\right) \right| |(\text{delta})|$$

and the result will be subject to increasingly large
amplification of errors. However the above relation breaks down
for large values of x (i.e., when $\frac{1}{x^2}$ is of the order of the
machine precision in this region the relative error in the result
is essentially bounded by $\frac{2}{(\text{pi})x}$).

Hence the effects of error amplification are limited and at worst

```
the relative error loss should not exceed half the possible
number of significant figures.

8. Further Comments

None.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

```
S20 -- Approximations of Special Functions                 S20ADF
        S20ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S20ADF returns a value for the Fresnel Integral C(x), via the
routine name.

2. Specification

      DOUBLE PRECISION FUNCTION S20ADF (X, IFAIL)
      INTEGER          IFAIL
      DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Fresnel Integral

                           x
                           /   ( (pi) 2)
                    C(x)= |cos( ----t )dt.
                           /   (  2    )
                           0

Note: C(x)=-C(-x), so the approximation need only consider x>=0.0

The routine is based on three Chebyshev expansions:
```

For 0<x<=3,

$$C(x)=x\sum_{r=0}^{\infty} a_r T_r(t), \text{ with } t=2\left(\frac{x}{3}\right)^4-1;$$

For x>3,

$$C(x)=\frac{1}{2}\frac{f(x)}{x}\sin\left(\frac{(\pi)}{2}x^4\right)-\frac{g(x)}{x^3}\cos\left(\frac{(\pi)}{2}x^4\right),$$

where $f(x)=\sum_{r=0}^{\infty} b_r T_r(t)$,

and $g(x)=\sum_{r=0}^{\infty} c_r T_r(t)$, with $t=2\left(\frac{3}{x}\right)^4-1$.

For small x, C(x)~=x. This approximation is used when x is sufficiently small for the result to be correct to machine precision.

For large x, $f(x)\sim=\frac{1}{(\pi)}$ and $g(x)\sim=\frac{1}{(\pi)^2}$. Therefore for moderately large x, when $\frac{1}{2^3(\pi)x^3}$ is negligible compared with $\frac{1}{2}$, the second term in the approximation for x>3 may be dropped. For very large x, when $\frac{1}{(\pi)x}$ becomes negligible, $C(x)\sim=\frac{1}{2}$. However there will be considerable difficulties in calculating $\sin\left(\frac{(\pi)}{2}x^4\right)$ accurately before this final limiting value can be used. Since $\sin\left(\frac{(\pi)}{2}x^4\right)$ is periodic, its value is essentially determined by the fractional part of x^4. If $x=N+(\text{theta})$, where N

```
                                        ( (pi) 2)
is an integer and 0<=(theta)<1, then sin( ----x ) depends on
                                        (  2    )
(theta) and on N modulo 4. By exploiting this fact, it is
possible to retain some significance in the calculation of
   ( (pi) 2)
sin( ----x ) either all the way to the very large x limit, or at
   (  2    )
                                  x
least until the integer part of   - is equal to the maximum
                                  2
integer allowed on the machine.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input
     On entry: the argument x of the function.

 2:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL
has been included for consistency with other routines in this
chapter.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument
and result respectively.

If (delta) is somewhat larger than the machine precision (i.e if
(delta) is due to data errors etc), then (epsilon) and (delta)
are approximately related by:

                          |     ( (pi) 2)|
                          | xcos( ----x )|
                          |     (  2    )|
               (epsilon)~=| -------------|(delta).
```

```
                          |     C(x)     |
```

Figure 1 shows the behaviour of the error amplification factor

```
|     ( (pi) 2)|
| xcos( ----x )|
|     (  2    )|
| -------------|.
|     C(x)     |
```

Figure 1
Please see figure in printed Reference Manual

However if (delta) is of the same order as the machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, (epsilon)~=(delta) and there is no amplification of relative error.

For moderately large values of x,

```
                          |     ( (pi) 2)|
            |(epsilon)|~=|2xcos( ----x )||(delta)|
                          |     (  2    )|
```

and the result will be subject to increasingly large amplification of errors. However the above relation breaks down
for large values of x (i.e., when $\frac{1}{2x^2}$ is of the order of the machine precision); in this region the relative error in the result is essentially bounded by $\frac{2}{(\pi)x}$.

Hence the effects of error amplification are limited and at worst the relative error loss should not exceed half the possible number of significant figures.

8. Further Comments

None.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
S21 -- Approximations of Special Functions                 S21BAF
        S21BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S21BAF returns a value of an elementary integral, which occurs as
a degenerate case of an elliptic integral of the first kind, via
the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S21BAF (X, Y, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X, Y

3. Description

This routine calculates an approximate value for the integral

                                 infty
                              1 /         dt
                    R (x,y)= - |     ----------
                     C       2 /
                               0     \/t+x(t+y)

where x>=0 and y/=0.

This function, which is related to the logarithm or inverse
hyperbolic functions for y<x and to inverse circular functions if
x<y, arises as a degenerate form of the elliptic integral of the
first kind. If y<0, the result computed is the Cauchy principal
value of the integral.

The basic algorithm, which is due to Carlson [2] and [3], is to
reduce the arguments recursively towards their mean by the
system:

    x =x,
```

```
 0

y =y
 0

(mu) =(x +2y )/3,
    n   n   n

S =(y -x )/3(mu)
 n   n  n       n



(lambda) =y +2  /x y
        n  n  \/  n n

x   =(x +(lambda) )/4,
 n+1   n         n

y   =(y +(lambda) )/4.
 n+1   n         n
```

The quantity $|S_n|$ for n=0,1,2,3,... decreases with increasing n, eventually $|S_n| \tilde{} 1/4$. For small enough S_n the required function value can be approximated by the first few terms of the Taylor series about the mean. That is

$$R_C(x,y) = \left(1 + \frac{3S_n}{10} + \frac{S_n^2}{7} + \frac{3S_n^3}{8} + \frac{9S_n^4}{22}\right) / \sqrt{\mu_n}\,.$$

The truncation error involved in using this approximation is bounded by $16|S_n|^6/(1-2|S_n|)$ and the recursive process is stopped when S_n is small enough for this truncation error to be negligible compared to the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the

result is done before returning to the calling program.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Carlson B C (1978) Computing Elliptic Integrals by Duplication. (Preprint) Department of Physics, Iowa State University.

[3] Carlson B C (1988) A Table of Elliptic Integrals of the Third Kind. Math. Comput. 51 267--280.

5. Parameters

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input
On entry: the arguments x and y of the function, respectively. Constraint: X >= 0.0 and Y /= 0.0.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry X < 0.0; the function is undefined.

IFAIL= 2
On entryY = 0.0; the function is undefined.

On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be

```
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme
arguments which are used with the routine to produce the table of
low accuracy results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%%

```
S21 -- Approximations of Special Functions                S21BBF
        S21BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S21BBF returns a value of the symmetrised elliptic integral of
the first kind, via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S21BBF (X, Y, Z, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X, Y, Z

3. Description

This routine calculates an approximation to the integral

                           infty
                         1 /            dt
               R (x,y,z)= - |     -----------------
                F         2 /
```

```
                              0     \/(t+x)(t+y)(t+z)
```

where x, y, z>=0 and at most one is zero.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:

```
     x =min(x,y,z),z =max(x,y,z),
      0             0

     y = remaining third intermediate value argument.
      0
```

(This ordering, which is possible because of the symmetry of the function, is done for technical reasons related to the avoidance of overflow and underflow.)

```
    (mu) =   (x +y +3z )/3
        n      n  n   n

    X    =   (1-x )/(mu)
     n           n      n

    Y    =   (1-y )/(mu)
     n           n      n

    Z    =   (1-z )/(mu)
     n           n      n



    (lambda) =  /x y +  /y z +  /z x
           n \/  n n /  n n /  n n

    x    =   (x +(lambda) )/4
     n+1       n         n

    y    =   (y +(lambda) )/4
     n+1       n         n

    z    =   (z +(lambda) )/4
     n+1       n         n
```

(epsilon) =max(|X |,|Y |,|Z |) and the function may be approximated adequately by a 5th order power series:

```
        n       n    n    n

                                 (        2           )
                                 (   E    E   3E E    E )
                           1     (    2   2    2 3    3)
```

$$R_F(x,y,z) = \frac{1}{2\sqrt{\mu_n}}\left(1 - \frac{10}{24} + \frac{44}{44} - \frac{14}{}\right)$$

where $E_2 = X_n Y_n + Y_n Z_n + Z_n X_n$, $E_3 = X_n Y_n Z_n$.

The truncation error involved in using this approximation is bounded by $(\text{epsilon})^6/4(1-(\text{epsilon})^3)$ and the recursive process is stopped when this truncation error is negligible compared with the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the result is done before returning to the calling program.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Carlson B C (1978) Computing Elliptic Integrals by Duplication. (Preprint) Department of Physics, Iowa State University.

[3] Carlson B C (1988) A Table of Elliptic Integrals of the Third Kind. Math. Comput. 51 267--280.

5. Parameters

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input

3: Z -- DOUBLE PRECISION Input
On entry: the arguments x, y and z of the function.
Constraint: X, Y, Z >= 0.0 and only one of X, Y and Z may be zero.

4: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

```
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry one or more of X, Y and Z is negative; the function
     is undefined.

IFAIL= 2
     On entry two or more of X, Y and Z are zero; the function is
     undefined.

     On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.

If two arguments are equal, the function reduces to the
elementary integral R , computed by S21BAF.
                      C

9. Example

This example program simply generates a small set of non-extreme
arguments which are used with the routine to produce the table of
low accuracy results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

%%

S21 -- Approximations of Special Functions S21BCF
S21BCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S21BCF returns a value of the symmetrised elliptic integral of the second kind, via the routine name.

2. Specification

```
DOUBLE PRECISION FUNCTION S21BCF (X, Y, Z, IFAIL)
INTEGER          IFAIL
DOUBLE PRECISION X, Y, Z
```

3. Description

This routine calculates an approximate value for the integral

$$R_D(x,y,z) = \frac{3}{2}\int_0^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)^3}}$$

where x, y>=0, at most one of x and y is zero, and z>0.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:

$$x_0 = x$$

$$y_0 = y$$

$$z_0 = z$$

$$(\mu)_n = (x_n + y_n + 3z_n)/5$$

$$X_n = (1-x_n)/(\mu_n)$$

$$Y_n = (1-y_n)/(\mu_n)$$

$$Z_n = (1-z_n)/(\mu_n)$$

$$(\lambda) = \sqrt{x_n y_n + y_n z_n + z_n x_n}$$

$$x_{n+1} = (x_n + (\lambda))/4$$

$$y_{n+1} = (y_n + (\lambda))/4$$

$$z_{n+1} = (z_n + (\lambda))/4$$

For n sufficiently large,

$$(\epsilon)_n = \max(|X_n|,|Y_n|,|Z_n|)^{(-)^{(n+1)}/4}$$

and the function may be approximated adequately by a 5th order power series

$$R_D(x,y,z) = 3 \sum_{m=0}^{n-1} \frac{4^{-m}}{(z_n + (\lambda))^m \sqrt{z_n}^m}$$

$$\frac{4^{-n}}{\sqrt{(\mu)_n^3}}\left[1 + \frac{3}{7n}S_n^{(2)} + \frac{1}{3n}S_n^{(3)} + \frac{3}{22n}S_n^{(2)2} + \frac{3}{11n}S_n^{(4)} + \frac{3}{13n}S_n^{(2)}S_n^{(3)} + \frac{3}{n}S_n^{(5)} + \frac{3}{13n}S_n^{(6)}\right]$$

where

$$S_n^{(m)} = (X_n^m + Y_n^m + 3Z_n^m)/2m.$$

```
                    n   ( n  n   n)

The truncation error in this expansion is bounded by
              6
     3(epsilon)
              n
 ------------------- and the recursive process is terminated when


    /             3
   / (1-(epsilon) )
 \/             n
this quantity is negligible compared with the machine precision.

The routine may fail either because it has been called with
arguments outside the domain of definition, or with arguments so
extreme that there is an unavoidable danger of setting underflow
or overflow.

                -3/2
Note: R (x,x,x)=x    , so there exists a region of extreme
       D
arguments for which the function value is not representable.

4. References

[1]   Abramowitz M and Stegun I A (1968) Handbook of Mathematical
      Functions. Dover Publications.

[2]   Carlson B C (1978) Computing Elliptic Integrals by
      Duplication. (Preprint) Department of Physics, Iowa State
      University.

[3]   Carlson B C (1988) A Table of Elliptic Integrals of the
      Third Kind. Math. Comput. 51 267--280.

5. Parameters

 1:  X -- DOUBLE PRECISION                                  Input

 2:  Y -- DOUBLE PRECISION                                  Input

 3:  Z -- DOUBLE PRECISION                                  Input
     On entry: the arguments x, y and z of the function.
     Constraint: X, Y >= 0.0, Z > 0.0 and only one of X and Y may
     be zero.

 4:  IFAIL -- INTEGER                                 Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
```

```
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry either X or Y is negative, or both X and Y are
     zero; the function is undefined.

IFAIL= 2
     On entry Z <= 0.0; the function is undefined.

IFAIL= 3
     On entry either Z is too close to zero or both X and Y are
     too close to zero: there is a danger of setting overflow.

IFAIL= 4
     On entry at least one of X, Y and Z is too large: there is a
     danger of setting underflow.

     On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme
arguments which are used with the routine to produce the table of
low accuracy results.
```

```
The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S21 -- Approximations of Special Functions                 S21BDF
        S21BDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S21BDF returns a value of the symmetrised elliptic integral of
the third kind, via the routine name.

2. Specification

       DOUBLE PRECISION FUNCTION S21BDF (X, Y, Z, R, IFAIL)
       INTEGER          IFAIL
       DOUBLE PRECISION X, Y, Z, R

3. Description

This routine calculates an approximation to the integral

                         infty
                       3 /                 dt
       R (x,y,z,(rho))= - |     -------------------------
        J               2 /
                         0     (t+(rho))\/(t+x)(t+y)(t+z)

where x, y, z>=0, (rho)/=0 and at most one of x, y and z is zero.

If p<0, the result computed is the Cauchy principal value of the
integral.

The basic algorithm, which is due to Carlson [2] and [3], is to
reduce the arguments recursively towards their mean by the rule:

    x    =   x
     0

    y    =   y
     0
```

```
z    =   z
 0

(rho) =  (rho)
     0

(mu) =   (x +y +z +2(rho) )/5
    n      n  n  n       n

X    =   (1-x )/(mu)
 n           n      n

Y    =   (1-y )/(mu)
 n           n      n

Z    =   (1-z )/(mu)
 n           n      n

P    =   (1-(rho) )/(mu)
 n               n      n



(lambda) =   /x y +  /y z +  /z x
          n\/  n n /  n n /  n n

x    =   (x +(lambda) )/4
 n+1       n         n

y    =   (y +(lambda) )/4
 n+1       n         n

z    =   (z +(lambda) )/4
 n+1       n         n

(rho) =  ((rho) +(lambda) )/4
     n+1       n         n
                                          2
          [                              ]
(alpha) =[(rho) (  /x +  /y +  /z )+  /x y z ]
       n [     n \/  n /  n /  n  /  n n n]

                                     2
(beta) = (rho) ((rho) +(lambda) )
      n       n      n         n
```

For n sufficiently large,

```
                                                 1
          (epsilon) =max(|X |,|Y |,|Z |,|P |)~ --
```

```
                      n       n    n    n    n     n
                                                 4
```

and the function may be approximated by a 5th order power series

```
                  n-1
                  --  -m
R (x,y,z,(rho))=3 >  4  R ((alpha) ,(beta) )
 J                --     C        m       m
                  m=0
```

```
      -n
     4     [   3 (2)  1 (3)  3   (2) 2  3  (4)  3  (2) (3)  3  (5)]
+ ---------[1+ -S   + -S   + --(S   ) + --S   + --S   S   + --S   ]
           [   7 n    3 n    22  n      11 n    13 n   n    13 n  ]
     /    3
    / (mu)
  \/      n
```

```
       (m)     m  m  m   m
where S    = (X +Y +Z +2P )/2m.
       n       n  n  n   n
```

The truncation error in this expansion is bounded by

```
          6    /              3
3(epsilon) /  / (1-(epsilon) )  and the recursion process is
          n \/              n
```

terminated when this quantity is negligible compared with the machine precision. The routine may fail either because it has been called with arguments outside the domain of definition or with arguments so extreme that there is an unavoidable danger of setting underflow or overflow.

```
                    3
                  - -
                    2
```

Note: R (x,x,x,x)=x , so there exists a region of extreme
J
arguments for which the function value is not representable.

4. References

[1] Abramowitz M and Stegun I A (1968) Handbook of Mathematical Functions. Dover Publications.

[2] Carlson B C (1978) Computing Elliptic Integrals by Duplication. (Preprint) Department of Physics, Iowa State

```
      University.

[3]   Carlson B C (1988) A Table of Elliptic Integrals of the
      Third Kind. Math. Comput. 51 267--280.

5. Parameters

 1:  X -- DOUBLE PRECISION                                    Input

 2:  Y -- DOUBLE PRECISION                                    Input

 3:  Z -- DOUBLE PRECISION                                    Input

 4:  R -- DOUBLE PRECISION                                    Input
     On entry: the arguments x, y, z and (rho) of the function.
     Constraint: X, Y, Z >= 0.0, R /= 0.0 and at most one of X, Y
     and Z may be zero.

 5:  IFAIL -- INTEGER                                  Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry at least one of X, Y and Z is negative, or at least
     two of them are zero; the function is undefined.

IFAIL= 2
     On entry R = 0.0; the function is undefined.

IFAIL= 3
     On entry either R is too close to zero, or any two of X, Y
     and Z are too close to zero; there is a danger of setting
     overflow.

IFAIL= 4
     On entry at least one of X, Y, Z and R is too large; there
     is a danger of setting underflow.

IFAIL= 5
```

```
     An error has occurred in a call to S21BAF. Any such
     occurrence should be reported to NAG.

     On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.

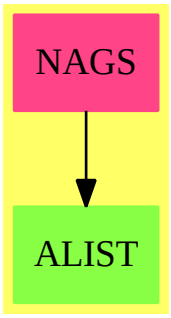
If the argument R is equal to any of the other arguments, the
function reduces to the integral R , computed by S21BCF.
                                  D

9. Example

This example program simply generates a small set of non-extreme
arguments which are used with the routine to produce the table of
low accuracy results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
```

15.15.1 NagSpecialFunctionsPackage (NAGS)



Exports:

s01eaf	s13aaf	s13acf	s13adf	s14aaf
s14abf	s14baf	s15adf	s15aef	s17acf
s17adf	s17aef	s17aff	s17agf	s17ahf
s17ajf	s17akf	s17dcf	s17def	s17dgf
s17dhf	s17dlf	s18acf	s18adf	s18aef
s18aff	s18dcf	s18def	s19aaf	s19abf
s19acf	s19adf	s20acf	s20adf	s21baf
s21bbf	s21bcf	s21bdf		

— package NAGS NagSpecialFunctionsPackage —

```
)abbrev package NAGS NagSpecialFunctionsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:44 1994
++ Description:
++ This package uses the NAG Library to compute some commonly
++ occurring physical and mathematical functions.

NagSpecialFunctionsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

  Exports ==> with
    s01eaf : (Complex DoubleFloat,Integer) -> Result
     ++ s01eaf(z,ifail)
     ++ S01EAF evaluates the exponential function exp(z) , for complex z.
     ++ See \downlink{Manual Page}{manpageXXs01eaf}.
    s13aaf : (DoubleFloat,Integer) -> Result
     ++ s13aaf(x,ifail)
     ++ returns the value of the exponential integral
     ++  E (x), via the routine name.
     ++   1
     ++ See \downlink{Manual Page}{manpageXXs13aaf}.
```

```
s13acf : (DoubleFloat,Integer) -> Result
 ++ s13acf(x,ifail)
 ++ returns the value of the cosine integral
 ++ See \downlink{Manual Page}{manpageXXs13acf}.
s13adf : (DoubleFloat,Integer) -> Result
 ++ s13adf(x,ifail)
 ++ returns the value of the sine integral
 ++ See \downlink{Manual Page}{manpageXXs13adf}.
s14aaf : (DoubleFloat,Integer) -> Result
 ++ s14aaf(x,ifail) returns the value of the Gamma function (Gamma)(x),
 ++ via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs14aaf}.
s14abf : (DoubleFloat,Integer) -> Result
 ++ s14abf(x,ifail) returns a value for the log, ln(Gamma(x)), via
 ++ the routine name.
 ++ See \downlink{Manual Page}{manpageXXs14abf}.
s14baf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
 ++ s14baf(a,x,tol,ifail)
 ++ computes values for the incomplete gamma functions P(a,x)
 ++ and Q(a,x).
 ++ See \downlink{Manual Page}{manpageXXs14baf}.
s15adf : (DoubleFloat,Integer) -> Result
 ++ s15adf(x,ifail)
 ++ returns the value of the complementary error function,
 ++ erfc(x), via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs15adf}.
s15aef : (DoubleFloat,Integer) -> Result
 ++ s15aef(x,ifail)
 ++ returns the value of the error function erf(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs15aef}.
s17acf : (DoubleFloat,Integer) -> Result
 ++ s17acf(x,ifail)
 ++ returns the value of the Bessel Function
 ++  Y (x), via the routine name.
 ++   0
 ++ See \downlink{Manual Page}{manpageXXs17acf}.
s17adf : (DoubleFloat,Integer) -> Result
 ++ s17adf(x,ifail)
 ++ returns the value of the Bessel Function
 ++  Y (x), via the routine name.
 ++   1
 ++ See \downlink{Manual Page}{manpageXXs17adf}.
s17aef : (DoubleFloat,Integer) -> Result
 ++ s17aef(x,ifail)
 ++ returns the value of the Bessel Function
 ++  J (x), via the routine name.
 ++   0
 ++ See \downlink{Manual Page}{manpageXXs17aef}.
s17aff : (DoubleFloat,Integer) -> Result
```

```
 ++ s17aff(x,ifail)
 ++ returns the value of the Bessel Function
 ++  J (x), via the routine name.
 ++   1
 ++ See \downlink{Manual Page}{manpageXXs17aff}.
s17agf : (DoubleFloat,Integer) -> Result
 ++ s17agf(x,ifail)
 ++ returns a value for the Airy function, Ai(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs17agf}.
s17ahf : (DoubleFloat,Integer) -> Result
 ++ s17ahf(x,ifail)
 ++ returns a value of the Airy function, Bi(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs17ahf}.
s17ajf : (DoubleFloat,Integer) -> Result
 ++ s17ajf(x,ifail)
 ++ returns a value of the derivative of the Airy function
 ++ Ai(x), via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs17ajf}.
s17akf : (DoubleFloat,Integer) -> Result
 ++ s17akf(x,ifail)
 ++ returns a value for the derivative of the Airy function
 ++ Bi(x), via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs17akf}.
s17dcf : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
 ++ s17dcf(fnu,z,n,scale,ifail)
 ++ returns a sequence of values for the Bessel functions
 ++  Y      (z) for complex z, non-negative (nu) and n=0,1,...,N-1,
 ++   (nu)+n
 ++ with an option for exponential scaling.
 ++ See \downlink{Manual Page}{manpageXXs17dcf}.
s17def : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
 ++ s17def(fnu,z,n,scale,ifail)
 ++ returns a sequence of values for the Bessel functions
 ++  J      (z) for complex z, non-negative (nu) and n=0,1,...,N-1,
 ++   (nu)+n
 ++ with an option for exponential scaling.
 ++ See \downlink{Manual Page}{manpageXXs17def}.
s17dgf : (String,Complex DoubleFloat,String,Integer) -> Result
 ++ s17dgf(deriv,z,scale,ifail)
 ++ returns the value of the Airy function Ai(z) or its
 ++ derivative Ai'(z) for complex z, with an option for exponential
 ++ scaling.
 ++ See \downlink{Manual Page}{manpageXXs17dgf}.
s17dhf : (String,Complex DoubleFloat,String,Integer) -> Result
 ++ s17dhf(deriv,z,scale,ifail)
 ++ returns the value of the Airy function Bi(z) or its
```

```
 ++ derivative Bi'(z) for complex z, with an option for exponential
 ++ scaling.
 ++ See \downlink{Manual Page}{manpageXXs17dhf}.
s17dlf : (Integer,DoubleFloat,Complex DoubleFloat,Integer,_
    String,Integer) -> Result
 ++ s17dlf(m,fnu,z,n,scale,ifail)
 ++ returns a sequence of values for the Hankel functions
 ++   (1)           (2)
 ++  H      (z) or H      (z) for complex z, non-negative (nu) and
 ++   (nu)+n        (nu)+n
 ++ n=0,1,...,N-1, with an option for exponential scaling.
 ++ See \downlink{Manual Page}{manpageXXs17dlf}.
s18acf : (DoubleFloat,Integer) -> Result
 ++ s18acf(x,ifail)
 ++ returns the value of the modified Bessel Function
 ++  K (x), via the routine name.
 ++   0
 ++ See \downlink{Manual Page}{manpageXXs18acf}.
s18adf : (DoubleFloat,Integer) -> Result
 ++ s18adf(x,ifail)
 ++ returns the value of the modified Bessel Function
 ++  K (x), via the routine name.
 ++   1
 ++ See \downlink{Manual Page}{manpageXXs18adf}.
s18aef : (DoubleFloat,Integer) -> Result
 ++ s18aef(x,ifail)
 ++ returns the value of the modified Bessel Function
 ++  I (x), via the routine name.
 ++   0
 ++ See \downlink{Manual Page}{manpageXXs18aef}.
s18aff : (DoubleFloat,Integer) -> Result
 ++ s18aff(x,ifail)
 ++ returns a value for the modified Bessel Function
 ++  I (x), via the routine name.
 ++   1
 ++ See \downlink{Manual Page}{manpageXXs18aff}.
s18dcf : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
 ++ s18dcf(fnu,z,n,scale,ifail)
 ++ returns a sequence of values for the modified Bessel functions
 ++  K      (z) for complex z, non-negative (nu) and
 ++   (nu)+n
 ++ n=0,1,...,N-1, with an option for exponential scaling.
 ++ See \downlink{Manual Page}{manpageXXs18dcf}.
s18def : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
 ++ s18def(fnu,z,n,scale,ifail)
 ++ returns a sequence of values for the modified Bessel functions
 ++  I      (z) for complex z, non-negative (nu) and
 ++   (nu)+n
```

```
 ++ n=0,1,...,N-1, with an option for exponential scaling.
 ++ See \downlink{Manual Page}{manpageXXs18def}.
s19aaf : (DoubleFloat,Integer) -> Result
 ++ s19aaf(x,ifail)
 ++ returns a value for the Kelvin function ber(x) via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs19aaf}.
s19abf : (DoubleFloat,Integer) -> Result
 ++ s19abf(x,ifail)
 ++ returns a value for the Kelvin function bei(x) via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs19abf}.
s19acf : (DoubleFloat,Integer) -> Result
 ++ s19acf(x,ifail)
 ++ returns a value for the Kelvin function ker(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs19acf}.
s19adf : (DoubleFloat,Integer) -> Result
 ++ s19adf(x,ifail)
 ++ returns a value for the Kelvin function kei(x) via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs19adf}.
s20acf : (DoubleFloat,Integer) -> Result
 ++ s20acf(x,ifail)
 ++ returns a value for the Fresnel Integral S(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs20acf}.
s20adf : (DoubleFloat,Integer) -> Result
 ++ s20adf(x,ifail)
 ++ returns a value for the Fresnel Integral C(x), via the
 ++ routine name.
 ++ See \downlink{Manual Page}{manpageXXs20adf}.
s21baf : (DoubleFloat,DoubleFloat,Integer) -> Result
 ++ s21baf(x,y,ifail)
 ++ returns a value of an elementary integral, which occurs as
 ++ a degenerate case of an elliptic integral of the first kind, via
 ++ the routine name.
 ++ See \downlink{Manual Page}{manpageXXs21baf}.
s21bbf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
 ++ s21bbf(x,y,z,ifail)
 ++ returns a value of the symmetrised elliptic integral of
 ++ the first kind, via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs21bbf}.
s21bcf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
 ++ s21bcf(x,y,z,ifail)
 ++ returns a value of the symmetrised elliptic integral of
 ++ the second kind, via the routine name.
 ++ See \downlink{Manual Page}{manpageXXs21bcf}.
s21bdf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,_
    Integer) -> Result
```

```
    ++ s21bdf(x,y,z,r,ifail)
    ++ returns a value of the symmetrised elliptic integral of
    ++ the third kind, via the routine name.
    ++ See \downlink{Manual Page}{manpageXXs21bdf}.
  Implementation ==> add

    import Lisp
    import DoubleFloat
    import Any
    import Record
    import Integer
    import Matrix DoubleFloat
    import Boolean
    import NAGLinkSupportPackage
    import AnyFunctions1(Complex DoubleFloat)
    import AnyFunctions1(Integer)
    import AnyFunctions1(DoubleFloat)
    import AnyFunctions1(String)

    s01eaf(zArg:Complex DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s01eaf",_
        ["z"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["integer"::S,"ifail"::S]$Lisp_
        ,["double complex"::S,"s01eafResult"::S,"z"::S]$Lisp_
        ]$Lisp,_
        ["s01eafResult"::S,"ifail"::S]$Lisp,_
        [([zArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s13aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s13aaf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s13aafResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s13aafResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s13acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s13acf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
```

```
        [["double"::S,"s13acfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s13acfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s13adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s13adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s13adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s13adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s14aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s14aaf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s14aafResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s14aafResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s14abf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s14abf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s14abfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s14abfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s14baf(aArg:DoubleFloat,xArg:DoubleFloat,tolArg:DoubleFloat,_
        ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
```

```
        "s14baf",_
        ["a"::S,"x"::S,"tol"::S,"p"::S,"q"::S_
        ,"ifail"::S]$Lisp,_
        ["p"::S,"q"::S]$Lisp,_
        [["double"::S,"a"::S,"x"::S,"tol"::S,"p"::S_
        ,"q"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["p"::S,"q"::S,"ifail"::S]$Lisp,_
        [([aArg::Any,xArg::Any,tolArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s15adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s15adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s15adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s15adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s15aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s15aef",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s15aefResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s15aefResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17acf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17acfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17acfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
```

```
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17aef",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17aefResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17aefResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17aff(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17aff",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17affResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17affResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17agf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17agf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17agfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17agfResult"::S,"ifail"::S]$Lisp,_
```

```
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17ahf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17ahf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17ahfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17ahfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17ajf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17ajf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17ajfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17ajfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17akf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17akf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s17akfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s17akfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result


    s17dcf(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
        scaleArg:String,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17dcf",_
        ["fnu"::S,"z"::S,"n"::S,"scale"::S,"nz"::S_
        ,"ifail"::S,"cy"::S,"cwrk"::S]$Lisp,_
```

```
    ["cy"::S,"nz"::S,"cwrk"::S]$Lisp,_
    [["double"::S,"fnu"::S]$Lisp_
    ,["integer"::S,"n"::S,"nz"::S,"ifail"::S]$Lisp_
    ,["character"::S,"scale"::S]$Lisp_
    ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp,_
    ["cwrk"::S,"n"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["cy"::S,"nz"::S,"ifail"::S]$Lisp,_
    [([fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s17def(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
    scaleArg:String,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s17def",_
    ["fnu"::S,"z"::S,"n"::S,"scale"::S,"nz"::S_
    ,"ifail"::S,"cy"::S]$Lisp,_
    ["cy"::S,"nz"::S]$Lisp,_
    [["double"::S,"fnu"::S]$Lisp_
    ,["integer"::S,"n"::S,"nz"::S,"ifail"::S]$Lisp_
    ,["character"::S,"scale"::S]$Lisp_
    ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp]$Lisp_
    ]$Lisp,_
    ["cy"::S,"nz"::S,"ifail"::S]$Lisp,_
    [([fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s17dgf(derivArg:String,zArg:Complex DoubleFloat,scaleArg:String,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s17dgf",_
    ["deriv"::S,"z"::S,"scale"::S,"ai"::S,"nz"::S_
    ,"ifail"::S]$Lisp,_
    ["ai"::S,"nz"::S]$Lisp,_
    [["integer"::S,"nz"::S,"ifail"::S]$Lisp_
    ,["character"::S,"deriv"::S,"scale"::S]$Lisp_
    ,["double complex"::S,"z"::S,"ai"::S]$Lisp_
    ]$Lisp,_
    ["ai"::S,"nz"::S,"ifail"::S]$Lisp,_
    [([derivArg::Any,zArg::Any,scaleArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s17dhf(derivArg:String,zArg:Complex DoubleFloat,scaleArg:String,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s17dhf",_
    ["deriv"::S,"z"::S,"scale"::S,"bi"::S,"ifail"::S_
```

```
        ]$Lisp,_
        ["bi"::S]$Lisp,_
        [["integer"::S,"ifail"::S]$Lisp_
        ,["character"::S,"deriv"::S,"scale"::S]$Lisp_
        ,["double complex"::S,"z"::S,"bi"::S]$Lisp_
        ]$Lisp,_
        ["bi"::S,"ifail"::S]$Lisp,_
        [([derivArg::Any,zArg::Any,scaleArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s17dlf(mArg:Integer,fnuArg:DoubleFloat,zArg:Complex DoubleFloat,_
        nArg:Integer,scaleArg:String,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s17dlf",_
        ["m"::S,"fnu"::S,"z"::S,"n"::S,"scale"::S_
        ,"nz"::S,"ifail"::S,"cy"::S]$Lisp,_
        ["cy"::S,"nz"::S]$Lisp,_
        [["double"::S,"fnu"::S]$Lisp_
        ,["integer"::S,"m"::S,"n"::S,"nz"::S,"ifail"::S_
        ]$Lisp_
        ,["character"::S,"scale"::S]$Lisp_
        ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["cy"::S,"nz"::S,"ifail"::S]$Lisp,_
        [([mArg::Any,fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,_
        ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18acf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s18acfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s18acfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s18adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
```

```
        ]$Lisp,_
        ["s18adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18aef",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s18aefResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s18aefResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18aff(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18aff",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s18affResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s18affResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18dcf(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
        scaleArg:String,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18dcf",_
        ["fnu"::S,"z"::S,"n"::S,"scale"::S,"nz"::S_
        ,"ifail"::S,"cy"::S]$Lisp,_
        ["cy"::S,"nz"::S]$Lisp,_
        [["double"::S,"fnu"::S]$Lisp_
        ,["integer"::S,"n"::S,"nz"::S,"ifail"::S]$Lisp_
        ,["character"::S,"scale"::S]$Lisp_
        ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["cy"::S,"nz"::S,"ifail"::S]$Lisp,_
        [([fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s18def(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,_
```

```
        scaleArg:String,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s18def",_
        ["fnu"::S,"z"::S,"n"::S,"scale"::S,"nz"::S_
        ,"ifail"::S,"cy"::S]$Lisp,_
        ["cy"::S,"nz"::S]$Lisp,_
        [["double"::S,"fnu"::S]$Lisp_
        ,["integer"::S,"n"::S,"nz"::S,"ifail"::S]$Lisp_
        ,["character"::S,"scale"::S]$Lisp_
        ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp]$Lisp_
        ]$Lisp,_
        ["cy"::S,"nz"::S,"ifail"::S]$Lisp,_
        [([fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s19aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s19aaf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s19aafResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s19aafResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s19abf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s19abf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s19abfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s19abfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s19acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s19acf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s19acfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
```

```
        ["s19acfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s19adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s19adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s19adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s19adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s20acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s20acf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s20acfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s20acfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s20adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s20adf",_
        ["x"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s20adfResult"::S,"x"::S]$Lisp_
        ,["integer"::S,"ifail"::S]$Lisp_
        ]$Lisp,_
        ["s20adfResult"::S,"ifail"::S]$Lisp,_
        [([xArg::Any,ifailArg::Any ])_
        @List Any]$Lisp)$Lisp)_
        pretend List (Record(key:Symbol,entry:Any))]$Result

    s21baf(xArg:DoubleFloat,yArg:DoubleFloat,ifailArg:Integer): Result ==
        [(invokeNagman(NIL$Lisp,_
        "s21baf",_
        ["x"::S,"y"::S,"ifail"::S]$Lisp,_
        []$Lisp,_
        [["double"::S,"s21bafResult"::S,"x"::S,"y"::S_
```

```
    ]$Lisp_
    ,["integer"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["s21bafResult"::S,"ifail"::S]$Lisp,_
    [([xArg::Any,yArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s21bbf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s21bbf",_
    ["x"::S,"y"::S,"z"::S,"ifail"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,"s21bbfResult"::S,"x"::S,"y"::S_
    ,"z"::S]$Lisp_
    ,["integer"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["s21bbfResult"::S,"ifail"::S]$Lisp,_
    [([xArg::Any,yArg::Any,zArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s21bcf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,_
    ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s21bcf",_
    ["x"::S,"y"::S,"z"::S,"ifail"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,"s21bcfResult"::S,"x"::S,"y"::S_
    ,"z"::S]$Lisp_
    ,["integer"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["s21bcfResult"::S,"ifail"::S]$Lisp,_
    [([xArg::Any,yArg::Any,zArg::Any,ifailArg::Any ])_
    @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))]$Result

s21bdf(xArg:DoubleFloat,yArg:DoubleFloat,zArg:DoubleFloat,_
    rArg:DoubleFloat,ifailArg:Integer): Result ==
    [(invokeNagman(NIL$Lisp,_
    "s21bdf",_
    ["x"::S,"y"::S,"z"::S,"r"::S,"ifail"::S_
    ]$Lisp,_
    []$Lisp,_
    [["double"::S,"s21bdfResult"::S,"x"::S,"y"::S_
    ,"z"::S,"r"::S]$Lisp_
    ,["integer"::S,"ifail"::S]$Lisp_
    ]$Lisp,_
    ["s21bdfResult"::S,"ifail"::S]$Lisp,_
```

```
[([xArg::Any,yArg::Any,zArg::Any,rArg::Any,ifailArg::Any ])_
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))]$Result
```

———-

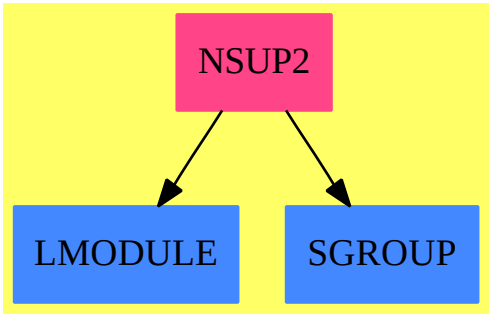
— **NAGS.dotabb** —

```
"NAGS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGS" -> "ALIST"
```

———-

15.16 package NSUP2 NewSparseUnivariatePolynomialFunctions2

15.16.1 NewSparseUnivariatePolynomialFunctions2 (NSUP2)



Exports:
map

— **package NSUP2 NewSparseUnivariatePolynomialFunctions2** —

```
)abbrev package NSUP2 NewSparseUnivariatePolynomialFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
```

```
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package lifts a mapping from coefficient rings R to S to
++ a mapping from sparse univariate polynomial over R to
++ a sparse univariate polynomial over S.
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

NewSparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
  map:(R->S,NewSparseUnivariatePolynomial R) -> NewSparseUnivariatePolynomial S
    ++ \axiom{map(func, poly)} creates a new polynomial by applying func to
    ++ every non-zero coefficient of the polynomial poly.
 == add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
           NewSparseUnivariatePolynomial R, S, NewSparseUnivariatePolynomial S)
```

———-

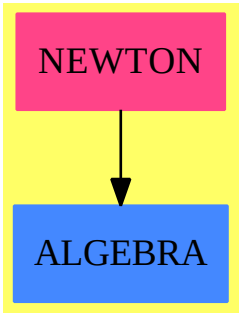
— **NSUP2.dotabb** —

```
"NSUP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NSUP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"NSUP2" -> "LMODULE"
"NSUP2" -> "SGROUP"
```

———-

15.17 package NEWTON NewtonInterpolation

15.17.1 NewtonInterpolation (NEWTON)



Exports:
newton

— package NEWTON NewtonInterpolation —

```
)abbrev package NEWTON NewtonInterpolation
++ Description:
++ This package exports Newton interpolation for the special case where the
++ result is known to be in the original integral domain
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

NewtonInterpolation F: Exports == Implementation where
    F: IntegralDomain
    Exports == with

      newton: List F -> SparseUnivariatePolynomial F

      ++ \spad{newton}(l) returns the interpolating polynomial for the values
      ++ l, where the x-coordinates are assumed to be [1,2,3,...,n] and the
      ++ coefficients of the interpolating polynomial are known to be in the
      ++ domain F. I.e., it is a very streamlined version for a special case of
      ++ interpolation.

    Implementation == add

      differences(yl: List F): List F ==
        [y2-y1 for y1 in yl for y2 in rest yl]

      z: SparseUnivariatePolynomial(F) := monomial(1,1)

-- we assume x=[1,2,3,...,n]
```

```
      newtonAux(k: F, fact: F, yl: List F): SparseUnivariatePolynomial(F) ==
        if empty? rest yl
        then ((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
        else ((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
             + (z-k::SparseUnivariatePolynomial(F)) _
               * newtonAux(k+1$F, fact*k, differences yl)


      newton yl == newtonAux(1$F, 1$F, yl)
```

———-

— **NEWTON.dotabb** —

```
"NEWTON" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NEWTON"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"NEWTON" -> "ALGEBRA"
```

———-

15.18 package NPOLYGON NewtonPolygon

— **NewtonPolygon.input** —

```
)set break resume
)sys rm -f NewtonPolygon.output
)spool NewtonPolygon.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewtonPolygon
--R
--R NewtonPolygon(K: Ring,PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(dim,NonNegativeInt
--R Abbreviation for NewtonPolygon is NPOLYGON
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NPOLYGON
--R
--R------------------------------ Operations --------------------------------
--R multiplicity : List(List(PolyRing)) -> NonNegativeInteger
--R negAndPosEdge : (PolyRing,List(List(PolyRing))) -> List(List(PolyRing))
--R newtonPolygon : (PolyRing,Integer,Integer,Union(left,center,right,vertical,horizontal)) -> List(List(Pol
--R slope : (PolyRing,PolyRing) -> Record(height: Integer,base: Integer,quotient: Integer,reste: Integer,typ
```

```
--R slope : List(PolyRing) -> Record(height: Integer,base: Integer,quotient: Integer,reste: Inte
--R
--E 1

)spool
)lisp (bye)
```

———-

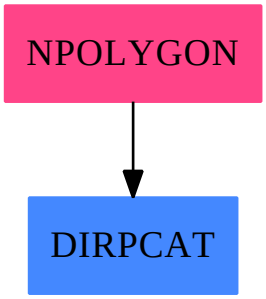
— **NewtonPolygon.help** —

```
=======================================================================
NewtonPolygon examples
=======================================================================

See Also:
o )show NewtonPolygon
```

———-

15.18.1 NewtonPolygon (NPOLYGON)



Exports:
multiplicity negAndPosEdge newtonPolygon slope

— **package NPOLYGON NewtonPolygon** —

```
)abbrev package NPOLYGON NewtonPolygon
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
```

```
++ The following is part of the PAFF package
NewtonPolygon(K,PolyRing,E,dim):Exports == Implementation where
  K:Ring
  dim:NonNegativeInteger
  PolyRing: FiniteAbelianMonoidRing(K,E)
  E: DirectProductCategory(dim,NonNegativeInteger)

  PackPoly ==> PackageForPoly(K,PolyRing,E,dim)

  recSlope ==> Record( height:Integer, base:Integer, quotient:Integer, _
                       reste:Integer, _
                  type:Union("left","center","right","vertical","horizontal"))

  Exports ==> with

    newtonPolygon: (PolyRing,Integer,Integer,Union("left","center", "right",_
                               "vertical","horizontal")) -> List List PolyRing

    multiplicity: List List PolyRing -> NonNegativeInteger

    negAndPosEdge: (PolyRing, List List PolyRing) -> List List PolyRing

    slope: (PolyRing,PolyRing) -> recSlope
    slope: List PolyRing -> recSlope

  Implementation ==> add

    slope(p1,p2)==
      -- calcule la pente de p1 a p2 et change le signe.
      e1:=degree p1
      e2:=degree p2
      hgt:= ( e1.2 pretend Integer) - ( e2.2 pretend Integer)
      bs:= ( e2.1 pretend Integer) - ( e1.1 pretend Integer )
      zero? bs =>  [hgt, bs, 0$Integer, 0$Integer, "vertical" ]$recSlope
      zero? hgt => [hgt, bs, 0$Integer, 0$Integer, "horizontal" ]$recSlope
      hgt = bs  => [hgt, bs, 1$Integer, 0$Integer, "center" ]$recSlope
      hgt > bs =>
         eucl:=divide(hgt,bs)
         [hgt, bs,  eucl.quotient, eucl.remainder , "left" ]$recSlope
      eucl:=divide(bs, hgt)
      [hgt, bs,  eucl.quotient, eucl.remainder , "right" ]$recSlope

    oneToPos:  List List PolyRing -> List List PolyRing

    oneToPos(lpol)==
      fedge:= first lpol
      sl:= slope fedge
      one? ( #(lpol) ) =>
        if sl.height > sl.base then [ fedge, empty() ]
        else [ empty() , fedge ]
```

```
  ^( sl.base < sl.height ) => [ empty() , fedge ]
  restPANE:= oneToPos  rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ^( sl2.base < sl2.height ) => [ fedge , fedge2 ]
  restPANE

oneToNeg:  List List PolyRing -> List List PolyRing

oneToNeg(lpol)==
  fedge:= first lpol
  sl:= slope fedge
  one? ( #(lpol) ) =>
    if sl.height < sl.base then [ empty(), fedge ]
    else [ fedge , empty()  ]
  ( sl.height < sl.base ) => [ empty() , fedge ]
  restPANE:= oneToNeg  rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ( sl2.height < sl2.base ) => [ fedge , fedge2 ]
  restPANE

negAndPosEdge(pol, lpol)==
  -- cette fonction retourne deux liste de polynomes:
  -- la premiere est liee a
  -- la transformation x = x y^l (i.e v(x) >= v(y) ).
  -- la deuxieme est liee a la transformation
  -- y = x^l y (i.e. v(x) <= v(y) ).
  -- si le degree en Y est inferieur a celui en X on
  -- previligie la transformation
  -- y = x^l y.
  degree( pol , 2 )$PackPoly < degree( pol, 1 )$PackPoly => oneToPos lpol
  oneToNeg lpol

localNewtonPolygon: List PolyRing -> List PolyRing

slEq: (recSlope, recSlope) -> Boolean

regroup: List PolyRing -> List List PolyRing

multiplicity( lpol )==
  nl:=#(lpol)
  flpol:= first lpol
  one? nl=> totalDegree( last flpol)$PackPoly
  s:=slope flpol
  s.height < s.base => totalDegree( first flpol )$PackPoly
  multiplicity( rest lpol )

slEq(s1,s2)==
  s1.height * s2.base = s2.height * s1.base
```

```
    regroup(lpol)==
      -- Note : les elements de lpol sont sur la frontiere d'un poly.
      -- de Newton et il sont deja trie's.
      nl:=#(lpol)
      one? nl => [lpol]
      2 = nl => [lpol]
      f:=first lpol
      r:= regroup rest lpol
      -- Note : les listes de "r" contiennent au moins 2 elements !!
      fg:=first r
      s1:=slope(f, first fg)
      s2:=slope(fg.1,fg.2)
      slEq(s1,s2) => cons( cons(f, fg) , rest r)
      cons( [f, first fg], r)

-- ================================================
-- sortMono :  trie les monomes par ordre croissant
-- ================================================

    sortMono: (PolyRing, PolyRing) -> Boolean
    sortMono(p1,p2)==
      a:= degree p1
      b:= degree p2
      a.1 < b.1 => true                  -- p1 est a gauche de p2
      a.1 = b.1 and  a.2 > b.2 => true   -- p1 est au dessus de p2
      false

-- ================================================
-- newtonPolygon :  retourne tous les monomes sur la
-- frontiere de du polygone de Newton,
-- regroupes selon leur pente.
-- ================================================

    properSlope: ( List PolyRing, Integer, Integer, _
            Union("left","center","right","vertical","horizontal")) -> Boolean

    properSlope(lpol,hgt,bs, tp)==
      s:=slope lpol
      tp case "left" and s.height = hgt and s.base = bs => true
      tp case "right" and s.height = bs and s.base = hgt => true
      false


    newtonPolygon(pol,hgt,bs,tp)==
      ans:=regroup localNewtonPolygon _
                              sort( sortMono(#1,#2) , monomials(pol)$PackPoly)
      zero?(bs) => ans
      [ l for l in ans | properSlope(l,hgt,bs,tp)]
```

```
comp2pol: (PolyRing,PolyRing) -> List PolyRing
comp2pol(p1,p2)==
  rs:= slope(p1,p2)
  zero? rs.base =>               -- p1 et p2 sont alignes verticalement !!
    zero? rs.height => [p1 + p2] -- les monomes sont identiques !
    rs.height < 0 => [p1]        -- p2 est au dessus de p1,
                                 -- il faut retourner p1 !!
    [p2]                         -- sinon p1 est au dessus de p2 .
  rs.base > 0 =>                 -- p1 est a gauche de p2
    rs.height > 0 => [p1,p2]     -- p1 est plus haut que p2
    [p1]                         -- p1 est a la meme hauteur que p2
                                 -- ici p2 est a gauche de p1
  rs.height < 0 => [p2,p1]       -- p2 est plus haut que p1
  [p2]                           -- p2 est a la meme hauteur que p1.

slope(lpol) ==
  ^one?(#lpol) => slope( first lpol, second lpol)
  f:= first lpol
  ( degree(f,2)$PackPoly < degree(f,1)$PackPoly ) => _
                          [ 0$Integer, 1$Integer,0,0, "right" ]$recSlope
  [1$Integer, 0$Integer,0,0 , "left"  ]$recSlope

convex_?: (PolyRing,PolyRing,PolyRing) -> Boolean
convex_?(p1,p2,p3)==
  s1:=slope(p1,p2)
  s2:=slope(p2,p3)
  s1.type case "horizontal" => true
  s2.type case "vertical" => true
  s1.type case "vertical" => false  -- car ici il faut c2 vertical
  s2.type case "horizontal" => false
  (s1.height * s2.base) < (s2.height  * s1.base)

consBondary: (PolyRing , List PolyRing) -> List PolyRing
consBondary(lt, lpol)==
  -- "lt" est un monome a ajouter ou non a "lpol" qui est une
  empty? lpol => [lt]
  st:=first lpol
  nl:NonNegativeInteger:= # lpol
  one? nl => comp2pol(lt,st)
  degree(lt).1 = degree(st).1 and degree(lt).2 > degree(st).2 => lpol
  ^convex?(lt , st , lpol.2) => cons(lt, lpol)
  consBondary( lt, rest lpol )

localNewtonPolygon(lpol)==
  -- lpol doit etre trie' par sortMono
  empty? lpol =>  empty()
  nl:= #(lpol)
  one?  nl => lpol
  lt:=first lpol
  polgRest:= localNewtonPolygon rest lpol
```

```
consBondary( lt , polgRest )
```

———-

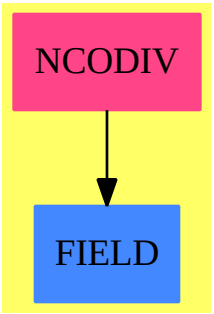
— **NPOLYGON.dotabb** —

```
"NPOLYGON" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPOLYGON"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"NPOLYGON" -> "DIRPCAT"
```

———-

15.19 package NCODIV NonCommutativeOperatorDivision

15.19.1 NonCommutativeOperatorDivision (NCODIV)



Exports:
leftGcd leftLcm leftQuotient leftRemainder leftDivide leftExactQuotient

— **package NCODIV NonCommutativeOperatorDivision** —

```
)abbrev package NCODIV NonCommutativeOperatorDivision
++ Author: Jean Della Dora, Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Basic Operations:
++ Related Domains: LinearOrdinaryDifferentialOperator
++ Also See:
++ AMS Classifications:
++ Keywords: gcd, lcm, division, non-commutative
```

```
++ Examples:
++ References:
++ Description:
++ This package provides a division and related operations for
++ \spadtype{MonogenicLinearOperator}s over a \spadtype{Field}.
++ Since the multiplication is in general non-commutative,
++ these operations all have left- and right-hand versions.
++ This package provides the operations based on left-division.\br
++ \tab{5}[q,r] = leftDivide(a,b) means a=b*q+r

NonCommutativeOperatorDivision(P, F): PDcat == PDdef  where
    P: MonogenicLinearOperator(F)
    F: Field

    PDcat == with
        leftDivide:   (P, P) -> Record(quotient: P, remainder: P)
            ++ leftDivide(a,b) returns the pair \spad{[q,r]} such that
            ++ \spad{a = b*q + r} and the degree of \spad{r} is
            ++ less than the degree of \spad{b}.
            ++ This process is called ``left division''.
        leftQuotient:  (P, P) -> P
            ++ leftQuotient(a,b) computes the pair \spad{[q,r]} such that
            ++ \spad{a = b*q + r} and the degree of \spad{r} is
            ++ less than the degree of \spad{b}.
            ++ The value \spad{q} is returned.
        leftRemainder:  (P, P) -> P
            ++ leftRemainder(a,b) computes the pair \spad{[q,r]} such that
            ++ \spad{a = b*q + r} and the degree of \spad{r} is
            ++ less than the degree of \spad{b}.
            ++ The value \spad{r} is returned.
        leftExactQuotient:(P, P) -> Union(P, "failed")
            ++ leftExactQuotient(a,b) computes the value \spad{q}, if it exists,
            ++  such that \spad{a = b*q}.

        leftGcd:   (P, P) -> P
            ++ leftGcd(a,b) computes the value \spad{g} of highest degree
            ++ such that
            ++    \spad{a = aa*g}
            ++    \spad{b = bb*g}
            ++ for some values \spad{aa} and \spad{bb}.
            ++ The value \spad{g} is computed using left-division.
        leftLcm:   (P, P) -> P
            ++ leftLcm(a,b) computes the value \spad{m} of lowest degree
            ++ such that \spad{m = a*aa = b*bb} for some values
            ++ \spad{aa} and \spad{bb}.  The value \spad{m} is
            ++ computed using left-division.

    PDdef == add
        leftDivide(a, b) ==
            q: P := 0
```

```
        r: P := a
        iv:F := inv leadingCoefficient b
        while degree r >= degree b and r ^= 0 repeat
            h := monomial(iv*leadingCoefficient r,
                          (degree r - degree b)::NonNegativeInteger)$P
            r := r - b*h
            q := q + h
        [q,r]

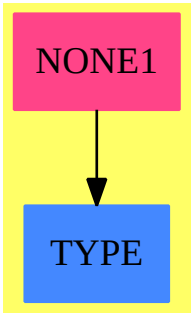
    -- leftQuotient(a,b) is the quotient from left division, etc.
    leftQuotient(a,b)   == leftDivide(a,b).quotient
    leftRemainder(a,b)   == leftDivide(a,b).remainder
    leftExactQuotient(a,b) ==
         qr := leftDivide(a,b)
         if qr.remainder = 0 then qr.quotient else "failed"
    -- l = leftGcd(a,b) means  a = aa*l  b = bb*l.  Uses leftDivide.
    leftGcd(a,b) ==
         a = 0 =>b
         b = 0 =>a
         while degree b > 0 repeat (a,b) := (b, leftRemainder(a,b))
         if b=0 then a else b
    -- l = leftLcm(a,b) means  l = a*aa  l = b*bb   Uses leftDivide.
    leftLcm(a,b) ==
        a = 0 =>b
        b = 0 =>a
        b0 := b
        u  := monomial(1,0)$P
        v  := 0
        while leadingCoefficient b ^= 0 repeat
            qr     := leftDivide(a,b)
            (a, b) := (b, qr.remainder)
            (u, v) := (u*qr.quotient+v, u)
        b0*u
```

— NCODIV.dotabb —

```
"NCODIV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCODIV"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"NCODIV" -> "FIELD"
```

15.20 package NONE1 NoneFunctions1

15.20.1 NoneFunctions1 (NONE1)



Exports:
coerce

— package NONE1 NoneFunctions1 —

```
)abbrev package NONE1 NoneFunctions1
++ Author:
++ Date Created:
++ Change History:
++ Basic Functions: coerce
++ Related Constructors: None
++ Also See:
++ AMS Classification:
++ Keywords:
++ Description:
++ \spadtype{NoneFunctions1} implements functions on \spadtype{None}.
++ It particular it includes a particulary dangerous coercion from
++ any other type to \spadtype{None}.

NoneFunctions1(S:Type): Exports == Implementation where
  Exports ==> with
    coerce: S -> None
      ++ coerce(x) changes \spad{x} into an object of type
      ++ \spadtype{None}.

  Implementation ==> add
    coerce(s:S):None == s pretend None
```

———-

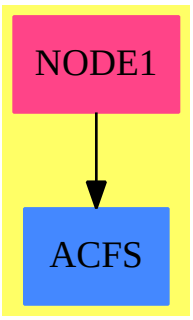
— NONE1.dotabb —

```
"NONE1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NONE1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"NONE1" -> "TYPE"
```

———-

15.21 package NODE1 NonLinearFirstOrderODESolver

15.21.1 NonLinearFirstOrderODESolver (NODE1)



Exports:
solve

— package NODE1 NonLinearFirstOrderODESolver —

```
)abbrev package NODE1 NonLinearFirstOrderODESolver
++ Author: Manuel Bronstein
++ Date Created: 2 September 1991
++ Date Last Updated: 14 October 1994
++ Keywords: differential equation, ODE
++ Description:
++ NonLinearFirstOrderODESolver provides a function
++ for finding closed form first integrals of nonlinear ordinary
++ differential equations of order 1.

NonLinearFirstOrderODESolver(R, F): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
          PrimitiveFunctionCategory)

  N   ==> NonNegativeInteger
  Q   ==> Fraction Integer
  UQ  ==> Union(Q, "failed")
```

```
  OP  ==> BasicOperator
  SY  ==> Symbol
  K   ==> Kernel F
  U   ==> Union(F, "failed")
  P   ==> SparseMultivariatePolynomial(R, K)
  REC ==> Record(coef:Q, logand:F)
  SOL ==> Record(particular: F,basis: List F)
  BER ==> Record(coef1:F, coefn:F, exponent:N)

  Exports ==> with
    solve: (F, F, OP, SY) -> U
      ++ solve(M(x,y), N(x,y), y, x) returns \spad{F(x,y)} such that
      ++ \spad{F(x,y) = c} for a constant \spad{c} is a first integral
      ++ of the equation \spad{M(x,y) dx + N(x,y) dy  = 0}, or
      ++ "failed" if no first-integral can be found.

  Implementation ==> add
    import ODEIntegration(R, F)
    import ElementaryFunctionODESolver(R, F)    -- recursive dependency!

    checkBernoulli   : (F, F, K) -> Union(BER, "failed")
    solveBernoulli   : (BER, OP, SY, F) -> Union(F, "failed")
    checkRiccati     : (F, F, K) -> Union(List F, "failed")
    solveRiccati     : (List F, OP, SY, F) -> Union(F, "failed")
    partSolRiccati   : (List F, OP, SY, F) -> Union(F, "failed")
    integratingFactor: (F, F, SY, SY) -> U

    unk    := new()$SY
    kunk:K := kernel unk

    solve(m, n, y, x) ==
-- first replace the operator y(x) by a new symbol z in m(x,y) and n(x,y)
      lk:List(K) := [retract(yx := y(x::F))@K]
      lv:List(F) := [kunk::F]
      mm := eval(m, lk, lv)
      nn := eval(n, lk, lv)
-- put over a common denominator (to balance m and n)
      d := lcm(denom mm, denom nn)::F
      mm := d * mm
      nn := d * nn
-- look for an integrating factor mu
      (u := integratingFactor(mm, nn, unk, x)) case F =>
        mu := u::F
        mm := mm * mu
        nn := nn * mu
        eval(int(mm,x) + int(nn-int(differentiate(mm,unk),x), unk),[kunk],[yx])
-- check for Bernoulli equation
      (w := checkBernoulli(m, n, k1 := first lk)) case BER =>
        solveBernoulli(w::BER, y, x, yx)
-- check for Riccati equation
```

```
      (v := checkRiccati(m, n, k1)) case List(F) =>
        solveRiccati(v::List(F), y, x, yx)
      "failed"

-- look for an integrating factor
    integratingFactor(m, n, y, x) ==
-- check first for exactness
      zero?(d := differentiate(m, y) - differentiate(n, x)) => 1
-- look for an integrating factor involving x only
      not member?(y, variables(f := d / n)) => expint(f, x)
-- look for an integrating factor involving y only
      not member?(x, variables(f := - d / m)) => expint(f, y)
-- room for more techniques later on (e.g. Prelle-Singer etc...)
      "failed"

-- check whether the equation is of the form
--    dy/dx + p(x)y + q(x)y^N = 0   with N > 1
-- i.e. whether m/n is of the form  p(x) y + q(x) y^N
-- returns [p, q, N] if the equation is in that form
    checkBernoulli(m, n, ky) ==
      r := denom(f := m / n)::F
      (not freeOf?(r, y := ky::F))
          or (d := degree(p := univariate(numer f, ky))) < 2
            or degree(pp := reductum p) ^= 1 or reductum(pp) ^= 0
              or (not freeOf?(a := (leadingCoefficient(pp)::F), y))
                or (not freeOf?(b := (leadingCoefficient(p)::F), y)) => "failed"
      [a / r, b / r, d]

-- solves the equation dy/dx + rec.coef1 y + rec.coefn y^rec.exponent = 0
-- the change of variable v = y^{1-n} transforms the above equation to
--  dv/dx + (1 - n) p v + (1 - n) q = 0
    solveBernoulli(rec, y, x, yx) ==
      n1 := 1 - rec.exponent::Integer
      deq := differentiate(yx, x) + n1 * rec.coef1 * yx + n1 * rec.coefn
      sol := solve(deq, y, x)::SOL          -- can always solve for order 1
-- if v = vp + c v0 is the general solution of the linear equation, then
-- the general first integral for the Bernoulli equation is
-- (y^{1-n} - vp) / v0  =   c   for any constant c
      (yx**n1 - sol.particular) / first(sol.basis)

-- check whether the equation is of the form
--    dy/dx + q0(x) + q1(x)y + q2(x)y^2 = 0
-- i.e. whether m/n is a quadratic polynomial in y.
-- returns the list [q0, q1, q2] if the equation is in that form
    checkRiccati(m, n, ky) ==
      q := denom(f := m / n)::F
      (not freeOf?(q, y := ky::F)) or degree(p := univariate(numer f, ky)) > 2
         or (not freeOf?(a0 := (coefficient(p, 0)::F), y))
           or (not freeOf?(a1 := (coefficient(p, 1)::F), y))
             or (not freeOf?(a2 := (coefficient(p, 2)::F), y)) => "failed"
```

```
      [a0 / q, a1 / q, a2 / q]

-- solves the equation dy/dx + l.1 + l.2 y + l.3 y^2 = 0
    solveRiccati(l, y, x, yx) ==
-- get first a particular solution
      (u := partSolRiccati(l, y, x, yx)) case "failed" => "failed"
-- once a particular solution yp is known, the general solution is of the
-- form  y = yp + 1/v  where v satisfies the linear 1st order equation
-- v' - (l.2 + 2 l.3 yp) v = l.3
      deq := differentiate(yx, x) - (l.2 + 2 * l.3 * u::F) * yx - l.3
      gsol := solve(deq, y, x)::SOL         -- can always solve for order 1
-- if v = vp + c v0 is the general solution of the above equation, then
-- the general first integral for the Riccati equation is
--  (1/(y - yp) - vp) / v0  =   c   for any constant c
      (inv(yx - u::F) - gsol.particular) / first(gsol.basis)

-- looks for a particular solution of dy/dx + l.1 + l.2 y + l.3 y^2 = 0
    partSolRiccati(l, y, x, yx) ==
-- we first do the change of variable y = z / l.3, which transforms
-- the equation into  dz/dx + l.1 l.3 + (l.2 - l.3'/l.3) z + z^2 = 0
      q0 := l.1 * (l3 := l.3)
      q1 := l.2 - differentiate(l3, x) / l3
-- the equation dz/dx + q0 + q1 z + z^2 = 0 is transformed by the change
-- of variable z = w'/w into the linear equation w'' + q1 w' + q0 w = 0
      lineq := differentiate(yx, x, 2) + q1 * differentiate(yx, x) + q0 * yx
-- should be made faster by requesting a particular nonzero solution only
      (not((gsol := solve(lineq, y, x)) case SOL))
                              or empty?(bas := (gsol::SOL).basis) => "failed"
      differentiate(first bas, x) / (l3 * first bas)
```

———-

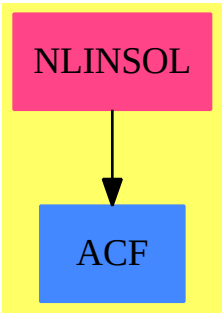
— NODE1.dotabb —

```
"NODE1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NODE1"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"NODE1" -> "ACFS"
```

———-

15.22 package NLINSOL NonLinearSolvePackage

15.22.1 NonLinearSolvePackage (NLINSOL)



Exports:
solve solveInField

— package NLINSOL NonLinearSolvePackage —

```
)abbrev package NLINSOL NonLinearSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 26 June 1992
++ Description:
++ NonLinearSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving. The solutions are given in the algebraic closure of R whenever
++ possible.

NonLinearSolvePackage(R:IntegralDomain): Exports == Implementation where
  Z   ==> Integer
  Q   ==> Fraction Z
  SY  ==> Symbol
  P   ==> Polynomial R
  F   ==> Fraction P
  EQ  ==> Equation F
  SSP ==> SystemSolvePackage
  SOL ==> RetractSolvePackage

  Exports ==> with
    solveInField:    (List P, List SY) -> List List EQ
      ++ solveInField(lp,lv) finds the solutions of the list lp of
      ++ rational functions with respect to the list of symbols lv.
    solveInField:     List P       -> List List EQ
      ++ solveInField(lp) finds the solution of the list lp of rational
      ++ functions with respect to all the symbols appearing in lp.
```

```
    solve:           (List P, List SY) -> List List EQ
      ++ solve(lp,lv) finds the solutions in the algebraic closure of R
      ++ of the list lp of
      ++ rational functions with respect to the list of symbols lv.
    solve:            List P       -> List List EQ
      ++ solve(lp) finds the solution in the algebraic closure of R
      ++ of the list lp of rational
      ++ functions with respect to all the symbols appearing in lp.

  Implementation ==> add
    solveInField l == solveInField(l, "setUnion"/[variables p for p in l])

    if R has AlgebraicallyClosedField then
      import RationalFunction(R)

      expandSol: List EQ -> List List EQ
      RIfCan   : F -> Union(R, "failed")
      addRoot  : (EQ, List List EQ) -> List List EQ
      allRoots : List P -> List List EQ
      evalSol  : (List EQ, List EQ) -> List EQ

      solve l        == solve(l, "setUnion"/[variables p for p in l])
      solve(lp, lv)  == concat([expandSol sol for sol in solveInField(lp, lv)])
      addRoot(eq, l) == [concat(eq, sol) for sol in l]
      evalSol(ls, l) == [equation(lhs eq, eval(rhs eq, l)) for eq in ls]

-- converts [p1(a1),...,pn(an)] to
-- [[a1=v1,...,an=vn]] where vi ranges over all the zeros of pi
      allRoots l ==
        empty? l => [empty()$List(EQ)]
        z := allRoots rest l
        s := mainVariable(p := first l)::SY::P::F
        concat [addRoot(equation(s, a::P::F), z) for a in zerosOf univariate p]

      expandSol l ==
        lassign := lsubs := empty()$List(EQ)
        luniv := empty()$List(P)
        for eq in l repeat
          if retractIfCan(lhs eq)@Union(SY, "failed") case SY then
            if RIfCan(rhs eq) case R then lassign := concat(eq, lassign)
                                     else lsubs := concat(eq, lsubs)
          else
            if ((u := retractIfCan(lhs eq)@Union(P, "failed")) case P) and
--               one?(# variables(u::P)) and ((r := RIfCan rhs eq) case R) then
                ((# variables(u::P)) = 1) and ((r := RIfCan rhs eq) case R) then
                  luniv := concat(u::P - r::R::P, luniv)
            else return [l]
        empty? luniv => [l]
        [concat(z, concat(evalSol(lsubs,z), lassign)) for z in allRoots luniv]
```

```
      RIfCan f ==
        ((n := retractIfCan(numer f)@Union(R,"failed")) case R) and
          ((d := retractIfCan(denom f)@Union(R,"failed")) case R) => n::R / d::R
        "failed"
    else
      solve l       == solveInField l
      solve(lp, lv) == solveInField(lp, lv)

 -- 'else if' is doubtful with this compiler so all 3 conditions are explicit
    if (not(R is Q)) and (R has RetractableTo Q) then
      solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Q, R)

    if (not(R is Z)) and (not(R has RetractableTo Q)) and
      (R has RetractableTo Z) then
        solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Z, R)

    if (not(R is Z)) and (not(R has RetractableTo Q)) and
      (not(R has RetractableTo Z)) then
        solveInField(lp, lv) == solve([p::F for p in lp]$List(F), lv)$SSP(R)
```

———-

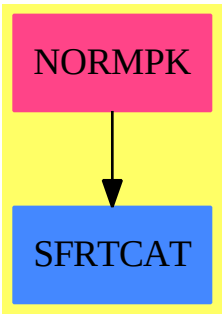
— **NLINSOL.dotabb** —

```
"NLINSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NLINSOL"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"NLINSOL" -> "ACF"
```

———-

15.23 package NORMPK NormalizationPackage

15.23.1 NormalizationPackage (NORMPK)



Exports:
normalizedAssociate normInvertible? normalize outputArgs recip

— package NORMPK NormalizationPackage —

```
)abbrev package NORMPK NormalizationPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++  [1] D. LAZARD "A new method for solving algebraic systems of
++      positive dimension" Discr. App. Math. 33:147-160,1991
++  [2] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++      algebraic towers of simple extensions" In proceedings of AAECC11
++      Paris, 1995.
++  [3] M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++      d'extensions simples et resolution des systemes d'equations
++      algebriques" These, Universite P.etM. Curie, Paris, 1997.
++ Description:
++ A package for computing normalized assocites of univariate polynomials
++ with coefficients in a tower of simple extensions of a field.

NormalizationPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : RegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  K ==> Fraction R
  LP ==> List P
  PWT ==> Record(val : P, tower : TS)

  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Split ==> List TS
  --KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
  --EntryGcd ==> List PWT
  --HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
  --KeyInvSet ==> Record(arg1: P, arg3: TS)
```

```
--EntryInvSet ==> List TS
--HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports ==  with

   recip: (P, TS) -> Record(num:P,den:P)
     ++ \axiom{recip(p,ts)} returns the inverse of \axiom{p} w.r.t \spad{ts}
     ++ assuming that \axiom{p} is invertible w.r.t \spad{ts}.
   normalizedAssociate: (P, TS) -> P
     ++ \axiom{normalizedAssociate(p,ts)} returns a normalized polynomial
     ++ \axiom{n} w.r.t. \spad{ts} such that \axiom{n} and \axiom{p} are
     ++ associates w.r.t \spad{ts} and assuming that \axiom{p} is invertible
     ++ w.r.t \spad{ts}.
   normalize: (P, TS) -> List PWT
     ++ \axiom{normalize(p,ts)} normalizes \axiom{p} w.r.t \spad{ts}.
   outputArgs: (S, S, P, TS) -> Void
     ++ \axiom{outputArgs(s1,s2,p,ts)}
     ++ is an internal subroutine, exported only for developement.
   normInvertible?: (P, TS) -> List BWT
     ++ \axiom{normInvertible?(p,ts)}
     ++ is an internal subroutine, exported only for developement.

Implementation == add

   if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P)
   then

     normInvertible?(p:P, ts:TS): List BWT ==
       stoseInvertible?_sqfreg(p,ts)$regsetgcdpack

   else

     normInvertible?(p:P, ts:TS): List BWT ==
       stoseInvertible?_reg(p,ts)$regsetgcdpack

   if (R has RetractableTo(Integer)) and (V has ConvertibleTo(Symbol))
   then

     outputArgs(s1:S, s2: S, p:P,ts:TS): Void ==
       if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
       if not empty? s1 then _
            output(s1,(convert(p)@String)::OutputForm)$OutputPackage
       output(" ")$OutputPackage
       if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
       empty? s2 => void()
       output(s2,("[")::OutputForm)$OutputPackage
       lp: List P := members(ts)
       for q in lp repeat
```

```
        output((convert(q)@String)::OutputForm)$OutputPackage
      output("]")$OutputPackage
      output(" ")$OutputPackage

  else

    outputArgs(s1:S, s2: S, p:P,ts:TS): Void ==
      if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
      output(" ")$OutputPackage
      if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
      output(" ")$OutputPackage

  recip(p:P,ts:TS): Record(num:P, den:P) ==
  -- ASSUME p is invertible w.r.t. ts
  -- ASSUME mvar(p) is algebraic w.r.t. ts
    v := mvar(p)
    ts_v := select(ts,v)::P
    if mdeg(p) < mdeg(ts_v)
      then
        hesrg: Record (gcd : P, coef2 : P)  := _
                 halfExtendedSubResultantGcd2(ts_v,p)$P
        d: P :=  hesrg.gcd; n: P := hesrg.coef2
      else
        hesrg: Record (gcd : P, coef1 : P) := _
                 halfExtendedSubResultantGcd1(p,ts_v)$P
        d: P :=  hesrg.gcd; n: P := hesrg.coef1
    g := gcd(n,d)
    (n, d) := ((n exquo g)::P, (d exquo g)::P)
    remn, remd: Record(rnum:R,polnum:P,den:R)
    remn := remainder(n,ts); remd := remainder(d,ts)
    cn := remn.rnum; pn := remn.polnum; dn := remn.den
    cd := remd.rnum; pd := remd.polnum; dp := remd.den
    k: K := (cn / cd) * (dp / dn)
    pn := removeZero(pn,ts)
    pd := removeZero(pd,ts)
    [numer(k) * pn, denom(k) * pd]$Record(num:P, den:P)

  normalizedAssociate(p:P,ts:TS): P ==
  -- ASSUME p is invertible or zero w.r.t. ts
    empty? ts => p
    zero?(p) => p
    ground?(p) => 1
    zero? initiallyReduce(init(p),ts) =>
      error "in normalizedAssociate$NORMPK: bad #1"
    vp := mvar(p)
    ip: P := p
    mp: P := 1
    tp: P := 0
    while not ground?(ip) repeat
      v := mvar(ip)
```

```
      if algebraic?(v,ts)
        then
          if v = vp
            then
              ts_v := select(ts,v)::P
              ip := lastSubResultant(ip,ts_v)$P
              ip := remainder(ip,ts).polnum
              -- ip := primitivePart stronglyReduce(ip,ts)
              ip := primitivePart initiallyReduce(ip,ts)
            else
              qr := recip(ip,ts)
              ip := qr.den
              tp := qr.num * tp
              zero? ip =>
                  outputArgs("p = ", " ts = ",p,ts)
                  error _
                    "in normalizedAssociate$NORMPK: should never happen !"
        else
          tp := tail(ip) * mp + tp
          mp := mainMonomial(ip) * mp
          ip := init(ip)
    r := ip * mp + tp
    r := remainder(r,ts).polnum
    -- primitivePart stronglyReduce(r,ts)
    primitivePart initiallyReduce(r,ts)

  normalize(p: P, ts: TS): List PWT ==
    zero? p => [[p,ts]$PWT]
    ground? p => [[1,ts]$PWT]
    zero? initiallyReduce(init(p),ts) =>
      error "in normalize$NORMPK: init(#1) reduces to 0 w.r.t. #2"
    --output("Entering  normalize")$OutputPackage
    --outputArgs("p = ", " ts = ",p,ts)
    --output("Calling  normInvertible?")$OutputPackage
    lbwt: List BWT := normInvertible?(p,ts)
    --output("Result is: ")$OutputPackage
    --output(lbwt::OutputForm)$OutputPackage
    lpwt: List PWT := []
    for bwt in lbwt repeat
      us := bwt.tower
      q := remainder(p,us).polnum
      q := removeZero(q,us)
      bwt.val =>
        --output("Calling  normalizedAssociate")$OutputPackage
        --outputArgs("q = ", " us = ",q,us)
        lpwt := cons([normalizedAssociate(q,us)@P,us]$PWT, lpwt)
        --output("Leaving  normalizedAssociate")$OutputPackage
      zero? q => lpwt := cons([0$P,us]$PWT, lpwt)
      lpwt := concat(normalize(q,us)@(List PWT),lpwt)
    lpwt
```

———-

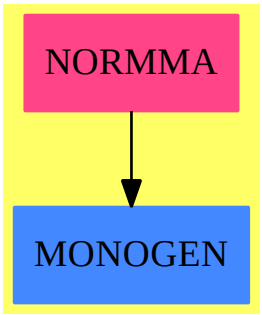
— NORMPK.dotabb —

```
"NORMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"NORMPK" -> "SFRTCAT"
```

———-

15.24 package NORMMA NormInMonogenicAlgebra

15.24.1 NormInMonogenicAlgebra (NORMMA)



Exports:
norm

— package NORMMA NormInMonogenicAlgebra —

```
)abbrev package NORMMA NormInMonogenicAlgebra
++ Author: Manuel Bronstein
++ Date Created: 23 February 1995
++ Date Last Updated: 23 February 1995
++ Basic Functions: norm
++ Description:
++ This package implements the norm of a polynomial with coefficients
++ in a monogenic algebra (using resultants)

NormInMonogenicAlgebra(R, PolR, E, PolE): Exports == Implementation where
  R: GcdDomain
  PolR: UnivariatePolynomialCategory R
```

```
  E: MonogenicAlgebra(R, PolR)
  PolE: UnivariatePolynomialCategory E

  SUP ==> SparseUnivariatePolynomial

  Exports ==> with
    norm: PolE -> PolR
      ++ norm q returns the norm of q,
      ++ i.e. the product of all the conjugates of q.

  Implementation ==> add
    import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, SUP PolR)

    PolR2SUP: PolR -> SUP PolR
    PolR2SUP q == map(x +-> x::PolR, q)

    defpol := PolR2SUP(definingPolynomial()$E)

    norm q ==
      p:SUP PolR := 0
      while q ~= 0 repeat
        p := p + monomial(1,degree q)$PolR * PolR2SUP lift leadingCoefficient q
        q := reductum q
      primitivePart resultant(p, defpol)
```

———-

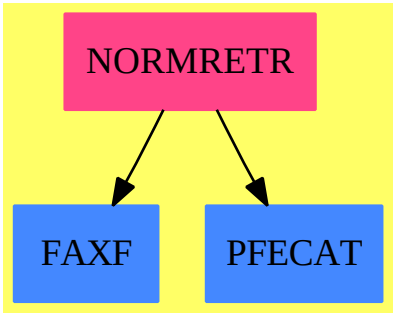
— **NORMMA.dotabb** —

```
"NORMMA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMMA"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"NORMMA" -> "MONOGEN"
```

———-

15.25 package NORMRETR NormRetractPackage

15.25.1 NormRetractPackage (NORMRETR)



Exports:
Frobenius normFactors retractIfCan

— **package NORMRETR NormRetractPackage** —

```
)abbrev package NORMRETR NormRetractPackage
++ Description:
++ This package has no description

NormRetractPackage(F, ExtF, SUEx, ExtP, n):C  == T where
  F          :   FiniteFieldCategory
  ExtF       :   FiniteAlgebraicExtensionField(F)
  SUEx       :   UnivariatePolynomialCategory ExtF
  ExtP       :   UnivariatePolynomialCategory SUEx
  n          :   PositiveInteger
  SUP       ==>  SparseUnivariatePolynomial
  R         ==>  SUP F
  P         ==>  SUP R

  C  ==> with
      normFactors : ExtP -> List ExtP
        ++ normFactors(x) \undocumented
      retractIfCan : ExtP -> Union(P, "failed")
        ++ retractIfCan(x) \undocumented
      Frobenius    : ExtP -> ExtP
        ++ Frobenius(x) \undocumented

  T  ==> add

      normFactors(p:ExtP):List ExtP ==
          facs : List ExtP := [p]
          for i in 1..n-1 repeat
```

```
            member?((p := Frobenius p), facs) => return facs
            facs := cons(p, facs)
         facs

      Frobenius(ff:ExtP):ExtP ==
         fft:ExtP:=0
         while ff^=0 repeat
           fft:=fft + monomial(map(Frobenius, leadingCoefficient ff),
                               degree ff)
           ff:=reductum ff
         fft

      retractIfCan(ff:ExtP):Union(P, "failed") ==
         fft:P:=0
         while ff ^= 0 repeat
           lc : SUEx := leadingCoefficient ff
           plc: SUP F := 0
           while lc ^= 0 repeat
              lclc:ExtF := leadingCoefficient lc
              (retlc := retractIfCan lclc) case "failed" => return "failed"
              plc := plc + monomial(retlc::F, degree lc)
              lc := reductum lc
           fft:=fft+monomial(plc, degree ff)
           ff:=reductum ff
         fft
```

———-

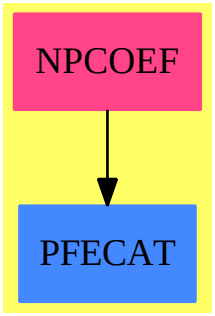
— NORMRETR.dotabb —

```
"NORMRETR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMRETR"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"NORMRETR" -> "FAXF"
"NORMRETR" -> "PFECAT"
```

———-

15.26 package NPCOEF NPCoef

15.26.1 NPCoef (NPCOEF)



Exports:
listexp npcoef

— package NPCOEF NPCoef —

```
)abbrev package NPCOEF NPCoef
++ Author : P.Gianni, revised May 1990
++ Description:
++ Package for the determination of the coefficients in the lifting
++ process. Used by \spadtype{MultivariateLifting}.
++ This package will work for every euclidean domain R which has property
++ F, i.e. there exists a factor operation in \spad{R[x]}.

NPCoef(BP,E,OV,R,P) : C == T where

 OV   :   OrderedSet
 E    :   OrderedAbelianMonoidSup
 R    :   EuclideanDomain  -- with property "F"
 BP   :   UnivariatePolynomialCategory R
 P    :   PolynomialCategory(R,E,OV)

 Z       ==> Integer
 NNI     ==> NonNegativeInteger
 USP     ==> SparseUnivariatePolynomial(P)
 Term    ==> Record(expt:NNI,pcoef:P)
 Detc    ==> Record(valexp:NNI,valcoef:P,posit:NNI)
 VTerm   ==> List(Term)
 DetCoef ==> Record(deter:List(USP),dterm:List(VTerm),
                    nfacts:List(BP),nlead:List(P))
 TermC   ==> Record(coefu:P,detfacts:List(VTerm))
 TCoef   ==> List(TermC)
```

```
C == with
     npcoef   :    (USP,List(BP),List(P))      ->   DetCoef
       ++ npcoef \undocumented
     listexp  :              BP                ->   List(NNI)
       ++ listexp \undocumented
T == add

               ----   Local  Functions  ----
 check      : (TermC,Vector P) -> Union(Detc,"failed")
 buildvect  : (List(VTerm),NNI) -> Vector(List(VTerm))
 buildtable : (Vector(P),List(List NNI),List P) -> TCoef
 modify : (TCoef,Detc) -> TCoef
 constructp : VTerm -> USP

 npcoef(u:USP,factlist:List(BP),leadlist:List(P)) :DetCoef ==
   detcoef:List(VTerm):=empty();detufact:List(USP):=empty()
   lexp:List(List(NNI)):=[listexp(v) for v in factlist]
   ulist :Vector(P):=vector [coefficient(u,i) for i in 0..degree u]
   tablecoef:=buildtable(ulist,lexp,leadlist)
   detcoef:=[[[ep.first,lcu]$Term]  for ep in lexp for lcu in leadlist]
   ldtcf:=detcoef
   lexp:=[ep.rest for ep in lexp]
   ndet:NNI:=#factlist
   changed:Boolean:=true
   ltochange:List(NNI):=empty()
   ltodel:List(NNI):=empty()
   while changed and ndet^=1 repeat
     changed :=false
     dt:=#tablecoef
     for i in 1..dt while ^changed repeat
       (cf:=check(tablecoef.i,ulist)) case "failed" => "next i"
       ltochange:=cons(i,ltochange)
       celtf:Detc:=cf::Detc
       tablecoef:=modify(tablecoef,celtf)
       vpos:=celtf.posit
       vexp:=celtf.valexp
       nterm:=[vexp,celtf.valcoef]$Term
       detcoef.vpos:=cons(nterm,detcoef.vpos)
       lexp.vpos:=delete(lexp.vpos,position(vexp,lexp.vpos))
       if lexp.vpos=[] then
        ltodel:=cons(vpos,ltodel)
        ndet:=(ndet-1):NNI
        detufact:=cons(constructp(detcoef.vpos),detufact)
       changed:=true
     for i in ltochange repeat tablecoef:=delete(tablecoef,i)
     ltochange:=[]
   if ndet=1 then
    uu:=u exquo */[pol for pol in detufact]
    if uu case "failed" then return
      [empty(),ldtcf,factlist,leadlist]$DetCoef
```

```
    else  detufact:=cons(uu::USP,detufact)
  else
    ltodel:=sort((n1:NNI,n2:NNI):Boolean +-> n1>n2,ltodel)
    for i in ltodel repeat
      detcoef:=delete(detcoef,i)
      factlist:=delete(factlist,i)
      leadlist:=delete(leadlist,i)
  [detufact,detcoef,factlist,leadlist]$DetCoef


check(tterm:TermC,ulist:Vector(P)) : Union(Detc,"failed") ==
  cfu:P:=1$P;doit:NNI:=0;poselt:NNI:=0;pp:Union(P,"failed")
  termlist:List(VTerm):=tterm.detfacts
  vterm:VTerm:=empty()
  #termlist=1 =>
    vterm:=termlist.first
    for elterm in vterm while doit<2 repeat
      (cu1:=elterm.pcoef)^=0 => cfu:=cu1*cfu
      doit:=doit+1
      poselt:=position(elterm,vterm):NNI
    doit=2  or (pp:=tterm.coefu exquo cfu) case "failed" => "failed"
    [vterm.poselt.expt,pp::P,poselt]$Detc
  "failed"

buildvect(lvterm:List(VTerm),n:NNI) : Vector(List(VTerm)) ==
  vtable:Vector(List(VTerm)):=new(n,empty())
  (#lvterm)=1 =>
    for term in lvterm.first repeat vtable.(term.expt+1):=[[term]]
    vtable

  vtable:=buildvect(lvterm.rest,n)
  ntable:Vector(List(VTerm)):=new(n,empty())
  for term in lvterm.first repeat
    nexp:=term.expt
    for i in 1..n while (nexp+i)<(n+1) repeat
      ntable.(nexp+i):=append(
                          [cons(term,lvterm) for lvterm in vtable.i],
                             ntable.(nexp+i))
  ntable

buildtable(vu:Vector(P),lvect:List(List(NNI)),leadlist:List(P)):TCoef==
  nfact:NNI:=#leadlist
  table:TCoef:=empty()
  degu:=(#vu-1)::NNI
  prelim:List(VTerm):=[[[e,0$P]$Term for e in lv] for lv in lvect]
  for i in 1..nfact repeat prelim.i.first.pcoef:=leadlist.i
  partialv:Vector(List(VTerm)):=new(nfact,empty())
  partialv:=buildvect(prelim,degu)
  for i in 1..degu repeat
    empty? partialv.i => "next i"
```

```
      table:=cons([vu.i,partialv.i]$TermC, table)
    table

  modify(tablecoef:TCoef,cfter:Detc) : TCoef ==
    cfexp:=cfter.valexp;cfcoef:=cfter.valcoef;cfpos:=cfter.posit
    lterase:List(NNI):=empty()
    for cterm in tablecoef | ^empty?(ctdet:=cterm.detfacts) repeat
      (+/[term.expt for term in ctdet.first])<cfexp => "next term"
      for celt in ctdet repeat
        if celt.cfpos.expt=cfexp then
          celt.cfpos.pcoef:=cfcoef
          if (and/[cc.pcoef ^=0 for cc in celt]) then
            k:=position(celt,ctdet):NNI
            lterase:=cons(k,lterase)
            cterm.coefu:=(cterm.coefu - */[cc.pcoef for cc in celt])
      if not empty? lterase then
        lterase:=sort((n1:NNI,n2:NNI):Boolean +-> n1>n2,lterase)
        for i in lterase repeat ctdet:=delete(ctdet,i)
        cterm.detfacts:=ctdet
        lterase:=empty()
    tablecoef

  listexp(up:BP) :List(NNI) ==
    degree up=0 => [0]
    [degree up,:listexp(reductum up)]

  constructp(lterm:VTerm):USP ==
    +/[monomial(term.pcoef,term.expt) for term in lterm]
```

———-

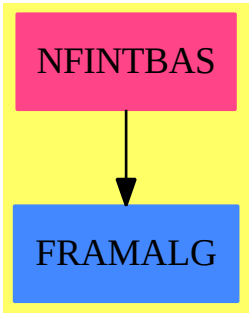
— NPCOEF.dotabb —

```
"NPCOEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NPCOEF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"NPCOEF" -> "PFECAT"
```

———-

15.27 package NFINTBAS NumberFieldIntegralBasis

15.27.1 NumberFieldIntegralBasis (NFINTBAS)



Exports:
discriminant integralBasis localIntegralBasis

— package NFINTBAS NumberFieldIntegralBasis —

```
)abbrev package NFINTBAS NumberFieldIntegralBasis
++ Author: Victor Miller, Clifton Williamson
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Basic Operations: discriminant, integralBasis
++ Related Domains: IntegralBasisTools, TriangularMatrixOperations
++ Also See: FunctionFieldIntegralBasis, WildFunctionFieldIntegralBasis
++ AMS Classifications:
++ Keywords: number field, integral basis, discriminant
++ Examples:
++ References:
++ Description:
++ In this package F is a framed algebra over the integers (typically
++ \spad{F = Z[a]} for some algebraic integer a).  The package provides
++ functions to compute the integral closure of Z in the quotient
++ quotient field of F.

NumberFieldIntegralBasis(UP,F): Exports == Implementation where
  UP : UnivariatePolynomialCategory Integer
  F  : FramedAlgebra(Integer,UP)

  FR  ==> Factored Integer
  I   ==> Integer
  Mat ==> Matrix I
  NNI ==> NonNegativeInteger
  Ans ==> Record(basis: Mat, basisDen: I, basisInv:Mat,discr: I)
```

```
Exports ==> with
  discriminant: () -> Integer
    ++ \spad{discriminant()} returns the discriminant of the integral
    ++ closure of Z in the quotient field of the framed algebra F.
  integralBasis : () -> Record(basis: Mat, basisDen: I, basisInv:Mat)
    ++ \spad{integralBasis()} returns a record
    ++ \spad{[basis,basisDen,basisInv]}
    ++ containing information regarding the integral closure of Z in the
    ++ quotient field of F, where F is a framed algebra with Z-module
    ++ basis \spad{w1,w2,...,wn}.
    ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of \spad{basis} contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
    ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
    ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.
  localIntegralBasis : I -> Record(basis: Mat, basisDen: I, basisInv:Mat)
    ++ \spad{integralBasis(p)} returns a record
    ++ \spad{[basis,basisDen,basisInv]} containing information regarding
    ++ the local integral closure of Z at the prime \spad{p} in the quotient
    ++ field of F, where F is a framed algebra with Z-module basis
    ++ \spad{w1,w2,...,wn}.
    ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of \spad{basis} contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
    ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
    ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.

Implementation ==> add
  import IntegralBasisTools(I, UP, F)
  import ModularHermitianRowReduction(I)
  import TriangularMatrixOperations(I, Vector I, Vector I, Matrix I)

  frobMatrix              : (Mat,Mat,I,NNI) -> Mat
  wildPrimes              : (FR,I) -> List I
  tameProduct             : (FR,I) -> I
  iTameLocalIntegralBasis : (Mat,I,I) -> Ans
  iWildLocalIntegralBasis : (Mat,I,I) -> Ans

  frobMatrix(rb,rbinv,rbden,p) ==
    n := rank()$F; b := basis()$F
    v : Vector F := new(n,0)
    for i in minIndex(v)..maxIndex(v)
```

```
        for ii in minRowIndex(rb)..maxRowIndex(rb) repeat
          a : F := 0
          for j in minIndex(b)..maxIndex(b)
            for jj in minColIndex(rb)..maxColIndex(rb) repeat
              a := a + qelt(rb,ii,jj) * qelt(b,j)
          qsetelt_!(v,i,a**p)
      mat := transpose coordinates v
      ((transpose(rbinv) * mat) exquo (rbden ** p)) :: Mat

    wildPrimes(factoredDisc,n) ==
      -- returns a list of the primes <=n which divide factoredDisc to a
      -- power greater than 1
      ans : List I := empty()
      for f in factors(factoredDisc) repeat
        if f.exponent > 1 and f.factor <= n then ans := concat(f.factor,ans)
      ans

    tameProduct(factoredDisc,n) ==
      -- returns the product of the primes > n which divide factoredDisc
      -- to a power greater than 1
      ans : I := 1
      for f in factors(factoredDisc) repeat
        if f.exponent > 1 and f.factor > n then ans := f.factor * ans
      ans

    integralBasis() ==
      traceMat := traceMatrix()$F; n := rank()$F
      disc := determinant traceMat  -- discriminant of current order
      disc0 := disc                 -- this is disc(F)
      factoredDisc := factor(disc0)$IntegerFactorizationPackage(Integer)
      wilds := wildPrimes(factoredDisc,n)
      sing := tameProduct(factoredDisc,n)
      runningRb := scalarMatrix(n, 1); runningRbinv := scalarMatrix(n, 1)
      -- runningRb    = basis matrix of current order
      -- runningRbinv = inverse basis matrix of current order
      -- these are wrt the original basis for F
      runningRbden : I := 1
      -- runningRbden = denominator for current basis matrix
--      one? sing and empty? wilds => [runningRb, runningRbden, runningRbinv]
      (sing = 1) and empty? wilds => [runningRb, runningRbden, runningRbinv]
      -- id = basis matrix of the ideal (p-radical) wrt current basis
      matrixOut : Mat := scalarMatrix(n,0)
      for p in wilds repeat
        lb := iWildLocalIntegralBasis(matrixOut,disc,p)
        rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
        disc := lb.discr
        -- update 'running integral basis' if newly computed
        -- local integral basis is non-trivial
        if sizeLess?(1,rbden) then
          mat := vertConcat(rbden * runningRb,runningRbden * rb)
```

```
      runningRbden := runningRbden * rbden
      runningRb := squareTop rowEchelon(mat,runningRbden)
      runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
  lb := iTameLocalIntegralBasis(traceMat,disc,sing)
  rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
  disc := lb.discr
  -- update 'running integral basis' if newly computed
  -- local integral basis is non-trivial
  if sizeLess?(1,rbden) then
    mat := vertConcat(rbden * runningRb,runningRbden * rb)
    runningRbden := runningRbden * rbden
    runningRb := squareTop rowEchelon(mat,runningRbden)
    runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
  [runningRb,runningRbden,runningRbinv]

localIntegralBasis p ==
  traceMat := traceMatrix()$F; n := rank()$F
  disc := determinant traceMat  -- discriminant of current order
  (disc exquo (p*p)) case "failed" =>
    [scalarMatrix(n, 1), 1, scalarMatrix(n, 1)]
  lb :=
    p > rank()$F =>
      iTameLocalIntegralBasis(traceMat,disc,p)
    iWildLocalIntegralBasis(scalarMatrix(n,0),disc,p)
  [lb.basis,lb.basisDen,lb.basisInv]

iTameLocalIntegralBasis(traceMat,disc,sing) ==
  n := rank()$F; disc0 := disc
  rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
  -- rb    = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  rbden : I := 1; index : I := 1; oldIndex : I := 1
  -- rbden = denominator for current basis matrix
  -- id = basis matrix of the ideal (p-radical) wrt current basis
  tfm := traceMat
  repeat
    -- compute the p-radical = p-trace-radical
    idinv := transpose squareTop rowEchelon(tfm,sing)
    -- [u1,..,un] are the coordinates of an element of the p-radical
    -- iff [u1,..,un] * idinv is in p * Z^n
    id := rowEchelon LowTriBddDenomInv(idinv, sing)
    -- id = basis matrix of the p-radical
    idinv := UpTriBddDenomInv(id, sing)
    -- id * idinv = sing * identity
    -- no need to check for inseparability in this case
    rbinv := idealiser(id * rb, rbinv * idinv, sing * rbden)
    index := diagonalProduct rbinv
    rb := rowEchelon LowTriBddDenomInv(rbinv, sing * rbden)
    g := matrixGcd(rb,sing,n)
```

```
        if sizeLess?(1,g) then rb := (rb exquo g) :: Mat
        rbden := rbden * (sing quo g)
        rbinv := UpTriBddDenomInv(rb, rbden)
        disc := disc0 quo (index * index)
        indexChange := index quo oldIndex; oldIndex := index
--        one? indexChange => return [rb, rbden, rbinv, disc]
        (indexChange = 1) => return [rb, rbden, rbinv, disc]
        tfm := ((rb * traceMat * transpose rb) exquo (rbden * rbden)) :: Mat

    iWildLocalIntegralBasis(matrixOut,disc,p) ==
      n := rank()$F; disc0 := disc
      rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
      -- rb    = basis matrix of current order
      -- rbinv = inverse basis matrix of current order
      -- these are wrt the original basis for F
      rbden : I := 1; index : I := 1; oldIndex : I := 1
      -- rbden = denominator for current basis matrix
      -- id = basis matrix of the ideal (p-radical) wrt current basis
      p2 := p * p; lp := leastPower(p::NNI,n)
      repeat
        tfm := frobMatrix(rb,rbinv,rbden,p::NNI) ** lp
        -- compute Rp = p-radical
        idinv := transpose squareTop rowEchelon(tfm, p)
        -- [u1,..,un] are the coordinates of an element of Rp
        -- iff [u1,..,un] * idinv is in p * Z^n
        id := rowEchelon LowTriBddDenomInv(idinv,p)
        -- id = basis matrix of the p-radical
        idinv := UpTriBddDenomInv(id,p)
        -- id * idinv = p * identity
        -- no need to check for inseparability in this case
        rbinv := idealiser(id * rb, rbinv * idinv, p * rbden)
        index := diagonalProduct rbinv
        rb := rowEchelon LowTriBddDenomInv(rbinv, p * rbden)
        if divideIfCan_!(rb,matrixOut,p,n) = 1
          then rb := matrixOut
          else rbden := p * rbden
        rbinv := UpTriBddDenomInv(rb, rbden)
        indexChange := index quo oldIndex; oldIndex := index
        disc := disc quo (indexChange * indexChange)
--        one? indexChange or gcd(p2,disc) ^= p2 =>
        (indexChange = 1) or gcd(p2,disc) ^= p2 =>
          return [rb, rbden, rbinv, disc]

    discriminant() ==
      disc := determinant traceMatrix()$F
      intBas := integralBasis()
      rb := intBas.basis; rbden := intBas.basisDen
      index := ((rbden ** rank()$F) exquo (determinant rb)) :: Integer
      (disc exquo (index * index)) :: Integer
```

———-

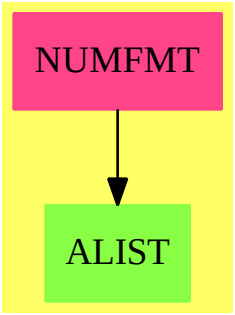
— **NFINTBAS.dotabb** —

```
"NFINTBAS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NFINTBAS"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"NFINTBAS" -> "FRAMALG"
```

———-

15.28 package NUMFMT NumberFormats

15.28.1 NumberFormats (NUMFMT)



Exports:

FormatArabic	FormatRoman	ScanArabic	ScanFloatIgnoreSpaces
ScanFloatIgnoreSpacesIfCan	ScanRoman		

— **package NUMFMT NumberFormats** —

```
)abbrev package NUMFMT NumberFormats
++ SMW March 88
++ Keywords: string manipulation, roman numerals, format
++ Description:
++ NumberFormats provides function to format and read arabic and
++ roman numbers, to convert numbers to strings and to read
++ floating-point numbers.

NumberFormats(): NFexports == NFimplementation where
    PI ==> PositiveInteger
    I  ==> Integer
    C  ==> Character
    F  ==> Float
```

```
    S  ==> String
    V  ==> PrimitiveArray

    NFexports ==> with
        FormatArabic: PI -> S
            ++ FormatArabic(n) forms an Arabic numeral
            ++ string from an integer n.
        ScanArabic:   S -> PI
            ++ ScanArabic(s) forms an integer from an Arabic numeral string s.
        FormatRoman:  PI -> S
            ++ FormatRoman(n) forms a Roman numeral string from an integer n.
        ScanRoman:    S -> PI
            ++ ScanRoman(s) forms an integer from a Roman numeral string s.
        ScanFloatIgnoreSpaces: S -> F
            ++ ScanFloatIgnoreSpaces(s) forms a floating point number from
            ++ the string s ignoring any spaces. Error is generated if the
            ++ string is not recognised as a floating point number.
        ScanFloatIgnoreSpacesIfCan: S -> Union(F, "failed")
            ++ ScanFloatIgnoreSpacesIfCan(s) tries to form a floating point
            ++ number from the string s ignoring any spaces.


    NFimplementation ==> add
        import SExpression
        import Symbol
        replaceD: C -> C
        replaced: C -> C
        contract: S -> S
        check: S ->Boolean
        replaceD c ==
          if c = char "D" then char "E" else c
        replaced c ==
          if c = char "d" then char "E" else c
        contract s ==
          s:= map(replaceD,s)
          s:= map(replaced,s)
          ls:List S := split(s,char " ")$String
          s:= concat ls
        check s ==
          NUMBERP(READ_-FROM_-STRING(s)$Lisp)$Lisp and
           -- if there is an "E" then there must be a "."
           -- this is not caught by code above
           -- also if the exponent is v.big the above returns false
           not (any?((c1:C):Boolean +-> c1=char "E",s)
             and not any?((c2:C):Boolean +-> c2=char ".",s) )

--        Original interpreter function:
--        )lis (defun scanstr(x) (spadcomp::|parseFromString| x))
        sexfloat:SExpression:=convert(coerce("Float")@Symbol)$SExpression
        ScanFloatIgnoreSpaces s ==
```

```
    s := contract s
    not check s => error "Non-numeric value"
    sex := interpret(ncParseFromString(s)$Lisp)$Lisp
    sCheck := car(car(sex))
    if (sCheck=sexfloat) = true then
       f := (cdr cdr sex) pretend Float
    else
       if integer?(cdr sex) = true then
          f := (cdr sex) pretend Integer
          f::F
       else
          error "Non-numeric value"

  ScanFloatIgnoreSpacesIfCan s ==
    s := contract s
    not check s => "failed"
    sex := interpret(ncParseFromString(s)$Lisp)$Lisp
    sCheck := car(car(sex))
    if (sCheck=sexfloat) = true then
       f := (cdr cdr sex) pretend Float
    else
       if integer?(cdr sex) = true then
          f := (cdr sex) pretend Integer
          f::F
       else
          "failed"

  units:V S :=
     construct ["","I","II","III","IV","V","VI","VII","VIII","IX"]
  tens :V S :=
     construct ["","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"]
  hunds:V S :=
     construct ["","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"]
  umin := minIndex units
  tmin := minIndex tens
  hmin := minIndex hunds
  romval:V I := new(256, -1)
  romval ord char(" ")$C := 0
  romval ord char("I")$C := 1
  romval ord char("V")$C := 5
  romval ord char("X")$C := 10
  romval ord char("L")$C := 50
  romval ord char("C")$C := 100
  romval ord char("D")$C := 500
  romval ord char("M")$C := 1000
  thou:C  := char "M"
  plen:C  := char "("
  pren:C  := char ")"
  ichar:C := char "I"
```

```
FormatArabic n == STRINGIMAGE(n)$Lisp
ScanArabic   s == PARSE_-INTEGER(s)$Lisp

FormatRoman pn ==
    n := pn::Integer
    -- Units
    d := (n rem 10) + umin
    n := n quo 10
    s := units.d
    zero? n => s
    -- Tens
    d := (n rem 10) + tmin
    n := n quo 10
    s := concat(tens.d, s)
    zero? n => s
    -- Hundreds
    d := (n rem 10) + hmin
    n := n quo 10
    s := concat(hunds.d, s)
    zero? n => s
    -- Thousands
    d := n rem 10
    n := n quo 10
    s := concat(new(d::NonNegativeInteger, thou), s)
    zero? n => s
    -- Ten thousand and higher
    for i in 2.. while not zero? n repeat
        -- Coefficient of 10**(i+2)
        d := n rem 10
        n := n quo 10
        zero? d => "iterate"
        m0:String := concat(new(i,plen),concat("I",new(i,pren)))
        mm := concat([m0 for j in 1..d]$List(String))
        -- strictly speaking the blank is gratuitous
        if #s > 0 then s := concat(" ", s)
        s  := concat(mm, s)
    s

-- ScanRoman
--
-- The Algorithm:
--    Read number from right to left.  When the current
--    numeral is lower in magnitude than the previous maximum
--    then subtract otherwise add.
--    Shift left and repeat until done.

ScanRoman s ==
    s      := upperCase s
    tot: I := 0
    Max: I := 0
```

```
i:   I := maxIndex s
while i >= minIndex s repeat
    -- Read a single roman digit
    c := s.i; i := i-1
    n := romval ord c
    -- (I)=1000, ((I))=10000, (((I)))=100000, etc
    if n < 0 then
        c ^= pren =>
           error ["Improper character in Roman numeral: ",c]
        nprens: PI := 1
        while c = pren and i >= minIndex s repeat
           c := s.i; i := i-1
           if c = pren then nprens := nprens+1
        c ^= ichar =>
           error "Improper Roman numeral: (x)"
        for k in 1..nprens while i >= minIndex s repeat
           c := s.i; i := i-1
           c ^= plen =>
              error "Improper Roman numeral: unbalanced ')'"
        n := 10**(nprens + 2)
    if n < Max then
        tot := tot - n
    else
        tot := tot + n
        Max := n
tot < 0 => error ["Improper Roman numeral: ", tot]
tot::PI
```

———-

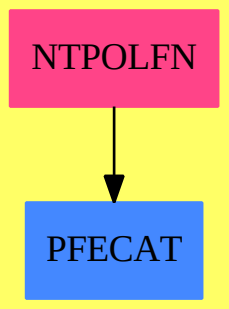
— NUMFMT.dotabb —

```
"NUMFMT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMFMT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NUMFMT" -> "ALIST"
```

———-

15.29 package NTPOLFN NumberTheoreticPolynomialFunctions

15.29.1 NumberTheoreticPolynomialFunctions (NTPOLFN)



Exports:
bernoulliB cyclotomic eulerE

— package NTPOLFN NumberTheoreticPolynomialFunctions —

```
)abbrev package NTPOLFN NumberTheoreticPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created:  1990
++ Date Last Updated: June 25, 1991
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides polynomials as functions on a ring.

NumberTheoreticPolynomialFunctions(R: CommutativeRing): Exports == Impl where
    NNI ==> NonNegativeInteger
    RN  ==> Fraction Integer

    Exports ==> with

        cyclotomic: (NNI, R) -> R
         ++ cyclotomic(n,r) \undocumented

        if R has Algebra RN then
            bernoulliB: (NNI, R) -> R
```

```
            ++ bernoulliB(n,r) \undocumented
           eulerE:     (NNI, R) -> R
            ++ eulerE(n,r) \undocumented

    Impl ==> add

        import PolynomialNumberTheoryFunctions()

        I   ==> Integer
        SUP ==> SparseUnivariatePolynomial

        -- This is the wrong way to evaluate the polynomial.
        cyclotomic(k, x) ==
            p: SUP(I) := cyclotomic(k)
            r: R      := 0
            while p ^= 0 repeat
                d := degree p
                c := leadingCoefficient p
                p := reductum p
                r := c*x**d + r
            r

        if R has Algebra RN then
            eulerE(k, x) ==
                p: SUP(RN) := euler(k)
                r: R       := 0
                while p ^= 0 repeat
                    d := degree p
                    c := leadingCoefficient p
                    p := reductum p
                    r := c*x**d + r
                r
            bernoulliB(k, x) ==
                p: SUP(RN) := bernoulli(k)
                r: R       := 0
                while p ^= 0 repeat
                    d := degree p
                    c := leadingCoefficient p
                    p := reductum p
                    r := c*x**d + r
                r


            ———-
```

— NTPOLFN.dotabb —

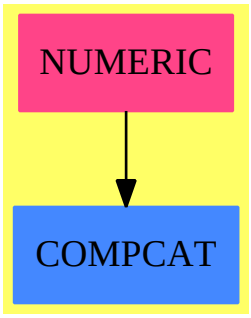
```
"NTPOLFN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NTPOLFN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
```

```
"NTPOLFN" -> "PFECAT"
```

———-

15.30 package NUMERIC Numeric

15.30.1 Numeric (NUMERIC)



Exports:
numeric complexNumeric complexNumericIfCan numeric numericIfCan

— package NUMERIC Numeric —

```
)abbrev package NUMERIC Numeric
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 17 August 1995, Mike Dewar
++                    24 January 1997, Miked Dewar (added partial operators)
++ Basic Operations: numeric, complexNumeric, numericIfCan, complexNumericIfCan
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Numeric provides real and complex numerical evaluation
++ functions for various symbolic types.

Numeric(S:ConvertibleTo Float): with
  numeric: S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (S, PositiveInteger) -> Float
    ++ numeric(x, n) returns a real approximation of x up to n decimal
    ++ places.
```

```
complexNumeric: S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
if S has CommutativeRing then
  complexNumeric: Complex S -> Complex Float
    ++ complexNumeric(x) returns a complex approximation of x.
  complexNumeric: (Complex S, PositiveInteger) -> Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x up
    ++ to n decimal places.
  complexNumeric: Polynomial Complex S -> Complex Float
    ++ complexNumeric(x) returns a complex approximation of x.
  complexNumeric: (Polynomial Complex S, PositiveInteger) -> Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x up
    ++ to n decimal places.
if S has Ring then
  numeric: Polynomial S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (Polynomial S, PositiveInteger) -> Float
    ++ numeric(x,n) returns a real approximation of x up to n decimal
    ++ places.
  complexNumeric: Polynomial S -> Complex Float
    ++ complexNumeric(x) returns a complex approximation of x.
  complexNumeric: (Polynomial S, PositiveInteger) -> Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x
    ++ up to n decimal places.
if S has IntegralDomain then
  numeric: Fraction Polynomial S -> Float
    ++ numeric(x) returns a real approximation of x.
  numeric: (Fraction Polynomial S, PositiveInteger) -> Float
    ++ numeric(x,n) returns a real approximation of x up to n decimal
    ++ places.
  complexNumeric: Fraction Polynomial S -> Complex Float
    ++ complexNumeric(x) returns a complex approximation of x.
  complexNumeric: (Fraction Polynomial S, PositiveInteger) -> Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x
  complexNumeric: Fraction Polynomial Complex S -> Complex Float
    ++ complexNumeric(x) returns a complex approximation of x.
  complexNumeric: (Fraction Polynomial Complex S, PositiveInteger) ->
                                                          Complex Float
    ++ complexNumeric(x, n) returns a complex approximation of x
    ++ up to n decimal places.
  if S has OrderedSet then
    numeric: Expression S -> Float
      ++ numeric(x) returns a real approximation of x.
    numeric: (Expression S, PositiveInteger) -> Float
      ++ numeric(x, n) returns a real approximation of x up to n
      ++ decimal places.
    complexNumeric: Expression S -> Complex Float
```

```
      ++ complexNumeric(x) returns a complex approximation of x.
    complexNumeric: (Expression S, PositiveInteger) -> Complex Float
      ++ complexNumeric(x, n) returns a complex approximation of x
      ++ up to n decimal places.
    complexNumeric: Expression Complex S -> Complex Float
      ++ complexNumeric(x) returns a complex approximation of x.
    complexNumeric: (Expression Complex S, PositiveInteger) -> Complex Float
      ++ complexNumeric(x, n) returns a complex approximation of x
      ++ up to n decimal places.
  if S has CommutativeRing then
    complexNumericIfCan: Polynomial Complex S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not constant.
    complexNumericIfCan: (Polynomial Complex S, PositiveInteger) -> Union(Complex Float,"failed"
      ++ complexNumericIfCan(x, n) returns a complex approximation of x up
      ++ to n decimal places, or "failed" if \axiom{x} is not a constant.
  if S has Ring then
    numericIfCan: Polynomial S -> Union(Float,"failed")
      ++ numericIfCan(x) returns a real approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    numericIfCan: (Polynomial S, PositiveInteger) -> Union(Float,"failed")
      ++ numericIfCan(x,n) returns a real approximation of x up to n decimal
      ++ places, or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: Polynomial S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: (Polynomial S, PositiveInteger) -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x, n) returns a complex approximation of x
      ++ up to n decimal places, or "failed" if \axiom{x} is not a constant.
  if S has IntegralDomain then
    numericIfCan: Fraction Polynomial S -> Union(Float,"failed")
      ++ numericIfCan(x) returns a real approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    numericIfCan: (Fraction Polynomial S, PositiveInteger) -> Union(Float,"failed")
      ++ numericIfCan(x,n) returns a real approximation of x up to n decimal
      ++ places, or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: Fraction Polynomial S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: (Fraction Polynomial S, PositiveInteger) -> Union(Complex Float,"failed
      ++ complexNumericIfCan(x, n) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: Fraction Polynomial Complex S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: (Fraction Polynomial Complex S, PositiveInteger) ->
                                              Union(Complex Float,"failed")
      ++ complexNumericIfCan(x, n) returns a complex approximation of x
      ++ up to n decimal places, or "failed" if \axiom{x} is not a constant.
    if S has OrderedSet then
```

```
    numericIfCan: Expression S -> Union(Float,"failed")
      ++ numericIfCan(x) returns a real approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    numericIfCan: (Expression S, PositiveInteger) -> Union(Float,"failed")
      ++ numericIfCan(x, n) returns a real approximation of x up to n
      ++ decimal places, or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: Expression S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: (Expression S, PositiveInteger) ->
                                            Union(Complex Float,"failed")
      ++ complexNumericIfCan(x, n) returns a complex approximation of x
      ++ up to n decimal places, or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: Expression Complex S -> Union(Complex Float,"failed")
      ++ complexNumericIfCan(x) returns a complex approximation of x,
      ++ or "failed" if \axiom{x} is not a constant.
    complexNumericIfCan: (Expression Complex S, PositiveInteger) ->
                                             Union(Complex Float,"failed")
      ++ complexNumericIfCan(x, n) returns a complex approximation of x
      ++ up to n decimal places, or "failed" if \axiom{x} is not a constant.
== add

 if S has CommutativeRing then
   complexNumericIfCan(p:Polynomial Complex S) ==
     p' : Union(Complex(S),"failed") := retractIfCan p
     p' case "failed" => "failed"
     complexNumeric(p')

   complexNumericIfCan(p:Polynomial Complex S,n:PositiveInteger) ==
     p' : Union(Complex(S),"failed") := retractIfCan p
     p' case "failed" => "failed"
     complexNumeric(p',n)

 if S has Ring then
   numericIfCan(p:Polynomial S) ==
     p' : Union(S,"failed") := retractIfCan p
     p' case "failed" => "failed"
     numeric(p')

   complexNumericIfCan(p:Polynomial S) ==
     p' : Union(S,"failed") := retractIfCan p
     p' case "failed" => "failed"
     complexNumeric(p')

   complexNumericIfCan(p:Polynomial S, n:PositiveInteger) ==
     p' : Union(S,"failed") := retractIfCan p
     p' case "failed" => "failed"
     complexNumeric(p', n)

   numericIfCan(p:Polynomial S, n:PositiveInteger) ==
```

```
      old := digits(n)$Float
      ans := numericIfCan p
      digits(old)$Float
      ans

  if S has IntegralDomain then
    numericIfCan(f:Fraction Polynomial S)==
      num := numericIfCan(numer(f))
      num case "failed" => "failed"
      den := numericIfCan(denom f)
      den case "failed" => "failed"
      num/den

    complexNumericIfCan(f:Fraction Polynomial S) ==
      num := complexNumericIfCan(numer f)
      num case "failed" => "failed"
      den := complexNumericIfCan(denom f)
      den case "failed" => "failed"
      num/den

    complexNumericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
      num := complexNumericIfCan(numer f, n)
      num case "failed" => "failed"
      den := complexNumericIfCan(denom f, n)
      den case "failed" => "failed"
      num/den

    numericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
      old := digits(n)$Float
      ans := numericIfCan f
      digits(old)$Float
      ans

    complexNumericIfCan(f:Fraction Polynomial Complex S) ==
      num := complexNumericIfCan(numer f)
      num case "failed" => "failed"
      den := complexNumericIfCan(denom f)
      den case "failed" => "failed"
      num/den

    complexNumericIfCan(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
      num := complexNumericIfCan(numer f, n)
      num case "failed" => "failed"
      den := complexNumericIfCan(denom f, n)
      den case "failed" => "failed"
      num/den

    if S has OrderedSet then
      numericIfCan(x:Expression S) ==
        retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
```

```
    --s2cs(u:S):Complex(S) == complex(u,0)

    complexNumericIfCan(x:Expression S) ==
       complexNumericIfCan map(coerce, x)$ExpressionFunctions2(S,Complex S)

    numericIfCan(x:Expression S, n:PositiveInteger) ==
      old := digits(n)$Float
      x' : Expression Float := map(convert, x)$ExpressionFunctions2(S, Float)
      ans : Union(Float,"failed") := retractIfCan x'
      digits(old)$Float
      ans

    complexNumericIfCan(x:Expression S, n:PositiveInteger) ==
      old := digits(n)$Float
      x' : Expression Complex S := map(coerce, x)$ExpressionFunctions2(S, Complex S)
      ans : Union(Complex Float,"failed") := complexNumericIfCan(x')
      digits(old)$Float
      ans

    if S has RealConstant then
      complexNumericIfCan(x:Expression Complex S) ==
        retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

      complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
        old := digits(n)$Float
        x' : Expression Complex Float :=
         map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
        ans : Union(Complex Float,"failed") := retractIfCan x'
        digits(old)$Float
        ans
    else
      convert(x:Complex S):Complex(Float)==map(convert,x)$ComplexFunctions2(S,Float)

      complexNumericIfCan(x:Expression Complex S) ==
        retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

      complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
        old := digits(n)$Float
        x' : Expression Complex Float :=
         map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
        ans : Union(Complex Float,"failed") := retractIfCan x'
        digits(old)$Float
        ans
  numeric(s:S) == convert(s)@Float

  if S has ConvertibleTo Complex Float then
    complexNumeric(s:S) == convert(s)@Complex(Float)

    complexNumeric(s:S, n:PositiveInteger) ==
```

```
      old := digits(n)$Float
      ans := complexNumeric s
      digits(old)$Float
      ans

  else
    complexNumeric(s:S) == convert(s)@Float :: Complex(Float)

    complexNumeric(s:S,n:PositiveInteger) ==
      numeric(s, n)::Complex(Float)

  if S has CommutativeRing then
    complexNumeric(p:Polynomial Complex S) ==
      p' : Union(Complex(S),"failed") := retractIfCan p
      p' case "failed" =>
        error "Cannot compute the numerical value of a non-constant polynomial"
      complexNumeric(p')

    complexNumeric(p:Polynomial Complex S,n:PositiveInteger) ==
      p' : Union(Complex(S),"failed") := retractIfCan p
      p' case "failed" =>
        error "Cannot compute the numerical value of a non-constant polynomial"
      complexNumeric(p',n)

    if S has RealConstant then
      complexNumeric(s:Complex S) == convert(s)$Complex(S)

      complexNumeric(s:Complex S, n:PositiveInteger) ==
        old := digits(n)$Float
        ans := complexNumeric s
        digits(old)$Float
        ans

    else if Complex(S) has ConvertibleTo(Complex Float) then
      complexNumeric(s:Complex S) == convert(s)@Complex(Float)

      complexNumeric(s:Complex S, n:PositiveInteger) ==
        old := digits(n)$Float
        ans := complexNumeric s
        digits(old)$Float
        ans

    else
      complexNumeric(s:Complex S) ==
        s' : Union(S,"failed") := retractIfCan s
        s' case "failed" =>
          error "Cannot compute the numerical value of a non-constant object"
        complexNumeric(s')

      complexNumeric(s:Complex S, n:PositiveInteger) ==
```

```
      s' : Union(S,"failed") := retractIfCan s
      s' case "failed" =>
        error "Cannot compute the numerical value of a non-constant object"
      old := digits(n)$Float
      ans := complexNumeric s'
      digits(old)$Float
      ans

  numeric(s:S, n:PositiveInteger) ==
    old := digits(n)$Float
    ans := numeric s
    digits(old)$Float
    ans

  if S has Ring then
    numeric(p:Polynomial S) ==
      p' : Union(S,"failed") := retractIfCan p
      p' case "failed" => error
       "Can only compute the numerical value of a constant, real-valued polynomial"
      numeric(p')

    complexNumeric(p:Polynomial S) ==
      p' : Union(S,"failed") := retractIfCan p
      p' case "failed" =>
        error "Cannot compute the numerical value of a non-constant polynomial"
      complexNumeric(p')

    complexNumeric(p:Polynomial S, n:PositiveInteger) ==
      p' : Union(S,"failed") := retractIfCan p
      p' case "failed" =>
        error "Cannot compute the numerical value of a non-constant polynomial"
      complexNumeric(p', n)

    numeric(p:Polynomial S, n:PositiveInteger) ==
      old := digits(n)$Float
      ans := numeric p
      digits(old)$Float
      ans

  if S has IntegralDomain then
    numeric(f:Fraction Polynomial S)==
        numeric(numer(f)) / numeric(denom f)

    complexNumeric(f:Fraction Polynomial S) ==
      complexNumeric(numer f)/complexNumeric(denom f)

    complexNumeric(f:Fraction Polynomial S, n:PositiveInteger) ==
      complexNumeric(numer f, n)/complexNumeric(denom f, n)

    numeric(f:Fraction Polynomial S, n:PositiveInteger) ==
```

```
    old := digits(n)$Float
    ans := numeric f
    digits(old)$Float
    ans

  complexNumeric(f:Fraction Polynomial Complex S) ==
    complexNumeric(numer f)/complexNumeric(denom f)

  complexNumeric(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
    complexNumeric(numer f, n)/complexNumeric(denom f, n)

  if S has OrderedSet then
    numeric(x:Expression S) ==
      x' : Union(Float,"failed") :=
       retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
      x' case "failed" => error
       "Can only compute the numerical value of a constant, real-valued Expression"
      x'

    complexNumeric(x:Expression S) ==
      x' : Union(Complex Float,"failed") := retractIfCan(
       map(complexNumeric, x)$ExpressionFunctions2(S,Complex Float))
      x' case "failed" =>
       error "Cannot compute the numerical value of a non-constant expression"
      x'

    numeric(x:Expression S, n:PositiveInteger) ==
      old := digits(n)$Float
      x' : Expression Float := map(convert, x)$ExpressionFunctions2(S, Float)
      ans : Union(Float,"failed") := retractIfCan x'
      digits(old)$Float
      ans case "failed" => error
       "Can only compute the numerical value of a constant, real-valued Expression"
      ans

    complexNumeric(x:Expression S, n:PositiveInteger) ==
      old := digits(n)$Float
      x' : Expression Complex Float :=
       map(complexNumeric, x)$ExpressionFunctions2(S,Complex Float)
      ans : Union(Complex Float,"failed") := retractIfCan x'
      digits(old)$Float
      ans case "failed" =>
       error "Cannot compute the numerical value of a non-constant expression"
      ans

    complexNumeric(x:Expression Complex S) ==
      x' : Union(Complex Float,"failed") := retractIfCan(
       map(complexNumeric, x)$ExpressionFunctions2(Complex S,Complex Float))
      x' case "failed" =>
       error "Cannot compute the numerical value of a non-constant expression"
```

```
    x'

  complexNumeric(x:Expression Complex S, n:PositiveInteger) ==
    old := digits(n)$Float
    x' : Expression Complex Float :=
     map(complexNumeric, x)$ExpressionFunctions2(Complex S,Complex Float)
    ans : Union(Complex Float,"failed") := retractIfCan x'
    digits(old)$Float
    ans case "failed" =>
     error "Cannot compute the numerical value of a non-constant expression"
    ans
```

———-

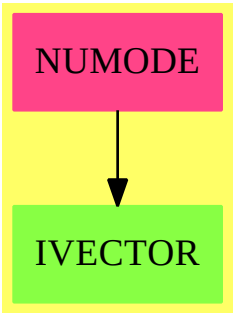
— **NUMERIC.dotabb** —

```
"NUMERIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMERIC"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NUMERIC" -> "COMPCAT"
```

———-

15.31 package NUMODE NumericalOrdinaryDifferentialEquations

15.31.1 NumericalOrdinaryDifferentialEquations (NUMODE)



Exports:
rk4 rk4a rk4f rk4qc rk4qc

— **package NUMODE NumericalOrdinaryDifferentialEquations** —

```
)abbrev package NUMODE NumericalOrdinaryDifferentialEquations
++ Author: Yurij Baransky
++ Date Created: October 90
++ Date Last Updated: October 90
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package is a suite of functions for the numerical integration of an
++ ordinary differential equation of n variables:\br
++ \tab{5}dy/dx = f(y,x)\tab{5}y is an n-vector\br
++ All the routines are based on a 4-th order Runge-Kutta kernel.
++ These routines generally have as arguments:\br
++ n, the number of dependent variables;\br
++ x1, the initial point;\br
++ h, the step size;\br
++ y, a vector of initial conditions of length n\br
++ which upon exit contains the solution at \spad{x1 + h};\br
++
++ \spad{derivs}, a function which computes the right hand side of the
++ ordinary differential equation: \spad{derivs(dydx,y,x)} computes
++ \spad{dydx}, a vector which contains the derivative information.
++
++ In order of increasing complexity:\br
++ \tab{5}\spad{rk4(y,n,x1,h,derivs)} advances the solution vector to\br
++ \tab{5}\spad{x1 + h} and return the values in y.\br
++
++ \tab{5}\spad{rk4(y,n,x1,h,derivs,t1,t2,t3,t4)} is the same as\br
++ \tab{5}\spad{rk4(y,n,x1,h,derivs)} except that you must provide 4 scratch\br
++ \tab{5}arrays t1-t4 of size n.\br
++
++ \tab{5}Starting with y at x1, \spad{rk4f(y,n,x1,x2,ns,derivs)}\br
++ \tab{5}uses \spad{ns} fixed steps of a 4-th order Runge-Kutta\br
++ \tab{5}integrator to advance the solution vector to x2 and return\br
++ \tab{5}the values in y.  Argument x2, is the final point, and\br
++ \tab{5}\spad{ns}, the number of steps to take.
++
++ \spad{rk4qc(y,n,x1,step,eps,yscal,derivs)} takes a 5-th order
++ Runge-Kutta step with monitoring of local truncation to ensure
++ accuracy and adjust stepsize.
++ The function takes two half steps and one full step and scales
++ the difference in solutions at the final point. If the error is
++ within \spad{eps}, the step is taken and the result is returned.
++ If the error is not within \spad{eps}, the stepsize if decreased
++ and the procedure is tried again until the desired accuracy is
++ reached. Upon input, an trial step size must be given and upon
++ return, an estimate of the next step size to use is returned as
```

```
++ well as the step size which produced the desired accuracy.
++ The scaled error is computed as\br
++ \tab{5}\spad{error = MAX(ABS((y2steps(i) - y1step(i))/yscal(i)))}\br
++ and this is compared against \spad{eps}. If this is greater
++ than \spad{eps}, the step size is reduced accordingly to\br
++ \tab{5}\spad{hnew = 0.9 * hdid * (error/eps)**(-1/4)}\br
++ If the error criterion is satisfied, then we check if the
++ step size was too fine and return a more efficient one. If
++ \spad{error > \spad{eps} * (6.0E-04)} then the next step size should be\br
++ \tab{5}\spad{hnext = 0.9 * hdid * (error/\spad{eps})**(-1/5)}\br
++ Otherwise \spad{hnext = 4.0 * hdid} is returned.
++ A more detailed discussion of this and related topics can be
++ found in the book "Numerical Recipies" by W.Press, B.P. Flannery,
++ S.A. Teukolsky, W.T. Vetterling published by Cambridge University Press.
++
++ Argument \spad{step} is a record of 3 floating point
++ numbers \spad{(try , did , next)},
++ \spad{eps} is the required accuracy,
++ \spad{yscal} is the scaling vector for the difference in solutions.
++ On input, \spad{step.try} should be the guess at a step
++ size to achieve the accuracy.
++ On output, \spad{step.did} contains the step size which achieved the
++ accuracy and \spad{step.next} is the next step size to use.
++
++ \spad{rk4qc(y,n,x1,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,t7)} is the
++ same as \spad{rk4qc(y,n,x1,step,eps,yscal,derivs)} except that the user
++ must provide the 7 scratch arrays \spad{t1-t7} of size n.
++
++ \spad{rk4a(y,n,x1,x2,eps,h,ns,derivs)}
++ is a driver program which uses \spad{rk4qc} to integrate n ordinary
++ differential equations starting at x1 to x2, keeping the local
++ truncation error to within \spad{eps} by changing the local step size.
++ The scaling vector is defined as\br
++ \tab{5}\spad{yscal(i) = abs(y(i)) + abs(h*dydx(i)) + tiny}\br
++ where \spad{y(i)} is the solution at location x, \spad{dydx} is the
++ ordinary differential equation's right hand side, h is the current
++ step size and \spad{tiny} is 10 times the
++ smallest positive number representable.
++
++ The user must supply an estimate for a trial step size and
++ the maximum number of calls to \spad{rk4qc} to use.
++ Argument x2 is the final point,
++ \spad{eps} is local truncation,
++ \spad{ns} is the maximum number of call to \spad{rk4qc} to use.

NumericalOrdinaryDifferentialEquations(): Exports == Implementation where
  L     ==> List
  V     ==> Vector
  B     ==> Boolean
  I     ==> Integer
```

```
  E     ==> OutputForm
  NF    ==> Float
  NNI   ==> NonNegativeInteger
  VOID  ==> Void
  OFORM ==> OutputForm
  RK4STEP ==> Record(try:NF, did:NF, next:NF)

  Exports ==> with
--header definitions here
   rk4   : (V NF,I,NF,NF,  (V NF,V NF,NF) -> VOID) -> VOID
    ++ rk4(y,n,x1,h,derivs) uses a 4-th order Runge-Kutta method
    ++ to numerically integrate the ordinary differential equation
    ++ dy/dx = f(y,x) of n variables, where y is an n-vector.
    ++ Argument y is a vector of initial conditions of length n which upon exit
    ++ contains the solution at \spad{x1 + h}, n is the number of dependent
    ++ variables, x1 is the initial point, h is the step size, and
    ++ \spad{derivs} is a function which computes the right hand side of the
    ++ ordinary differential equation.
    ++ For details, see \spadtype{NumericalOrdinaryDifferentialEquations}.
   rk4   : (V NF,I,NF,NF,  (V NF,V NF,NF) -> VOID
           ,V NF,V NF,V NF,V NF) -> VOID
    ++ rk4(y,n,x1,h,derivs,t1,t2,t3,t4) is the same as
    ++ \spad{rk4(y,n,x1,h,derivs)} except that you must provide 4 scratch
    ++ arrays t1-t4 of size n.
    ++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
   rk4a  : (V NF,I,NF,NF,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
    ++ rk4a(y,n,x1,x2,eps,h,ns,derivs) is a driver function for the
    ++ numerical integration of an ordinary differential equation
    ++ dy/dx = f(y,x) of n variables, where y is an n-vector
    ++ using a 4-th order Runge-Kutta method.
    ++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
   rk4qc : (V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID) -> VOID
    ++ rk4qc(y,n,x1,step,eps,yscal,derivs) is a subfunction for the
    ++ numerical integration of an ordinary differential equation
    ++ dy/dx = f(y,x) of n variables, where y is an n-vector
    ++ using a 4-th order Runge-Kutta method.
    ++ This function takes a 5-th order Runge-Kutta step with monitoring
    ++ of local truncation to ensure accuracy and adjust stepsize.
    ++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
   rk4qc : (V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID
           ,V NF,V NF,V NF,V NF,V NF,V NF,V NF) -> VOID
    ++ rk4qc(y,n,x1,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,t7) is a
    ++ subfunction for the numerical integration of an ordinary differential
    ++ equation dy/dx = f(y,x) of n variables, where y is an n-vector
    ++ using a 4-th order Runge-Kutta method.
    ++ This function takes a 5-th order Runge-Kutta step with monitoring
    ++ of local truncation to ensure accuracy and adjust stepsize.
    ++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
   rk4f  : (V NF,I,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
    ++ rk4f(y,n,x1,x2,ns,derivs) uses a 4-th order Runge-Kutta method
```

```
    ++ to numerically integrate the ordinary differential equation
    ++ dy/dx = f(y,x) of n variables, where y is an n-vector.
    ++ Starting with y at x1, this function uses \spad{ns} fixed
    ++ steps of a 4-th order Runge-Kutta integrator to advance the
    ++ solution vector to x2 and return the values in y.
    ++ For details, see \con{NumericalOrdinaryDifferentialEquations}.

  Implementation ==>  add
  --some local function definitions here
   rk4qclocal : (V NF,V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID
                 ,V NF,V NF,V NF,V NF,V NF,V NF) -> VOID
   rk4local   : (V NF,V NF,I,NF,NF,V NF,(V NF,V NF,NF) -> VOID
                 ,V NF,V NF,V NF) -> VOID
   import OutputPackage

------------------------------------------------------------

   rk4a(ystart,nvar,x1,x2,eps,htry,nstep,derivs) ==
      y       : V NF := new(nvar::NNI,0.0)
      yscal   : V NF := new(nvar::NNI,1.0)
      dydx    : V NF := new(nvar::NNI,0.0)
      t1      : V NF := new(nvar::NNI,0.0)
      t2      : V NF := new(nvar::NNI,0.0)
      t3      : V NF := new(nvar::NNI,0.0)
      t4      : V NF := new(nvar::NNI,0.0)
      t5      : V NF := new(nvar::NNI,0.0)
      t6      : V NF := new(nvar::NNI,0.0)
      step    : RK4STEP := [htry,0.0,0.0]
      x       : NF   := x1
      tiny    : NF   := 10.0**(-(digits()+1)::I)
      m       : I    := nvar
      outlist : L OFORM := [x::E,x::E,x::E]
      i       : I
      iter    : I

      eps  := 1.0/eps
      for i in 1..m repeat
         y(i)  := ystart(i)
      for iter in 1..nstep repeat
--compute the derivative
         derivs(dydx,y,x)
--if overshoot, the set h accordingly
         if (x + step.try - x2) > 0.0 then
            step.try := x2 - x
--find the correct scaling
         for i in 1..m repeat
            yscal(i) := abs(y(i)) + abs(step.try * dydx(i)) + tiny
--take a quality controlled runge-kutta step
         rk4qclocal(y,dydx,nvar,x,step,eps,yscal,derivs
                    ,t1,t2,t3,t4,t5,t6)
```

```
         x         := x + step.did
--       outlist.0 := x::E
--       outlist.1 := y(0)::E
--       outlist.2 := y(1)::E
--       output(blankSeparate(outlist)::E)
--check to see if done
         if (x-x2) >= 0.0 then
            leave
--next stepsize to use
         step.try := step.next
--end nstep repeat
      if iter = (nstep+1) then
         output("ode: ERROR ")
         outlist.1 := nstep::E
         outlist.2 := " steps to small, last h = "::E
         outlist.3 := step.did::E
         output(blankSeparate(outlist))
         output(" y= ",y::E)
      for i in 1..m repeat
         ystart(i) := y(i)

------------------------------------------------------------------

   rk4qc(y,n,x,step,eps,yscal,derivs) ==
      t1 : V NF := new(n::NNI,0.0)
      t2 : V NF := new(n::NNI,0.0)
      t3 : V NF := new(n::NNI,0.0)
      t4 : V NF := new(n::NNI,0.0)
      t5 : V NF := new(n::NNI,0.0)
      t6 : V NF := new(n::NNI,0.0)
      t7 : V NF := new(n::NNI,0.0)
      derivs(t7,y,x)
      eps := 1.0/eps
      rk4qclocal(y,t7,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)

------------------------------------------------------------

   rk4qc(y,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,dydx) ==
      derivs(dydx,y,x)
      eps := 1.0/eps
      rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)

------------------------------------------------------------

   rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs
             ,t1,t2,t3,ysav,dysav,ytemp) ==
      xsav   : NF := x
      h      : NF := step.try
      fcor   : NF := 1.0/15.0
      safety : NF := 0.9
```

```
      grow   : NF := -0.20
      shrink : NF := -0.25
      errcon : NF := 0.6E-04  --(this is 4/safety)**(1/grow)
      hh     : NF
      errmax : NF
      i      : I
      m      : I  := n
--
      for i in 1..m repeat
         dysav(i) := dydx(i)
         ysav(i)  := y(i)
--cut down step size till error criterion is met
      repeat
--take two little steps to get to x + h
         hh := 0.5 * h
         rk4local(ysav,dysav,n,xsav,hh,ytemp,derivs,t1,t2,t3)
         x  := xsav + hh
         derivs(dydx,ytemp,x)
         rk4local(ytemp,dydx,n,x,hh,y,derivs,t1,t2,t3)
         x  := xsav + h
--take one big step get to x + h
         rk4local(ysav,dysav,n,xsav,h,ytemp,derivs,t1,t2,t3)

--compute the maximum scaled difference
         errmax := 0.0
         for i in 1..m repeat
            ytemp(i) := y(i) - ytemp(i)
            errmax   := max(errmax,abs(ytemp(i)/yscal(i)))
--scale relative to required accuracy
         errmax := errmax * eps
--update integration stepsize
         if (errmax > 1.0) then
            h := safety * h * (errmax ** shrink)
         else
            step.did := h
            if errmax > errcon then
               step.next := safety * h * (errmax ** grow)
            else
               step.next := 4 * h
            leave
--make fifth order with 4-th order error estimate
      for i in 1..m repeat
         y(i) := y(i) + ytemp(i) * fcor

-----------------------------------------

   rk4f(y,nvar,x1,x2,nstep,derivs) ==
     yt   : V NF := new(nvar::NNI,0.0)
     dyt  : V NF := new(nvar::NNI,0.0)
     dym  : V NF := new(nvar::NNI,0.0)
```

```
     dydx : V NF := new(nvar::NNI,0.0)
     ynew : V NF := new(nvar::NNI,0.0)
     h    : NF := (x2-x1) / (nstep::NF)
     x    : NF := x1
     i    : I
     j    : I
-- start integrating
     for i in 1..nstep repeat
        derivs(dydx,y,x)
        rk4local(y,dydx,nvar,x,h,y,derivs,yt,dyt,dym)
        x := x + h

------------------------------------------------------

   rk4(y,n,x,h,derivs) ==
      t1 : V NF := new(n::NNI,0.0)
      t2 : V NF := new(n::NNI,0.0)
      t3 : V NF := new(n::NNI,0.0)
      t4 : V NF := new(n::NNI,0.0)
      derivs(t1,y,x)
      rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)

------------------------------------------------------------

   rk4(y,n,x,h,derivs,t1,t2,t3,t4) ==
      derivs(t1,y,x)
      rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)

------------------------------------------------------------

   rk4local(y,dydx,n,x,h,yout,derivs,yt,dyt,dym) ==
      hh : NF := h*0.5
      h6 : NF := h/6.0
      xh : NF := x+hh
      m  : I  := n
      i  : I
-- first step
      for i in 1..m repeat
         yt(i) := y(i) + hh*dydx(i)
-- second step
      derivs(dyt,yt,xh)
      for i in 1..m repeat
         yt(i) := y(i) + hh*dyt(i)
-- third step
      derivs(dym,yt,xh)
      for i in 1..m repeat
         yt(i)  := y(i)   + h*dym(i)
         dym(i) := dyt(i) + dym(i)
-- fourth step
      derivs(dyt,yt,x+h)
```

```
      for i in 1..m repeat
         yout(i) := y(i) + h6*( dydx(i) + 2.0*dym(i) + dyt(i) )
```

———-

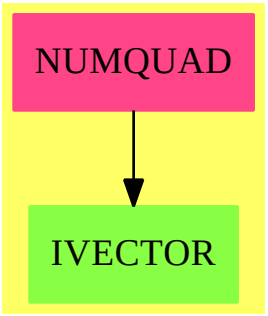
— **NUMODE.dotabb** —

```
"NUMODE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMODE"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"NUMODE" -> "IVECTOR"
```

———-

15.32 package NUMQUAD NumericalQuadrature

15.32.1 NumericalQuadrature (NUMQUAD)



Exports:

aromberg	asimpson	atrapezoidal	romberg	rombergo
simpson	simpsono	trapezoidal	trapezoidalo	

— **package NUMQUAD NumericalQuadrature** —

```
)abbrev package NUMQUAD NumericalQuadrature
++ Author: Yurij A. Baransky
++ Date Created: October 90
++ Date Last Updated: October 90
++ Basic Operations:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ This suite of routines performs numerical quadrature using
++ algorithms derived from the basic trapezoidal rule. Because
++ the error term of this rule contains only even powers of the
++ step size (for open and closed versions), fast convergence
++ can be obtained if the integrand is sufficiently smooth.
++
++ Each routine returns a Record of type TrapAns, which contains
++ value Float: estimate of the integral
++ error Float: estimate of the error in the computation
++ totalpts Integer: total number of function evaluations
++ success Boolean: if the integral was computed within the user
++ specified error criterion
++ To produce this estimate, each routine generates an internal
++ sequence of sub-estimates, denoted by S(i), depending on the
++ routine, to which the various convergence criteria are applied.
++ The user must supply a relative accuracy, \spad{eps_r}, and an absolute
++ accuracy, \spad{eps_a}. Convergence is obtained when either\br
++ \tab{5}\spad{ABS(S(i) - S(i-1)) < eps_r * ABS(S(i-1))}\br
++ \tab{5}or \spad{ABS(S(i) - S(i-1)) < eps_a}
++ are true statements.
++
++ The routines come in three families and three flavors:
++ closed: romberg, simpson, trapezoidal
++ open: rombergo, simpsono, trapezoidalo
++ adaptive closed: aromberg, asimpson, atrapezoidal
++
++ The S(i) for the trapezoidal family is the value of the
++ integral using an equally spaced absicca trapezoidal rule for
++ that level of refinement.
++
++ The S(i) for the simpson family is the value of the integral
++ using an equally spaced absicca simpson rule for that level of
++ refinement.
++
++ The S(i) for the romberg family is the estimate of the integral
++ using an equally spaced absicca romberg method. For
++ the i-th level, this is an appropriate combination of all the
++ previous trapezodial estimates so that the error term starts
++ with the 2*(i+1) power only.
++
++ The three families come in a closed version, where the formulas
++ include the endpoints, an open version where the formulas do not
++ include the endpoints and an adaptive version, where the user
++ is required to input the number of subintervals over which the
++ appropriate closed family integrator will apply with the usual
++ convergence parmeters for each subinterval. This is useful
++ where a large number of points are needed only in a small fraction
++ of the entire domain.
```

```
++
++ Each routine takes as arguments:\br
++ f integrand\br
++ a starting point\br
++ b ending point\br
++ eps_r relative error\br
++ eps_a absolute error\br
++ nmin refinement level when to start checking for convergence (> 1)\br
++ nmax maximum level of refinement\br
++
++ The adaptive routines take as an additional parameter,
++ nint, the number of independent intervals to apply a closed
++ family integrator of the same name.
++
++ Notes:\br
++ Closed family level i uses \spad{1 + 2**i} points.\br
++ Open family level i uses \spad{1 + 3**i} points.\br

NumericalQuadrature(): Exports == Implementation where
  L        ==> List
  V        ==> Vector
  I        ==> Integer
  B        ==> Boolean
  E        ==> OutputForm
  F       ==> Float
  PI       ==> PositiveInteger
  OFORM    ==> OutputForm
  TrapAns  ==> Record(value:F, error:F, totalpts:I, success:B )

  Exports ==> with
   aromberg     : (F -> F,F,F,F,F,I,I,I) -> TrapAns
     ++ aromberg(fn,a,b,epsrel,epsabs,nmin,nmax,nint)
     ++ uses the adaptive romberg method to numerically integrate function
     ++ \spad{fn} over the closed interval from \spad{a} to \spad{b},
     ++ with relative accuracy \spad{epsrel} and absolute accuracy
     ++ \spad{epsabs}, with the refinement levels for convergence checking
     ++ vary from \spad{nmin} to \spad{nmax}, and where \spad{nint}
     ++ is the number of independent intervals to apply the integrator.
     ++ The value returned is a record containing the value of the integral,
     ++ the estimate of the error in the computation, the total number of
     ++ function evaluations, and either a boolean value which is true if
     ++ the integral was computed within the user specified error criterion.
     ++ See \spadtype{NumericalQuadrature} for details.
   asimpson     : (F -> F,F,F,F,F,I,I,I) -> TrapAns
     ++ asimpson(fn,a,b,epsrel,epsabs,nmin,nmax,nint) uses the
     ++ adaptive simpson method to numerically integrate function \spad{fn}
     ++ over the closed interval from \spad{a} to \spad{b}, with relative
     ++ accuracy \spad{epsrel} and absolute accuracy \spad{epsabs}, with the
     ++ refinement levels for convergence checking vary from \spad{nmin}
     ++ to \spad{nmax}, and where \spad{nint} is the number of independent
```

```
    ++ intervals to apply the integrator. The value returned is a record
    ++ containing the value of the integral, the estimate of the error in
    ++ the computation, the total number of function evaluations, and
    ++ either a boolean value which is true if the integral was computed
    ++ within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  atrapezoidal : (F -> F,F,F,F,F,I,I,I) -> TrapAns
    ++ atrapezoidal(fn,a,b,epsrel,epsabs,nmin,nmax,nint) uses the
    ++ adaptive trapezoidal method to numerically integrate function
    ++ \spad{fn} over the closed interval from \spad{a} to \spad{b}, with
    ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
    ++ with the refinement levels for convergence checking vary from
    ++ \spad{nmin} to \spad{nmax}, and where \spad{nint} is the number
    ++ of independent intervals to apply the integrator. The value returned
    ++ is a record containing the value of the integral, the estimate of
    ++ the error in the computation, the total number of function
    ++ evaluations, and either a boolean value which is true if
    ++ the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  romberg      : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ romberg(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
    ++ method to numerically integrate function \spadvar{fn} over the closed
    ++ interval \spad{a} to \spad{b}, with relative accuracy \spad{epsrel}
    ++ and absolute accuracy \spad{epsabs}, with the refinement levels
    ++ for convergence checking vary from \spad{nmin} to \spad{nmax}.
    ++ The value returned is a record containing the value
    ++ of the integral, the estimate of the error in the computation, the
    ++ total number of function evaluations, and either a boolean value
    ++ which is true if the integral was computed within the user specified
    ++ error criterion. See \spadtype{NumericalQuadrature} for details.
  simpson      : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ simpson(fn,a,b,epsrel,epsabs,nmin,nmax) uses the simpson
    ++ method to numerically integrate function \spad{fn} over the closed
    ++ interval \spad{a} to \spad{b}, with
    ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
    ++ with the refinement levels for convergence checking vary from
    ++ \spad{nmin} to \spad{nmax}. The value returned
    ++ is a record containing the value of the integral, the estimate of
    ++ the error in the computation, the total number of function
    ++ evaluations, and either a boolean value which is true if
    ++ the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  trapezoidal  : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ trapezoidal(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
    ++ trapezoidal method to numerically integrate function \spadvar{fn} over
    ++ the closed interval \spad{a} to \spad{b}, with relative accuracy
    ++ \spad{epsrel} and absolute accuracy \spad{epsabs}, with the
    ++ refinement levels for convergence checking vary
    ++ from \spad{nmin} to \spad{nmax}. The value
    ++ returned is a record containing the value of the integral, the
```

```
    ++ estimate of the error in the computation, the total number of
    ++ function evaluations, and either a boolean value which is true
    ++ if the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  rombergo     : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ rombergo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
    ++ method to numerically integrate function \spad{fn} over
    ++ the open interval from \spad{a} to \spad{b}, with
    ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
    ++ with the refinement levels for convergence checking vary from
    ++ \spad{nmin} to \spad{nmax}. The value returned
    ++ is a record containing the value of the integral, the estimate of
    ++ the error in the computation, the total number of function
    ++ evaluations, and either a boolean value which is true if
    ++ the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  simpsono     : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ simpsono(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
    ++ simpson method to numerically integrate function \spad{fn} over
    ++ the open interval from \spad{a} to \spad{b}, with
    ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
    ++ with the refinement levels for convergence checking vary from
    ++ \spad{nmin} to \spad{nmax}. The value returned
    ++ is a record containing the value of the integral, the estimate of
    ++ the error in the computation, the total number of function
    ++ evaluations, and either a boolean value which is true if
    ++ the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.
  trapezoidalo : (F -> F,F,F,F,F,I,I) -> TrapAns
    ++ trapezoidalo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
    ++ trapezoidal method to numerically integrate function \spad{fn}
    ++ over the open interval from \spad{a} to \spad{b}, with
    ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
    ++ with the refinement levels for convergence checking vary from
    ++ \spad{nmin} to \spad{nmax}. The value returned
    ++ is a record containing the value of the integral, the estimate of
    ++ the error in the computation, the total number of function
    ++ evaluations, and either a boolean value which is true if
    ++ the integral was computed within the user specified error criterion.
    ++ See \spadtype{NumericalQuadrature} for details.

 Implementation ==> add
  trapclosed : (F -> F,F,F,F,I) -> F
  trapopen   : (F -> F,F,F,F,I) -> F
  import OutputPackage

-----------------------------------------------

  aromberg(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
     ans  : TrapAns
```

```
      sum  : F := 0.0
      err  : F := 0.0
      pts  : I  := 1
      done : B  := true
      hh   : F := (b-a) / nint
      x1   : F := a
      x2   : F := a + hh
      io   : L OFORM := [x1::E,x2::E]
      i    : I
      for i in 1..nint repeat
         ans := romberg(func,x1,x2,epsrel,epsabs,nmin,nmax)
         if (not ans.success) then
           io.1 := x1::E
           io.2 := x2::E
           print blankSeparate cons("accuracy not reached in interval"::E,io)
         sum  := sum + ans.value
         err  := err + abs(ans.error)
         pts  := pts + ans.totalpts-1
         done := (done and ans.success)
         x1   := x2
         x2   := x2 + hh
      return( [sum , err , pts , done] )

------------------------------------------------

   asimpson(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
      ans  : TrapAns
      sum  : F := 0.0
      err  : F := 0.0
      pts  : I  := 1
      done : B  := true
      hh   : F := (b-a) / nint
      x1   : F := a
      x2   : F := a + hh
      io   : L OFORM := [x1::E,x2::E]
      i    : I
      for i in 1..nint repeat
         ans := simpson(func,x1,x2,epsrel,epsabs,nmin,nmax)
         if (not ans.success) then
           io.1 := x1::E
           io.2 := x2::E
           print blankSeparate cons("accuracy not reached in interval"::E,io)
         sum  := sum + ans.value
         err  := err + abs(ans.error)
         pts  := pts + ans.totalpts-1
         done := (done and ans.success)
         x1   := x2
         x2   := x2 + hh
      return( [sum , err , pts , done] )
```

```
------------------------------------------------

   atrapezoidal(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
      ans  : TrapAns
      sum  : F := 0.0
      err  : F := 0.0
      pts  : I  := 1
      i    : I
      done : B := true
      hh   : F := (b-a) / nint
      x1   : F := a
      x2   : F := a + hh
      io   : L OFORM := [x1::E,x2::E]
      for i in 1..nint repeat
         ans := trapezoidal(func,x1,x2,epsrel,epsabs,nmin,nmax)
         if (not ans.success) then
           io.1 := x1::E
           io.2 := x2::E
           print blankSeparate cons("accuracy not reached in interval"::E,io)
         sum  := sum + ans.value
         err  := err + abs(ans.error)
         pts  := pts + ans.totalpts-1
         done := (done and ans.success)
         x1   := x2
         x2   := x2 + hh
      return( [sum , err , pts , done] )

------------------------------------------------

   romberg(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length
      newsum : F := 0.5 * length * (func(a)+func(b))
      newest : F := 0.0
      oldsum : F := 0.0
      oldest : F := 0.0
      change : F := 0.0
      qx1    : F := newsum
      table  : V F := new((nmax+1)::PI,0.0)
      n      : I  := 1
      pts    : I  := 1
      four   : I
      j      : I
      i      : I
      if (nmin < 2) then
         output("romberg: nmin to small (nmin > 1) nmin = ",nmin::E)
         return([0.0,0.0,0,false])
      if (nmax < nmin) then
         output("romberg: nmax < nmin : nmax = ",nmax::E)
         output("                       nmin = ",nmin::E)
```

```
         return([0.0,0.0,0,false])
      if (a = b) then
         output("romberg: integration limits are equal  = ",a::E)
         return([0.0,0.0,1,true])
      if (epsrel < 0.0) then
         output("romberg: eps_r < 0.0            eps_r  = ",epsrel::E)
         return([0.0,0.0,0,false])
      if (epsabs < 0.0) then
         output("romberg: eps_a < 0.0            eps_a  = ",epsabs::E)
         return([0.0,0.0,0,false])
      for n in 1..nmax repeat
         oldsum := newsum
         newsum := trapclosed(func,a,delta,oldsum,pts)
         newest := (4.0 * newsum - oldsum) / 3.0
         four   := 4
         table(n) := newest
         for j in 2..n repeat
            i        := n+1-j
            four     := four * 4
            table(i) := table(i+1) + (table(i+1)-table(i)) / (four-1)
         if n > nmin then
            change := abs(table(1) - qx1)
            if change < abs(epsrel*qx1) then
               return( [table(1) , change , 2*pts+1 , true] )
            if change < epsabs then
               return( [table(1) , change , 2*pts+1 , true] )
         oldsum := newsum
         oldest := newest
         delta  := 0.5*delta
         pts    := 2*pts
         qx1    := table(1)
      return( [table(1) , 1.25*change , pts+1 ,false] )

-------------------------------------------------

   simpson(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length
      newsum : F := 0.5*(b-a)*(func(a)+func(b))
      newest : F := 0.0
      oldsum : F := 0.0
      oldest : F := 0.0
      change : F := 0.0
      n      : I  := 1
      pts    : I  := 1
      if (nmin < 2) then
         output("simpson: nmin to small (nmin > 1) nmin = ",nmin::E)
         return([0.0,0.0,0,false])
      if (nmax < nmin) then
         output("simpson: nmax < nmin : nmax = ",nmax::E)
```

```
         output("                        nmin = ",nmin::E)
         return([0.0,0.0,0,false])
      if (a = b) then
         output("simpson: integration limits are equal  = ",a::E)
         return([0.0,0.0,1,true])
      if (epsrel < 0.0) then
         output("simpson: eps_r < 0.0 : eps_r = ",epsrel::E)
         return([0.0,0.0,0,false])
      if (epsabs < 0.0) then
         output("simpson: eps_a < 0.0 : eps_a = ",epsabs::E)
         return([0.0,0.0,0,false])
      for n in 1..nmax repeat
         oldsum := newsum
         newsum := trapclosed(func,a,delta,oldsum,pts)
         newest := (4.0 * newsum - oldsum) / 3.0
         if n > nmin then
            change := abs(newest-oldest)
            if change < abs(epsrel*oldest) then
               return( [newest , 1.25*change , 2*pts+1 , true] )
            if change < epsabs then
               return( [newest , 1.25*change , 2*pts+1 , true] )
         oldsum := newsum
         oldest := newest
         delta  := 0.5*delta
         pts    := 2*pts
      return( [newest , 1.25*change , pts+1 ,false] )

------------------------------------------------

   trapezoidal(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length
      newsum : F := 0.5*(b-a)*(func(a)+func(b))
      change : F := 0.0
      oldsum : F
      n      : I  := 1
      pts    : I  := 1
      if (nmin < 2) then
         output("trapezoidal: nmin to small (nmin > 1) nmin = ",nmin::E)
         return([0.0,0.0,0,false])
      if (nmax < nmin) then
         output("trapezoidal: nmax < nmin : nmax = ",nmax::E)
         output("                           nmin = ",nmin::E)
         return([0.0,0.0,0,false])
      if (a = b) then
         output("trapezoidal: integration limits are equal  = ",a::E)
         return([0.0,0.0,1,true])
      if (epsrel < 0.0) then
         output("trapezoidal: eps_r < 0.0 : eps_r = ",epsrel::E)
         return([0.0,0.0,0,false])
```

```
      if (epsabs < 0.0) then
         output("trapezoidal: eps_a < 0.0 : eps_a = ",epsabs::E)
         return([0.0,0.0,0,false])
      for n in 1..nmax repeat
         oldsum := newsum
         newsum := trapclosed(func,a,delta,oldsum,pts)
         if n > nmin then
            change := abs(newsum-oldsum)
            if change < abs(epsrel*oldsum) then
               return( [newsum , 1.25*change , 2*pts+1 , true] )
            if change < epsabs then
               return( [newsum , 1.25*change , 2*pts+1 , true] )
         delta := 0.5*delta
         pts   := 2*pts
      return( [newsum , 1.25*change , pts+1 ,false] )

-------------------------------------------------

   rombergo(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length / 3.0
      newsum : F := length * func( 0.5*(a+b) )
      newest : F := 0.0
      oldsum : F := 0.0
      oldest : F := 0.0
      change : F := 0.0
      qx1    : F := newsum
      table  : V F := new((nmax+1)::PI,0.0)
      four   : I
      j      : I
      i      : I
      n      : I  := 1
      pts    : I  := 1
      for n in 1..nmax repeat
         oldsum   := newsum
         newsum   := trapopen(func,a,delta,oldsum,pts)
         newest   := (9.0 * newsum - oldsum) / 8.0
         table(n) := newest
         nine     := 9
         output(newest::E)
         for j in 2..n repeat
            i        := n+1-j
            nine     := nine * 9
            table(i) := table(i+1) + (table(i+1)-table(i)) / (nine-1)
         if n > nmin then
            change := abs(table(1) - qx1)
            if change < abs(epsrel*qx1) then
               return( [table(1) , 1.5*change , 3*pts , true] )
            if change < epsabs then
               return( [table(1) , 1.5*change , 3*pts , true] )
```

```
         output(table::E)
         oldsum := newsum
         oldest := newest
         delta  := delta / 3.0
         pts    := 3*pts
         qx1    := table(1)
      return( [table(1) , 1.5*change , pts ,false] )

------------------------------------------------

   simpsono(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length / 3.0
      newsum : F := length * func( 0.5*(a+b) )
      newest : F := 0.0
      oldsum : F := 0.0
      oldest : F := 0.0
      change : F := 0.0
      n      : I  := 1
      pts    : I  := 1
      for n in 1..nmax repeat
         oldsum := newsum
         newsum := trapopen(func,a,delta,oldsum,pts)
         newest := (9.0 * newsum - oldsum) / 8.0
         output(newest::E)
         if n > nmin then
            change := abs(newest - oldest)
            if change < abs(epsrel*oldest) then
               return( [newest , 1.5*change , 3*pts , true] )
            if change < epsabs then
               return( [newest , 1.5*change , 3*pts , true] )
         oldsum := newsum
         oldest := newest
         delta  := delta / 3.0
         pts    := 3*pts
      return( [newest , 1.5*change , pts ,false] )

------------------------------------------------

   trapezoidalo(func,a,b,epsrel,epsabs,nmin,nmax) ==
      length : F := (b-a)
      delta  : F := length/3.0
      newsum : F := length*func( 0.5*(a+b) )
      change : F := 0.0
      pts    : I  := 1
      oldsum : F
      n      : I
      for n in 1..nmax repeat
         oldsum := newsum
         newsum := trapopen(func,a,delta,oldsum,pts)
```

```
         output(newsum::E)
         if n > nmin then
            change := abs(newsum-oldsum)
            if change < abs(epsrel*oldsum) then
               return([newsum , 1.5*change , 3*pts , true] )
            if change < epsabs then
               return([newsum , 1.5*change , 3*pts , true] )
         delta := delta / 3.0
         pts   := 3*pts
      return([newsum , 1.5*change , pts ,false] )

------------------------------------------------

   trapclosed(func,start,h,oldsum,numpoints) ==
      x   : F := start + 0.5*h
      sum : F := 0.0
      i   : I
      for i in 1..numpoints repeat
          sum := sum + func(x)
          x   := x + h
      return( 0.5*(oldsum + sum*h) )

------------------------------------------------

   trapopen(func,start,del,oldsum,numpoints) ==
      ddel : F := 2.0*del
      x   : F := start + 0.5*del
      sum : F := 0.0
      i   : I
      for i in 1..numpoints repeat
          sum := sum + func(x)
          x   := x + ddel
          sum := sum + func(x)
          x   := x + del
      return( (oldsum/3.0 + sum*del) )
```

———-

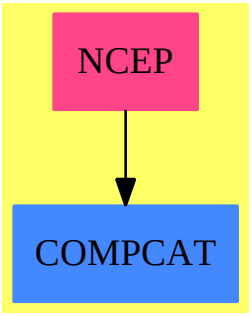
— NUMQUAD.dotabb —

```
"NUMQUAD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMQUAD"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"NUMQUAD" -> "IVECTOR"
```

———-

15.33 package NCEP NumericComplexEigenPackage

15.33.1 NumericComplexEigenPackage (NCEP)



Exports:
characteristicPolynomial complexEigenvalues complexEigenvectors

— package NCEP NumericComplexEigenPackage —

```
)abbrev package NCEP NumericComplexEigenPackage
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: Spring 1991
++ Basic Functions:
++ Related Constructors: FloatingComplexPackage
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package computes explicitly eigenvalues and eigenvectors of
++ matrices with entries over the complex rational numbers.
++ The results are expressed either as complex floating numbers or as
++ complex rational numbers depending on the type of the precision parameter.

NumericComplexEigenPackage(Par) : C == T
 where
   Par   :   Join(Field,OrderedRing)   -- Float or RationalNumber

   SE    ==> Symbol()
   RN    ==> Fraction Integer
   I     ==> Integer
   NF    ==> Float
   CF    ==> Complex Float
   GRN   ==> Complex RN
   GI    ==> Complex Integer
```

```
PI    ==> PositiveInteger
NNI   ==> NonNegativeInteger
MRN   ==> Matrix RN

MCF        ==> Matrix CF
MGRN       ==> Matrix GRN
MCPar      ==> Matrix Complex Par
SUPGRN     ==> SparseUnivariatePolynomial GRN
outForm    ==> Record(outval:Complex Par,outmult:Integer,outvect:List MCPar)

C == with
  characteristicPolynomial :   MGRN    -> Polynomial GRN
    ++ characteristicPolynomial(m) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over complex rationals with a new symbol as variable.
    -- while the function in EigenPackage returns Fraction P GRN.
  characteristicPolynomial : (MGRN,SE) -> Polynomial GRN
    ++ characteristicPolynomial(m,x) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over Complex Rationals with variable x.
    -- while the function in EigenPackage returns Fraction P GRN.
  complexEigenvalues  :   (MGRN,Par)   ->  List Complex Par
    ++ complexEigenvalues(m,eps) computes the eigenvalues of the matrix
    ++ m to precision eps. The eigenvalues are expressed as complex
    ++ floats or complex rational numbers depending on the type of
    ++ eps (float or rational).
  complexEigenvectors    : (MGRN,Par)   ->  List(outForm)
    ++ complexEigenvectors(m,eps)  returns a list of
    ++ records each one containing
    ++ a complex eigenvalue, its algebraic multiplicity, and a list of
    ++ associated eigenvectors. All these results
    ++ are computed to precision eps and are expressed as complex floats
    ++ or complex rational numbers depending on the type of
    ++ eps (float or rational).
T == add

  import InnerNumericEigenPackage(GRN,Complex Par,Par)

  characteristicPolynomial(m:MGRN) : Polynomial GRN  ==
    x:SE:=new()$SE
    multivariate(charpol m, x)

        ----  characteristic polynomial of a matrix A ----
  characteristicPolynomial(A:MGRN,x:SE):Polynomial GRN ==
    multivariate(charpol A, x)

  complexEigenvalues(m:MGRN,eps:Par) : List Complex Par  ==
    solve1(charpol m, eps)

  complexEigenvectors(m:MGRN,eps:Par) :List outForm ==
```

```
innerEigenvectors(m,eps,factor$ComplexFactorization(RN,SUPGRN))
```

———-

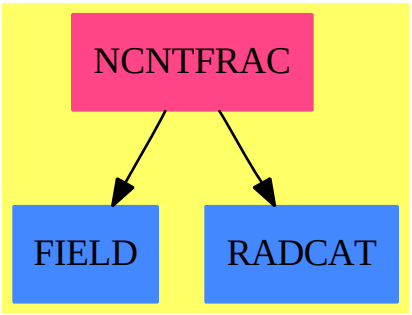
— **NCEP.dotabb** —

```
"NCEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCEP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NCEP" -> "COMPCAT"
```

———-

15.34 package NCNTFRAC NumericContinuedFraction

15.34.1 NumericContinuedFraction (NCNTFRAC)



Exports:
continuedFraction

— **package NCNTFRAC NumericContinuedFraction** —

```
)abbrev package NCNTFRAC NumericContinuedFraction
++ Author: Clifton J. Williamson
++ Date Created: 12 April 1990
++ Change History:
++ Basic Operations: continuedFraction
++ Related Constructors: ContinuedFraction, Float
++ Also See: Fraction
++ AMS Classifications: 11J70 11A55 11K50 11Y65 30B70 40A15
++ Keywords: continued fraction
++ References:
++ Description:
```

```
++ \spadtype{NumericContinuedFraction} provides functions
++ for converting floating point numbers to continued fractions.

NumericContinuedFraction(F): Exports == Implementation where
  F :     FloatingPointSystem
  CFC ==> ContinuedFraction Integer
  I   ==> Integer
  ST  ==> Stream I

  Exports ==> with
    continuedFraction: F -> CFC
      ++ continuedFraction(f) converts the floating point number
      ++ \spad{f} to a reduced continued fraction.

  Implementation ==> add

    cfc: F -> ST
    cfc(a) == delay
      aa := wholePart a
      zero?(b := a - (aa :: F)) => concat(aa,empty()$ST)
      concat(aa,cfc inv b)

    continuedFraction a ==
      aa := wholePart a
      zero?(b := a - (aa :: F)) =>
        reducedContinuedFraction(aa,empty()$ST)
      if negative? b then (aa := aa - 1; b := b + 1)
      reducedContinuedFraction(aa,cfc inv b)
```

———-

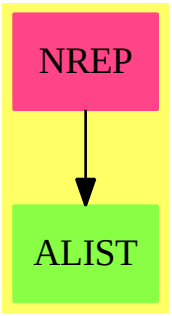
— NCNTFRAC.dotabb —

```
"NCNTFRAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCNTFRAC"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"NCNTFRAC" -> "FIELD"
"NCNTFRAC" -> "RADCAT"
```

———-

15.35 package NREP NumericRealEigenPackage

15.35.1 NumericRealEigenPackage (NREP)



Exports:
characteristicPolynomial realEigenvalues realEigenvectors

— package NREP NumericRealEigenPackage —

```
)abbrev package NREP NumericRealEigenPackage
++ Author:P. Gianni
++ Date Created:Summer 1990
++ Date Last Updated:Spring 1991
++ Basic Functions:
++ Related Constructors: FloatingRealPackage
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package computes explicitly eigenvalues and eigenvectors of
++ matrices with entries over the Rational Numbers.
++ The results are expressed as floating numbers or as rational numbers
++ depending on the type of the parameter Par.

NumericRealEigenPackage(Par) : C == T
 where
   Par   :   Join(Field,OrderedRing) -- Float or RationalNumber

   SE    ==> Symbol()
   RN    ==> Fraction Integer
   I     ==> Integer
   NF    ==> Float
   CF    ==> Complex Float
   GRN   ==> Complex RN
   GI    ==> Complex Integer
```

```
PI    ==> PositiveInteger
NNI   ==> NonNegativeInteger
MRN   ==> Matrix RN

MPar        ==> Matrix Par
outForm     ==> Record(outval:Par,outmult:Integer,outvect:List MPar)

C == with
  characteristicPolynomial :   MRN    -> Polynomial RN
    ++ characteristicPolynomial(m) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over RN with a new symbol as variable.
    -- while the function in EigenPackage returns Fraction P RN.
  characteristicPolynomial : (MRN,SE) -> Polynomial RN
    ++ characteristicPolynomial(m,x) returns the characteristic polynomial
    ++ of the matrix m expressed as polynomial
    ++ over RN with variable x.
    -- while the function in EigenPackage returns
    ++ Fraction P RN.
  realEigenvalues  :   (MRN,Par)   ->  List Par
    ++ realEigenvalues(m,eps) computes the eigenvalues of the matrix
    ++ m to precision eps. The eigenvalues are expressed as floats or
    ++ rational numbers depending on the type of eps (float or rational).
  realEigenvectors    : (MRN,Par)   ->  List(outForm)
    ++ realEigenvectors(m,eps)  returns a list of
    ++ records each one containing
    ++ a real eigenvalue, its algebraic multiplicity, and a list of
    ++ associated eigenvectors. All these results
    ++ are computed to precision eps as floats or rational
    ++ numbers depending on the type of eps .


T == add

  import InnerNumericEigenPackage(RN, Par, Par)

  characteristicPolynomial(m:MRN) : Polynomial RN ==
    x:SE:=new()$SE
    multivariate(charpol(m),x)

        ----  characteristic polynomial of a matrix A ----
  characteristicPolynomial(A:MRN,x:SE):Polynomial RN ==
    multivariate(charpol(A),x)

  realEigenvalues(m:MRN,eps:Par) : List Par  ==
    solve1(charpol m, eps)

  realEigenvectors(m:MRN,eps:Par) :List outForm ==
    innerEigenvectors(m,eps,factor$GenUFactorize(RN))
```

———-

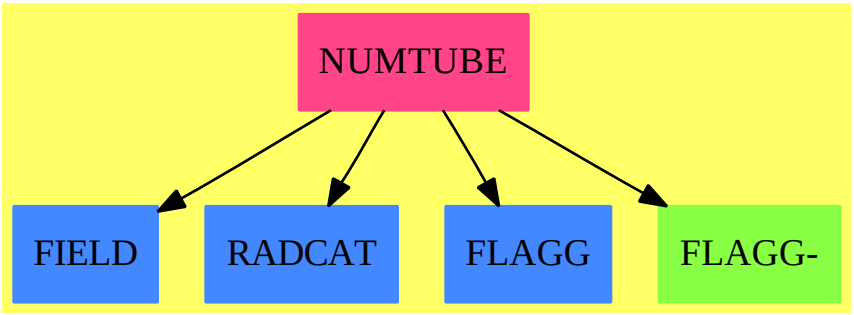
— NREP.dotabb —

```
"NREP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NREP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NREP" -> "ALIST"
```

———-

15.36 package NUMTUBE NumericTubePlot

15.36.1 NumericTubePlot (NUMTUBE)



Exports:
tube

— package NUMTUBE NumericTubePlot —

```
)abbrev package NUMTUBE NumericTubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Keywords:
++ Examples:
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.

NumericTubePlot(Curve): Exports == Implementation where
  Curve : PlottableSpaceCurveCategory
  B   ==> Boolean
  I   ==> Integer
  SF  ==> DoubleFloat
```

```
L   ==> List
S   ==> String
SEG ==> Segment
Pt  ==> Point SF
TUBE ==> TubePlot Curve
Triad ==> Record(tang:Pt,norm:Pt,bin:Pt)

Exports ==> with
  tube: (Curve,SF,I) -> TUBE
    ++ tube(c,r,n) creates a tube of radius r around the curve c.

Implementation ==> add
  import TubePlotTools

  LINMAX  := convert(0.995)@SF
  XHAT := point(1,0,0,0)
  YHAT := point(0,1,0,0)
  PREV0 := point(1,1,0,0)
  PREV := PREV0

  colinearity: (Pt,Pt) -> SF
  colinearity(x,y) == dot(x,y)**2/(dot(x,x) * dot(y,y))

  orthog: (Pt,Pt) -> Pt
  orthog(x,y) ==
    if colinearity(x,y) > LINMAX then y := PREV
    if colinearity(x,y) > LINMAX then
      y := (colinearity(x,XHAT) < LINMAX => XHAT; YHAT)
    a := -dot(x,y)/dot(x,x)
    PREV := a*x + y

  poTriad:(Pt,Pt,Pt) -> Triad
  poTriad(pl,po,pr) ==
    -- use divided difference for t.
    t := unitVector(pr - pl)
    -- compute n as orthogonal to t in plane containing po.
    pol := pl - po
    n   := unitVector orthog(t,pol)
    [t,n,cross(t,n)]

  curveTriads: L Pt -> L Triad
  curveTriads l ==
    (k := #l) < 2 => error "Need at least 2 points to specify a curve"
    PREV := PREV0
    k = 2 =>
      t := unitVector(second l - first l)
      n := unitVector(t - XHAT)
      b := cross(t,n)
      triad : Triad := [t,n,b]
      [triad,triad]
```

```
    -- compute interior triads using divided differences
    midtriads : L Triad :=
      [poTriad(pl,po,pr) for pl in l for po in rest l _
            for pr in rest rest l]
    -- compute first triad using a forward difference
    x := first midtriads
    t := unitVector(second l - first l)
    n := unitVector orthog(t,x.norm)
    begtriad : Triad := [t,n,cross(t,n)]
    -- compute last triad using a backward difference
    x := last midtriads
    -- efficiency!!
    t := unitVector(l.k - l.(k-1))
    n := unitVector orthog(t,x.norm)
    endtriad : Triad := [t,n,cross(t,n)]
    concat(begtriad,concat(midtriads,endtriad))

  curveLoops: (L Pt,SF,I) -> L L Pt
  curveLoops(pts,r,nn) ==
    triads := curveTriads pts
    cosSin := cosSinInfo nn
    loops : L L Pt := nil()
    for pt in pts for triad in triads repeat
      n := triad.norm; b := triad.bin
      loops := concat(loopPoints(pt,n,b,r,cosSin),loops)
    reverse_! loops

  tube(curve,r,n) ==
    n < 3 => error "tube: n should be at least 3"
    brans := listBranches curve
    loops : L L Pt := nil()
    for bran in brans repeat
      loops := concat(loops,curveLoops(bran,r,n))
    tube(curve,loops,false)
```

———-

— **NUMTUBE.dotabb** —

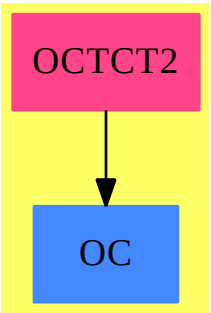
```
"NUMTUBE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMTUBE"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"NUMTUBE" -> "FIELD"
"NUMTUBE" -> "RADCAT"
"NUMTUBE" -> "FLAGG"
"NUMTUBE" -> "FLAGG-"
```

Chapter 16

Chapter O

16.1 package OCTCT2 OctonionCategoryFunctions2

16.1.1 OctonionCategoryFunctions2 (OCTCT2)



Exports:
map

— package OCTCT2 OctonionCategoryFunctions2 —

```
)abbrev package OCTCT2 OctonionCategoryFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 10 September 1990
++ Date Last Updated: 10 September 1990
++ Basic Operations: map
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: octonion, non-associative algebra, Cayley-Dixon
```

```
++ References:
++ Description:
++ OctonionCategoryFunctions2 implements functions between
++ two octonion domains defined over different rings.
++ The function map is used to coerce between octonion types.

OctonionCategoryFunctions2(OR,R,OS,S) : Exports ==
  Implementation where
    R  : CommutativeRing
    S  : CommutativeRing
    OR : OctonionCategory R
    OS : OctonionCategory S
    Exports == with
      map:     (R -> S, OR) -> OS
        ++ map(f,u) maps f onto the component parts of the octonion
        ++ u.
    Implementation == add
      map(fn : R -> S, u : OR): OS ==
        octon(fn real u, fn imagi u, fn imagj u, fn imagk u,_
        fn imagE u, fn imagI u, fn imagJ u, fn imagK u)$OS
```

———-

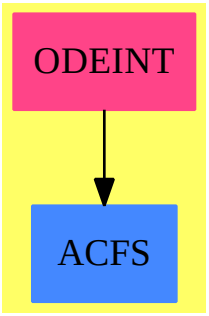
— OCTCT2.dotabb —

```
"OCTCT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OCTCT2"]
"OC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OC"]
"OCTCT2" -> "OC"
```

———-

16.2 package ODEINT ODEIntegration

16.2.1 ODEIntegration (ODEINT)



Exports:
diff expint int

— package ODEINT ODEIntegration —

```
)abbrev package ODEINT ODEIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 November 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spadtype{ODEIntegration} provides an interface to the integrator.
++ This package is intended for use
++ by the differential equations solver but not at top-level.

ODEIntegration(R, F): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
                                              PrimitiveFunctionCategory)

  Q   ==> Fraction Integer
  UQ  ==> Union(Q, "failed")
  SY  ==> Symbol
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  REC ==> Record(coef:Q, logand:F)

  Exports ==> with
    int   : (F, SY) -> F
      ++ int(f, x) returns the integral of f with respect to x.
    expint: (F, SY) -> F
      ++ expint(f, x) returns e^{the integral of f with respect to x}.
```

```
    diff  : SY -> (F -> F)
      ++ diff(x) returns the derivation with respect to x.

  Implementation ==> add
    import FunctionSpaceIntegration(R, F)
    import ElementaryFunctionStructurePackage(R, F)

    isQ   : List F -> UQ
    isQlog: F -> Union(REC, "failed")
    mkprod: List REC -> F

    diff x == (f1:F):F +-> differentiate(f1, x)

-- This is the integration function to be used for quadratures
    int(f, x) ==
      (u := integrate(f, x)) case F => u::F
      first(u::List(F))

-- mkprod([q1, f1],...,[qn,fn]) returns */(fi^qi) but groups the
-- qi having the same denominator together
    mkprod l ==
      empty? l => 1
      rec := first l
      d := denom(rec.coef)
      ll := select((z1:REC):Boolean +-> denom(z1.coef) = d, l)
      nthRoot(*/[r.logand ** numer(r.coef) for r in ll], d) *
        mkprod setDifference(l, ll)

-- computes exp(int(f,x)) in a non-naive way
    expint(f, x) ==
      a := int(f, x)
      (u := validExponential(tower a, a, x)) case F => u::F
      da := denom a
      l :=
        (v := isPlus(na := numer a)) case List(P) => v::List(P)
        [na]
      exponent:P := 0
      lrec:List(REC) := empty()
      for term in l repeat
        if (w := isQlog(term / da)) case REC then
          lrec := concat(w::REC, lrec)
        else
          exponent := exponent + term
      mkprod(lrec) * exp(exponent / da)

-- checks if all the elements of l are rational numbers, returns their product
    isQ l ==
      prod:Q := 1
      for x in l repeat
        (u := retractIfCan(x)@UQ) case "failed" => return "failed"
```

```
        prod := prod * u::Q
      prod

-- checks if a non-sum expr is of the form c * log(g) for a rational number c
    isQlog f ==
      is?(f, "log"::SY) => [1, first argument(retract(f)@K)]
      (v := isTimes f) case List(F) and (#(l := v::List(F)) <= 3) =>
          l := reverse_! sort_! l
          is?(first l, "log"::SY) and ((u := isQ rest l) case Q) =>
              [u::Q, first argument(retract(first(l))@K)]
          "failed"
      "failed"
```

———-

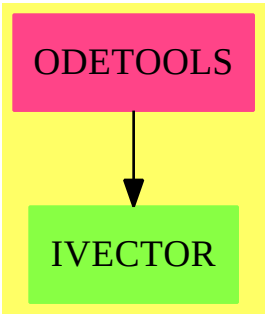
— **ODEINT.dotabb** —

```
"ODEINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEINT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODEINT" -> "ACFS"
```

———-

16.3 package ODETOOLS ODETools

16.3.1 ODETools (ODETOOLS)



Exports:
particularSolution variationOfParameters wronskianMatrix

— **package ODETOOLS ODETools** —

```
)abbrev package ODETOOLS ODETools
++ Author: Manuel Bronstein
++ Date Created: 20 March 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spad{ODETools} provides tools for the linear ODE solver.

ODETools(F, LODO): Exports == Implementation where
  N ==> NonNegativeInteger
  L ==> List F
  V ==> Vector F
  M ==> Matrix F

  F:    Field
  LODO: LinearOrdinaryDifferentialOperatorCategory F

  Exports ==> with
    wronskianMatrix: L -> M
      ++ wronskianMatrix([f1,...,fn]) returns the \spad{n x n} matrix m
      ++ whose i^th row is \spad{[f1^(i-1),...,fn^(i-1)]}.
    wronskianMatrix: (L, N) -> M
      ++ wronskianMatrix([f1,...,fn], q, D) returns the \spad{q x n} matrix m
      ++ whose i^th row is \spad{[f1^(i-1),...,fn^(i-1)]}.
    variationOfParameters: (LODO, F, L) -> Union(V, "failed")
      ++ variationOfParameters(op, g, [f1,...,fm])
      ++ returns \spad{[u1,...,um]} such that a particular solution of the
      ++ equation \spad{op y = g} is \spad{f1 int(u1) + ... + fm int(um)}
      ++ where \spad{[f1,...,fm]} are linearly independent and \spad{op(fi)=0}.
      ++ The value "failed" is returned if \spad{m < n} and no particular
      ++ solution is found.
    particularSolution: (LODO, F, L, F -> F) -> Union(F, "failed")
      ++ particularSolution(op, g, [f1,...,fm], I) returns a particular
      ++ solution h of the equation \spad{op y = g} where \spad{[f1,...,fm]}
      ++ are linearly independent and \spad{op(fi)=0}.
      ++ The value "failed" is returned if no particular solution is found.
      ++ Note that the method of variations of parameters is used.

  Implementation ==> add
    import LinearSystemMatrixPackage(F, V, V, M)

    diff := D()$LODO

    wronskianMatrix l == wronskianMatrix(l, #l)

    wronskianMatrix(l, q) ==
      v:V := vector l
      m:M := zero(q, #v)
      for i in minRowIndex m .. maxRowIndex m repeat
        setRow_!(m, i, v)
        v := map_!((f1:F):F +-> diff f1, v)
```

```
      m

    variationOfParameters(op, g, b) ==
      empty? b => "failed"
      v:V := new(n := degree op, 0)
      qsetelt_!(v, maxIndex v, g / leadingCoefficient op)
      particularSolution(wronskianMatrix(b, n), v)

    particularSolution(op, g, b, integration) ==
      zero? g => 0
      (sol := variationOfParameters(op, g, b)) case "failed" => "failed"
      ans:F := 0
      for f in b for i in minIndex(s := sol::V) .. repeat
        ans := ans + integration(qelt(s, i)) * f
      ans
```

———-

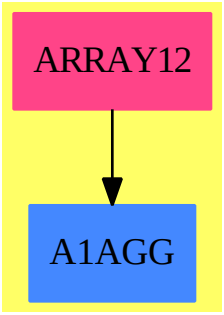
— **ODETOOLS.dotabb** —

```
"ODETOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODETOOLS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ODETOOLS" -> "IVECTOR"
```

———-

16.4 package ARRAY12 OneDimensionalArrayFunctions2

16.4.1 OneDimensionalArrayFunctions2 (ARRAY12)



Exports:
map reduce scan

— package ARRAY12 OneDimensionalArrayFunctions2 —

```
)abbrev package ARRAY12 OneDimensionalArrayFunctions2
++ Description:
++ This package provides tools for operating on one-dimensional arrays
++ with unary and binary functions involving different underlying types

OneDimensionalArrayFunctions2(A, B): Exports == Implementation where
  A, B: Type

  VA ==> OneDimensionalArray A
  VB ==> OneDimensionalArray B
  O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)

  Exports ==> with
   scan   : ((A, B) -> B, VA, B) -> VB
    ++ scan(f,a,r) successively applies
    ++ \spad{reduce(f,x,r)} to more and more leading sub-arrays
    ++ x of one-dimensional array \spad{a}.
    ++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
    ++ \spad{scan(f,a,r)} returns
    ++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
    ++
    ++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
    ++X adder(a:Integer,b:Integer):Integer == a+b
    ++X scan(adder,[i for i in 1..10],0)$T1

   reduce : ((A, B) -> B, VA, B) -> B
    ++ reduce(f,a,r) applies function f to each
    ++ successive element of the
    ++ one-dimensional array \spad{a} and an accumulant initialized to r.
    ++ For example, \spad{reduce(_+$Integer,[1,2,3],0)}
    ++ does \spad{3+(2+(1+0))}. Note that third argument r
    ++ may be regarded as the identity element for the function f.
    ++
    ++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
    ++X adder(a:Integer,b:Integer):Integer == a+b
    ++X reduce(adder,[i for i in 1..10],0)$T1

   map : (A -> B, VA) -> VB
    ++ map(f,a) applies function f to each member of one-dimensional array
    ++ \spad{a} resulting in a new one-dimensional array over a
    ++ possibly different underlying domain.
    ++
    ++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
    ++X map(x+->x+2,[i for i in 1..10])$T1

  Implementation ==> add
    map(f, v)       == map(f, v)$O2
```

```
scan(f, v, b)   == scan(f, v, b)$O2
reduce(f, v, b) == reduce(f, v, b)$O2
```

———-

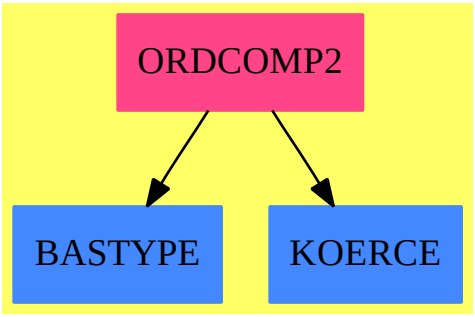
— **ARRAY12.dotabb** —

```
"ARRAY12" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ARRAY12"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARRAY12" -> "A1AGG"
```

———-

16.5 package ONECOMP2 OnePointCompletionFunctions2

16.5.1 OnePointCompletionFunctions2 (ONECOMP2)



Exports:
map

— **package ONECOMP2 OnePointCompletionFunctions2** —

```
)abbrev package ONECOMP2 OnePointCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to one-point completions.

OnePointCompletionFunctions2(R, S): Exports == Implementation where
  R, S: SetCategory
```

```
  OPR ==> OnePointCompletion R
  OPS ==> OnePointCompletion S

  Exports ==> with
    map: (R -> S, OPR) -> OPS
      ++ map(f, r) lifts f and applies it to r, assuming that
      ++ f(infinity) = infinity.
    map: (R -> S, OPR, OPS) -> OPS
      ++ map(f, r, i) lifts f and applies it to r, assuming that
      ++ f(infinity) = i.

  Implementation ==> add
    map(f, r) == map(f, r, infinity())

    map(f, r, i) ==
      (u := retractIfCan r) case R => (f(u::R))::OPS
      i
```

———-

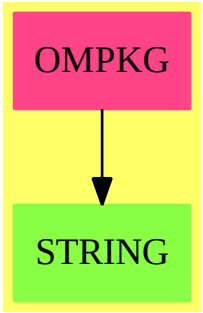
— ONECOMP2.dotabb —

```
"ONECOMP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ONECOMP2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ONECOMP2" -> "BASTYPE"
"ONECOMP2" -> "KOERCE"
```

———-

16.6 package OMPKG OpenMathPackage

16.6.1 OpenMathPackage (OMPKG)



Exports:

OMlistCDs	OMread	OMreadFile	OMreadStr	OMsupportsCD?
OMlistSymbols	OMsupportsSymbol?	OMunhandledSymbol		

— package OMPKG OpenMathPackage —

```
)abbrev package OMPKG OpenMathPackage
++ Author: Vilya Harvey
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{OpenMathPackage} provides some simple utilities
++ to make reading OpenMath objects easier.

OpenMathPackage(): with
  OMread            : OpenMathDevice -> Any
  ++ OMread(dev) reads an OpenMath object from \axiom{dev} and passes it
  ++ to AXIOM.
  OMreadFile        : String -> Any
  ++ OMreadFile(f) reads an OpenMath object from \axiom{f} and passes it
  ++ to AXIOM.
  OMreadStr         : String -> Any
  ++ OMreadStr(f) reads an OpenMath object from \axiom{f} and passes it
  ++ to AXIOM.
  OMlistCDs         : () -> List(String)
  ++ OMlistCDs() lists all the CDs supported by AXIOM.
  OMlistSymbols     : String -> List(String)
  ++ OMlistSymbols(cd) lists all the symbols in \axiom{cd}.
  OMsupportsCD?      : String -> Boolean
  ++ OMsupportsCD?(cd) returns true if AXIOM supports \axiom{cd}, false
  ++ otherwise.
  OMsupportsSymbol? : (String, String) -> Boolean
  ++ OMsupportsSymbol?(s,cd) returns true if AXIOM supports symbol \axiom{s}
  ++ from CD \axiom{cd}, false otherwise.
  OMunhandledSymbol : (String, String) -> Exit
  ++ OMunhandledSymbol(s,cd) raises an error if AXIOM reads a symbol which it
  ++ is unable to handle.  Note that this is different from an unexpected
  ++ symbol.
 == add
  import OpenMathEncoding
  import OpenMathDevice
  import String

  OMunhandledSymbol(u,v) ==
```

```
    error concat ["AXIOM is unable to process the symbol ",u," from CD ",v,"."]

  OMread(dev: OpenMathDevice): Any ==
    interpret(OM_-READ(dev)$Lisp :: InputForm)

  OMreadFile(filename: String): Any ==
    dev := OMopenFile(filename, "r", OMencodingUnknown())
    res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
    OMclose(dev)
    res

  OMreadStr(str: String): Any ==
    strp := OM_-STRINGTOSTRINGPTR(str)$Lisp
    dev := OMopenString(strp pretend String, OMencodingUnknown())
    res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
    OMclose(dev)
    res

  OMlistCDs(): List(String) ==
    OM_-LISTCDS()$Lisp pretend List(String)

  OMlistSymbols(cd: String): List(String) ==
    OM_-LISTSYMBOLS(cd)$Lisp pretend List(String)

  import SExpression

  OMsupportsCD?(cd: String): Boolean ==
    not null? OM_-SUPPORTSCD(cd)$Lisp

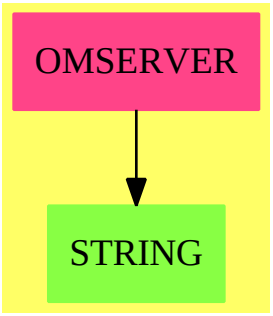
  OMsupportsSymbol?(cd: String, name: String): Boolean ==
    not null? OM_-SUPPORTSSYMBOL(cd, name)$Lisp
```

— OMPKG.dotabb —

```
"OMPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMPKG"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMPKG" -> "STRING"
```

16.7 package OMSERVER OpenMathServerPackage

16.7.1 OpenMathServerPackage (OMSERVER)



Exports:
OMreceive OMsend OMserve

— package OMSERVER OpenMathServerPackage —

```
)abbrev package OMSERVER OpenMathServerPackage
++ Author: Vilya Harvey
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{OpenMathServerPackage} provides the necessary
++ operations to run AXIOM as an OpenMath server, reading/writing objects
++ to/from a port.  Please note the facilities available here are very basic.
++ The idea is that a user calls e.g. \axiom{Omserve(4000,60)} and then
++ another process sends OpenMath objects to port 4000 and reads the result.

OpenMathServerPackage(): with
  OMreceive : OpenMathConnection -> Any
  ++ OMreceive(c) reads an OpenMath object from connection \axiom{c} and
  ++ returns the appropriate AXIOM object.
  OMsend    : (OpenMathConnection, Any) -> Void
  ++ OMsend(c,u) attempts to output \axiom{u} on \axiom{c} in OpenMath.
  OMserve   : (SingleInteger, SingleInteger) -> Void
  ++ OMserve(portnum,timeout) puts AXIOM into server mode on port number
  ++ \axiom{portnum}.  The parameter \axiom{timeout} specifies the timeout
  ++ period for the connection.
```

```
== add
 import OpenMathDevice
 import OpenMathConnection
 import OpenMathPackage
 import OpenMath



 OMreceive(conn: OpenMathConnection): Any ==
   dev: OpenMathDevice := OMconnInDevice(conn)
   OMsetEncoding(dev, OMencodingUnknown);
   OMread(dev)

 OMsend(conn: OpenMathConnection, value: Any): Void ==
   dev: OpenMathDevice := OMconnOutDevice(conn)
   OMsetEncoding(dev, OMencodingXML);
   --retractable?(value)$AnyFunctions1(Expression Integer) =>
   --  OMwrite(dev, retract(value)$AnyFunctions1(Expression Integer), true)
   retractable?(value)$AnyFunctions1(Integer) =>
     OMwrite(dev, retract(value)$AnyFunctions1(Integer), true)
   retractable?(value)$AnyFunctions1(Float) =>
     OMwrite(dev, retract(value)$AnyFunctions1(Float), true)
   retractable?(value)$AnyFunctions1(SingleInteger) =>
     OMwrite(dev, retract(value)$AnyFunctions1(SingleInteger), true)
   retractable?(value)$AnyFunctions1(DoubleFloat) =>
     OMwrite(dev, retract(value)$AnyFunctions1(DoubleFloat), true)
   retractable?(value)$AnyFunctions1(String) =>
     OMwrite(dev, retract(value)$AnyFunctions1(String), true)

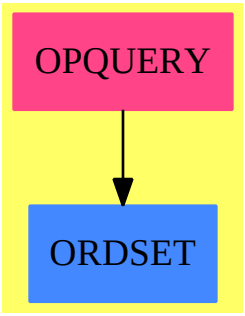
 OMserve(portNum: SingleInteger, timeout: SingleInteger): Void ==
   conn: OpenMathConnection := OMmakeConn(timeout)
   OMbindTCP(conn, portNum)
   val: Any
   while true repeat
     val := OMreceive(conn)
     OMsend(conn, val)
```

— OMSERVER.dotabb —

```
"OMSERVER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMSERVER"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMSERVER" -> "STRING"
```

16.8 package OPQUERY OperationsQuery

16.8.1 OperationsQuery (OPQUERY)



Exports:
getDatabase

— **package OPQUERY OperationsQuery** —

```
)abbrev package OPQUERY OperationsQuery
++ Description:
++ This package exports tools to create AXIOM Library information databases.

OperationsQuery(): Exports == Implementation where
  Exports == with
    getDatabase: String -> Database(IndexCard)
      ++ getDatabase("char") returns a list of appropriate entries in the
      ++ browser database.  The legal values for "char" are "o" (operations),
      ++ "k" (constructors), "d" (domains), "c" (categories) or "p" (packages).
  Implementation == add
    getDatabase(s) == getBrowseDatabase(s)$Lisp
```

———-

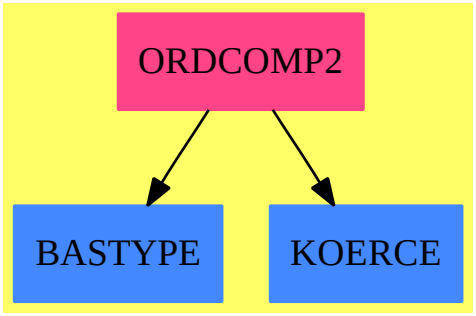
— **OPQUERY.dotabb** —

```
"OPQUERY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OPQUERY"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"OPQUERY" -> "ORDSET"
```

———-

16.9 package ORDCOMP2 OrderedCompletionFunctions2

16.9.1 OrderedCompletionFunctions2 (ORDCOMP2)



Exports:
map

— **package ORDCOMP2 OrderedCompletionFunctions2** —

```
)abbrev package ORDCOMP2 OrderedCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to ordered completions.

OrderedCompletionFunctions2(R, S): Exports == Implementation where
  R, S: SetCategory

  ORR ==> OrderedCompletion R
  ORS ==> OrderedCompletion S

  Exports ==> with
    map: (R -> S, ORR) -> ORS
      ++ map(f, r) lifts f and applies it to r, assuming that
      ++ f(plusInfinity) = plusInfinity and that
      ++ f(minusInfinity) = minusInfinity.
    map: (R -> S, ORR, ORS, ORS) -> ORS
      ++ map(f, r, p, m) lifts f and applies it to r, assuming that
      ++ f(plusInfinity) = p and that f(minusInfinity) = m.

  Implementation ==> add
    map(f, r) == map(f, r, plusInfinity(), minusInfinity())

    map(f, r, p, m) ==
      zero?(n := whatInfinity r) => (f retract r)::ORS
```

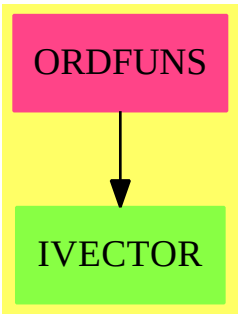
```
--       one? n => p
       (n = 1) => p
       m
```

— ORDCOMP2.dotabb —

```
"ORDCOMP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORDCOMP2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ORDCOMP2" -> "BASTYPE"
"ORDCOMP2" -> "KOERCE"
```

16.10 package ORDFUNS OrderingFunctions

16.10.1 OrderingFunctions (ORDFUNS)



Exports:
pureLex reverseLex totalLex

— package ORDFUNS OrderingFunctions —

```
)abbrev package ORDFUNS OrderingFunctions
++ Author: Barry Trager
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: OrderedDirectProduct
++ Also See:
```

```
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides ordering functions on vectors which
++ are suitable parameters for OrderedDirectProduct.

OrderingFunctions(dim,S) : T == C  where
  dim : NonNegativeInteger
  S         : OrderedAbelianMonoid
  VS       == Vector S

  T ==  with
     pureLex    :  (VS,VS)  -> Boolean
       ++ pureLex(v1,v2) return true if the vector v1 is less than the
       ++ vector v2 in the lexicographic ordering.
     totalLex   :  (VS,VS)  -> Boolean
       ++ totalLex(v1,v2) return true if the vector v1 is less than the
       ++ vector v2 in the ordering which is total degree refined by
       ++ lexicographic ordering.
     reverseLex :  (VS,VS)  -> Boolean
       ++ reverseLex(v1,v2) return true if the vector v1 is less than the
       ++ vector v2 in the ordering which is total degree refined by
       ++ the reverse lexicographic ordering.

  C == add
    n:NonNegativeInteger:=dim

 -- pure lexicographical ordering
    pureLex(v1:VS,v2:VS) : Boolean ==
      for i in 1..n repeat
        if qelt(v1,i) < qelt(v2,i) then return true
        if qelt(v2,i) < qelt(v1,i) then return false
      false

 -- total ordering refined with lex
    totalLex(v1:VS,v2:VS) :Boolean ==
      n1:S:=0
      n2:S:=0
      for i in 1..n repeat
        n1:= n1+qelt(v1,i)
        n2:=n2+qelt(v2,i)
      n1<n2 => true
      n2<n1 => false
      for i in 1..n repeat
        if qelt(v1,i) < qelt(v2,i) then return true
        if qelt(v2,i) < qelt(v1,i) then return false
      false

 -- reverse lexicographical ordering
```

```
    reverseLex(v1:VS,v2:VS) :Boolean ==
      n1:S:=0
      n2:S:=0
      for i in 1..n repeat
        n1:= n1+qelt(v1,i)
        n2:=n2+qelt(v2,i)
      n1<n2 => true
      n2<n1 => false
      for i in reverse(1..n) repeat
        if qelt(v2,i) < qelt(v1,i) then return true
        if qelt(v1,i) < qelt(v2,i) then return false
      false
```

———-

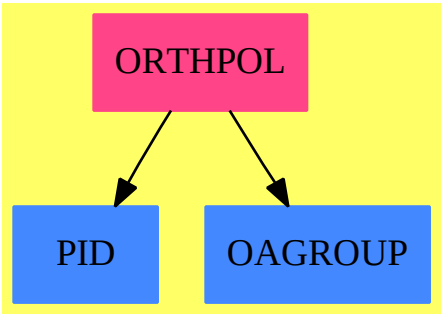
— **ORDFUNS.dotabb** —

```
"ORDFUNS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORDFUNS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ORDFUNS" -> "IVECTOR"
```

———-

16.11 package ORTHPOL OrthogonalPolynomialFunctions

16.11.1 OrthogonalPolynomialFunctions (ORTHPOL)



Exports:
chebyshevT chebyshevU hermiteH laguerreL laguerreL legendreP

— **package ORTHPOL OrthogonalPolynomialFunctions** —

```
)abbrev package ORTHPOL OrthogonalPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created:  1990
++ Date Last Updated: June 25, 1991
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides orthogonal polynomials as functions on a ring.

OrthogonalPolynomialFunctions(R: CommutativeRing): Exports == Impl where
    NNI ==> NonNegativeInteger
    RN  ==> Fraction Integer

    Exports ==> with

        chebyshevT: (NNI, R) -> R
           ++ chebyshevT(n,x) is the n-th Chebyshev polynomial of the first
           ++ kind, \spad{T[n](x)}.  These are defined by
           ++ \spad{(1-t*x)/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}.

        chebyshevU: (NNI, R) -> R
           ++ chebyshevU(n,x) is the n-th Chebyshev polynomial of the second
           ++ kind, \spad{U[n](x)}. These are defined by
           ++ \spad{1/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}.

        hermiteH:   (NNI, R) -> R
           ++ hermiteH(n,x) is the n-th Hermite polynomial, \spad{H[n](x)}.
           ++ These are defined by
           ++ \spad{exp(2*t*x-t**2) = sum(H[n](x)*t**n/n!, n = 0..)}.

        laguerreL:  (NNI, R) -> R
           ++ laguerreL(n,x) is the n-th Laguerre polynomial, \spad{L[n](x)}.
           ++ These are defined by
           ++ \spad{exp(-t*x/(1-t))/(1-t) = sum(L[n](x)*t**n/n!, n = 0..)}.

        laguerreL:  (NNI, NNI, R) -> R
           ++ laguerreL(m,n,x) is the associated Laguerre polynomial,
           ++ \spad{L<m>[n](x)}.  This is the m-th derivative of \spad{L[n](x)}.

        if R has Algebra RN then
            legendreP:  (NNI, R) -> R
                ++ legendreP(n,x) is the n-th Legendre polynomial,
                ++ \spad{P[n](x)}.  These are defined by
                ++ \spad{1/sqrt(1-2*x*t+t**2) = sum(P[n](x)*t**n, n = 0..)}.
```

```
Impl ==> add
    p0, p1: R
    cx:     Integer

    import IntegerCombinatoricFunctions()

    laguerreL(n, x) ==
        n = 0 => 1
        (p1, p0) := (-x + 1, 1)
        for i in 1..n-1 repeat
            (p1, p0) := ((2*i::R + 1 - x)*p1 - i**2*p0, p1)
        p1
    laguerreL(m, n, x) ==
        ni := n::Integer
        mi := m::Integer
        cx := (-1)**m * binomial(ni,ni-mi) * factorial(ni)
        p0 := 1
        p1 := cx::R
        for j in 1..ni-mi repeat
            cx := -cx*(ni-mi-j+1)
            cx := (cx exquo ((mi+j)*j))::Integer
            p0 := p0 * x
            p1 := p1 + cx*p0
        p1
    chebyshevT(n, x) ==
        n = 0 => 1
        (p1, p0) := (x, 1)
        for i in 1..n-1 repeat
            (p1, p0) := (2*x*p1 - p0, p1)
        p1
    chebyshevU(n, x) ==
        n = 0 => 1
        (p1, p0) := (2*x, 1)
        for i in 1..n-1 repeat
            (p1, p0) := (2*x*p1 - p0, p1)
        p1
    hermiteH(n, x) ==
        n = 0 => 1
        (p1, p0) := (2*x, 1)
        for i in 1..n-1 repeat
            (p1, p0) := (2*x*p1 - 2*i*p0, p1)
        p1
    if R has Algebra RN then
        legendreP(n, x) ==
            n = 0 => 1
            p0 := 1
            p1 := x
            for i in 1..n-1 repeat
                c: RN := 1/(i+1)
                (p1, p0) := (c*((2*i+1)*x*p1 - i*p0), p1)
```

```
p1
```

———-

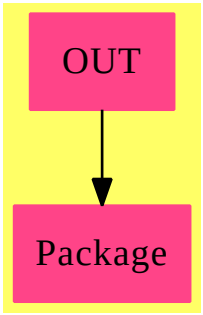
— **ORTHPOL.dotabb** —

```
"ORTHPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORTHPOL"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ORTHPOL" -> "PID"
"ORTHPOL" -> "OAGROUP"
```

———-

16.12 package OUT OutputPackage

16.12.1 OutputPackage (OUT)



Exports:
output outputList

— **package OUT OutputPackage** —

```
)abbrev package OUT OutputPackage
++ Author: Stephen M. Watt
++ Date Created: February 1986
++ Date Last Updated: October 27 1995 (MCD)
++ Basic Operations: output
++ Related Constructors: OutputForm
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ OutPackage allows pretty-printing from programs.

OutputPackage: with
        output: String -> Void
            ++ output(s) displays the string s on the ‘‘algebra output’’
            ++ stream, as defined by \spadsyscom{set output algebra}.
        output: OutputForm -> Void
            ++ output(x) displays the output form x on the
            ++ ‘‘algebra output’’ stream, as defined by
            ++ \spadsyscom{set output algebra}.
        output: (String, OutputForm) -> Void
            ++ output(s,x) displays the string s followed by the form x
            ++ on the ‘‘algebra output’’ stream, as defined by
            ++ \spadsyscom{set output algebra}.
        outputList: (List Any) -> Void
            ++ outputList(l) displays the concatenated components of the
            ++ list l on the ‘‘algebra output’’ stream, as defined by
            ++ \spadsyscom{set output algebra}; quotes are stripped
            ++ from strings.

    == add
        --ExpressionPackage()
        E      ==> OutputForm
        putout ==> mathprint$Lisp

        s: String
        e: OutputForm
        l: List Any

        output e ==
            mathprint(e)$Lisp
            void()
        output s ==
            output(s:E)
        output(s,e) ==
            output blankSeparate [s:E, e]
        outputList(l) ==                                -- MGR
          output hconcat
            [if retractable?(x)$AnyFunctions1(String) then
                message(retract(x)$AnyFunctions1(String))$OutputForm
              else
                x::OutputForm
             for x in l]


            _______-
```

— OUT.dotabb —

```
"OUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OUT"]
"Package" [color="#FF4488"]
"OUT" -> "Package"
```

———-

Chapter 17

Chapter P

17.1 package PAFF PackageForAlgebraicFunctionField

— PackageForAlgebraicFunctionField.input —

```
)set break resume
)sys rm -f PackageForAlgebraicFunctionField.output
)spool PackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForAlgebraicFunctionField
--R PackageForAlgebraicFunctionField(K: Field,symb: List Symbol,BLMET: BlowUpMethodCategory)  is a package c
--R Abbreviation for PackageForAlgebraicFunctionField is PAFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFF
--R
--R------------------------------ Operations --------------------------------
--R fullDesTree : () -> Void              fullInfClsPt : () -> Void
--R genus : () -> NonNegativeInteger      genusNeg : () -> Integer
--R LPolynomial : () -> SparseUnivariatePolynomial Integer if K has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial Integer if K has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R adjunctionDivisor : () -> Divisor Places K
--R classNumber : () -> Integer if K has FINITE
--R desingTree : () -> List DesingTree InfClsPt(K,symb,BLMET)
--R desingTreeWoFullParam : () -> List DesingTree InfClsPt(K,symb,BLMET)
--R eval : (DistributedMultivariatePolynomial(symb,K),Places K) -> K
--R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),Places K) ->
```

```
--R eval : (Fraction DistributedMultivariatePolynomial(symb,K),Places K) -> K
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),Places K) -> Union(K,"failed")
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(sym
--R evalIfCan : (Fraction DistributedMultivariatePolynomial(symb,K),Places K) -> Union(K,"failed
--R findOrderOfDivisor : (Divisor Places K,Integer,Integer) -> Record(ord: Integer,num: Distribu
--R goppaCode : (Divisor Places K,Divisor Places K) -> Matrix K
--R goppaCode : (Divisor Places K,List Places K) -> Matrix K
--R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariateP
--R interpolateForms : (Divisor Places K,NonNegativeInteger) -> List DistributedMultivariatePoly
--R interpolateFormsForFact : (Divisor Places K,List DistributedMultivariatePolynomial(symb,K))
--R intersectionDivisor : DistributedMultivariatePolynomial(symb,K) -> Divisor Places K
--R lBasis : (Divisor Places K,NonNegativeInteger) -> List Fraction DistributedMultivariatePolyn
--R lBasis : Divisor Places K -> Record(num: List DistributedMultivariatePolynomial(symb,K),den:
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R parametrize : (DistributedMultivariatePolynomial(symb,K),Places K) -> NeitherSparseOrDensePo
--R placesAbove : ProjectivePlane K -> List Places K
--R placesOfDegree : PositiveInteger -> List Places K if K has FINITE
--R pointDominateBy : Places K -> ProjectivePlane K
--R projectivePoint : List K -> ProjectivePlane K
--R rationalPlaces : () -> List Places K
--R rationalPoints : () -> List ProjectivePlane K
--R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(sy
--R setSingularPoints : List ProjectivePlane K -> List ProjectivePlane K
--R singularPoints : () -> List ProjectivePlane K
--R theCurve : () -> DistributedMultivariatePolynomial(symb,K)
--R
--E 1

)spool
)lisp (bye)
```

———-

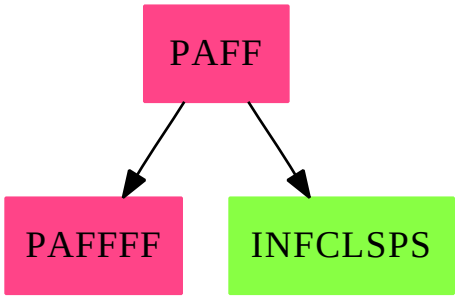
— PackageForAlgebraicFunctionField.help —

```
=====================================================================
PackageForAlgebraicFunctionField examples
=====================================================================

See Also:
o )show PackageForAlgebraicFunctionField
```

———-

17.1.1 PackageForAlgebraicFunctionField (PAFF)



Exports:

adjunctionDivisor	classNumber	desingTree
desingTreeWoFullParam	eval	evalIfCan
findOrderOfDivisor	fullDesTree	fullInfClsPt
genus	genusNeg	goppaCode
homogenize	interpolateForms	interpolateFormsForFact
intersectionDivisor	lBasis	LPolynomial
numberOfPlacesOfDegree	numberPlacesDegExtDeg	numberRatPlacesExtDeg
parametrize	placesAbove	placesOfDegree
pointDominateBy	projectivePoint	rationalPlaces
rationalPoints	setCurve	setSingularPoints
singularPoints	theCurve	ZetaFunction

— **package PAFF PackageForAlgebraicFunctionField** —

```
)abbrev package PAFF PackageForAlgebraicFunctionField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that implements the Brill-Noether algorithm.
++ Part of the PAFF package
PackageForAlgebraicFunctionField(K,symb,BLMET):Exports == Implementation where
  K:Field
  symb :  List(Symbol)
  PolyRing ==> DistributedMultivariatePolynomial(symb,K)
  E  ==>  DirectProduct(#symb,NonNegativeInteger)
  BLMET : BlowUpMethodCategory

  AFP         ==> AffinePlane(K)
  ProjPt      ==> ProjectivePlane(K)
  PCS         ==> NeitherSparseOrDensePowerSeries(K)
  Plc         ==> Places(K)
  DIVISOR     ==> Divisor(Plc)
```

```
InfClsPoint ==> InfClsPt(K,symb,BLMET)
DesTree     ==> DesingTree(InfClsPoint)
FRACPOLY    ==> Fraction PolyRing
NNI         ==> NonNegativeInteger
PI          ==> PositiveInteger
UTSZ        ==> UnivariateTaylorSeriesCZero(Integer,t)
PAFFPC      ==> GeneralPackageForAlgebraicFunctionField
PACKPOLY    ==> PackageForPoly(K,PolyRing,E,#symb)
BP ==> PAFFPC(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

Exports ==>  with
  homogenize: (PolyRing,Integer) -> PolyRing

  interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

  fullDesTree: () -> Void

  fullInfClsPt: () -> Void

  setCurve: PolyRing -> PolyRing

  projectivePoint: List K -> ProjPt

  pointDominateBy : Plc -> ProjPt
    ++ pointDominateBy(pl) returns the projective point dominated
    ++ by the place pl.

  placesAbove: ProjPt -> List Plc

  setSingularPoints: List ProjPt -> List ProjPt

  goppaCode: (DIVISOR,DIVISOR) -> Matrix K

  goppaCode: (DIVISOR,List(Plc)) -> Matrix K

  rationalPlaces: () -> List Plc
    ++ rationalPlaces returns all the rational places of  the
    ++ curve defined by the polynomial given to the package.

  theCurve: () -> PolyRing
    ++ theCurve returns the specified polynomial for the package.

  genus: () -> NNI
    ++ genus returns the genus of the curve defined by the polynomial
    ++ given to the package.

  genusNeg: () -> Integer

  desingTreeWoFullParam  : () -> List DesTree
    ++ desingTreeWoFullParam returns the desingularisation trees at all
```

```
    ++ singular points of the curve defined by the polynomial given to
    ++ the package. The local parametrizations are not computed.

  desingTree: () -> List DesTree
    ++ desingTree returns the desingularisation trees at all singular
    ++ points of the curve defined by the polynomial given to the package.

  rationalPoints: () -> List(ProjPt)

  singularPoints: () -> List(ProjPt)
    ++ rationalPoints() returns the singular points of the
    ++ curve defined by the polynomial given to the package.
    ++ If the singular points lie in an extension of the specified
    ++ ground field an error message is issued specifying the extension
    ++ degree needed to find all singular points.

  parametrize: (PolyRing,Plc) -> PCS
   ++ parametrize(f,pl) returns a local parametrization of f at the place pl.

  lBasis: (DIVISOR,NNI) -> List FRACPOLY

  lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
    ++ lBasis computes a basis associated to the specified divisor

  findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _
    Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)

  interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
    ++ interpolateForms(d,n) returns a basis of the interpolate forms of
    ++ degree n of the divisor d.

  eval: (PolyRing,Plc) -> K
    ++ eval(f,pl) evaluate f at the place pl.

  eval: (PolyRing,PolyRing,Plc) -> K
    ++ eval(f,g,pl) evaluate the function f/g at the place pl.

  eval: (FRACPOLY,Plc) -> K
    ++ eval(u,pl) evaluate the function u at the place pl.

  evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
    ++ evalIfCan(f,pl) evaluate f at the place pl
    ++ (returns "failed" if it is a pole).

  evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
    ++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
    ++ (returns "failed" if it is a pole).

  evalIfCan: (FRACPOLY,Plc) -> Union(K,"failed")
    ++ evalIfCan(u,pl) evaluate the function u at the place pl
```

```
      ++ (returns "failed" if it is a pole).

    intersectionDivisor: PolyRing -> DIVISOR
      ++ intersectionDivisor(pol) compute the intersection divisor (the
      ++ Cartier divisor) of the form pol with the curve. If some
      ++ intersection points lie in an extension of the ground field,
      ++ an error message is issued specifying the extension degree
      ++ needed to find all the intersection points.
      ++ (If pol is not homogeneous an error message is issued).

    adjunctionDivisor: () -> DIVISOR
      ++ adjunctionDivisor computes the adjunction divisor of the plane
      ++ curve given by the polynomial set with the function setCurve.

    if K has Finite then --should we say LocallyAlgebraicallyClosedField??

      LPolynomial: () -> SparseUnivariatePolynomial Integer
        ++ Returns the L-Polynomial of the curve.

      LPolynomial: PI -> SparseUnivariatePolynomial Integer
        ++ LPolynomial(d) returns the L-Polynomial of the curve in
        ++ constant field extension of degree d.

      classNumber: () -> Integer
        ++ Returns the class number of the curve.

      placesOfDegree: PI -> List Plc
        ++ placesOfDegree(d) returns all places of degree d of the curve.

      numberOfPlacesOfDegree: PI -> Integer
        ++ returns the number of places of the given degree

      numberRatPlacesExtDeg: PI -> Integer
        ++ numberRatPlacesExtDeg(n) returns the number of rational
        ++ places in the constant field extenstion of degree n

      numberPlacesDegExtDeg: (PI, PI) -> Integer
        ++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
        ++ places of degree d in the constant field extension of degree n

      ZetaFunction: () -> UTSZ
        ++ Returns the Zeta function of the curve. Calculated by
        ++ using the L-Polynomial

      ZetaFunction: PI -> UTSZ
        ++ Returns the Zeta function of the curve in constant field
        ++ extension. Calculated by using the L-Polynomial

  Implementation ==>  add
    import BP
```

```
homogenize(pol,n) == homogenize(pol,n)$PACKPOLY

pointDominateBy(pl)== pointDominateBy(pl)$BP

placesAbove(pt)== placesAbove(pt)$BP

setSingularPoints(lspt)==    setSingularPoints(lspt)$BP

projectivePoint(lpt)==projectivePoint(lpt)$ProjPt

interpolateFormsForFact(d,lm)==
  interpolateFormsForFact(d,lm)$BP

if K has Finite then

  goppaCode(d:DIVISOR,lp:List(Plc))==
    lb:=lBasis(d)
    dd:=lb.den
    ll:=[[eval(f,dd,pl) for pl in lp] for f in lb.num]
    matrix ll

  goppaCode(d:DIVISOR,p:DIVISOR)==
    lp:=supp p
    goppaCode(d,lp)

  ZetaFunction == ZetaFunction()$BP

  ZetaFunction(d) == ZetaFunction(d)$BP

  numberOfPlacesOfDegree(i)==numberOfPlacesOfDegree(i)$BP

  placesOfDegree(i) ==placesOfDegree(i)$BP

  numberRatPlacesExtDeg(extDegree)==numberRatPlacesExtDeg(extDegree)$BP

  numberPlacesDegExtDeg(degree,extDegree)==
    numberPlacesDegExtDeg(degree,extDegree)$BP

  LPolynomial == LPolynomial()$BP

  LPolynomial(extDeg)==LPolynomial(extDeg)$BP

  classNumber== classNumber()$BP

  rationalPlaces == rationalPlaces()$BP

  rationalPoints==rationalPoints()$BP

crvLocal:PolyRing
```

```
eval(f:PolyRing,pl:Plc)==
  dd:= degree pl
  ^one?(dd) => error " cannot evaluate at place of degree greater than one"
  eval(f,pl)$BP

evalIfCan(f:PolyRing,pl:Plc)==
  dd:= degree pl
  ^one?(dd) => error " cannot evaluate at place of degree greater than one"
  evalIfCan(f,pl)$BP

setCurve(pol)==setCurve(pol)$BP

lBasis(divis)==lBasis(divis)$BP

genus==genus()$BP

genusNeg==genusNeg()$BP

theCurve==theCurve()$BP

desingTree==desingTree()$BP

desingTreeWoFullParam== desingTreeWoFullParam()$BP

-- compute the adjunction divisor of the curve using
-- adjunctionDivisor from DesingTreePackage
adjunctionDivisor == adjunctionDivisor()$BP

singularPoints==singularPoints()$BP

parametrize(f,pl)==parametrize(f,pl)$BP

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==interpolateForms(d,n)$BP

eval(f:PolyRing,g:PolyRing,pl:Plc)==eval(f,g,pl)$BP

eval(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
  eval(ff,gg,pl)

evalIfCan(f:PolyRing,g:PolyRing,pl:Plc)==evalIfCan(f,g,pl)$BP

evalIfCan(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
  evalIfCan(ff,gg,pl)
```

```
    intersectionDivisor(pol)==intersectionDivisor(pol)$BP

    fullDesTree==
      fullOutput()$DesTree => fullOutput(false())$DesTree
      fullOutput(true())$DesTree

    fullInfClsPt==
      fullOutput()$InfClsPoint => fullOutput(false())$InfClsPoint
      fullOutput(true())$InfClsPoint
```

———-

— **PAFF.dotabb** —

```
"PAFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PAFF"]
"PAFFFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PAFFFF"]
"INFCLSPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFCLSPS"]
"PAFF" -> "PAFFFF"
"PAFF" -> "INFCLSPS"
```

———-

17.2 package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField

— **PackageForAlgebraicFunctionFieldOverFiniteField.input** —

```
)set break resume
)sys rm -f PackageForAlgebraicFunctionFieldOverFiniteField.output
)spool PackageForAlgebraicFunctionFieldOverFiniteField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForAlgebraicFunctionFieldOverFiniteField
--R PackageForAlgebraicFunctionFieldOverFiniteField(K: FiniteFieldCategory,symb: List Symbol,BLMET: BlowUpM
--R Abbreviation for PackageForAlgebraicFunctionFieldOverFiniteField is PAFFFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFFFF
--R
--R------------------------------ Operations --------------------------------
--R fullDesTree : () -> Void              fullInfClsPt : () -> Void
```

```
--R genus : () -> NonNegativeInteger      genusNeg : () -> Integer
--R LPolynomial : () -> SparseUnivariatePolynomial Integer if PseudoAlgebraicClosureOfFiniteFiel
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial Integer if PseudoAlgebraicClosur
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicClosureOfFinit
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicC
--R adjunctionDivisor : () -> Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K
--R classNumber : () -> Integer if PseudoAlgebraicClosureOfFiniteField K has FINITE
--R desingTree : () -> List DesingTree InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteFiel
--R desingTreeWoFullParam : () -> List DesingTree InfinitlyClosePointOverPseudoAlgebraicClosureO
--R eval : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteFi
--R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),
--R eval : (Fraction DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureO
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFin
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(sym
--R evalIfCan : (Fraction DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClo
--R findOrderOfDivisor : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,Integer,Intege
--R goppaCode : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,Divisor PlacesOverPseud
--R goppaCode : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,List PlacesOverPseudoAl
--R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariateP
--R interpolateForms : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,NonNegativeInteg
--R interpolateFormsForFact : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,List Dist
--R intersectionDivisor : DistributedMultivariatePolynomial(symb,K) -> Divisor PlacesOverPseudoA
--R lBasis : (Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K,NonNegativeInteger) -> Lis
--R lBasis : Divisor PlacesOverPseudoAlgebraicClosureOfFiniteField K -> Record(num: List Distrib
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField K
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if PseudoAlgebraicClosu
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField K
--R parametrize : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfF
--R placesAbove : ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField K -> List PlacesOverPse
--R placesOfDegree : PositiveInteger -> List PlacesOverPseudoAlgebraicClosureOfFiniteField K if
--R pointDominateBy : PlacesOverPseudoAlgebraicClosureOfFiniteField K -> ProjectivePlaneOverPseu
--R projectivePoint : List PseudoAlgebraicClosureOfFiniteField K -> ProjectivePlaneOverPseudoAlg
--R rationalPlaces : () -> List PlacesOverPseudoAlgebraicClosureOfFiniteField K
--R rationalPoints : () -> List ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField K
--R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(sy
--R setSingularPoints : List ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField K -> List Pr
--R singularPoints : () -> List ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField K
--R theCurve : () -> DistributedMultivariatePolynomial(symb,K)
--R translateToOrigin : (DistributedMultivariatePolynomial(symb,K),ProjectivePlaneOverPseudoAlge
--R
--E 1

)spool
)lisp (bye)
```

———-

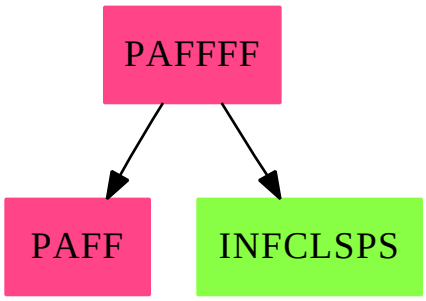
— PackageForAlgebraicFunctionFieldOverFiniteField.help —

```
=======================================================================
PackageForAlgebraicFunctionFieldOverFiniteField examples
=======================================================================

See Also:
o )show PackageForAlgebraicFunctionFieldOverFiniteField
```

———-

17.2.1 PackageForAlgebraicFunctionFieldOverFiniteField (PAFFFF)



Exports:

adjunctionDivisor	classNumber	desingTree
desingTreeWoFullParam	eval	evalIfCan
findOrderOfDivisor	fullDesTree	fullInfClsPt
genus	genusNeg	goppaCode
homogenize	interpolateForms	interpolateFormsForFact
intersectionDivisor	lBasis	LPolynomial
numberOfPlacesOfDegree	numberPlacesDegExtDeg	numberRatPlacesExtDeg
parametrize	placesAbove	placesOfDegree
pointDominateBy	projectivePoint	rationalPlaces
rationalPoints	setCurve	setSingularPoints
singularPoints	theCurve	translateToOrigin
ZetaFunction		

— **package PAFFFF PackageForAlgebraicFunctionFieldOverFinite-Field** —

```
)abbrev package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
```

```
++ A package that implements the Brill-Noether algorithm.
++ Part of the PAFF package
PackageForAlgebraicFunctionFieldOverFiniteField(K,symb,BLMET):Exp == Impl where

  K:FiniteFieldCategory -- Field
  symb :  List(Symbol)
  BLMET : BlowUpMethodCategory

  DK          ==> PseudoAlgebraicClosureOfFiniteField(K)
  PolyRing    ==> DistributedMultivariatePolynomial(symb,K)
  PolyRing2   ==> DistributedMultivariatePolynomial(symb,DK)
  ProjPt      ==> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
  NNI         ==> NonNegativeInteger
  PI          ==> PositiveInteger
  UTSZ        ==> UnivariateTaylorSeriesCZero(Integer,t)
  PAFFPC      ==> GeneralPackageForAlgebraicFunctionField
  PCS         ==> NeitherSparseOrDensePowerSeries(DK)
  Plc         ==> PlacesOverPseudoAlgebraicClosureOfFiniteField(K)
  DIVISOR     ==> Divisor(Plc)
  InfClsPoint ==> InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)
  DesTree     ==> DesingTree(InfClsPoint)
  FracPoly    ==> Fraction PolyRing
  PackPoly    ==> PackageForPoly(DK,PolyRing2,E,#symb)
  E           ==> DirectProduct(#symb,NNI)
  BP ==> PAFFPC(DK,symb,PolyRing2,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

  Exp ==  with
    homogenize: (PolyRing,Integer) -> PolyRing

    fullDesTree: () -> Void

    fullInfClsPt: () -> Void

    setCurve: PolyRing -> PolyRing

    translateToOrigin: (PolyRing, ProjPt) -> PolyRing2

    goppaCode: (DIVISOR,DIVISOR) -> Matrix K

    goppaCode: (DIVISOR,List(Plc)) -> Matrix K

    pointDominateBy : Plc -> ProjPt
      ++ pointDominateBy(pl) returns the projective point dominated
      ++ by the place pl.

    placesAbove: ProjPt -> List Plc

    projectivePoint: List DK -> ProjPt

    setSingularPoints: List ProjPt -> List ProjPt
```

```
rationalPlaces: () -> List Plc
  ++ rationalPlaces returns all the rational places of  the
  ++ curve defined by the polynomial given to the package.

theCurve: () -> PolyRing
  ++ theCurve returns the specified polynomial for the package.

genus: () -> NNI
  ++ genus returns the genus of the curve defined by the polynomial
  ++ given to the package.

genusNeg: () -> Integer

desingTreeWoFullParam  : () -> List DesTree
  ++ desingTreeWoFullParam returns the desingularisation trees at all
  ++ singular points of the curve defined by the polynomial given to
  ++ the package. The local parametrizations are not computed.

desingTree: () -> List DesTree
  ++ desingTree returns the desingularisation trees at all singular points
  ++ of the curve defined by the polynomial given to the package.

rationalPoints: () -> List(ProjPt)

singularPoints: () -> List(ProjPt)
  ++ rationalPoints() returns the singular points of the
  ++ curve defined by the polynomial given to the package.
  ++ If the singular points lie in an extension of the specified
  ++ ground field an error message is issued specifying the extension
  ++ degree needed to find all singular points.

parametrize: (PolyRing,Plc) -> PCS
 ++ parametrize(f,pl) returns a local parametrization of f at the place pl.

lBasis: (DIVISOR,NNI) -> List FracPoly

lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
  ++ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _
  Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)

interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing2)

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
  ++ interpolateForms(d,n) returns a basis of the interpolate forms of
  ++ degree n of the divisor d.

eval: (PolyRing,Plc) -> K
```

```
      ++ eval(f,pl) evaluate f at the place pl.

    eval: (PolyRing,PolyRing,Plc) -> K
      ++ eval(f,g,pl) evaluate the function f/g at the place pl.

    eval: (FracPoly,Plc) -> K
      ++ eval(u,pl) evaluate the function u at the place pl.

    evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
      ++ evalIfCan(f,pl) evaluate f at the place pl
      ++ (returns "failed" if it is a pole).

    evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
      ++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
      ++ (returns "failed" if it is a pole).

    evalIfCan: (FracPoly,Plc) -> Union(K,"failed")
      ++ evalIfCan(u,pl) evaluate the function u at the place pl
      ++ (returns "failed" if it is a pole).

    intersectionDivisor: PolyRing -> DIVISOR
      ++ intersectionDivisor(pol) compute the intersection divisor  of the
      ++ form pol with the curve.
      ++ (If pol is not homogeneous an error message is issued).

    adjunctionDivisor: () -> DIVISOR
      ++ adjunctionDivisor computes the adjunction divisor of the plane
      ++ curve given by the polynomial defined by setCurve.

    if DK has Finite then --should we say LocallyAlgebraicallyClosedField??

      LPolynomial: () -> SparseUnivariatePolynomial Integer
        ++ Returns the L-Polynomial of the curve.

      LPolynomial: PI -> SparseUnivariatePolynomial Integer
        ++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

      classNumber: () -> Integer
        ++ Returns the class number of the curve.

      placesOfDegree: PI -> List Plc
        ++ placesOfDegree(d) returns all places of degree d of the
++ curve.

      numberOfPlacesOfDegree: PI -> Integer
        ++ returns the number of places of the given degree

      numberRatPlacesExtDeg: PI -> Integer
        ++ numberRatPlacesExtDeg(n) returns the number of rational
```

```
++ places in the constant field extenstion of degree n

      numberPlacesDegExtDeg: (PI, PI) -> Integer
        ++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of
++ degree n

      ZetaFunction: () -> UTSZ
        ++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

      ZetaFunction: PI -> UTSZ
        ++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial

  Impl ==>  add
    import BP

    homogenize(pol,n) == homogenize(pol,n)$PackageForPoly(K,PolyRing,E,#symb)

    toPolyRing2: PolyRing -> PolyRing2

    toPolyRing: PolyRing2 -> PolyRing

    projectivePoint(lpt)==projectivePoint(lpt)$ProjPt

    pointDominateBy(pl)== pointDominateBy(pl)$BP

    placesAbove(pt)== placesAbove(pt)$BP

    setSingularPoints(lspt)==    setSingularPoints(lspt)$BP

    findOrderOfDivisor(divis,lb,hb) ==
      ens:=findOrderOfDivisor(divis,lb,hb)$BP
      [ens.ord, toPolyRing ens.num, toPolyRing ens.den, ens.upTo]

    setCurve(pol)==
      ooo:=setCurve(toPolyRing2 pol)$BP
      pol

    ZetaFunction == ZetaFunction()$BP

    ZetaFunction(d) == ZetaFunction(d)$BP

    numberOfPlacesOfDegree(i)==numberOfPlacesOfDegree(i)$BP

    placesOfDegree(i) ==placesOfDegree(i)$BP

    numberRatPlacesExtDeg(extDegree)==numberRatPlacesExtDeg(extDegree)$BP
```

```
numberPlacesDegExtDeg(degree,extDegree)==
  numberPlacesDegExtDeg(degree,extDegree)$BP

LPolynomial == LPolynomial()$BP

LPolynomial(extDeg)==LPolynomial(extDeg)$BP

classNumber== classNumber()$BP

rationalPlaces == rationalPlaces()$BP

rationalPoints==rationalPoints()$BP

goppaCode(d:DIVISOR,lp:List(Plc))==
  lb:=lBasis(d)
  dd:=lb.den
  ll:=[[eval(f,dd,pl) for pl in lp] for f in lb.num]
  matrix ll

goppaCode(d:DIVISOR,p:DIVISOR)==
  lp:=supp p
  goppaCode(d,lp)

toPolyRing(pol)==
  zero?(pol) => 0$PolyRing
  lc:=leadingCoefficient pol
  lce:K:= retract lc
  lm:=leadingMonomial pol
  lt:=degree lm
  monomial(lce,lt)$PolyRing + toPolyRing( reductum pol )

toPolyRing2(pol)==
  zero?(pol) => 0$PolyRing2
  lc:=leadingCoefficient pol
  lce:DK:= lc :: DK
  lm:=leadingMonomial pol
  lt:=degree lm
  monomial(lce,lt)$PolyRing2 + toPolyRing2( reductum pol )

evalIfCan(f:PolyRing,pl:Plc)==
  dd:= degree pl
  ^one?(dd) => error " cannot evaluate at place of degree greater than one"
  ee:=evalIfCan(toPolyRing2 f,pl)$BP
  ee case "failed" => "failed"
  retract ee

eval(f:PolyRing,pl:Plc)==
  dd:= degree pl
  ^one?(dd) => error " cannot evaluate at place of degree greater than one"
  ee:=eval(toPolyRing2 f,pl)$BP
```

```
    retract ee

  lBasis(divis)==
    ans:=lBasis(divis)$BP
    nn:=ans.num
    dd:=ans.den
    nnd:=[toPolyRing pol for pol in nn]
    ddd:=toPolyRing dd
    [nnd,ddd]

  genus==genus()$BP

  genusNeg==genusNeg()$BP

  theCurve==
    ccc:= theCurve()$BP
    toPolyRing ccc

  desingTree==desingTree()$BP

  desingTreeWoFullParam== desingTreeWoFullParam()$BP

  -- compute the adjunction divisor of the curve using
  -- adjunctionDivisor from DesingTreePackage
  adjunctionDivisor == adjunctionDivisor()$BP

  singularPoints==singularPoints()$BP

  parametrize(f,pl)==
    ff:= toPolyRing2 f
    parametrize(ff,pl)$BP

  -- compute the interpolating forms (see package InterpolateFormsPackage)
  interpolateForms(d,n)==
    ans:=interpolateForms(d,n)$BP
    [toPolyRing pol for pol in ans]

  interpolateFormsForFact(d,lm)==
    lm2:List PolyRing2 := [ toPolyRing2 p for p in lm]
    interpolateFormsForFact(d,lm2)$BP

  evalIfCan(ff:PolyRing,gg:PolyRing,pl:Plc)==
    dd:= degree pl
    ^one?(dd) => error " cannot evaluate at place of degree greater than one"
    f:=toPolyRing2 ff
    g:=toPolyRing2 gg
    ee:=evalIfCan(f,g,pl)$BP
    ee case "failed" => "failed"
    retract ee
```

```
    eval(ff:PolyRing,gg:PolyRing,pl:Plc)==
      dd:= degree pl
      ^one?(dd) => error " cannot evaluate at place of degree greater than one"
      f:=toPolyRing2 ff
      g:=toPolyRing2 gg
      ee:=eval(f,g,pl)$BP
      retract ee

    evalIfCan(u:FracPoly,pl:Plc)==
      ff:=numer u
      gg:=denom u
      evalIfCan(ff,gg,pl)

    eval(u:FracPoly,pl:Plc)==
      ff:=numer u
      gg:=denom u
      eval(ff,gg,pl)

    intersectionDivisor(pol)==
      polu:=toPolyRing2 pol
      intersectionDivisor(polu)$BP

    fullDesTree==
      fullOutput()$DesTree => fullOutput(false())$DesTree
      fullOutput(true())$DesTree

    fullInfClsPt==
      fullOutput()$InfClsPoint => fullOutput(false())$InfClsPoint
      fullOutput(true())$InfClsPoint
```

———-

— PAFFFF.dotabb —

```
"PAFFFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PAFFFF"]
"PAFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PAFF"]
"INFCLSPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=INFCLSPS"]
"PAFFFF" -> "PAFF"
"PAFFFF" -> "INFCLSPS"
```

———-

17.3 package PFORP PackageForPoly

— **PackageForPoly.input** —

```
)set break resume
)sys rm -f PackageForPoly.output
)spool PackageForPoly.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForPoly
--R
--R PackageForPoly(R: Ring,PolyRing: FiniteAbelianMonoidRing(R,E),E: DirectProductCategory(dim,NonNegativeI
--R Abbreviation for PackageForPoly is PFORP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFORP
--R
--R------------------------------ Operations --------------------------------
--R constant : PolyRing -> R              firstExponent : PolyRing -> E
--R minimalForm : PolyRing -> PolyRing
--R degOneCoef : (PolyRing,PositiveInteger) -> R
--R degree : (PolyRing,Integer) -> NonNegativeInteger
--R degreeOfMinimalForm : PolyRing -> NonNegativeInteger
--R homogenize : (PolyRing,Integer) -> PolyRing
--R listAllMono : NonNegativeInteger -> List(PolyRing)
--R listAllMonoExp : Integer -> List(E)
--R listVariable : () -> List(PolyRing)
--R mapExponents : ((E -> E),PolyRing) -> PolyRing
--R monomials : PolyRing -> List(PolyRing)
--R replaceVarByOne : (PolyRing,Integer) -> PolyRing
--R replaceVarByZero : (PolyRing,Integer) -> PolyRing
--R subs1stVar : (PolyRing,PolyRing) -> PolyRing
--R subs2ndVar : (PolyRing,PolyRing) -> PolyRing
--R subsInVar : (PolyRing,PolyRing,Integer) -> PolyRing
--R totalDegree : PolyRing -> NonNegativeInteger
--R translate : (PolyRing,List(R),Integer) -> PolyRing
--R translate : (PolyRing,List(R)) -> PolyRing
--R univariate : PolyRing -> SparseUnivariatePolynomial(R)
--R
--E 1

)spool
)lisp (bye)
```

———-

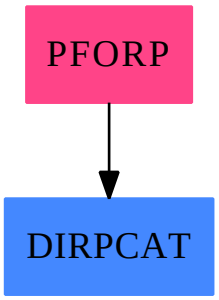
— PackageForPoly.help —

```
=============================================================================
PackageForPoly examples
=============================================================================

See Also:
o )show PackageForPoly
```

———-

17.3.1 PackageForPoly (PFORP)



Exports:

constant	firstExponent	listAllMonoExp
listVariable	minimalForm	degOneCoef
degree	degreeOfMinimalForm	homogenize
listAllMono	mapExponents	monomials
replaceVarByOne	replaceVarByZero	subs1stVar
subs2ndVar	subsInVar	totalDegree
translate	univariate	

— package PFORP PackageForPoly —

```
)abbrev package PFORP PackageForPoly
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PackageForPoly(R,PolyRing,E,dim): public == private where
  R: Ring -- was Field but change for SolveTree package. 21/01/98 )
```

```
dim: NonNegativeInteger
E: DirectProductCategory(dim,NonNegativeInteger)
PolyRing : FiniteAbelianMonoidRing(R,E)

Term ==> Record(k:E,c:R)
PI   ==> PositiveInteger
NNI  ==> NonNegativeInteger
INT  ==> Integer

public == with
    mapExponents: (E->E, PolyRing) -> PolyRing

    degree: (PolyRing , Integer) -> NNI

    univariate: PolyRing -> SparseUnivariatePolynomial(R)

    totalDegree: PolyRing -> NNI

    subs1stVar: (PolyRing , PolyRing) -> PolyRing

    subs2ndVar: (PolyRing , PolyRing) -> PolyRing

    subsInVar: (PolyRing, PolyRing, Integer) -> PolyRing

    minimalForm: PolyRing -> PolyRing
      ++ minimalForm(pol) returns the minimal forms of the polynomial pol.

    firstExponent: PolyRing -> E
      ++ firstExponent(pol) returns the exponent of the first term in the
      ++ representation of pol. Not to be confused with the leadingExponent
      ++  which is the highest exponent according to the order
      ++ over the monomial.

    replaceVarByZero: (PolyRing,Integer) -> PolyRing
      ++ replaceVarByZero(pol,a) evaluate to zero the variable in pol
      ++ specified by the integer a.

    replaceVarByOne: (PolyRing,Integer) -> PolyRing
      ++ replaceVarByOne(pol,a) evaluate to one the variable in pol
      ++ specified by the integer a.

    translate: (PolyRing,List R,Integer) -> PolyRing
      ++ translate(pol,[a,b,c],3) apply to pol the
      ++ linear change of coordinates, x->x+a, y->y+b, z->1.

    translate: (PolyRing,List R) -> PolyRing
      ++ translate(pol,[a,b,c]) apply to pol the
      ++ linear change of coordinates, x->x+a, y->y+b, z->z+c

    degOneCoef: (PolyRing,PI) -> R
```

```
      ++ degOneCoef(pol,n) returns the coefficient in front of the monomial
      ++ specified by the positive integer.

    constant: PolyRing -> R
      ++ constant(pol) returns the constant term of the polynomial.

    homogenize: (PolyRing,INT) -> PolyRing
      ++ homogenize(pol,n) returns the homogenized polynomial of pol
      ++ with respect to the n-th variable.

    listAllMonoExp: Integer -> List E
      ++ listAllMonoExp(l) returns all the exponents of degree l

    listAllMono: NNI -> List PolyRing
      ++ listAllMono(l) returns all the monomials of degree l

    degreeOfMinimalForm: PolyRing -> NNI
      ++ degreeOfMinimalForm does what it says

    listVariable: () -> List PolyRing

    monomials: PolyRing -> List PolyRing


  private == add
    import PolyRing

    monomials(pol)==
      zero? pol => empty()
      lt:=leadingMonomial pol
      cons( lt , monomials reductum pol )

    lll: Integer -> E
    lll(i) ==
      le:=new( dim , 0$NNI)$List(NNI)
      le.i := 1
      directProduct( vector(le)$Vector(NNI) )$E

    listVariable==
      [monomial(1,ee)$PolyRing for ee in [lll(i) for i in 1..dim]]

    univariate(pol)==
      zero? pol => 0
      d:=degree pol
      lc:=leadingCoefficient pol
      td := reduce("+", entries d)
      monomial(lc,td)$SparseUnivariatePolynomial(R)+univariate(reductum pol)

    collectExpon: List Term -> PolyRing
```

```
      translateLocal: (PolyRing,List R,Integer) -> PolyRing

      lA: (Integer,Integer) -> List List NNI

      toListRep: PolyRing -> List Term

      exponentEntryToZero: (E,Integer) -> E

      exponentEntryZero?: (E,Integer) -> Boolean

      homogenizeExp: (E,NNI,INT) -> E

      translateMonomial: (PolyRing,List R,INT,R) -> PolyRing

      leadingTerm: PolyRing -> Term

      mapExponents(f,pol)==
        zero?(pol) => 0
lt:=leadingTerm pol
newExp:E:= f(lt.k)
newMono:PolyRing:= monomial(lt.c,newExp)$PolyRing
newMono + mapExponents(f,reductum pol)

      collectExpon(pol)==
        empty? pol => 0
ft:=first pol
monomial(ft.c,ft.k) + collectExpon( rest pol )

      subs1stVar(pol, spol)==
        zero? pol => 0
        lexpE:E:= degree pol
lexp:List NNI:= parts lexpE
        coef:= leadingCoefficient pol
coef * spol ** lexp.1  * second(listVariable())**lexp.2 _
          + subs1stVar( reductum pol, spol )

      subs2ndVar(pol, spol)==
        zero? pol => 0
        lexpE:E:= degree pol
lexp:List NNI:= parts lexpE
        coef:= leadingCoefficient pol
coef * first(listVariable())**lexp.1   *  spol ** lexp.2 _
          + subs2ndVar( reductum pol, spol )

      subsInVar( pol, spol, n)==
        one?( n ) => subs1stVar( pol, spol)
        subs2ndVar(pol,spol)

      translate(pol,lpt)==
        zero? pol => 0
```

```
        lexpE:E:= degree pol
lexp:List NNI:= parts lexpE
        coef:= leadingCoefficient pol
trVar:=[(listVariable().i + (lpt.i)::PolyRing)**lexp.i for i in 1..dim]
        coef * reduce("*",trVar,1) + translate(reductum pol , lpt)

      translate(poll,lpt,nV)==
        pol:=replaceVarByOne(poll,nV)
        translateLocal(pol,lpt,nV)

      translateLocal(pol,lpt,nV)==
zero?(pol) => 0
        lll:List R:=[l for l in lpt | ^zero?(l)]
        nbOfNonZero:=# lll
ltk:=leadingMonomial pol
ltc:=leadingCoefficient pol
        if one?(nbOfNonZero) then
  pol
        else
  translateMonomial(ltk,lpt,nV,ltc) + _
           translateLocal(reductum(pol),lpt,nV)

      exponentEntryToZero(exp,nV)==
        pexp:= parts exp
pexp(nV):=0
directProduct(vector(pexp)$Vector(NonNegativeInteger))

      exponentEntryZero?(exp,nV)==
        pexp:= parts exp
zero?(pexp(nV))

      replaceVarByZero(pol,nV)==
        -- surement le collectExpon ici n'est pas necessaire  !!!!
        zero?(pol) => 0
        lRep:= toListRep pol
        reduce("+",_
               [monomial(p.c,p.k)$PolyRing _
                 for p in lRep | exponentEntryZero?(p.k,nV) ],0)

      replaceVarByOne(pol,nV)==
        zero?(pol) => 0
        lRep:= toListRep pol
reduce("+",_
         [monomial(p.c,exponentEntryToZero(p.k,nV))$PolyRing for p in lRep],0)

      homogenizeExp(exp,deg,nV)==
        lv:List NNI:=parts(exp)
        lv.nV:=(deg+lv.nV - reduce("+",lv)) pretend NNI
        directProduct(vector(lv)$Vector(NNI))$E
```

```
      listTerm: PolyRing -> List E
      listTerm(pol)==
        zero? pol => empty
        cons( degree pol,  listTerm reductum pol )

      degree( a : PolyRing , n : Integer )==
        zero? a => error "Degree for 0 is not defined for this degree fnc"
        "max" / [ ee.n for ee in listTerm a ]

      totalDegree p ==
         zero? p => 0
         "max"/[reduce("+",t::(Vector NNI), 0) for t in listTerm p]

      homogenize(pol,nV)==
        degP:=totalDegree(pol)
        mapExponents(homogenizeExp(#1,degP,nV),pol)

      degOneCoef(p:PolyRing,i:PI)==
        vv:=new(dim,0)$Vector(NNI)
        vv.i:=1
        pd:=directProduct(vv)$E
lp:=toListRep p
        lc:=[t.c for t in lp | t.k=pd]
        reduce("+",lc,0)

      constant(p)==
        vv:=new(dim,0)$Vector(NNI)
        pd:=directProduct(vv)$E
lp:=toListRep p
        lc:=[t.c for t in lp | t.k=pd]
        reduce("+",lc,0)

      degreeOfMinimalForm(pol)==
        totalDegree minimalForm pol

      minimalForm(pol)==
zero?(pol) => pol
lpol:=toListRep pol
actTerm:Term:=  first lpol
minDeg:NNI:=reduce("+", parts(actTerm.k))
actDeg:NNI
lminForm:List(Term):= [actTerm]
for p in rest(lpol) repeat
  actDeg:= reduce("+", parts(p.k))
  if actDeg = minDeg then
    lminForm := concat(lminForm,p)
  if actDeg < minDeg then
    minDeg:=actDeg
    lminForm:=[p]
collectExpon lminForm
```

```
    -- le code de collectExponSort a ete emprunte a D. Augot.

      leadingTerm(pol)==
        zero?(pol) => error "no leading term for 0  (message from package)"
lcoef:R:=leadingCoefficient(pol)$PolyRing
lterm:PolyRing:=leadingMonomial(pol)$PolyRing
tt:E:=degree(lterm)$PolyRing
[tt,lcoef]$Term

      toListRep(pol)==
        zero?(pol) => empty()
lt:=leadingTerm pol
cons(lt, toListRep reductum pol)

      lA(n,l)==
        zero?(n) => [new((l pretend NNI),0)$List(NNI)]
one?(l) => [[(n pretend NNI)]]
concat [[ concat([i],lll) for lll in lA(n-i,l-1)] for i in 0..n]

      listAllMonoExp(l)==
        lst:=lA(l,(dim pretend Integer))
        [directProduct(vector(pexp)$Vector(NNI)) for pexp in lst]

      translateMonomial(mono,pt,nV,coef)==
        lexpE:E:= degree mono
lexp:List NNI:= parts lexpE
        lexp(nV):=0
trVar:=[(listVariable().i + (pt.i)::PolyRing)** lexp.i for i in 1..dim]
        coef * reduce("*",trVar,1)

      listAllMono(l)==
        [monomial(1,e)$PolyRing for e in listAllMonoExp(l)]
```

———-

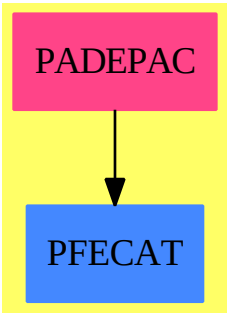
— PFORP.dotabb —

```
"PFORP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFORP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFORP" -> "DIRPCAT"
```

———-

17.4 package PADEPAC PadeApproximantPackage

17.4.1 PadeApproximantPackage (PADEPAC)



Exports:
pade

— package PADEPAC PadeApproximantPackage —

```
)abbrev package PADEPAC PadeApproximantPackage
++ Authors: Trager,Burge, Hassner & Watt.
++ Date Created: April 1987
++ Date Last Updated: 12 April 1990
++ Keywords: Pade, series
++ Examples:
++ References:
++   "Pade Approximants, Part I: Basic Theory", Baker & Graves-Morris.
++ Description:
++ This package computes reliable Pad&ea. approximants using
++ a generalized Viskovatov continued fraction algorithm.

PadeApproximantPackage(R: Field, x:Symbol, pt:R): Exports == Implementation where
   PS ==> UnivariateTaylorSeries(R,x,pt)
   UP ==> UnivariatePolynomial(x,R)
   QF ==> Fraction UP
   CF  ==> ContinuedFraction UP
   NNI ==> NonNegativeInteger
   Exports ==> with
     pade:   (NNI,NNI,PS,PS) -> Union(QF,"failed")
      ++ pade(nd,dd,ns,ds) computes the approximant as a quotient of polynomials
      ++ (if it exists) for arguments
      ++ nd (numerator degree of approximant),
      ++ dd (denominator degree of approximant),
      ++ ns (numerator series of function), and
      ++ ds (denominator series of function).
     pade:   (NNI,NNI,PS) -> Union(QF,"failed")
```

```
      ++ pade(nd,dd,s)
      ++ computes the quotient of polynomials
      ++ (if it exists) with numerator degree at
      ++ most nd and denominator degree at most dd
      ++ which matches the series s to order \spad{nd + dd}.

   Implementation ==> add
     n,m : NNI
     u,v : PS
     pa := PadeApproximants(R,PS,UP)
     pade(n,m,u,v) ==
       ans:=pade(n,m,u,v)$pa
       ans case "failed" => ans
       pt = 0 => ans
       num := numer(ans::QF)
       den := denom(ans::QF)
       xpt : UP := monomial(1,1)-monomial(pt,0)
       num := num(xpt)
       den := den(xpt)
       num/den
     pade(n,m,u) == pade(n,m,u,1)
```

———-

— PADEPAC.dotabb —

```
"PADEPAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADEPAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PADEPAC" -> "PFECAT"
```

———-

17.5 package PADE PadeApproximants

— PadeApproximants.input —

```
)set break resume
)sys rm -f PadeApproximants.output
)spool PadeApproximants.output
)set message test on
)set message auto off
)clear all

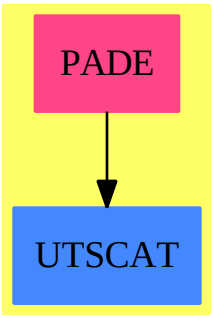
--S 1 of 1
```

```
)show PadeApproximants
--R
--R PadeApproximants(R: Field,PS: UnivariateTaylorSeriesCategory(R),UP: UnivariatePolynomialCategory(R))  is
--R Abbreviation for PadeApproximants is PADE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PADE
--R
--R------------------------------ Operations --------------------------------
--R pade : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(Fraction(UP),"failed")
--R padecf : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(ContinuedFraction(UP),"failed")
--R
--E 1

)spool
)lisp (bye)
```

———-

17.5.1 PadeApproximants (PADE)



Exports:
pade padecf

— **package PADE PadeApproximants** —

```
)abbrev package PADE PadeApproximants
++ Authors: Burge, Hassner & Watt.
++ Date Created: April 1987
++ Date Last Updated: 12 April 1990
++ Keywords: Pade, series
++ Examples:
++ References:
++   "Pade Approximants, Part I: Basic Theory", Baker & Graves-Morris.
++ Description:
```

```
++ This package computes reliable Pad&ea. approximants using
++ a generalized Viskovatov continued fraction algorithm.

PadeApproximants(R,PS,UP): Exports == Implementation where
  R:  Field -- IntegralDomain
  PS: UnivariateTaylorSeriesCategory R
  UP: UnivariatePolynomialCategory R

  NNI ==> NonNegativeInteger
  QF  ==> Fraction UP
  CF  ==> ContinuedFraction UP

  Exports ==> with
    pade:   (NNI,NNI,PS,PS) -> Union(QF,"failed")
      ++ pade(nd,dd,ns,ds)
      ++ computes the approximant as a quotient of polynomials
      ++ (if it exists) for arguments
      ++ nd (numerator degree of approximant),
      ++ dd (denominator degree of approximant),
      ++ ns (numerator series of function), and
      ++ ds (denominator series of function).
    padecf: (NNI,NNI,PS,PS) -> Union(CF, "failed")
      ++ padecf(nd,dd,ns,ds)
      ++ computes the approximant as a continued fraction of
      ++ polynomials (if it exists) for arguments
      ++ nd (numerator degree of approximant),
      ++ dd (denominator degree of approximant),
      ++ ns (numerator series of function), and
      ++ ds (denominator series of function).

  Implementation ==> add
    -- The approximant is represented as
    --   p0 + x**a1/(p1 + x**a2/(...))

    PadeRep ==> Record(ais: List UP, degs: List NNI) -- #ais= #degs
    PadeU   ==> Union(PadeRep, "failed")             -- #ais= #degs+1

    constInner(up:UP):PadeU == [[up], []]

    truncPoly(p:UP,n:NNI):UP ==
      while n < degree p repeat p := reductum p
      p

    truncSeries(s:PS,n:NNI):UP ==
      p: UP := 0
      for i in 0..n repeat p := p + monomial(coefficient(s,i),i)
      p

    -- Assumes s starts with a<n>*x**n + ... and divides out x**n.
    divOutDegree(s:PS,n:NNI):PS ==
```

```
    for i in 1..n repeat s := quoByVar s
    s

  padeNormalize: (NNI,NNI,PS,PS) -> PadeU
  padeInner:     (NNI,NNI,PS,PS) -> PadeU

  pade(l,m,gps,dps) ==
    (ad := padeNormalize(l,m,gps,dps)) case "failed" => "failed"
    plist := ad.ais; dlist := ad.degs
    approx := first(plist) :: QF
    for d in dlist for p in rest plist repeat
      approx := p::QF + (monomial(1,d)$UP :: QF)/approx
    approx

  padecf(l,m,gps,dps) ==
    (ad := padeNormalize(l,m,gps,dps)) case "failed" => "failed"
    alist := reverse(ad.ais)
    blist := [monomial(1,d)$UP for d in reverse ad.degs]
    continuedFraction(first(alist),_
                      blist::Stream UP,(rest alist) :: Stream UP)

  padeNormalize(l,m,gps,dps) ==
    zero? dps => "failed"
    zero? gps => constInner 0
    -- Normalize so numerator or denominator has constant term.
    ldeg:= min(order dps,order gps)
    if ldeg > 0 then
      dps := divOutDegree(dps,ldeg)
      gps := divOutDegree(gps,ldeg)
    padeInner(l,m,gps,dps)

  padeInner(l, m, gps, dps) ==
    zero? coefficient(gps,0) and zero? coefficient(dps,0) =>
      error "Pade' problem not normalized."
    plist: List UP := nil()
    alist: List NNI := nil()
    -- Ensure denom has constant term.
    if zero? coefficient(dps,0) then
      -- g/d = 0 + z**0/(d/g)
      (gps,dps) := (dps,gps)
      (l,m)     := (m,l)
      plist := concat(0,plist)
      alist := concat(0,alist)
    -- Ensure l >= m, maintaining coef(dps,0)^=0.
    if l < m then
      --   (a<n>*x**n + a<n+1>*x**n+1 + ...)/b
      -- = x**n/b + (a<n> + a<n+1>*x + ...)/b
      alpha := order gps
      if alpha > l then return "failed"
      gps := divOutDegree(gps, alpha)
```

```
    (l,m) := (m,(l-alpha) :: NNI)
    (gps,dps) := (dps,gps)
    plist := concat(0,plist)
    alist := concat(alpha,alist)
  degbd: NNI := l + m + 1
  g := truncSeries(gps,degbd)
  d := truncSeries(dps,degbd)
  for j in 0.. repeat
    -- Normalize d so constant coefs cancel. (B&G-M is wrong)
    d0 := coefficient(d,0)
    d := (1/d0) * d; g := (1/d0) * g
    p : UP := 0; s := g
    if l-m+1 < 0 then error "Internal pade error"
    degbd := (l-m+1) :: NNI
    for k in 1..degbd repeat
      pk := coefficient(s,0)
      p  := p + monomial(pk,(k-1) :: NNI)
      s  := s - pk*d
      s  := (s exquo monomial(1,1)) :: UP
    plist := concat(p,plist)
    s = 0 => return [plist,alist]
    alpha := minimumDegree(s) + degbd
    alpha > l + m => return [plist,alist]
    alpha > l     => return "failed"
    alist := concat(alpha,alist)
    h := (s exquo monomial(1,minimumDegree s)) :: UP
    degbd := (l + m - alpha) :: NNI
    g := truncPoly(d,degbd)
    d := truncPoly(h,degbd)
    (l,m) := (m,(l-alpha) :: NNI)
```

———-

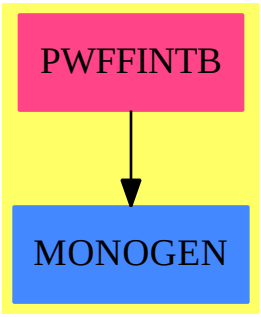
— PADE.dotabb —

```
"PADE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADE"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"PADE" -> "UTSCAT"
```

———-

17.6 package PWFFINTB PAdicWildFunctionFieldIntegralBasis

17.6.1 PAdicWildFunctionFieldIntegralBasis (PWFFINTB)



Exports:
reducedDiscriminant integralBasis localIntegralBasis

— package PWFFINTB PAdicWildFunctionFieldIntegralBasis —

```
)abbrev package PWFFINTB PAdicWildFunctionFieldIntegralBasis
++ Author: Clifton Williamson
++ Date Created: 5 July 1993
++ Date Last Updated: 17 August 1993
++ Basic Operations: integralBasis, localIntegralBasis
++ Related Domains: WildFunctionFieldIntegralBasis(K,R,UP,F)
++ Also See: FunctionFieldIntegralBasis
++ AMS Classifications:
++ Keywords: function field, finite field, integral basis
++ Examples:
++ References:
++ Description:
++ In this package K is a finite field, R is a ring of univariate
++ polynomials over K, and F is a monogenic algebra over R.
++ We require that F is monogenic, i.e. that \spad{F = K[x,y]/(f(x,y))},
++ because the integral basis algorithm used will factor the polynomial
++ \spad{f(x,y)}.  The package provides a function to compute the integral
++ closure of R in the quotient field of F as well as a function to compute
++ a "local integral basis" at a specific prime.

PAdicWildFunctionFieldIntegralBasis(K,R,UP,F): Exports == Implementation where
  K  : FiniteFieldCategory
  R  : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R
  F  : MonogenicAlgebra(R,UP)
```

```
I        ==> Integer
L        ==> List
L2       ==> ListFunctions2
Mat      ==> Matrix R
NNI      ==> NonNegativeInteger
PI       ==> PositiveInteger
Q        ==> Fraction R
SAE      ==> SimpleAlgebraicExtension
SUP      ==> SparseUnivariatePolynomial
CDEN     ==> CommonDenominator
DDFACT   ==> DistinctDegreeFactorize
WFFINTBS ==> WildFunctionFieldIntegralBasis
Result   ==> Record(basis: Mat, basisDen: R, basisInv:Mat)
IResult  ==> Record(basis: Mat, basisDen: R, basisInv:Mat,discr: R)
IBPTOOLS ==> IntegralBasisPolynomialTools
IBACHIN  ==> ChineseRemainderToolsForIntegralBases
IRREDFFX ==> IrredPolyOverFiniteField
GHEN     ==> GeneralHenselPackage

Exports ==> with
  integralBasis : () -> Result
    ++ \spad{integralBasis()} returns a record
    ++ \spad{[basis,basisDen,basisInv] } containing information regarding
    ++ the integral closure of R in the quotient field of the framed
    ++ algebra F.  F is a framed algebra with R-module basis
    ++ \spad{w1,w2,...,wn}.
    ++ If 'basis' is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of 'basis' contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix 'basisInv' contains the coordinates of \spad{wi} with respect
    ++ to the basis \spad{v1,...,vn}: if 'basisInv' is the matrix
    ++ \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.
  localIntegralBasis : R -> Result
    ++ \spad{integralBasis(p)} returns a record
    ++ \spad{[basis,basisDen,basisInv] } containing information regarding
    ++ the local integral closure of R at the prime \spad{p} in the quotient
    ++ field of the framed algebra F.  F is a framed algebra with R-module
    ++ basis \spad{w1,w2,...,wn}.
    ++ If 'basis' is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the local integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of 'basis' contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix 'basisInv' contains the coordinates of \spad{wi} with respect
    ++ to the basis \spad{v1,...,vn}: if 'basisInv' is the matrix
    ++ \spad{(bij, i = 1..n, j = 1..n)}, then
```

```
      ++ \spad{wi = sum(bij * vj, j = 1..n)}.
    reducedDiscriminant: UP -> R
      ++ reducedDiscriminant(up) \undocumented

  Implementation ==> add
    import IntegralBasisTools(R, UP, F)
    import GeneralHenselPackage(R,UP)
    import ModularHermitianRowReduction(R)
    import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

    reducedDiscriminant f ==
      ff : SUP Q := mapUnivariate((r1:R):Q+->r1 :: Q,f)$IBPTOOLS(R,UP,SUP UP,Q)
      ee := extendedEuclidean(ff,differentiate ff)
      cc := concat(coefficients(ee.coef1),coefficients(ee.coef2))
      cden := splitDenominator(cc)$CDEN(R,Q,L Q)
      denom := cden.den
      gg := gcd map(numer,cden.num)$L2(Q,R)
      (ans := denom exquo gg) case "failed" =>
        error "PWFFINTB: error in reduced discriminant computation"
      ans :: R

    compLocalBasis: (UP,R) -> Result
    compLocalBasis(poly,prime) ==
      -- compute a local integral basis at 'prime' for k[x,y]/(poly(x,y)).
      sae := SAE(R,UP,poly)
      localIntegralBasis(prime)$WFFINTBS(K,R,UP,sae)

    compLocalBasisOverExt: (UP,R,UP,NNI) -> Result
    compLocalBasisOverExt(poly0,prime0,irrPoly0,k) ==
      -- poly0 = irrPoly0**k (mod prime0)
      n := degree poly0; disc0 := discriminant poly0
      (disc0 exquo prime0) case "failed" =>
        [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
      r := degree irrPoly0
      -- extend scalars:
      -- construct irreducible polynomial of degree r over K
      irrPoly := generateIrredPoly(r :: PI)$IRREDFFX(K)
      -- construct extension of degree r over K
      E := SAE(K,SUP K,irrPoly)
      -- lift coefficients to elements of E
      poly := mapBivariate((k1:K):E +-> k1::E,poly0)$IBPTOOLS(K,R,UP,E)
      redDisc0 := reducedDiscriminant poly0
      redDisc := mapUnivariate((k1:K):E +-> k1::E,redDisc0)$IBPTOOLS(K,R,UP,E)
      prime := mapUnivariate((k1:K):E +-> k1::E,prime0)$IBPTOOLS(K,R,UP,E)
      sae := SAE(E,SUP E,prime)
      -- reduction (mod prime) of polynomial of which poly is the kth power
      redIrrPoly :=
        pp := mapBivariate((k1:K):E +-> k1::E,irrPoly0)$IBPTOOLS(K,R,UP,E)
        mapUnivariate(reduce,pp)$IBPTOOLS(SUP E,SUP SUP E,SUP SUP SUP E,sae)
      -- factor the reduction
```

```
  factorListSAE := factors factor(redIrrPoly)$DDFACT(sae,SUP sae)
  -- list the 'primary factors' of the reduction of poly
  redFactors : List SUP sae := [(f.factor)**k for f in factorListSAE]
  -- lift these factors to elements of SUP SUP E
  primaries : List SUP SUP E :=
    [mapUnivariate(lift,ff)$IBPTOOLS(SUP E,SUP SUP E,SUP SUP SUP E,sae) _
         for ff in redFactors]
  -- lift the factors to factors modulo a suitable power of 'prime'
  deg := (1 + order(redDisc,prime) * degree(prime)) :: PI
  henselInfo := HenselLift(poly,primaries,prime,deg)$GHEN(SUP E,SUP SUP E)
  henselFactors := henselInfo.plist
  psi1 := first henselFactors
  FF := SAE(SUP E,SUP SUP E,psi1)
  factorIb := localIntegralBasis(prime)$WFFINTBS(E,SUP E,SUP SUP E,FF)
  bs := listConjugateBases(factorIb,size()$K,r)$IBACHIN(E,SUP E,SUP SUP E)
  ib := chineseRemainder(henselFactors,bs,n)$IBACHIN(E,SUP E,SUP SUP E)
  b : Matrix R :=
    bas := mapMatrixIfCan(retractIfCan,ib.basis)$IBPTOOLS(K,R,UP,E)
    bas case "failed" => error "retraction of basis failed"
    bas :: Matrix R
  bInv : Matrix R :=
    --bas := mapMatrixIfCan(ric,ib.basisInv)$IBPTOOLS(K,R,UP,E)
    bas := mapMatrixIfCan(retractIfCan,ib.basisInv)$IBPTOOLS(K,R,UP,E)
    bas case "failed" => error "retraction of basis inverse failed"
    bas :: Matrix R
  bDen : R :=
    p := mapUnivariateIfCan(retractIfCan,ib.basisDen)$IBPTOOLS(K,R,UP,E)
    p case "failed" => error "retraction of basis denominator failed"
    p :: R
  [b,bDen,bInv]

padicLocalIntegralBasis: (UP,R,R,R) -> IResult
padicLocalIntegralBasis(p,disc,redDisc,prime) ==
  -- polynomials in x modulo 'prime'
  sae := SAE(K,R,prime)
  -- find the factorization of 'p' modulo 'prime' and lift the
  -- prime powers to elements of UP:
  -- reduce 'p' modulo 'prime'
  reducedP := mapUnivariate(reduce,p)$IBPTOOLS(R,UP,SUP UP,sae)
  -- factor the reduced polynomial
  factorListSAE := factors factor(reducedP)$DDFACT(sae,SUP sae)
  -- if only one prime factor, perform usual integral basis computation
  (# factorListSAE) = 1 =>
    ib := localIntegralBasis(prime)$WFFINTBS(K,R,UP,F)
    index := diagonalProduct(ib.basisInv)
    [ib.basis,ib.basisDen,ib.basisInv,disc quo (index * index)]
  -- list the 'prime factors' of the reduced polynomial
  redPrimes : List SUP sae :=
    [f.factor for f in factorListSAE]
  -- lift these factors to elements of UP
```

```
    primes : List UP :=
      [mapUnivariate(lift,ff)$IBPTOOLS(R,UP,SUP UP,sae) for ff in redPrimes]
    -- list the exponents
    expons : List NNI := [((f.exponent) :: NNI) for f in factorListSAE]
    -- list the 'primary factors' of the reduced polynomial
    redPrimaries : List SUP sae :=
      [(f.factor) **((f.exponent) :: NNI) for f in factorListSAE]
    -- lift these factors to elements of UP
    primaries : List UP :=
      [mapUnivariate(lift,ff)$IBPTOOLS(R,UP,SUP UP,sae) for ff in redPrimaries]
    -- lift the factors to factors modulo a suitable power of 'prime'
    deg := (1 + order(redDisc,prime) * degree(prime)) :: PI
    henselInfo := HenselLift(p,primaries,prime,deg)
    henselFactors := henselInfo.plist
    -- compute integral bases for the factors
    factorBases : List Result := empty(); degPrime := degree prime
    for pp in primes for k in expons for qq in henselFactors repeat
      base :=
        degPp := degree pp
        degPp > 1 and gcd(degPp,degPrime) = 1 =>
          compLocalBasisOverExt(qq,prime,pp,k)
        compLocalBasis(qq,prime)
      factorBases := concat(base,factorBases)
    factorBases := reverse_! factorBases
    ib := chineseRemainder(henselFactors,factorBases,rank()$F)$IBACHIN(K,R,UP)
    index := diagonalProduct(ib.basisInv)
    [ib.basis,ib.basisDen,ib.basisInv,disc quo (index * index)]

  localIntegralBasis prime ==
    p := definingPolynomial()$F; disc := discriminant p
    --disc := determinant traceMatrix()$F
    redDisc := reducedDiscriminant p
    ib := padicLocalIntegralBasis(p,disc,redDisc,prime)
    [ib.basis,ib.basisDen,ib.basisInv]

  listSquaredFactors: R -> List R
  listSquaredFactors px ==
    -- returns a list of the factors of px which occur with
    -- exponent > 1
    ans : List R := empty()
    factored := factor(px)$DistinctDegreeFactorize(K,R)
    for f in factors(factored) repeat
      if f.exponent > 1 then ans := concat(f.factor,ans)
    ans

  integralBasis() ==
    p := definingPolynomial()$F; disc := discriminant p; n := rank()$F
    --traceMat := traceMatrix()$F; n := rank()$F
    --disc := determinant traceMat        -- discriminant of current order
    singList := listSquaredFactors disc -- singularities of relative Spec
```

```
redDisc := reducedDiscriminant p
runningRb := runningRbinv := scalarMatrix(n,1)$Mat
-- runningRb    = basis matrix of current order
-- runningRbinv = inverse basis matrix of current order
-- these are wrt the original basis for F
runningRbden : R := 1
-- runningRbden = denominator for current basis matrix
empty? singList => [runningRb, runningRbden, runningRbinv]
for prime in singList repeat
  lb := padicLocalIntegralBasis(p,disc,redDisc,prime)
  rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
  disc := lb.discr
  mat := vertConcat(rbden * runningRb,runningRbden * rb)
  runningRbden := runningRbden * rbden
  runningRb := squareTop rowEchelon(mat,runningRbden)
  --runningRb := squareTop rowEch mat
  runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
[runningRb, runningRbden, runningRbinv]
```

———-

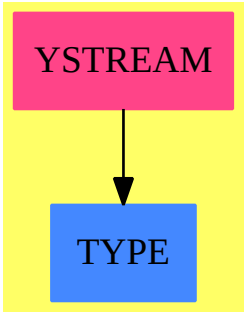
— PWFFINTB.dotabb —

```
"PWFFINTB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PWFFINTB"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"PWFFINTB" -> "MONOGEN"
```

———-

17.7 package YSTREAM ParadoxicalCombinatorsForStreams

17.7.1 ParadoxicalCombinatorsForStreams (YSTREAM)



Exports:
Y

— package YSTREAM ParadoxicalCombinatorsForStreams —

```
)abbrev package YSTREAM ParadoxicalCombinatorsForStreams
++ Author: Burge, Watt (revised by Williamson)
++ Date Created: 1986
++ Date Last Updated: 21 October 1989
++ Keywords: stream, fixed point
++ Examples:
++ References:
++ Description:
++ Computation of fixed points of mappings on streams

ParadoxicalCombinatorsForStreams(A):Exports == Implementation where
  ++ This package implements fixed-point computations on streams.
  A  :   Type
  ST ==> Stream
  L  ==> List
  I  ==> Integer

  Exports ==> with
    Y: (ST A -> ST A) -> ST A
      ++ Y(f) computes a fixed point of the function f.
    Y: (L ST A -> L ST A,I) -> L ST A
      ++ Y(g,n) computes a fixed point of the function g, where g takes
      ++ a list of n streams and returns a list of n streams.

  Implementation ==> add

    Y f ==
      y : ST A := CONS(0$I,0$I)$Lisp
      j := f y
      RPLACA(y,frst j)$Lisp
      RPLACD(y,rst j)$Lisp
      y

    Y(g,n) ==
      x : L ST A := [CONS(0$I,0$I)$Lisp for i in 1..n]
      j := g x
      for xi in x for ji in j repeat
        RPLACA(xi,frst ji)$Lisp
        RPLACD(xi,rst ji)$Lisp
      x
```

———-

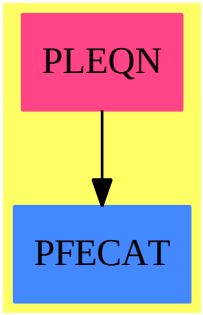
— **YSTREAM.dotabb** —

```
"YSTREAM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=YSTREAM"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"YSTREAM" -> "TYPE"
```

———-

17.8 package PLEQN ParametricLinearEquations

17.8.1 ParametricLinearEquations (PLEQN)



Exports:

bsolve	B1solve	dmp2rfi	dmp2rfi	dmp2rfi
factorset	hasoln	inconsistent?	maxrank	minrank
minset	nextSublist	overset?	ParCond	ParCondList
pr2dmp	psolve	rdregime	redmat	redpps
regime	se2rfi	sqfree	wrregime	

```
-- modified to conform with new runtime system 06/04/90
-- updated with comments for MB, 02/16/94
-- also cleaned up some unnecessary arguments in regime routine
--
-- MB: In order to allow the rhs to be indeterminate, while working
-- mainly with the parametric variables on the lhs (less number of
-- variables), certain conversions of internal representation from
-- GR to Polynomial R and Fraction Polynomial R are done.  At the time
-- of implementation, I thought that this may be more efficient.  I
-- have not done any comparison since that requires rewriting the
-- package.  My own application needs to call this package quite often,
-- and most computations involves only polynomials in the parametric
-- variables.
```

```
-- The 12 modes of psolve are probably not all necessary.  Again, I
-- was thinking that if there are many regimes and many ranks, then
-- the output is quite big, and it may be nice to be able to save it
-- and read the results in later to continue computing rather than
-- recomputing.  Because of the combinatorial nature of the algorithm
-- (computing all subdeterminants!), it does not take a very big matrix
-- to get into many regimes.  But I now have second thoughts of this
-- design, since most of the time, the results are just intermediate,
-- passed to be further processed.  On the other hand, there is probably
-- no penalty in leaving the options as is.
```

— package PLEQN ParametricLinearEquations —

```
)abbrev package PLEQN ParametricLinearEquations
++ Author: William Sit, spring 89
++ Description:
++ This package completely solves a parametric linear system of equations
++ by decomposing the set of all parametric values for which the linear
++ system is consistent into a union of quasi-algebraic  sets (which need
++ not be irredundant, but most of the time is). Each quasi-algebraic
++ set is described by a list of polynomials that vanish on the set, and
++ a list of polynomials that vanish at no point of the set.
++ For each quasi-algebraic set, the solution of the linear system
++ is given, as a particular solution and  a basis of the homogeneous
++ system.
++
++ The parametric linear system should be given in matrix form, with
++ a coefficient matrix and a right hand side vector. The entries
++ of the coefficient matrix and right hand side vector should be
++ polynomials in the parametric variables, over a Euclidean domain
++ of characteristic zero.
++
++ If the system is homogeneous, the right hand side need not be given.
++ The right hand side can also be  replaced by an indeterminate vector,
++ in which case, the conditions required for consistency will also be
++ given.
++
++ The package has other facilities for saving results to external
++ files, as well as solving the system for a specified minimum rank.
++ Altogether there are 12 mode maps for psolve, as explained below.

ParametricLinearEquations(R,Var,Expon,GR):
           Declaration == Definition where

  R         :  Join(EuclideanDomain, CharacteristicZero)
  -- Warning: does not work if R is a field! because of Fraction R
  Var       :  Join(OrderedSet,ConvertibleTo (Symbol))
  Expon     :  OrderedAbelianMonoidSup
  GR        :  PolynomialCategory(R,Expon,Var)
```

```
  F        == Fraction R
  FILE ==> FileCategory
  FNAME ==> FileName
  GB  ==> EuclideanGroebnerBasisPackage
--  GBINTERN ==> GroebnerInternalPackage
  I   ==> Integer
  L   ==> List
  M   ==> Matrix
  NNI ==> NonNegativeInteger
  OUT ==> OutputForm
  P   ==> Polynomial
  PI  ==> PositiveInteger
  SEG ==> Segment
  SM  ==> SquareMatrix
  S   ==> String
  V   ==> Vector
  mf    ==> MultivariateFactorize(Var,Expon,R,GR)
  rp    ==> GB(R,Expon,Var,GR)
  gb    ==> GB(R,Expon,Var,GR)
  PR    ==> P R
  GF    ==> Fraction PR
  plift ==> PolynomialCategoryLifting(Expon,Var,R,GR,GF)
  Inputmode ==> Integer
  groebner ==> euclideanGroebner
  redPol  ==> euclideanNormalForm

-- MB: The following macros are data structures to store mostly
-- intermediate results
-- Rec stores a subdeterminant with corresponding row and column indices
-- Fgb is a Groebner basis for the ideal generated by the subdeterminants
--     of a given rank.
-- Linsys specifies a linearly independent system of a given system
--     assuming a given rank, using given row and column indices
-- Linsoln stores the solution to the parametric linear system as a basis
--     and a particular solution (for a given regime)
-- Rec2 stores the rank, and a list of subdeterminants of that rank,
--     and a Groebner basis for the ideal they generate.
-- Rec3 stores a regime and the corresponding solution; the regime is
--     given by a list of equations (eqzro) and one inequation (neqzro)
--     describing the quasi-algebraic set which is the regime; the
--     additional consistency conditions due to the rhs is given by wcond.
-- Ranksolns stores a list of regimes and their solutions, and the number
--     of regimes all together.
-- Rec8 (temporary) stores a quasi-algebraic set with an indication
-- whether it is empty (sysok = false) or not (sysok = true).

-- I think psolve should be renamed parametricSolve, or even
-- parametricLinearSolve.  On the other hand, may be just solve will do.
-- Please feel free to change it to conform with system conventions.
-- Most psolve routines return a list of regimes and solutions,
```

```
-- except those that output to file when the number of regimes is
-- returned instead.
-- This version has been tested on the pc version 1.608 March 13, 1992

  Rec   ==> Record(det:GR,rows:L I,cols:L I)
  Eqns  ==> L Rec
  Fgb   ==> L GR  -- groebner basis
  Linsoln   ==> Record(partsol:V GF,basis:L V GF)
  Linsys    ==> Record(mat:M GF,vec:L GF,rank:NNI,rows:L I,cols:L I)
  Rec2  ==> Record(rank:NNI,eqns:Eqns,fgb:Fgb)
  RankConds ==> L Rec2
  Rec3  ==> Record(eqzro:L GR, neqzro:L GR,wcond:L PR, bsoln:Linsoln)
  Ranksolns ==> Record(rgl:L Rec3,rgsz:I)
  Rec8 ==> Record(sysok:Boolean, z0:L GR, n0:L GR)


  Declaration == with
      psolve: (M GR, L GR) -> L Rec3
        ++ psolve(c,w) solves c z = w for all possible ranks
        ++ of the matrix c and given right hand side vector w
        -- this is mode 1
      psolve: (M GR, L Symbol) -> L Rec3
        ++ psolve(c,w) solves c z = w for all possible ranks
        ++ of the matrix c and indeterminate right hand side w
        -- this is mode 2
      psolve:  M GR        -> L Rec3
        ++ psolve(c) solves the homogeneous linear system
        ++ c z = 0 for all possible ranks of the matrix c
        -- this is mode 3
      psolve: (M GR, L GR, PI) -> L Rec3
        ++ psolve(c,w,k) solves c z = w for all possible ranks >= k
        ++ of the matrix c and given right hand side vector w
        -- this is mode 4
      psolve: (M GR, L Symbol, PI) -> L Rec3
        ++ psolve(c,w,k) solves c z = w for all possible ranks >= k
        ++ of the matrix c and indeterminate right hand side w
        -- this is mode 5
      psolve: (M GR,       PI) -> L Rec3
        ++ psolve(c) solves the homogeneous linear system
        ++ c z = 0 for all possible ranks >= k of the matrix c
        -- this is mode 6
      psolve: (M GR, L GR, S) -> I
        ++ psolve(c,w,s) solves c z = w for all possible ranks
        ++ of the matrix c and given right hand side vector w,
        ++ writes the results to a file named s, and returns the
        ++ number of regimes
        -- this is mode 7
      psolve: (M GR, L Symbol, S) -> I
        ++ psolve(c,w,s) solves c z = w for all possible ranks
        ++ of the matrix c and indeterminate right hand side w,
```

```
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 8
psolve: (M GR,       S) -> I
  ++ psolve(c,s) solves c z = 0 for all possible ranks
  ++ of the matrix c and given right hand side vector w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 9
psolve: (M GR, L GR, PI, S) -> I
  ++ psolve(c,w,k,s) solves c z = w for all possible ranks >= k
  ++ of the matrix c and given right hand side w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 10
psolve: (M GR, L Symbol, PI, S) -> I
  ++ psolve(c,w,k,s) solves c z = w for all possible ranks >= k
  ++ of the matrix c and indeterminate right hand side w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 11
psolve: (M GR,       PI, S) -> I
  ++ psolve(c,k,s) solves c z = 0 for all possible ranks >= k
  ++ of the matrix c,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 12

wrregime   : (L Rec3, S) -> I
  ++ wrregime(l,s) writes a list of regimes to a file named s
  ++ and returns the number of regimes written
rdregime   : S -> L Rec3
  ++ rdregime(s) reads in a list from a file with name s

  -- for internal use only --
-- these are exported so my other packages can use them

bsolve: (M GR, L GF, NNI, S, Inputmode) -> Ranksolns
  ++ bsolve(c, w, r, s, m) returns a list of regimes and
  ++ solutions of the system c z = w for ranks at least r;
  ++ depending on the mode m chosen, it writes the output to
  ++ a file given by the string s.
dmp2rfi: GR -> GF
  ++ dmp2rfi(p) converts p to target domain
dmp2rfi: M GR -> M GF
  ++ dmp2rfi(m) converts m to target domain
dmp2rfi: L GR -> L GF
  ++ dmp2rfi(l) converts l to target domain
se2rfi:  L Symbol -> L GF
  ++ se2rfi(l) converts l to target domain
```

```
      pr2dmp: PR -> GR
        ++ pr2dmp(p) converts p to target domain
      hasoln: (Fgb, L GR) -> Rec8
        ++ hasoln(g, l) tests whether the quasi-algebraic set
        ++ defined by p = 0 for p in g and q ^= 0 for q in l
        ++ is empty or not and returns a simplified definition
        ++ of the quasi-algebraic set
        -- this is now done in QALGSET package
      ParCondList: (M GR,NNI) -> RankConds
        ++ ParCondList(c,r) computes a list of subdeterminants of each
        ++ rank >= r of the matrix c and returns
        ++ a groebner basis for the
        ++ ideal they generate
      redpps: (Linsoln, Fgb) -> Linsoln
        ++ redpps(s,g) returns the simplified form of s after reducing
        ++ modulo a groebner basis g


--              L O C A L    F U N C T I O N S

      B1solve: Linsys -> Linsoln
        ++ B1solve(s) solves the system (s.mat) z = s.vec
        ++ for the variables given by the column indices of s.cols
        ++ in terms of the other variables and the right hand side s.vec
        ++ by assuming that the rank is s.rank,
        ++ that the system is consistent, with the linearly
        ++ independent equations indexed by the given row indices s.rows;
        ++ the coefficients in s.mat involving parameters are treated as
        ++ polynomials.  B1solve(s) returns a particular solution to the
        ++ system and a basis of the homogeneous system (s.mat) z = 0.
      factorset: GR -> L GR
        ++ factorset(p) returns the set of irreducible factors of p.
      maxrank: RankConds -> NNI
        ++ maxrank(r) returns the maximum rank in the list r of regimes
      minrank: RankConds -> NNI
        ++ minrank(r) returns the minimum rank in the list r of regimes
      minset: L L GR -> L L GR
        ++ minset(sl) returns the sublist of sl consisting of the minimal
        ++ lists (with respect to inclusion) in the list sl of lists
      nextSublist: (I, I) -> L L I
        ++ nextSublist(n,k) returns a list of k-subsets of {1, ..., n}.
      overset?: (L GR, L L GR) -> Boolean
        ++ overset?(s,sl) returns true if s properly a sublist of a member
        ++ of sl; otherwise it returns false
      ParCond    : (M GR,NNI) -> Eqns
        ++ ParCond(m,k) returns the list of all k by k subdeterminants in
        ++ the matrix m
      redmat: (M GR, Fgb) -> M GR
        ++ redmat(m,g) returns a matrix whose entries are those of m
```

```
      ++ modulo the ideal generated by the groebner basis g
    regime: (Rec,M GR,L GF,L L GR,NNI,NNI,Inputmode) -> Rec3
      ++ regime(y,c, w, p, r, rm, m) returns a regime,
      ++ a list of polynomials specifying the consistency conditions,
      ++ a particular solution and basis representing the general
      ++ solution of the parametric linear system c z = w
      ++ on that regime. The regime returned depends on
      ++ the subdeterminant y.det and the row and column indices.
      ++ The solutions are simplified using the assumption that
      ++ the system has rank r and maximum rank rm. The list p
      ++ represents a list of list of factors of polynomials in
      ++ a groebner basis of the ideal generated by higher order
      ++ subdeterminants, and ius used for the simplification.
      ++ The mode m
      ++ distinguishes the cases when the system is homogeneous,
      ++ or the right hand side is arbitrary, or when there is no
      ++ new right hand side variables.
    sqfree: GR -> GR
      ++ sqfree(p) returns the product of square free factors of p
    inconsistent?: L GR -> Boolean
      ++ inconsistant?(pl) returns true if the system of equations
      ++ p = 0 for p in pl is inconsistent.  It is assumed
      ++ that pl is a groebner basis.
      -- this is needed because of change to
      -- EuclideanGroebnerBasisPackage
    inconsistent?: L PR -> Boolean
      ++ inconsistant?(pl) returns true if the system of equations
      ++ p = 0 for p in pl is inconsistent.  It is assumed
      ++ that pl is a groebner basis.
      -- this is needed because of change to
      -- EuclideanGroebnerBasisPackage

Definition == add

  inconsistent?(pl:L GR):Boolean ==
    for p in pl repeat
      ground? p => return true
    false
  inconsistent?(pl:L PR):Boolean ==
    for p in pl repeat
      ground? p => return true
    false

  B1solve (sys:Linsys):Linsoln ==
    i,j,i1,j1:I
    rss:L I:=sys.rows
    nss:L I:=sys.cols
    k:=sys.rank
    cmat:M GF:=sys.mat
    n:=ncols cmat
```

```
    frcols:L I:=setDifference$(L I) (expand$(SEG I) (1..n), nss)
    w:L GF:=sys.vec
    p:V GF:=new(n,0)
    pbas:L V GF:=[]
    if k ^= 0 then
      augmat:M GF:=zero(k,n+1)
      for i in rss for i1 in 1.. repeat
        for j in nss for j1 in 1.. repeat
          augmat(i1,j1):=cmat(i,j)
        for j in frcols for j1 in k+1.. repeat
          augmat(i1,j1):=-cmat(i,j)
        augmat(i1,n+1):=w.i
      augmat:=rowEchelon$(M GF) augmat
      for i in nss for i1 in 1.. repeat p.i:=augmat(i1,n+1)
      for j in frcols for j1 in k+1.. repeat
        pb:V GF:=new(n,0)
        pb.j:=1
        for i in nss for i1 in 1.. repeat
          pb.i:=augmat(i1,j1)
        pbas:=cons(pb,pbas)
    else
      for j in frcols for j1 in k+1.. repeat
        pb:V GF:=new(n,0)
        pb.j:=1
        pbas:=cons(pb,pbas)
    [p,pbas]

  regime (y, coef, w, psbf, rk, rkmax, mode) ==
    i,j:I
    -- use the y.det nonzero to simplify the groebner basis
    -- of ideal generated by higher order subdeterminants
    ydetf:L GR:=factorset y.det
    yzero:L GR:=
      rk = rkmax => nil$(L GR)
      psbf:=[setDifference(x, ydetf) for x in psbf]
      groebner$gb [*/x for x in psbf]
    -- simplify coefficients by modulo ideal
    nc:M GF:=dmp2rfi redmat(coef,yzero)
    -- solve the system
    rss:L I:=y.rows;  nss:L I :=y.cols
    sys:Linsys:=[nc,w,rk,rss,nss]$Linsys
    pps:= B1solve(sys)
    pp:=pps.partsol
    frows:L I:=setDifference$(L I) (expand$(SEG I) (1..nrows coef),rss)
    wcd:L PR:= []
    -- case homogeneous rhs
    entry? (mode, [3,6,9,12]$(L I)) =>
             [yzero, ydetf,wcd, redpps(pps, yzero)]$Rec3
    -- case arbitrary rhs, pps not reduced
    for i in frows repeat
```

```
          weqn:GF:=+/[nc(i,j)*(pp.j) for j in nss]
          wnum:PR:=numer$GF (w.i - weqn)
          wnum = 0 => "trivially satisfied"
          ground? wnum => return [yzero, ydetf,[1$PR]$(L PR),pps]$Rec3
          wcd:=cons(wnum,wcd)
      entry? (mode, [2,5,8,11]$(L I)) => [yzero, ydetf, wcd, pps]$Rec3
      -- case no new rhs variable
      if not empty? wcd then _
        yzero:=removeDuplicates append(yzero,[pr2dmp pw for pw in wcd])
      test:Rec8:=hasoln (yzero, ydetf)
      not test.sysok => [test.z0, test.n0, [1$PR]$(L PR), pps]$Rec3
      [test.z0, test.n0, [], redpps(pps, test.z0)]$Rec3

    bsolve (coeff, w, h, outname, mode) ==
      r:=nrows coeff
--    n:=ncols coeff
      r ^= #w => error "number of rows unequal on lhs and rhs"
      newfile:FNAME
      rksoln:File Rec3
      count:I:=0
      lrec3:L Rec3:=[]
      filemode:Boolean:= entry? (mode, [7,8,9,10,11,12]$(L I))
      if filemode then
        newfile:=new$FNAME  ("",outname,"regime")
        rksoln:=open$(File Rec3) newfile
      y:Rec
      k:NNI
      rrcl:RankConds:=
        entry? (mode,[1,2,3,7,8,9]$(L I)) => ParCondList (coeff,0)
        entry? (mode,[4,5,6,10,11,12]$(L I)) => ParCondList (coeff,h)
      rkmax:=maxrank rrcl
      rkmin:=minrank rrcl
      for k in rkmax-rkmin+1..1 by -1 repeat
        rk:=rrcl.k.rank
        pc:Eqns:=rrcl.k.eqns
        psb:Fgb:= (if rk=rkmax then [] else rrcl.(k+1).fgb)
        psbf:L L GR:= [factorset x for x in psb]
        psbf:= minset(psbf)
        for y in pc repeat
          rec3:Rec3:= regime (y, coeff, w, psbf, rk, rkmax, mode)
          inconsistent? rec3.wcond => "incompatible system"
          if filemode then write_!(rksoln, rec3)
          else lrec3:= cons(rec3, lrec3)
          count:=count+1
      if filemode then close_! rksoln
      [lrec3, count]$Ranksolns

    factorset y ==
      ground? y => []
      [j.factor for j in factors(factor$mf y)]
```

```
    ParCondList (mat, h) ==
      rcl: RankConds:= []
      ps: L GR:=[]
      pc:Eqns:=[]
      npc: Eqns:=[]
      psbf: Fgb:=[]
      rc: Rec
      done: Boolean := false
      r:=nrows mat
      n:=ncols mat
      maxrk:I:=min(r,n)
      k:NNI
      for k in min(r,n)..h by -1 until done repeat
        pc:= ParCond(mat,k)
        npc:=[]
        (empty? pc) and (k >= 1) => maxrk:= k - 1
        if ground? pc.1.det -- only one is sufficient (neqzro = {})
        then (npc:=pc; done:=true; ps := [1$GR])
        else
          zro:L GR:= (if k = maxrk then [] else rcl.1.fgb)
          covered:Boolean:=false
          for rc in pc until covered repeat
            p:GR:= redPol$rp (rc.det, zro)
            p = 0 => "incompatible or covered subdeterminant"
            test:=hasoln(zro, [rc.det])
--          zroideal:=ideal(zro)
--          inRadical? (p, zroideal) => "incompatible or covered"
            ^test.sysok => "incompatible or covered"
-- The next line is WRONG! cannot replace zro by test.z0
--          zro:=groebner$gb (cons(*/test.n0, test.z0))
            zro:=groebner$gb (cons(p,zro))
            npc:=cons(rc,npc)
            done:= covered:= inconsistent? zro
          ps:=zro
        pcl: Rec2:= construct(k,npc,ps)
        rcl:=cons(pcl,rcl)
      rcl

    redpps(pps, zz) ==
      pv:=pps.partsol
      r:=#pv
      pb:=pps.basis
      n:=#pb + 1
      nummat:M GR:=zero(r,n)
      denmat:M GR:=zero(r,n)
      for i in  1..r repeat
        nummat(i,1):=pr2dmp numer$GF pv.i
        denmat(i,1):=pr2dmp denom$GF pv.i
      for j in 2..n repeat
```

```
    for i in 1..r  repeat
      nummat(i,j):=pr2dmp numer$GF (pb.(j-1)).i
      denmat(i,j):=pr2dmp denom$GF (pb.(j-1)).i
  nummat:=redmat(nummat, zz)
  denmat:=redmat(denmat, zz)
  for i in 1..r repeat
    pv.i:=(dmp2rfi nummat(i,1))/(dmp2rfi denmat(i,1))
  for j in 2..n repeat
    pbj:V GF:=new(r,0)
    for i in 1..r repeat
      pbj.i:=(dmp2rfi nummat(i,j))/(dmp2rfi  denmat(i,j))
    pb.(j-1):=pbj
  [pv, pb]

dmp2rfi (mat:M GR): M GF ==
  r:=nrows mat
  n:=ncols mat
  nmat:M GF:=zero(r,n)
  for i in 1..r repeat
    for j in 1..n repeat
      nmat(i,j):=dmp2rfi mat(i,j)
  nmat

dmp2rfi (vl: L GR):L GF ==
  [dmp2rfi v for v in vl]

psolve (mat:M GR, w:L GR): L Rec3 ==
  bsolve(mat, dmp2rfi w, 1, "nofile", 1).rgl
psolve (mat:M GR, w:L Symbol): L Rec3 ==
  bsolve(mat,  se2rfi w, 1, "nofile", 2).rgl
psolve (mat:M GR): L Rec3 ==
  bsolve(mat, [0$GF for i in 1..nrows mat], 1, "nofile", 3).rgl

psolve (mat:M GR, w:L GR, h:PI): L Rec3 ==
  bsolve(mat, dmp2rfi w, h::NNI, "nofile", 4).rgl
psolve (mat:M GR, w:L Symbol, h:PI): L Rec3 ==
  bsolve(mat, se2rfi w, h::NNI, "nofile", 5).rgl
psolve (mat:M GR, h:PI): L Rec3 ==
  bsolve(mat, [0$GF for i in 1..nrows mat], h::NNI, "nofile", 6).rgl

psolve (mat:M GR, w:L GR, outname:S): I ==
  bsolve(mat, dmp2rfi w, 1, outname, 7).rgsz
psolve (mat:M GR, w:L Symbol, outname:S): I ==
  bsolve(mat, se2rfi w, 1, outname, 8).rgsz
psolve (mat:M GR, outname:S): I ==
  bsolve(mat, [0$GF for i in 1..nrows mat], 1, outname, 9).rgsz

nextSublist (n,k) ==
  n <= 0 => []
  k <= 0 => [ nil$(List Integer) ]
```

```
      k > n => []
      n = 1 and k = 1 => [[1]]
      mslist: L L I:=[]
      for ms in nextSublist(n-1,k-1) repeat
        mslist:=cons(append(ms,[n]),mslist)
      append(nextSublist(n-1,k), mslist)

    psolve (mat:M GR, w:L GR, h:PI, outname:S): I ==
      bsolve(mat, dmp2rfi w, h::NNI, outname, 10).rgsz
    psolve (mat:M GR, w:L Symbol, h:PI, outname:S): I ==
      bsolve(mat, se2rfi w, h::NNI, outname, 11).rgsz
    psolve (mat:M GR, h:PI, outname:S): I ==
      bsolve(mat,[0$GF for i in 1..nrows mat],h::NNI,outname, 12).rgsz

    hasoln (zro,nzro) ==
      empty? zro => [true, zro, nzro]
      zro:=groebner$gb zro
      inconsistent? zro => [false, zro, nzro]
      empty? nzro =>[true, zro, nzro]
      pnzro:GR:=redPol$rp (*/nzro, zro)
      pnzro = 0 => [false, zro, nzro]
      nzro:=factorset pnzro
      psbf:L L GR:= minset [factorset p for p in zro]
      psbf:= [setDifference(x, nzro) for x in psbf]
      entry? ([], psbf) => [false, zro, nzro]
      zro:=groebner$gb [*/x for x in psbf]
      inconsistent? zro => [false, zro, nzro]
      nzro:=[redPol$rp (p,zro) for p in nzro]
      nzro:=[p for p in nzro | ^(ground? p)]
      [true, zro, nzro]



    se2rfi w == [coerce$GF monomial$PR (1$PR, wi, 1) for wi in w]

    pr2dmp p ==
      ground? p => (ground p)::GR
      algCoerceInteractive(p,PR,GR)$(Lisp) pretend GR

    wrregime (lrec3, outname) ==
      newfile:FNAME:=new$FNAME ("",outname,"regime")
      rksoln: File Rec3:=open$(File Rec3) newfile
      count:I:=0  -- number of distinct regimes
--      rec3: Rec3
      for rec3 in lrec3 repeat
          write_!(rksoln, rec3)
          count:=count+1
      close_!(rksoln)
      count
```

```
dmp2rfi (p:GR):GF ==
  map$plift ((v1:Var):GF +-> (convert v1)@Symbol::GF,
             (r1:R):GF +-> r1::PR::GF, p)


rdregime inname ==
  infilename:=filename$FNAME ("",inname, "regime")
  infile: File Rec3:=open$(File Rec3) (infilename, "input")
  rksoln:L Rec3:=[]
  rec3:Union(Rec3, "failed"):=readIfCan_!$(File Rec3) (infile)
  while rec3 case Rec3 repeat
    rksoln:=cons(rec3::Rec3,rksoln) -- replace : to :: for AIX
    rec3:=readIfCan_!$(File Rec3) (infile)
  close_!(infile)
  rksoln

maxrank rcl ==
  empty? rcl => 0
  "max"/[j.rank for j in rcl]

minrank rcl ==
  empty? rcl => 0
  "min"/[j.rank for j in rcl]

minset lset ==
  empty? lset => lset
  [x for x in lset | ^(overset?(x,lset))]

sqfree p == */[j.factor for j in factors(squareFree p)]


ParCond (mat, k) ==
  k = 0 => [[1, [], []]$Rec]
  j:NNI:=k::NNI
  DetEqn :Eqns := []
  r:I:= nrows(mat)
  n:I:= ncols(mat)
  k > min(r,n) => error "k exceeds maximum possible rank "
  found:Boolean:=false
  for rss in nextSublist(r, k) until found repeat
    for nss in nextSublist(n, k) until found repeat
      matsub := mat(rss, nss) pretend SM(j, GR)
      detmat := determinant(matsub)
      if detmat ^= 0 then
        found:= (ground? detmat)
        detmat:=sqfree detmat
        neweqn:Rec:=construct(detmat,rss,nss)
        DetEqn:=cons(neweqn, DetEqn)
  found => [first DetEqn]$Eqns
  sort((z1:Rec,z2:Rec):Boolean +-> degree z1.det < degree z2.det, DetEqn)
```

```
    overset?(p,qlist) ==
      empty? qlist => false
      or/[(brace$(Set GR) q) <$(Set GR) (brace$(Set GR) p) _
                                              for q in qlist]


    redmat (mat,psb) ==
      i,j:I
      r:=nrows(mat)
      n:=ncols(mat)
      newmat: M GR:=zero(r,n)
      for i in 1..r repeat
        for j in 1..n repeat
          p:GR:=mat(i,j)
          ground? p => newmat(i,j):=p
          newmat(i,j):=redPol$rp (p,psb)
      newmat
```

———-

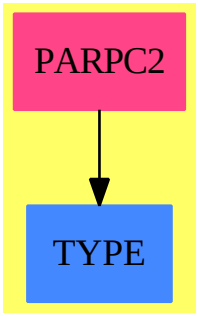
— PLEQN.dotabb —

```
"PLEQN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLEQN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PLEQN" -> "PFECAT"
```

———-

17.9 package PARPC2 ParametricPlaneCurveFunctions2

17.9.1 ParametricPlaneCurveFunctions2 (PARPC2)



Exports:
map

— package PARPC2 ParametricPlaneCurveFunctions2 —

```
)abbrev package PARPC2 ParametricPlaneCurveFunctions2
++ Description:
++ This package has no description

ParametricPlaneCurveFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricPlaneCurve(CF1)) -> ParametricPlaneCurve(CF2)
    ++ map(f,x) \undocumented
 == add
  map(f, c) == curve(f coordinate(c,1), f coordinate(c, 2))
```

———-

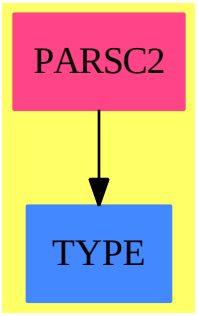
— PARPC2.dotabb —

```
"PARPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARPC2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARPC2" -> "TYPE"
```

———-

17.10 package PARSC2 ParametricSpaceCurveFunctions2

17.10.1 ParametricSpaceCurveFunctions2 (PARSC2)



Exports:
map

— package PARSC2 ParametricSpaceCurveFunctions2 —

```
)abbrev package PARSC2 ParametricSpaceCurveFunctions2
++ Description:
++ This package has no description

ParametricSpaceCurveFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricSpaceCurve(CF1)) -> ParametricSpaceCurve(CF2)
    ++ map(f,x) \undocumented
 == add
  map(f, c) == curve(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))
```

———-

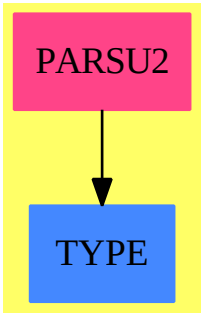
— PARSC2.dotabb —

```
"PARSC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARSC2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSC2" -> "TYPE"
```

———-

17.11 package PARSU2 ParametricSurfaceFunctions2

17.11.1 ParametricSurfaceFunctions2 (PARSU2)



Exports:
map

— **package PARSU2 ParametricSurfaceFunctions2** —

```
)abbrev package PARSU2 ParametricSurfaceFunctions2
++ Description:
++ This package has no description

ParametricSurfaceFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricSurface(CF1)) -> ParametricSurface(CF2)
    ++ map(f,x) \undocumented
 == add
  map(f, c) == surface(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))
```

———-

— **PARSU2.dotabb** —

```
"PARSU2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARSU2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARSU2" -> "TYPE"
```

———-

17.12 package PARAMP ParametrizationPackage

— **ParametrizationPackage.input** —

```
)set break resume
)sys rm -f ParametrizationPackage.output
)spool ParametrizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametrizationPackage
--R
--R ParametrizationPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList
--R Abbreviation for ParametrizationPackage is PARAMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARAMP
--R
--R------------------------------ Operations --------------------------------
--R parametrize : (PolyRing,List(PCS)) -> PCS
--R parametrize : (PolyRing,Plc) -> PCS
--R parametrize : (PolyRing,PolyRing,Plc) -> PCS
--R parametrize : (PolyRing,Plc,Integer) -> PCS
--R
--E 1

)spool
)lisp (bye)
```

———-

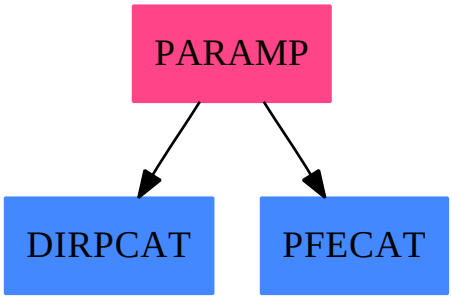
— **ParametrizationPackage.help** —

```
=====================================================================
ParametrizationPackage examples
=====================================================================

See Also:
o )show ParametrizationPackage
```

———-

17.12.1 ParametrizationPackage (PARAMP)



Exports:
parametrize

— package PARAMP ParametrizationPackage —

```
)abbreviation package PARAMP ParametrizationPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc):Exp == Impl where
  K:Field
  symb: List(Symbol)

  E : DirectProductCategory(#symb,NonNegativeInteger)
  OV  ==> OrderedVariableList(symb)

  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt   : ProjectiveSpaceCategory(K)
  PCS      : LocalPowerSeriesCategory(K)
  Plc      : PlacesCategory(K,PCS)

  Exp ==>  with

    parametrize: (PolyRing,List(PCS)) -> PCS

    parametrize: (PolyRing,Plc) -> PCS
      ++ parametrize(f,pl) returns the local parametrization of the
      ++ polynomial function f at the place pl. Note that local
      ++ parametrization of the place must have first been compute and set.
      ++ For simple point on a curve, this done with \spad{pointToPlace}.
      ++ The  local parametrization places corresponding to a leaf in a
      ++ desingularization tree are compute at the moment of
      ++ their "creation". (See package \spad{DesingTreePackage}.
```

```
  parametrize: (PolyRing,PolyRing,Plc) -> PCS
    ++ parametrize(f,g,pl) returns the local parametrization of the
    ++ rational function f/g at the place pl. Note that local
    ++ parametrization of the place must have first been compute and set.
    ++ For simple point on a curve, this done with \spad{pointToPlace}.
    ++ The  local parametrization places corresponding to a leaf in a
    ++ desingularization tree are compute at the moment of
    ++ their "creation". (See package \spad{DesingTreePackage}.

  parametrize: (PolyRing,Plc,Integer) -> PCS
    ++ parametrize(f,pl,n) returns t**n * parametrize(f,p).

Impl ==>  add

  import PCS
  import PolyRing

  -- the following returns the parametrization in term of
  -- the precomputed local parametrization
  -- of the point pt. Note if pl is a place and pl = pt::PLc then
  -- parametrize(f,pt) <> parametrize(pl) unless pt is a simple point
  parametrize(f:PolyRing,localPar:List(PCS))==
    zero?(f) => 0
    lc:K:=leadingCoefficient(f)
    ld:E:=degree f
    ldp:List NonNegativeInteger :=parts(ld)
    if empty?(localPar) then error _
            "the parametrization of the place or leaf has not been done yet!"
    monoPar:PCS:=reduce("*",[ s**e for s in localPar for e in ldp])
    lc* monoPar + parametrize(reductum(f),localPar)

  parametrize(f:PolyRing,pt:Plc)==
    zero?(f) => 0
    localPar:List PCS:=localParam pt
    parametrize(f,localPar)

  parametrize(f:PolyRing,g:PolyRing,pt:Plc)==
    sf:=parametrize(f,pt)
    sg:=parametrize(g,pt)
    sf * inv sg

  parametrize(f:PolyRing,pt:Plc,n:Integer)==
    s:=parametrize(f,pt)
    shift(s,n)
```

———-

— PARAMP.dotabb —

```
"PARAMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARAMP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PARAMP" -> "DIRPCAT"
"PARAMP" -> "PFECAT"
```

———-

17.13 package PFRPAC PartialFractionPackage

— PartialFractionPackage.input —

```
)set break resume
)spool PartialFractionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 4
a:=x+1/(y+1)
--R
--R        x y + x + 1
--R   (1)  -----------
--R           y + 1
--R                                          Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 4
partialFraction(a,y)$PFRPAC(INT)
--R
--R              1
--R   (2)  x + -----
--R            y + 1
--R Type: PartialFraction(UnivariatePolynomial(y,Fraction(Polynomial(Integer))))
--E 2

--S 3 of 4
b:=y+1/(x+1)
--R
--R        (x + 1)y + 1
--R   (3)  ------------
--R            x + 1
--R                                          Type: Fraction(Polynomial(Integer))
--E 3

--S 4 of 4
```

```
partialFraction(b,x)$PFRPAC(INT)
--R
--R              1
--R   (4)  y + -----
--R            x + 1
--R Type: PartialFraction(UnivariatePolynomial(x,Fraction(Polynomial(Integer))))
--E 4

)spool
)lisp (bye)
```

———-

— PartialFractionPackage.help —

```
=======================================================================
PartialFractionPackage examples
=======================================================================

When Axiom is given an expression such as:

  a:=x+1/(y+1)

it will end up with the result in Fraction Polynomial Integer.

        x y + x + 1
        -----------
           y + 1

We might want it in a partial fraction form. This can be done with:

  partialFraction(a,y)$PFRPAC(INT)

              1
        x + -----
            y + 1

We can do the same thing with a different variable:

  b:=y+1/(x+1)

        (x + 1)y + 1
        ------------
            x + 1

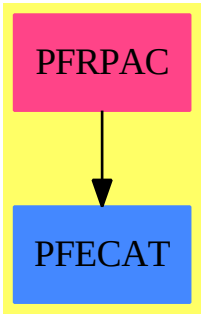
  partialFraction(b,x)$PFRPAC(INT)

              1
```

```
        y + -----
            x + 1

See Also:
o )show PartialFractionPackage
```

17.13.1 PartialFractionPackage (PFRPAC)



Exports:
partialFraction

— package PFRPAC PartialFractionPackage —

```
)abbrev package PFRPAC PartialFractionPackage
++ Author: Barry M. Trager
++ Date Created: 1992
++ BasicOperations:
++ Related Constructors: PartialFraction
++ Also See:
++ AMS Classifications:
++ Keywords: partial fraction, factorization, euclidean domain
++ References:
++ Description:
++ The package \spadtype{PartialFractionPackage} gives an easier
++ to use interfact the domain \spadtype{PartialFraction}.
++ The user gives a fraction of polynomials, and a variable and
++ the package converts it to the proper datatype for the
++ \spadtype{PartialFraction} domain.

PartialFractionPackage(R): Cat == Capsule where
--  R : UniqueFactorizationDomain -- not yet supported
```

```
  R : Join(EuclideanDomain, CharacteristicZero)
  FPR ==> Fraction Polynomial R
  INDE ==> IndexedExponents Symbol
  PR ==> Polynomial R
  SUP ==> SparseUnivariatePolynomial
  Cat == with
    partialFraction: (FPR, Symbol) -> Any
      ++ partialFraction(rf, var) returns the partial fraction decomposition
      ++ of the rational function rf with respect to the variable var.
      ++
      ++X a:=x+1/(y+1)
      ++X partialFraction(a,y)$PFRPAC(INT)
    partialFraction: (PR, Factored PR, Symbol) -> Any
      ++ partialFraction(num, facdenom, var) returns the partial fraction
      ++ decomposition of the rational function whose numerator is num and
      ++ whose factored denominator is facdenom with respect to the
      ++ variable var.
  Capsule == add
    partialFraction(rf, v) ==
      df := factor(denom rf)$MultivariateFactorize(Symbol, INDE,R,PR)
      partialFraction(numer rf, df, v)

    makeSup(p:Polynomial R, v:Symbol) : SparseUnivariatePolynomial FPR ==
      up := univariate(p,v)
      map((z1:PR):FPR +-> z1::FPR,up)_
        $UnivariatePolynomialCategoryFunctions2(PR, SUP PR, FPR, SUP FPR)

    partialFraction(p, facq, v) ==
      up := UnivariatePolynomial(v, Fraction Polynomial R)
      fup := Factored up
      ffact : List(Record(irr:up,pow:Integer))
      ffact:=[[makeSup(u.factor,v) pretend up,u.exponent]
                  for u in factors facq]
      fcont:=makeSup(unit facq,v) pretend up
      nflist:fup := fcont*(*/[primeFactor(ff.irr,ff.pow) for ff in ffact])
      pfup:=partialFraction(makeSup(p,v) pretend up,nflist)$PartialFraction(up)
      coerce(pfup)$AnyFunctions1(PartialFraction up)
```

———-

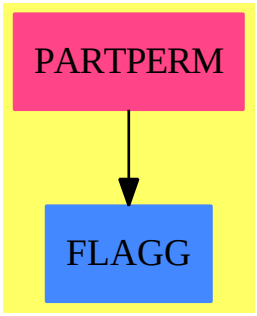
— PFRPAC.dotabb —

```
"PFRPAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFRPAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFRPAC" -> "PFECAT"
```

———-

17.14 package PARTPERM PartitionsAndPermutations

17.14.1 PartitionsAndPermutations (PARTPERM)



Exports:

conjugate	conjugates	partitions	permutations	sequences
shuffle	shufflein			

— package PARTPERM PartitionsAndPermutations —

```
)abbrev package PARTPERM PartitionsAndPermutations
++ Author: William H. Burge
++ Date Created: 29 October 1987
++ Date Last Updated: 3 April 1991
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: partition, permutation
++ References:
++ Description:
++ PartitionsAndPermutations contains functions for generating streams of
++ integer partitions, and streams of sequences of integers
++ composed from a multi-set.

PartitionsAndPermutations: Exports == Implementation where
  I   ==> Integer
  L   ==> List
  ST  ==> Stream
  ST1 ==> StreamFunctions1
  ST2 ==> StreamFunctions2
  ST3 ==> StreamFunctions3

  Exports ==> with

    partitions: (I,I,I) -> ST L I
      ++\spad{partitions(p,l,n)} is the stream of partitions
```

```
    ++ of n whose number of parts is no greater than p
    ++ and whose largest part is no greater than l.
  partitions: I -> ST L I
    ++\spad{partitions(n)} is the stream of all partitions of n.
  partitions: (I,I) -> ST L I
    ++\spad{partitions(p,l)} is the stream of all
    ++ partitions whose number of
    ++ parts and largest part are no greater than p and l.
  conjugate: L I -> L I
    ++\spad{conjugate(pt)} is the conjugate of the partition pt.
  conjugates: ST L I -> ST L I
    ++\spad{conjugates(lp)} is the stream of conjugates of a stream
    ++ of partitions lp.
  shuffle: (L I,L I) -> ST L I
    ++\spad{shuffle(l1,l2)} forms the stream of all shuffles of l1
    ++ and l2, i.e. all sequences that can be formed from
    ++ merging l1 and l2.
  shufflein: (L I,ST L I) -> ST L I
    ++\spad{shufflein(l,st)} maps shuffle(l,u) on to all
    ++ members u of st, concatenating the results.
  sequences: (L I,L I) -> ST L I
    ++\spad{sequences(l1,l2)} is the stream of all sequences that
    ++ can be composed from the multiset defined from
    ++ two lists of integers l1 and l2.
    ++ For example,the pair \spad{([1,2,4],[2,3,5])} represents
    ++ multi-set with 1 \spad{2}, 2 \spad{3}'s, and 4 \spad{5}'s.
  sequences: L I -> ST L I
    ++ \spad{sequences([l0,l1,l2,..,ln])} is the set of
    ++  all sequences formed from
    ++ \spad{l0} 0's,\spad{l1} 1's,\spad{l2} 2's,...,\spad{ln} n's.
  permutations: I -> ST L I
    ++\spad{permutations(n)} is the stream of permutations
    ++ formed from \spad{1,2,3,...,n}.

Implementation ==> add

  partitions(M,N,n) ==
    zero? n => concat(empty()$L(I),empty()$(ST L I))
    zero? M or zero? N or n < 0 => empty()
    c := map((l1:List(I)):List(I)+->concat(N,l1),partitions(M - 1,N,n - N))
    concat(c,partitions(M,N - 1,n))

  partitions n == partitions(n,n,n)

  partitions(M,N)==
    aaa : L ST L I := [partitions(M,N,i) for i in 0..M*N]
    concat(aaa :: ST ST L I)$ST1(L I)

  -- nogreq(n,l) is the number of elements of l that are greater or
  -- equal to n
```

```
    nogreq: (I,L I) -> I
    nogreq(n,x) == +/[1 for i in x | i >= n]

    conjugate x ==
      empty? x => empty()
      [nogreq(i,x) for i in 1..first x]

    conjugates z == map(conjugate,z)

    shuffle(x,y)==
      empty? x => concat(y,empty())$(ST L I)
      empty? y => concat(x,empty())$(ST L I)
      concat(map((l1:List(I)):List(I)+->concat(first x,l1),shuffle(rest x,y)),_
             map((l2:List(I)):List(I)+->concat(first y,l2),shuffle(x,rest y)))

    shufflein(x,yy) ==
      concat(map((l1:List(I)):ST(L I)+->shuffle(x,l1),yy)_
        $ST2(L I,ST L I))$ST1(L I)

    -- rpt(n,m) is the list of n m's
    rpt: (I,I) -> L I
    rpt(n,m) == [m for i in 1..n]

    -- zrpt(x,y) where x is [x0,x1,x2...] and y is [y0,y1,y2...]
    -- is the stream [rpt(x0,y0),rpt(x1,y1),...]
    zrpt: (L I,L I) -> ST L I
    zrpt(x,y) == map(rpt,x :: ST I,y :: ST I)$ST3(I,I,L I)

    sequences(x,y) ==
      reduce(concat(empty()$L(I),empty()$(ST L I)),_
                    shufflein,zrpt(x,y))$ST2(L I,ST L I)

    sequences x == sequences(x,[i for i in 0..#x-1])

    permutations n == sequences(rpt(n,1),[i for i in 1..n])
```

———-

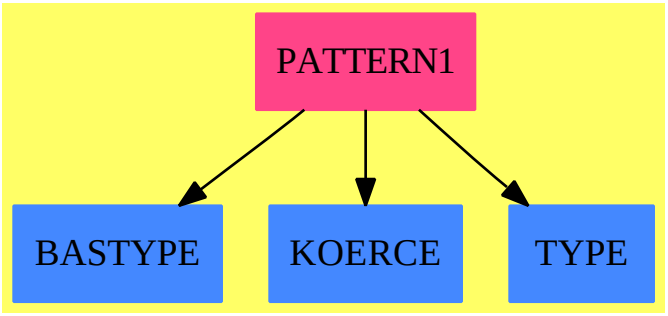
— PARTPERM.dotabb —

```
"PARTPERM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARTPERM"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PARTPERM" -> "FLAGG"
```

———-

17.15 package PATTERN1 PatternFunctions1

17.15.1 PatternFunctions1 (PATTERN1)



Exports:
badValues addBadValue predicate satisfy? suchThat

— package PATTERN1 PatternFunctions1 —

```
)abbrev package PATTERN1 PatternFunctions1
++ Author: Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 5 Jul 1990
++ Keywords: pattern, matching.
++ Description:
++ Utilities for handling patterns

PatternFunctions1(R:SetCategory, D:Type): with
    suchThat   : (Pattern R, D -> Boolean)       -> Pattern R
      ++ suchThat(p, f) makes a copy of p and adds the predicate
      ++ f to the copy, which is returned.
    suchThat   : (Pattern R, List(D -> Boolean)) -> Pattern R
      ++ \spad{suchThat(p, [f1,...,fn])} makes a copy of p and adds the
      ++ predicate f1 and ... and fn to the copy, which is returned.
    suchThat : (Pattern R, List Symbol, List D -> Boolean)  -> Pattern R
      ++ \spad{suchThat(p, [a1,...,an], f)} returns a copy of p with
      ++ the top-level predicate set to \spad{f(a1,...,an)}.
    predicate  : Pattern R -> (D -> Boolean)
      ++ predicate(p) returns the predicate attached to p, the
      ++ constant function true if p has no predicates attached to it.
    satisfy?   : (D, Pattern R) -> Boolean
      ++ satisfy?(v, p) returns f(v) where f is the predicate
      ++ attached to p.
    satisfy?   : (List D, Pattern R) -> Boolean
      ++ \spad{satisfy?([v1,...,vn], p)} returns \spad{f(v1,...,vn)}
      ++ where f is the
```

```
      ++ top-level predicate attached to p.
    addBadValue: (Pattern R, D) -> Pattern R
      ++ addBadValue(p, v) adds v to the list of "bad values" for p;
      ++ p is not allowed to match any of its "bad values".
    badValues  : Pattern R -> List D
      ++ badValues(p) returns the list of "bad values" for p;
      ++ p is not allowed to match any of its "bad values".
  == add
    A1D ==> AnyFunctions1(D)
    A1  ==> AnyFunctions1(D -> Boolean)
    A1L ==> AnyFunctions1(List D -> Boolean)

    applyAll: (List Any, D) -> Boolean
    st      : (Pattern R, List Any) -> Pattern R

    st(p, l)          == withPredicates(p, concat(predicates p, l))
    predicate p       == (d1:D):Boolean +-> applyAll(predicates p, d1)
    addBadValue(p, v) == addBadValue(p, coerce(v)$A1D)
    badValues p       == [retract(v)$A1D for v in getBadValues p]
    suchThat(p, l, f) == setTopPredicate(copy p, l, coerce(f)$A1L)
    suchThat(p:Pattern R, f:D -> Boolean) == st(p, [coerce(f)$A1])
    satisfy?(d:D, p:Pattern R)            == applyAll(predicates p, d)

    satisfy?(l:List D, p:Pattern R) ==
      empty?((rec := topPredicate p).var) => true
      retract(rec.pred)$A1L l

    applyAll(l, d) ==
      for f in l repeat
        not(retract(f)$A1 d) => return false
      true

    suchThat(p:Pattern R, l:List(D -> Boolean)) ==
      st(p, [coerce(f)$A1 for f in l])
```

———-

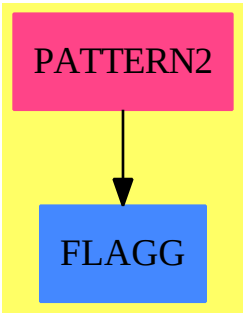
— PATTERN1.dotabb —

```
"PATTERN1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATTERN1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PATTERN1" -> "BASTYPE"
"PATTERN1" -> "KOERCE"
"PATTERN1" -> "TYPE"
```

———-

17.16 package PATTERN2 PatternFunctions2

17.16.1 PatternFunctions2 (PATTERN2)



Exports:
map

— package PATTERN2 PatternFunctions2 —

```
)abbrev package PATTERN2 PatternFunctions2
++ Author: Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 12 Jan 1990
++ Keywords: pattern, matching.
++ Description:
++ Lifts maps to patterns

PatternFunctions2(R:SetCategory, S:SetCategory): with
    map: (R -> S, Pattern R) -> Pattern S
      ++ map(f, p) applies f to all the leaves of p and
      ++ returns the result as a pattern over S.
  == add
    map(f, p) ==
      (r := (retractIfCan p)@Union(R, "failed")) case R =>
        f(r::R)::Pattern(S)
      (u := isOp p) case Record(op:BasicOperator, arg:List Pattern R) =>
        ur := u::Record(op:BasicOperator, arg:List Pattern R)
        (ur.op) [map(f, x) for x in ur.arg]
      (v := isQuotient p) case Record(num:Pattern R, den:Pattern R) =>
        vr := v::Record(num:Pattern R, den:Pattern R)
        map(f, vr.num) / map(f, vr.den)
      (l := isPlus p) case List(Pattern R) =>
        reduce("+", [map(f, x) for x in l::List(Pattern R)])
      (l := isTimes p) case List(Pattern R) =>
        reduce("*", [map(f, x) for x in l::List(Pattern R)])
      (x := isPower p) case
```

```
 Record(val:Pattern R, exponent: Pattern R) =>
  xr := x::Record(val:Pattern R, exponent: Pattern R)
  map(f, xr.val) ** map(f, xr.exponent)
(w := isExpt p) case
 Record(val:Pattern R, exponent: NonNegativeInteger) =>
  wr := w::Record(val:Pattern R, exponent: NonNegativeInteger)
  map(f, wr.val) ** wr.exponent
sy := retract(p)@Symbol
setPredicates(sy::Pattern(S), copy predicates p)
```

———-

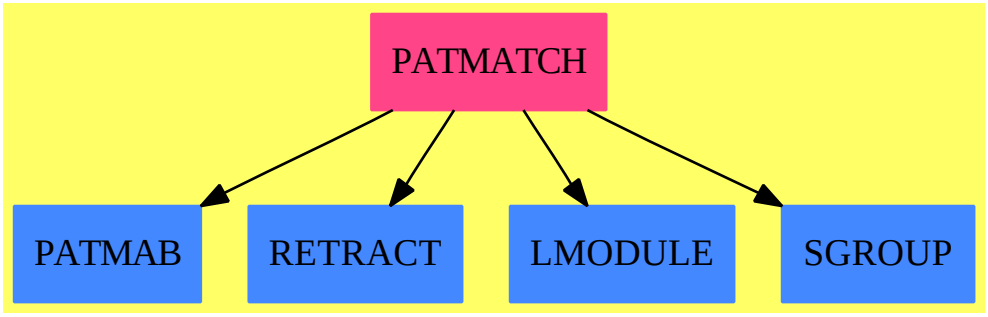
— PATTERN2.dotabb —

```
"PATTERN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATTERN2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PATTERN2" -> "FLAGG"
```

———-

17.17 package PATMATCH PatternMatch

17.17.1 PatternMatch (PATMATCH)



Exports:
is? Is

— package PATMATCH PatternMatch —

```
)abbrev package PATMATCH PatternMatch
++ Top-level pattern matching functions
++ Author: Manuel Bronstein
```

```
++ Date Created: 3 Dec 1989
++ Date Last Updated: 29 Jun 1990
++ Keywords: pattern, matching.
++ Description:
++ This package provides the top-level pattern macthing functions.

PatternMatch(Base, Subject, Pat): Exports == Implementation where
  Base   : SetCategory
  Subject: PatternMatchable Base
  Pat    : ConvertibleTo Pattern Base

  Exports ==> with
    is?: (Subject, Pat)      -> Boolean
      ++ is?(expr, pat) tests if the expression expr matches
      ++ the pattern pat.
    is?: (List Subject, Pat) -> Boolean
      ++ is?([e1,...,en], pat) tests if the list of
      ++ expressions \spad{[e1,...,en]} matches
      ++ the pattern pat.
    Is : (List Subject, Pat) ->
                    PatternMatchListResult(Base, Subject, List Subject)
      ++ Is([e1,...,en], pat) matches the pattern pat on the list of
      ++ expressions \spad{[e1,...,en]} and returns the result.
    if Subject has RetractableTo(Symbol) then
      Is: (Subject, Pat) -> List Equation Subject
        ++ Is(expr, pat) matches the pattern pat on the expression
        ++ expr and returns a list of matches \spad{[v1 = e1,...,vn = en]};
        ++ returns an empty list if either expr is exactly equal to
        ++ pat or if pat does not match expr.
    else
      if Subject has Ring then
        Is: (Subject, Pat) -> List Equation Polynomial Subject
          ++ Is(expr, pat) matches the pattern pat on the expression
          ++ expr and returns a list of matches \spad{[v1 = e1,...,vn = en]};
          ++ returns an empty list if either expr is exactly equal to
          ++ pat or if pat does not match expr.
      else
        Is: (Subject, Pat) -> PatternMatchResult(Base, Subject)
          ++ Is(expr, pat) matches the pattern pat on the expression
          ++ expr and returns a match of the form \spad{[v1 = e1,...,vn = en]};
          ++ returns an empty match if expr is exactly equal to pat.
          ++ returns a \spadfun{failed} match if pat does not match expr.

  Implementation ==> add
    import PatternMatchListAggregate(Base, Subject, List Subject)

    ist: (Subject, Pat) -> PatternMatchResult(Base, Subject)

    ist(s, p)                 == patternMatch(s, convert p, new())
    is?(s:     Subject, p:Pat) == not failed? ist(s, p)
```

```
    is?(s:List Subject, p:Pat) == not failed? Is(s, p)
    Is(s:List Subject,  p:Pat) == patternMatch(s, convert p, new())

    if Subject has RetractableTo(Symbol) then
      Is(s:Subject, p:Pat):List(Equation Subject) ==
        failed?(r := ist(s, p)) => empty()
        [rec.key::Subject = rec.entry for rec in destruct r]

    else
      if Subject has Ring then
        Is(s:Subject, p:Pat):List(Equation Polynomial Subject) ==
          failed?(r := ist(s, p)) => empty()
          [rec.key::Polynomial(Subject) =$Equation(Polynomial Subject)
           rec.entry::Polynomial(Subject) for rec in destruct r]

      else
        Is(s:Subject,p:Pat):PatternMatchResult(Base,Subject) == ist(s,p)
```

———-

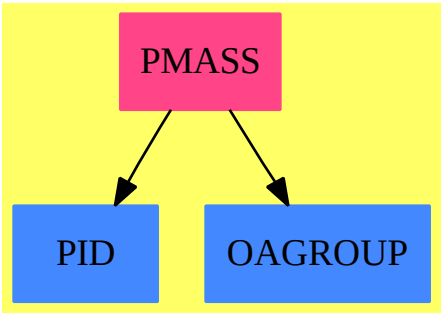
— PATMATCH.dotabb —

```
"PATMATCH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATMATCH"]
"PATMAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PATMAB"]
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"PATMATCH" -> "PATMAB"
"PATMATCH" -> "RETRACT"
"PATMATCH" -> "LMODULE"
"PATMATCH" -> "SGROUP"
```

———-

17.18 package PMASS PatternMatchAssertions

17.18.1 PatternMatchAssertions (PMASS)



Exports:
assert constant multiple optional

— package PMASS PatternMatchAssertions —

```
)abbrev package PMASS PatternMatchAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Keywords: pattern, matching.
++ Description:
++ Attaching assertions to symbols for pattern matching.

PatternMatchAssertions(): Exports == Implementation where
  FE ==> Expression Integer

  Exports ==> with
    assert  : (Symbol, String) -> FE
      ++ assert(x, s) makes the assertion s about x.
    constant: Symbol -> FE
      ++ constant(x) tells the pattern matcher that x should
      ++ match only the symbol 'x and no other quantity.
    optional: Symbol -> FE
      ++ optional(x) tells the pattern matcher that x can match
      ++ an identity (0 in a sum, 1 in a product or exponentiation).;
    multiple: Symbol -> FE
      ++ multiple(x) tells the pattern matcher that x should
      ++ preferably match a multi-term quantity in a sum or product.
      ++ For matching on lists, multiple(x) tells the pattern matcher
      ++ that x should match a list instead of an element of a list.

  Implementation ==> add
```

```
    import FunctionSpaceAssertions(Integer, FE)

    constant x   == constant(x::FE)
    multiple x   == multiple(x::FE)
    optional x   == optional(x::FE)
    assert(x, s) == assert(x::FE, s)
```

———-

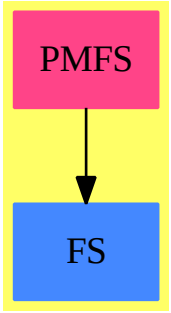
— **PMASS.dotabb** —

```
"PMASS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMASS"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PMASS" -> "PID"
"PMASS" -> "OAGROUP"
```

———-

17.19 package PMFS PatternMatchFunctionSpace

17.19.1 PatternMatchFunctionSpace (PMFS)



Exports:
patternMatch

— **package PMFS PatternMatchFunctionSpace** —

```
)abbrev package PMFS PatternMatchFunctionSpace
++ Author: Manuel Bronstein
++ Date Created: 15 Mar 1990
++ Date Last Updated: 20 June 1991
```

```
++ Keywords: pattern, matching, function, space.
++ Description:
++ This package provides pattern matching functions on function spaces.

PatternMatchFunctionSpace(S, R, F): Exports== Implementation where
  S: SetCategory
  R: Join(IntegralDomain, OrderedSet, PatternMatchable S)
  F: Join(FunctionSpace R, ConvertibleTo Pattern S, PatternMatchable S,
          RetractableTo Kernel %)  -- that one is redundant but won't
                                   -- compile without it

  N   ==> NonNegativeInteger
  K   ==> Kernel F
  PAT ==> Pattern S
  PRS ==> PatternMatchResult(S, F)
  RCP ==> Record(val:PAT, exponent:N)
  RCX ==> Record(var:K, exponent:Integer)

  Exports ==> with
    patternMatch: (F, PAT, PRS) -> PRS
      ++ patternMatch(expr, pat, res) matches the pattern pat to the
      ++ expression expr; res contains the variables of pat which
      ++ are already matched and their matches.

  Implementation ==> add
    import PatternMatchKernel(S, F)
    import PatternMatchTools(S, R, F)
    import PatternMatchPushDown(S, R, F)

    patternMatch(x, p, l) ==
      generic? p => addMatch(p, x, l)
      (r := retractIfCan(x)@Union(R, "failed")) case R =>
        patternMatch(r::R, p, l)
      (v := retractIfCan(x)@Union(K, "failed")) case K =>
        patternMatch(v::K, p, l)
      (q := isQuotient p) case Record(num:PAT, den:PAT) =>
        uq := q::Record(num:PAT, den:PAT)
        failed?(l := patternMatch(numer(x)::F, uq.num, l)) => l
        patternMatch(denom(x)::F, uq.den, l)
      (u := isPlus p) case List(PAT) =>
        (lx := isPlus x) case List(F) =>
          patternMatch(lx::List(F), u::List(PAT), l1 +-> +/l1, l, patternMatch)
        (u := optpair(u::List(PAT))) case List(PAT) =>
          failed?(l := addMatch(first(u::List(PAT)), 0, l)) => failed()
          patternMatch(x, second(u::List(PAT)), l)
        failed()
      (u := isTimes p) case List(PAT) =>
        (lx := isTimes x) case List(F) =>
          patternMatchTimes(lx::List(F), u::List(PAT), l, patternMatch)
        (u := optpair(u::List(PAT))) case List(PAT) =>
```

```
        failed?(l := addMatch(first(u::List(PAT)), 1, l)) => failed()
        patternMatch(x, second(u::List(PAT)), l)
      failed()
    (uu := isPower p) case Record(val:PAT, exponent:PAT) =>
      uur := uu::Record(val:PAT, exponent: PAT)
      (ex := isExpt x) case RCX =>
        failed?(l := patternMatch((ex::RCX).exponent::Integer::F,
                                      uur.exponent, l)) => failed()
        patternMatch((ex::RCX).var, uur.val, l)
      optional?(uur.exponent) =>
        failed?(l := addMatch(uur.exponent, 1, l)) => failed()
        patternMatch(x, uur.val, l)
      failed()
    ((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
         (ex::RCX).exponent = ((ep::RCP).exponent)::Integer =>
             patternMatch((ex::RCX).var, (ep::RCP).val, l)
    failed()
```

———-

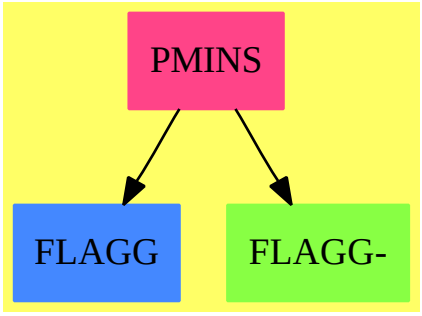
— **PMFS.dotabb** —

```
"PMFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMFS" -> "FS"
```

———-

17.20 package PMINS PatternMatchIntegerNumberSystem

17.20.1 PatternMatchIntegerNumberSystem (PMINS)



Exports:
patternMatch

— package PMINS PatternMatchIntegerNumberSystem —

```
)abbrev package PMINS PatternMatchIntegerNumberSystem
++ Author: Manuel Bronstein
++ Date Created: 29 Nov 1989
++ Date Last Updated: 22 Mar 1990
++ Keywords: pattern, matching, integer.
++ Description:
++ This package provides pattern matching functions on integers.

PatternMatchIntegerNumberSystem(I:IntegerNumberSystem): with
  patternMatch: (I, Pattern Integer, PatternMatchResult(Integer, I)) ->
                                        PatternMatchResult(Integer, I)
    ++ patternMatch(n, pat, res) matches the pattern pat to the
    ++ integer n; res contains the variables of pat which
    ++ are already matched and their matches.
 == add
   import IntegerRoots(I)

   PAT ==> Pattern Integer
   PMR ==> PatternMatchResult(Integer, I)

   patternMatchInner     : (I, PAT, PMR) -> PMR
   patternMatchRestricted: (I, PAT, PMR, I) -> PMR
   patternMatchSumProd   :
     (I, List PAT, PMR, (I, I) -> Union(I, "failed"), I) -> PMR

   patternMatch(x, p, l) ==
     generic? p => addMatch(p, x, l)
     patternMatchInner(x, p, l)

   patternMatchRestricted(x, p, l, y) ==
     generic? p => addMatchRestricted(p, x, l, y)
     patternMatchInner(x, p, l)

   patternMatchSumProd(x, lp, l, invOp, ident) ==
     #lp = 2 =>
       p2 := last lp
       if ((r:= retractIfCan(p1 := first lp)@Union(Integer,"failed"))
                          case "failed") then (p1 := p2; p2 := first lp)
       (r := retractIfCan(p1)@Union(Integer, "failed")) case "failed" =>
                                                          failed()
       (y := invOp(x, r::Integer::I)) case "failed" => failed()
       patternMatchRestricted(y::I, p2, l, ident)
     failed()
```

```
   patternMatchInner(x, p, l) ==
     constant? p =>
       (r := retractIfCan(p)@Union(Integer, "failed")) case Integer =>
         convert(x)@Integer = r::Integer => l
         failed()
       failed()
     (u := isExpt p) case Record(val:PAT,exponent:NonNegativeInteger) =>
       ur := u::Record(val:PAT, exponent:NonNegativeInteger)
       (v := perfectNthRoot(x, ur.exponent)) case "failed" => failed()
       patternMatchRestricted(v::I, ur.val, l, 1)
     (uu := isPower p) case Record(val:PAT, exponent:PAT) =>
       uur := uu::Record(val:PAT, exponent: PAT)
       pr := perfectNthRoot x
       failed?(l := patternMatchRestricted(pr.exponent::Integer::I,
                                           uur.exponent, l,1)) => failed()
       patternMatchRestricted(pr.base, uur.val, l, 1)
     (w := isTimes p) case List(PAT) =>
       patternMatchSumProd(x, w::List(PAT), l,
         (i1:I,i2:I):Union(I,"failed") +-> i1 exquo i2, 1)
     (w := isPlus p) case List(PAT) =>
      patternMatchSumProd(x,w::List(PAT),l,
        (i1:I,i2:I):Union(I,"failed") +-> (i1-i2)::Union(I,"failed"),0)
     (uv := isQuotient p) case Record(num:PAT, den:PAT) =>
       uvr := uv::Record(num:PAT, den:PAT)
       (r := retractIfCan(uvr.num)@Union(Integer,"failed")) case Integer
         and (v := r::Integer::I exquo x) case I =>
           patternMatchRestricted(v::I, uvr.den, l, 1)
       (r := retractIfCan(uvr.den)@Union(Integer,"failed")) case Integer
         => patternMatch(r::Integer * x, uvr.num, l)
       failed()
     failed()
```

———-

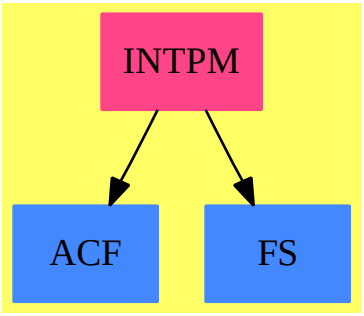
— PMINS.dotabb —

```
"PMINS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMINS"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"PMINS" -> "FLAGG"
"PMINS" -> "FLAGG-"
```

———-

17.21 package INTPM PatternMatchIntegration

17.21.1 PatternMatchIntegration (INTPM)



Exports:
pmComplexintegrate pmintegrate pmintegrate splitConstant

— package INTPM PatternMatchIntegration —

```
)abbrev package INTPM PatternMatchIntegration
++ Author: Manuel Bronstein
++ Date Created: 5 May 1992
++ Date Last Updated: 27 September 1995
++ Description:
++ \spadtype{PatternMatchIntegration} provides functions that use
++ the pattern matcher to find some indefinite and definite integrals
++ involving special functions and found in the litterature.

PatternMatchIntegration(R, F): Exports == Implementation where
  R : Join(OrderedSet, RetractableTo Integer, GcdDomain,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  N   ==> NonNegativeInteger
  Z   ==> Integer
  SY  ==> Symbol
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  SUP ==> SparseUnivariatePolynomial F
  PAT ==> Pattern Z
  RES ==> PatternMatchResult(Z, F)
  OFE ==> OrderedCompletion F
  REC ==> Record(which: Z, exponent: F, coeff: F)
  ANS ==> Record(special:F, integrand:F)
  NONE ==> 0
```

```
EI   ==> 1
ERF  ==> 2
SI   ==> 3
CI   ==> 4
GAM2 ==> 5
CI0  ==> 6

Exports ==> with
  splitConstant: (F, SY) -> Record(const:F, nconst:F)
    ++ splitConstant(f, x) returns \spad{[c, g]} such that
    ++ \spad{f = c * g} and \spad{c} does not involve \spad{t}.
  if R has ConvertibleTo Pattern Integer and
     R has PatternMatchable Integer then
       if F has LiouvillianFunctionCategory then
         pmComplexintegrate: (F, SY) -> Union(ANS, "failed")
           ++ pmComplexintegrate(f, x) returns either "failed" or
           ++ \spad{[g,h]} such that
           ++ \spad{integrate(f,x) = g + integrate(h,x)}.
           ++ It only looks for special complex integrals that pmintegrate
           ++ does not return.
         pmintegrate: (F, SY) -> Union(ANS, "failed")
           ++ pmintegrate(f, x) returns either "failed" or \spad{[g,h]} such
           ++ that \spad{integrate(f,x) = g + integrate(h,x)}.
       if F has SpecialFunctionCategory then
         pmintegrate: (F, SY, OFE, OFE) -> Union(F, "failed")
           ++ pmintegrate(f, x = a..b) returns the integral of
           ++ \spad{f(x)dx} from a to b
           ++ if it can be found by the built-in pattern matching rules.

Implementation ==> add
  import PatternMatch(Z, F, F)
  import ElementaryFunctionSign(R, F)
  import FunctionSpaceAssertions(R, F)
  import TrigonometricManipulations(R, F)
  import FunctionSpaceAttachPredicates(R, F, F)

  mkalist   : RES -> AssociationList(SY, F)

  pm := new()$SY
  pmw := new pm
  pmm := new pm
  pms := new pm
  pmc := new pm
  pma := new pm
  pmb := new pm

  c := optional(pmc::F)
  w := suchThat(optional(pmw::F),
    (x1:F):Boolean +-> empty? variables x1)
  s := suchThat(optional(pms::F),
```

```
    (x1:F):Boolean +-> empty? variables x1 and real? x1)
  m := suchThat(optional(pmm::F),
    (x1:F):Boolean+->(retractIfCan(x1)@Union(Z,"failed") case Z) and x1 >= 0)

  spi := sqrt(pi()$F)

  half := 1::F / 2::F

  mkalist res == construct destruct res

  splitConstant(f, x) ==
    not member?(x, variables f) => [f, 1]
    (retractIfCan(f)@Union(K, "failed")) case K => [1, f]
    (u := isTimes f) case List(F) =>
      cc := nc := 1$F
      for g in u::List(F) repeat
        rec := splitConstant(g, x)
        cc  := cc * rec.const
        nc  := nc * rec.nconst
      [cc, nc]
    (u := isPlus f) case List(F) =>
      rec := splitConstant(first(u::List(F)), x)
      cc  := rec.const
      nc  := rec.nconst
      for g in rest(u::List(F)) repeat
        rec := splitConstant(g, x)
        if rec.nconst = nc then cc := cc + rec.const
        else if rec.nconst = -nc then cc := cc - rec.const
        else return [1, f]
      [cc, nc]
    if (v := isPower f) case Record(val:F, exponent:Z) then
      vv := v::Record(val:F, exponent:Z)
      (vv.exponent ^= 1) =>
        rec := splitConstant(vv.val, x)
        return [rec.const ** vv.exponent, rec.nconst ** vv.exponent]
    error "splitConstant: should not happen"

  if R has ConvertibleTo Pattern Integer and
     R has PatternMatchable Integer then
       if F has LiouvillianFunctionCategory then
         import ElementaryFunctionSign(R, F)

         insqrt     : F -> F
         matchei    : (F, SY) -> REC
         matcherfei : (F, SY, Boolean) -> REC
         matchsici  : (F, SY) -> REC
         matchli    : (F, SY) -> List F
         matchli0   : (F, K, SY) -> List F
         matchdilog : (F, SY) -> List F
         matchdilog0: (F, K, SY, P, F) -> List F
```

```
          goodlilog? : (K, P) -> Boolean
          gooddilog? : (K, P, P) -> Boolean

--          goodlilog?(k, p) == is?(k, "log"::SY) and one? minimumDegree(p, k)
          goodlilog?(k, p) == is?(k, "log"::SY) and (minimumDegree(p, k) = 1)

          gooddilog?(k, p, q) ==
--            is?(k, "log"::SY) and one? degree(p, k) and zero? degree(q, k)
            is?(k, "log"::SY) and (degree(p, k) = 1) and zero? degree(q, k)

-- matches the integral to a result of the form d * erf(u) or d * ei(u)
-- returns [case, u, d]
          matcherfei(f, x, comp?) ==
            res0 := new()$RES
            pat := c * exp(pma::F)
            failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              comp? => [NONE, 0,0]
              matchei(f,x)
            l := mkalist res
            da := differentiate(a := l.pma, x)
            d := a * (cc := l.pmc) / da
            zero? differentiate(d, x) => [EI, a, d]
            comp? or (((u := sign a) case Z) and (u::Z) < 0) =>
              d := cc * (sa := insqrt(- a)) / da
              zero? differentiate(d, x) => [ERF, sa, - d * spi]
              [NONE, 0, 0]
            [NONE, 0, 0]

-- matches the integral to a result of the form d * ei(k * log u)
-- returns [case, k * log u, d]
          matchei(f, x) ==
            res0 := new()$RES
            a := pma::F
            pat := c * a**w / log a
            failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              [NONE, 0, 0]
            l := mkalist res
            da := differentiate(a := l.pma, x)
            d := (cc := l.pmc) / da
            zero? differentiate(d, x) => [EI, (1 + l.pmw) * log a, d]
            [NONE, 0, 0]

-- matches the integral to a result of the form d * dilog(u) + int(v),
-- returns [u,d,v] or []
          matchdilog(f, x) ==
            n := numer f
            df := (d := denom f)::F
            for k in select_!(
             (x1:K):Boolean +-> gooddilog?(x1,n,d),variables n)$List(K) repeat
                not empty?(l := matchdilog0(f, k, x, n, df)) => return l
```

```
            empty()

-- matches the integral to a result of the form d * dilog(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
          matchdilog0(f, k, x, p, q) ==
            zero?(da := differentiate(a := first argument k, x)) => empty()
            a1 := 1 - a
            d := coefficient(univariate(p, k), 1)::F * a1 / (q * da)
            zero? differentiate(d, x) => [a, d, f - d * da * (k::F) / a1]
            empty()

-- matches the integral to a result of the form d * li(u) + int(v),
-- returns [u,d,v] or []
          matchli(f, x) ==
            d := denom f
            for k in select_!(
              (x1:K):Boolean+->goodlilog?(x1,d), variables d)$List(K) repeat
                not empty?(l := matchli0(f, k, x)) => return l
            empty()

-- matches the integral to a result of the form d * li(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
          matchli0(f, k, x) ==
            g := (lg := k::F) * f
            zero?(da := differentiate(a := first argument k, x)) => empty()
            zero? differentiate(d := g / da, x) => [a, d, 0]
            ug := univariate(g, k)
            (u:=retractIfCan(ug)@Union(SUP,"failed")) case "failed" => empty()
            degree(p := u::SUP) > 1 => empty()
            zero? differentiate(d := coefficient(p, 0) / da, x) =>
              [a, d, leadingCoefficient p]
            empty()

-- matches the integral to a result of the form d * Si(u) or d * Ci(u)
-- returns [case, u, d]
          matchsici(f, x) ==
            res0 := new()$RES
            b := pmb::F
            t := tan(a := pma::F)
            patsi := c * t / (patden := b + b * t**2)
            patci := (c - c * t**2) / patden
            patci0 := c / patden
            ci0?:Boolean
            (ci? := failed?(res := patternMatch(f, convert(patsi)@PAT, res0)))
              and (ci0?:=failed?(res:=patternMatch(f,convert(patci)@PAT,res0)))
                and failed?(res := patternMatch(f,convert(patci0)@PAT,res0)) =>
                  [NONE, 0, 0]
            l := mkalist res
```

```
            (b := l.pmb) ^= 2 * (a := l.pma) => [NONE, 0, 0]
            db := differentiate(b, x)
            d := (cc := l.pmc) / db
            zero? differentiate(d, x) =>
              ci? =>
                 ci0? => [CI0, b, d / (2::F)]
                 [CI, b, d]
              [SI, b, d / (2::F)]
            [NONE, 0, 0]

-- returns a simplified sqrt(y)
          insqrt y ==
            rec := froot(y, 2)$PolynomialRoots(IndexedExponents K, K, R, P, F)
--            one?(rec.exponent) => rec.coef * rec.radicand
            ((rec.exponent) = 1) => rec.coef * rec.radicand
            rec.exponent ^=2 => error "insqrt: hould not happen"
            rec.coef * sqrt(rec.radicand)

          pmintegrate(f, x) ==
            (rc := splitConstant(f, x)).const ^= 1 =>
              (u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
              rec := u::ANS
              [rc.const * rec.special, rc.const * rec.integrand]
            not empty?(l := matchli(f, x)) => [second l * li first l, third l]
            not empty?(l := matchdilog(f, x)) =>
                                          [second l * dilog first l, third l]
            cse := (rec := matcherfei(f, x, false)).which
            cse = EI   => [rec.coeff * Ei(rec.exponent), 0]
            cse = ERF  => [rec.coeff * erf(rec.exponent), 0]
            cse := (rec := matchsici(f, x)).which
            cse = SI => [rec.coeff * Si(rec.exponent), 0]
            cse = CI => [rec.coeff * Ci(rec.exponent), 0]
            cse = CI0 => [rec.coeff * Ci(rec.exponent)
                          + rec.coeff * log(rec.exponent), 0]
            "failed"

          pmComplexintegrate(f, x) ==
            (rc := splitConstant(f, x)).const ^= 1 =>
              (u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
              rec := u::ANS
              [rc.const * rec.special, rc.const * rec.integrand]
            cse := (rec := matcherfei(f, x, true)).which
            cse = ERF  => [rec.coeff * erf(rec.exponent), 0]
            "failed"

         if F has SpecialFunctionCategory then
           match1    : (F, SY, F, F) -> List F
           formula1  : (F, SY, F, F) -> Union(F, "failed")

-- tries only formula (1) of the Geddes & al, AAECC 1 (1990) paper
```

```
          formula1(f, x, t, cc) ==
            empty?(l := match1(f, x, t, cc)) => "failed"
            mw := first l
            zero?(ms := third l) or ((sgs := sign ms) case "failed")=> "failed"
            ((sgz := sign(z := (mw + 1) / ms)) case "failed") or (sgz::Z < 0)
               => "failed"
            mmi := retract(mm := second l)@Z
            sgs * (last l) * ms**(- mmi - 1) *
                eval(differentiate(Gamma(x::F), x, mmi::N), [kernel(x)@K], [z])

-- returns [w, m, s, c] or []
-- matches only formula (1) of the Geddes & al, AAECC 1 (1990) paper
          match1(f, x, t, cc) ==
            res0 := new()$RES
            pat := cc * log(t)**m * exp(-t**s)
            not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              l := mkalist res
              [0, l.pmm, l.pms, l.pmc]
            pat := cc * t**w * exp(-t**s)
            not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              l := mkalist res
              [l.pmw, 0, l.pms, l.pmc]
            pat := cc / t**w * exp(-t**s)
            not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              l := mkalist res
              [- l.pmw, 0, l.pms, l.pmc]
            pat := cc * t**w * log(t)**m * exp(-t**s)
            not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              l := mkalist res
              [l.pmw, l.pmm, l.pms, l.pmc]
            pat := cc / t**w * log(t)**m * exp(-t**s)
            not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
              l := mkalist res
              [- l.pmw, l.pmm, l.pms, l.pmc]
            empty()

          pmintegrate(f, x, a, b) ==
--            zero? a and one? whatInfinity b =>
            zero? a and ((whatInfinity b) = 1) =>
              formula1(f, x, constant(x::F),
                suchThat(c, (x1:F):Boolean +-> freeOf?(x1, x)))
            "failed"
```

———-

— INTPM.dotabb —

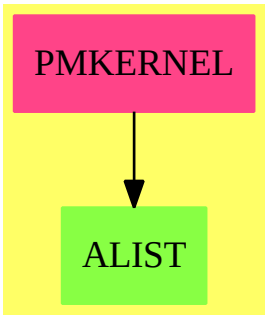
```
"INTPM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPM"]
```

```
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTPM" -> "ACF"
"INTPM" -> "FS"
```

———-

17.22 package PMKERNEL PatternMatchKernel

17.22.1 PatternMatchKernel (PMKERNEL)



Exports:
patternMatch

— package PMKERNEL PatternMatchKernel —

```
)abbrev package PMKERNEL PatternMatchKernel
++ Author: Manuel Bronstein
++ Date Created: 12 Jan 1990
++ Date Last Updated: 4 May 1992
++ Keywords: pattern, matching, kernel.
++ Description:
++ This package provides pattern matching functions on kernels.

PatternMatchKernel(S, E): Exports == Implementation where
  S: SetCategory
  E: Join(OrderedSet, RetractableTo Kernel %,
          ConvertibleTo Pattern S, PatternMatchable S)

  PAT ==> Pattern S
  PRS ==> PatternMatchResult(S, E)
  POWER ==> "%power"::Symbol
  NTHRT ==> "nthRoot"::Symbol
```

```
Exports ==> with
  patternMatch: (Kernel E, PAT, PRS) -> PRS
    ++ patternMatch(f(e1,...,en), pat, res) matches the pattern pat
    ++ to \spad{f(e1,...,en)}; res contains the variables of pat which
    ++ are already matched and their matches.

Implementation ==> add
  patternMatchArg  : (List E, List PAT, PRS) -> PRS
  patternMatchInner: (Kernel E, PAT, PRS) -> Union(PRS, "failed")

  -- matches the ordered lists ls and lp.
  patternMatchArg(ls, lp, l) ==
    #ls ^= #lp => failed()
    for p in lp for s in ls repeat
      generic? p and failed?(l := addMatch(p,s,l)) => return failed()
    for p in lp for s in ls repeat
      not(generic? p) and failed?(l := patternMatch(s, p, l)) =>
                                                    return failed()
    l

  patternMatchInner(s, p, l) ==
    generic? p => addMatch(p, s::E, l)
    (u := isOp p) case Record(op:BasicOperator, arg: List PAT) =>
      ur := u::Record(op:BasicOperator, arg: List PAT)
      ur.op = operator s => patternMatchArg(argument s, ur.arg, l)
      failed()
    constant? p =>
      ((v := retractIfCan(p)@Union(Symbol, "failed")) case Symbol)
        and ((w := symbolIfCan s) case Symbol) and
          (v::Symbol = w::Symbol) => l
      failed()
    "failed"

  if E has Monoid then
    patternMatchMonoid: (Kernel E, PAT, PRS) -> Union(PRS, "failed")
    patternMatchOpt   : (E, List PAT, PRS, E) -> PRS

    patternMatchOpt(x, lp, l, id) ==
      (u := optpair lp) case List(PAT) =>
        failed?(l := addMatch(first(u::List(PAT)), id, l)) => failed()
        patternMatch(x, second(u::List(PAT)), l)
      failed()

    patternMatchMonoid(s, p, l) ==
      (u := patternMatchInner(s, p, l)) case PRS => u::PRS
      (v := isPower p) case Record(val:PAT, exponent:PAT) =>
        vr := v::Record(val:PAT, exponent: PAT)
        is?(op := operator s, POWER) =>
          patternMatchArg(argument s, [vr.val, vr.exponent], l)
        is?(op,NTHRT) and ((r := recip(second(arg := argument s))) case E) =>
```

```
            patternMatchArg([first arg, r::E], [vr.val, vr.exponent], l)
          optional?(vr.exponent) =>
            failed?(l := addMatch(vr.exponent, 1, l)) => failed()
            patternMatch(s::E, vr.val, l)
          failed()
        (w := isTimes p) case List(PAT) =>
          patternMatchOpt(s::E, w::List(PAT), l, 1)
        "failed"

      if E has AbelianMonoid then
        patternMatch(s, p, l) ==
          (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
          (w := isPlus p) case List(PAT) =>
            patternMatchOpt(s::E, w::List(PAT), l, 0)
          failed()

      else
        patternMatch(s, p, l) ==
          (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
          failed()

    else
      patternMatch(s, p, l) ==
        (u := patternMatchInner(s, p, l)) case PRS => u::PRS
        failed()
```

———-

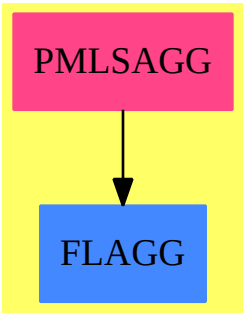
— PMKERNEL.dotabb —

```
"PMKERNEL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMKERNEL"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PMKERNEL" -> "ALIST"
```

———-

17.23 package PMLSAGG PatternMatchListAggregate

17.23.1 PatternMatchListAggregate (PMLSAGG)



Exports:
patternMatch

— package PMLSAGG PatternMatchListAggregate —

```
)abbrev package PMLSAGG PatternMatchListAggregate
++ Author: Manuel Bronstein
++ Date Created: 4 Dec 1989
++ Date Last Updated: 29 Jun 1990
++ Keywords: pattern, matching, list.
++ Description:
++ This package provides pattern matching functions on lists.

PatternMatchListAggregate(S, R, L): Exports == Implementation where
  S: SetCategory
  R: PatternMatchable S
  L: ListAggregate R

  PLR ==> PatternMatchListResult(S, R, L)

  Exports ==> with
    patternMatch: (L, Pattern S, PLR) -> PLR
      ++ patternMatch(l, pat, res) matches the pattern pat to the
      ++ list l; res contains the variables of pat which
      ++ are already matched and their matches.

  Implementation ==> add
    match: (L, List Pattern S, PLR, Boolean) -> PLR

    patternMatch(l, p, r) ==
      (u := isList p) case "failed" => failed()
      match(l, u::List Pattern S, r, true)
```

```
    match(l, lp, r, new?) ==
      empty? lp =>
        empty? l => r
        failed()
      multiple?(p0 := first lp) =>
        empty? rest lp =>
          if not new? then l := reverse_! l
          makeResult(atoms r, addMatchRestricted(p0,l,lists r,empty()))
        new? => match(reverse l, reverse lp, r, false)
        error "Only one multiple pattern allowed in list"
      empty? l => failed()
      failed?(r := makeResult(patternMatch(first l,p0,atoms r),lists r))
                                                          => failed()
      match(rest l, rest lp, r, new?)
```

———-

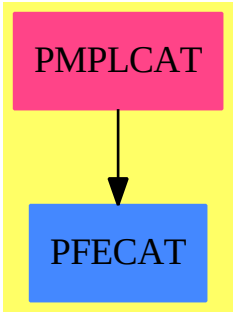
— **PMLSAGG.dotabb** —

```
"PMLSAGG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMLSAGG"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMLSAGG" -> "FLAGG"
```

———-

17.24 package PMPLCAT PatternMatchPolynomialCategory

17.24.1 PatternMatchPolynomialCategory (PMPLCAT)



Exports:
patternMatch

— package PMPLCAT PatternMatchPolynomialCategory —

```
)abbrev package PMPLCAT PatternMatchPolynomialCategory
++ Author: Manuel Bronstein
++ Date Created: 9 Jan 1990
++ Date Last Updated: 20 June 1991
++ Keywords: pattern, matching, polynomial.
++ Description:
++ This package provides pattern matching functions on polynomials.

PatternMatchPolynomialCategory(S,E,V,R,P):Exports== Implementation where
  S: SetCategory
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: Join(Ring, OrderedSet, PatternMatchable S)
  P: Join(PolynomialCategory(R, E, V), ConvertibleTo Pattern S)

  N   ==> NonNegativeInteger
  PAT ==> Pattern S
  PRS ==> PatternMatchResult(S, P)
  RCP ==> Record(val:PAT, exponent:N)
  RCX ==> Record(var:V, exponent:N)

  Exports ==> with
    patternMatch: (P, PAT, PRS, (V, PAT, PRS) -> PRS) -> PRS
      ++ patternMatch(p, pat, res, vmatch) matches the pattern pat to
      ++ the polynomial p. res contains the variables of pat which
      ++ are already matched and their matches; vmatch is the matching
      ++ function to use on the variables.
      -- This can be more efficient than pushing down when the variables
      -- are recursive over P (e.g. kernels)
    if V has PatternMatchable S then
      patternMatch: (P, PAT, PRS) -> PRS
        ++ patternMatch(p, pat, res) matches the pattern pat to
        ++ the polynomial p; res contains the variables of pat which
        ++ are already matched and their matches.

  Implementation ==> add
    import PatternMatchTools(S, R, P)
    import PatternMatchPushDown(S, R, P)

    if V has PatternMatchable S then
      patternMatch(x, p, l) ==
        patternMatch(x, p, l, patternMatch$PatternMatchPushDown(S,V,P))

    patternMatch(x, p, l, vmatch) ==
```

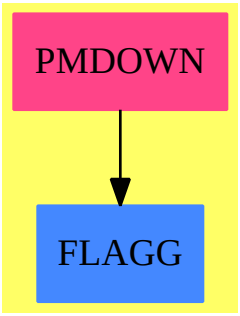
```
generic? p => addMatch(p, x, l)
(r := retractIfCan(x)@Union(R, "failed")) case R =>
  patternMatch(r::R, p, l)
(v := retractIfCan(x)@Union(V, "failed")) case V =>
  vmatch(v::V, p, l)
(u := isPlus p) case List(PAT) =>
  (lx := isPlus x) case List(P) =>
    patternMatch(lx::List(P), u::List(PAT),
      (l1:List(P)):P +-> +/l1, l,
        (p1:P,p2:PAT,p3:PRS):PRS +-> patternMatch(p1, p2, p3, vmatch))
  (u := optpair(u::List(PAT))) case List(PAT) =>
    failed?(l := addMatch(first(u::List(PAT)), 0, l)) => failed()
    patternMatch(x, second(u::List(PAT)), l, vmatch)
  failed()
(u := isTimes p) case List(PAT) =>
  (lx := isTimes x) case List(P) =>
    patternMatchTimes(lx::List(P), u::List(PAT), l,
      (p1:P,p2:PAT,p3:PRS):PRS +-> patternMatch(p1, p2, p3, vmatch))
  (u := optpair(u::List(PAT))) case List(PAT) =>
    failed?(l := addMatch(first(u::List(PAT)), 1, l)) => failed()
    patternMatch(x, second(u::List(PAT)), l, vmatch)
  failed()
(uu := isPower p) case Record(val:PAT, exponent:PAT) =>
  uur := uu::Record(val:PAT, exponent: PAT)
  (ex := isExpt x) case RCX =>
    failed?(l := patternMatch((ex::RCX).exponent::Integer::P,
                              uur.exponent, l, vmatch)) => failed()
    vmatch((ex::RCX).var, uur.val, l)
  optional?(uur.exponent) =>
    failed?(l := addMatch(uur.exponent, 1, l)) => failed()
    patternMatch(x, uur.val, l, vmatch)
  failed()
((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
     (ex::RCX).exponent = (ep::RCP).exponent =>
         vmatch((ex::RCX).var, (ep::RCP).val, l)
failed()
```

— PMPLCAT.dotabb —

```
"PMPLCAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPLCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PMPLCAT" -> "PFECAT"
```

17.25 package PMDOWN PatternMatchPushDown

17.25.1 PatternMatchPushDown (PMDOWN)



Exports:
fixPredicate patternMatch

— package PMDOWN PatternMatchPushDown —

```
)abbrev package PMDOWN PatternMatchPushDown
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 16 August 1995
++ Keywords: pattern, matching, quotient, field.
++ Description:
++ This packages provides tools for matching recursively in type towers.

PatternMatchPushDown(S, A, B): Exports == Implementation where
  S: SetCategory
  A: PatternMatchable S
  B: Join(SetCategory, RetractableTo A)

  PAT ==> Pattern S
  PRA ==> PatternMatchResult(S, A)
  PRB ==> PatternMatchResult(S, B)
  REC ==> Record(pat:PAT, res:PRA)

  Exports ==> with
    fixPredicate: (B -> Boolean) -> (A -> Boolean)
      ++ fixPredicate(f) returns g defined by g(a) = f(a::B);
    patternMatch: (A, PAT, PRB)  -> PRB
      ++ patternMatch(expr, pat, res) matches the pattern pat to the
      ++ expression expr; res contains the variables of pat which
      ++ are already matched and their matches.
      ++ Note that this function handles type towers by changing the predicates
      ++ and calling the matching function provided by \spad{A}.
```

```
Implementation ==> add
  import PatternMatchResultFunctions2(S, A, B)

  fixPred      : Any -> Union(Any, "failed")
  inA          : (PAT, PRB) -> Union(List A, "failed")
  fixPredicates: (PAT, PRB, PRA) -> Union(REC, "failed")
  fixList:(List PAT -> PAT, List PAT, PRB, PRA) -> Union(REC,"failed")

  fixPredicate f == (a1:A):Boolean +-> f(a1::B)

  patternMatch(a, p, l) ==
    (u := fixPredicates(p, l, new())) case "failed" => failed()
    union(l, map((a1:A):B +->a1::B,
              patternMatch(a, (u::REC).pat, (u::REC).res)))

  inA(p, l) ==
    (u := getMatch(p, l)) case "failed" => empty()
    (r := retractIfCan(u::B)@Union(A, "failed")) case A => [r::A]
    "failed"

  fixList(fn, l, lb, la) ==
    ll:List(PAT) := empty()
    for x in l repeat
      (f := fixPredicates(x, lb, la)) case "failed" => return "failed"
      ll := concat((f::REC).pat, ll)
      la := (f::REC).res
    [fn ll, la]

  fixPred f ==
    (u:= retractIfCan(f)$AnyFunctions1(B -> Boolean)) case "failed" =>
                                                          "failed"
    g := fixPredicate(u::(B -> Boolean))
    coerce(g)$AnyFunctions1(A -> Boolean)

  fixPredicates(p, lb, la) ==
    (r:=retractIfCan(p)@Union(S,"failed")) case S or quoted? p =>[p,la]
    (u := isOp p) case Record(op:BasicOperator, arg:List PAT) =>
      ur := u::Record(op:BasicOperator, arg:List PAT)
      fixList((l1:List(PAT)):PAT+-> (ur.op) l1, ur.arg, lb, la)
    (us := isPlus p) case List(PAT) =>
      fixList((l1:List(PAT)):PAT +-> reduce("+", l1), us::List(PAT), lb, la)
    (us := isTimes p) case List(PAT) =>
      fixList((l1:List(PAT)):PAT+->reduce("*", l1), us::List(PAT), lb, la)
    (v := isQuotient p) case Record(num:PAT, den:PAT) =>
      vr := v::Record(num:PAT, den:PAT)
      (fn := fixPredicates(vr.num, lb, la)) case "failed" => "failed"
      la  := (fn::REC).res
      (fd := fixPredicates(vr.den, lb, la)) case "failed" => "failed"
      [(fn::REC).pat / (fd::REC).pat, (fd::REC).res]
```

```
      (w:= isExpt p) case Record(val:PAT,exponent:NonNegativeInteger) =>
        wr := w::Record(val:PAT, exponent: NonNegativeInteger)
        (f := fixPredicates(wr.val, lb, la)) case "failed" => "failed"
        [(f::REC).pat ** wr.exponent, (f::REC).res]
      (uu := isPower p) case Record(val:PAT, exponent:PAT) =>
        uur := uu::Record(val:PAT, exponent: PAT)
        (fv := fixPredicates(uur.val, lb, la)) case "failed" => "failed"
        la  := (fv::REC).res
        (fe := fixPredicates(uur.exponent, lb, la)) case "failed" =>
          "failed"
        [(fv::REC).pat ** (fe::REC).pat, (fe::REC).res]
      generic? p =>
        (ua := inA(p, lb)) case "failed" => "failed"
        lp := [if (h := fixPred g) case Any then h::Any else
                        return "failed" for g in predicates p]$List(Any)
        q := setPredicates(patternVariable(retract p, constant? p,
                                           optional? p, multiple? p), lp)
        [q, (empty?(ua::List A) => la; insertMatch(q,first(ua::List A), la))]
      error "Should not happen"
```

———-

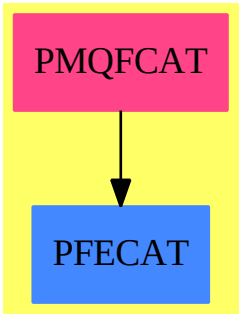
— **PMDOWN.dotabb** —

```
"PMDOWN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMDOWN"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMDOWN" -> "FLAGG"
```

———-

17.26 package PMQFCAT PatternMatchQuotientFieldCategory

17.26.1 PatternMatchQuotientFieldCategory (PMQFCAT)



Exports:
patternMatch

— **package PMQFCAT PatternMatchQuotientFieldCategory** —

```
)abbrev package PMQFCAT PatternMatchQuotientFieldCategory
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 20 June 1991
++ Keywords: pattern, matching, quotient, field.
++ Description:
++ This package provides pattern matching functions on quotients.

PatternMatchQuotientFieldCategory(S,R,Q):Exports == Implementation where
  S: SetCategory
  R: Join(IntegralDomain, PatternMatchable S, ConvertibleTo Pattern S)
  Q: QuotientFieldCategory R

  PAT ==> Pattern S
  PRQ ==> PatternMatchResult(S, Q)

  Exports ==> with
    patternMatch: (Q, PAT, PRQ) -> PRQ
      ++ patternMatch(a/b, pat, res) matches the pattern pat to the
      ++ quotient a/b; res contains the variables of pat which
      ++ are already matched and their matches.

  Implementation ==> add
    import PatternMatchPushDown(S, R, Q)
```

```
patternMatch(x, p, l) ==
  generic? p => addMatch(p, x, l)
  (r := retractIfCan x)@Union(R, "failed") case R =>
    patternMatch(r::R, p, l)
  (u := isQuotient p) case Record(num:PAT, den:PAT) =>
    ur := u::Record(num:PAT, den:PAT)
    failed?(l := patternMatch(numer x, ur.num, l)) => l
    patternMatch(denom x, ur.den, l)
  failed()
```

———-

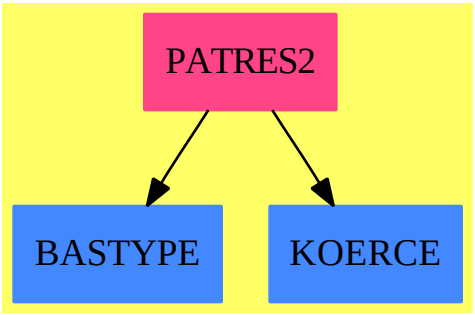
— **PMQFCAT.dotabb** —

```
"PMQFCAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMQFCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PMQFCAT" -> "PFECAT"
```

———-

17.27 package PATRES2 PatternMatchResultFunctions2

17.27.1 PatternMatchResultFunctions2 (PATRES2)



Exports:
map

— **package PATRES2 PatternMatchResultFunctions2** —

```
)abbrev package PATRES2 PatternMatchResultFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
```

```
++ Date Last Updated: 14 Dec 1989
++ Keywords: pattern, matching.
++ Description:
++ Lifts maps to pattern matching results.

PatternMatchResultFunctions2(R, A, B): Exports == Implementation where
  R: SetCategory
  A: SetCategory
  B: SetCategory

  Exports ==> with
    map: (A -> B, PatternMatchResult(R, A)) -> PatternMatchResult(R, B)
      ++ map(f, [(v1,a1),...,(vn,an)]) returns the matching result
      ++ [(v1,f(a1)),...,(vn,f(an))].

  Implementation ==> add
    map(f, r) ==
      failed? r => failed()
      construct [[rec.key, f(rec.entry)] for rec in destruct r]
```

———-

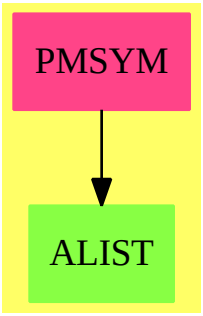
— PATRES2.dotabb —

```
"PATRES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATRES2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"PATRES2" -> "BASTYPE"
"PATRES2" -> "KOERCE"
```

———-

17.28 package PMSYM PatternMatchSymbol

17.28.1 PatternMatchSymbol (PMSYM)



Exports:
patternMatch

— **package PMSYM PatternMatchSymbol** —

```
)abbrev package PMSYM PatternMatchSymbol
++ Author: Manuel Bronstein
++ Date Created: 9 Jan 1990
++ Date Last Updated: 20 June 1991
++ Keywords: pattern, matching, symbol.
++ Description:
++ This package provides pattern matching functions on symbols.

PatternMatchSymbol(S:SetCategory): with
  patternMatch: (Symbol, Pattern S, PatternMatchResult(S, Symbol)) ->
                                          PatternMatchResult(S, Symbol)
    ++ patternMatch(expr, pat, res) matches the pattern pat to the
    ++ expression expr; res contains the variables of pat which
    ++ are already matched and their matches (necessary for recursion).
 == add
  import TopLevelPatternMatchControl

  patternMatch(s, p, l) ==
    generic? p  => addMatch(p, s, l)
    constant? p =>
      ((u := retractIfCan(p)@Union(Symbol, "failed")) case Symbol)
        and (u::Symbol) = s => l
      failed()
    failed()
```

———-

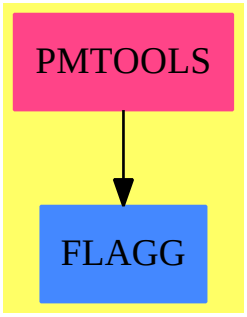
— PMSYM.dotabb —

```
"PMSYM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMSYM"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PMSYM" -> "ALIST"
```

———-

17.29 package PMTOOLS PatternMatchTools

17.29.1 PatternMatchTools (PMTOOLS)



Exports:
patternMatch patternMatchTimes

— package PMTOOLS PatternMatchTools —

```
)abbrev package PMTOOLS PatternMatchTools
++ Author: Manuel Bronstein
++ Date Created: 13 Mar 1990
++ Date Last Updated: 4 February 1992
++ Keywords: pattern, matching, tools.
++ Description:
++ This package provides tools for the pattern matcher.

PatternMatchTools(S, R, P): Exports == Implementation where
  S: SetCategory
  R: Join(Ring, OrderedSet)
  P: Join(Ring, ConvertibleTo Pattern S, RetractableTo R)

  PAT ==> Pattern S
  PRS ==> PatternMatchResult(S, P)
```

```
  REC ==> Record(res:PRS, s:List P)
  RC  ==> Record(pat:List PAT, s:List P)

  Exports ==> with
    patternMatch: (List P, List PAT, List P -> P, PRS,
                                          (P, PAT, PRS) -> PRS) -> PRS
      ++ patternMatch(lsubj, lpat, op, res, match) matches the list
      ++ of patterns lpat to the list of subjects lsubj, allowing for
      ++ commutativity; op is the operator such that op(lpat) should
      ++ match op(lsubj) at the end, r contains the previous matches,
      ++ and match is a pattern-matching function on P.
    patternMatchTimes: (List P, List PAT, PRS,
                                          (P, PAT, PRS) -> PRS) -> PRS
      ++ patternMatchTimes(lsubj, lpat, res, match) matches the
      ++ product of patterns \spad{reduce(*,lpat)}
      ++ to the product of subjects \spad{reduce(*,lsubj)};
      ++ r contains the previous matches
      ++ and match is a pattern-matching function on P.

  Implementation ==> add
    import PatternFunctions1(S, P)

    preprocessList: (PAT, List P, PRS) -> Union(List P, "failed")
    selBestGen    : List PAT -> List PAT
    negConstant   : List P -> Union(P, "failed")
    findMatch     : (PAT, List P, PRS, P, (P, PAT, PRS) -> PRS) -> REC
    tryToMatch    : (List PAT, REC, P, (P, PAT, PRS) -> PRS) ->
                                                  Union(REC, "failed")
    filterMatchedPatterns: (List PAT, List P, PRS) -> Union(RC, "failed")

    mn1 := convert(-1::P)@Pattern(S)

    negConstant l ==
      for x in l repeat
        ((r := retractIfCan(x)@Union(R, "failed")) case R) and
          (r::R < 0) => return x
      "failed"

-- tries to match the list of patterns lp to the list of subjects rc.s
-- with rc.res being the list of existing matches.
-- updates rc with the new result and subjects still to match
    tryToMatch(lp, rc, ident, pmatch) ==
      rec:REC := [l := rc.res, ls := rc.s]
      for p in lp repeat
        rec := findMatch(p, ls, l, ident, pmatch)
        failed?(l := rec.res) => return "failed"
        ls := rec.s
      rec

-- handles -1 in the pattern list.
```

```
    patternMatchTimes(ls, lp, l, pmatch) ==
      member?(mn1, lp) =>
        (u := negConstant ls) case "failed" => failed()
        if (u::P ^= -1::P) then ls := concat(-u::P, ls)
        patternMatch(remove(u::P,ls), remove(mn1,lp),
          (l1:List(P)):P +-> */l1, l, pmatch)
      patternMatch(ls, lp, (l1:List(P)):P +-> */l1, l, pmatch)

-- finds a match for p in ls, try not to match to a "bad" value
    findMatch(p, ls, l, ident, pmatch) ==
      bad:List(P) :=
        generic? p => setIntersection(badValues p, ls)
        empty()
      l1:PRS := failed()
      for x in setDifference(ls, bad)
        while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
      failed? l1 =>
        for x in bad
          while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
        failed? l1 => [addMatchRestricted(p, ident, l, ident), ls]
        [l1, remove(t, ls)]
      [l1, remove(t, ls)]

-- filters out pattern if it's generic and already matched.
    preprocessList(pattern, ls, l) ==
      generic? pattern =>
        (u := getMatch(pattern, l)) case P =>
          member?(u::P, ls) => [u::P]
          "failed"
        empty()
      empty()

-- take out already matched generic patterns
    filterMatchedPatterns(lp, ls, l) ==
      for p in lp repeat
        (rc := preprocessList(p, ls, l)) case "failed" => return "failed"
        if not empty?(rc::List(P)) then
          lp := remove(p,  lp)
          ls := remove(first(rc::List(P)), ls)
      [lp, ls]

-- select a generic pattern with no predicate if possible
    selBestGen l ==
      ans := empty()$List(PAT)
      for p in l | generic? p repeat
        ans := [p]
        not hasPredicate? p => return ans
      ans

-- matches unordered lists ls and lp
```

```
patternMatch(ls, lp, op, l, pmatch) ==
  ident := op empty()
  (rc := filterMatchedPatterns(lp, ls, l)) case "failed" => return failed()
  lp := (rc::RC).pat
  ls := (rc::RC).s
  empty? lp => l
  #(lpm := select(optional?, lp)) > 1 =>
    error "More than one optional pattern in sum/product"
  (#ls + #lpm) < #lp => failed()
  if (not empty? lpm) and (#ls + 1 = #lp) then
    lp := remove(first lpm, lp)
    failed?(l := addMatch(first lpm, ident, l)) => return l
  #(lpm := select(multiple?, lp)) > 1 =>
    error "More than one expandable pattern in sum/product"
  #ls > #lp and empty? lpm and empty?(lpm := selBestGen lp) =>
    failed()
  if not empty? lpm then lp := remove(first lpm, lp)
  -- this is the order in which we try to match predicates
  -- l1 = constant patterns (i.e. 'x, or sin('x))
  l1 := select(constant?, lp)
  -- l2 = patterns with a predicate attached to them
  l2 := select((p1:PAT):Boolean+->hasPredicate? p1 and not constant? p1,lp)
  -- l3 = non-generic patterns without predicates
  l3 := sort_!((z1:PAT,z2:PAT):Boolean+->depth(z1) > depth(z2),
    select((p2:PAT):Boolean+->not(hasPredicate? p2
               or generic? p2 or constant? p2),lp))
  -- l4 = generic patterns with predicates
  l4 := select((p1:PAT):Boolean +-> generic? p1 and
                          not(hasPredicate? p1 or constant? p1), lp)
  rec:REC := [l, ls]
  (u := tryToMatch(l1, rec, ident, pmatch)) case "failed" =>
    failed()
  (u := tryToMatch(l2, u::REC, ident, pmatch)) case "failed" =>
    failed()
  (u := tryToMatch(l3, u::REC, ident, pmatch)) case "failed" =>
    failed()
  rec := u::REC
  (rc := filterMatchedPatterns(l4,rec.s,rec.res)) case "failed" => failed()
  rec := [rec.res, (rc::RC).s]
  (u := tryToMatch((rc::RC).pat,rec,ident,pmatch)) case "failed" => failed()
  rec := u::REC
  l := rec.res
  ls := rec.s
  empty? lpm =>
    empty? ls => l
    failed()
  addMatch(first lpm, op ls, l)
```

———-

— **PMTOOLS.dotabb** —

```
"PMTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMTOOLS"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMTOOLS" -> "FLAGG"
```

———-

17.30 package PERMAN Permanent

— **Permanent.input** —

```
)set break resume
)spool Permanent.output
)set message test on
)set message auto off
)clear all
--S 1 of 3
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
    r.i.i := 0
  r
--R
--R                                                                    Type: Void
--E 1

--S 2 of 3
permanent(kn(5) :: SQMATRIX(5,INT))
--R
--R   Compiling function kn with type PositiveInteger -> Matrix(Integer)
--R
--R   (2)  44
--R                                                       Type: PositiveInteger
--E 2

--S 3 of 3
[permanent(kn(n) :: SQMATRIX(n,INT)) for n in 1..13]
--R
--R   Cannot compile conversion for types involving local variables. In
--R      particular, could not compile the expression involving ::
--R      SQMATRIX(n,INT)
--R   AXIOM will attempt to step through and interpret the code.
--R
--R   (3)
```

```
--R   [0,1,2,9,44,265,1854,14833,133496,1334961,14684570,176214841,2290792932]
--R                                                 Type: List(NonNegativeInteger)
--E 3
)spool
)lisp (bye)
```

———-

— Permanent.help —

```
========================================================================
Permanent examples
========================================================================

The package Permanent provides the function permanent for square
matrices.  The permanent of a square matrix can be computed in the
same way as the determinant by expansion of minors except that for the
permanent the sign for each element is 1, rather than being 1 if the
row plus column indices is positive and -1 otherwise.  This function
is much more difficult to compute efficiently than the determinant.
An example of the use of permanent is the calculation of the n-th
derangement number, defined to be the number of different possibilities
for n couples to dance but never with their own spouse.

Consider an n by n matrix with entries 0 on the diagonal and 1 elsewhere.
Think of the rows as one-half of each couple (for example, the males) and
the columns the other half.  The permanent of such a matrix gives the
desired derangement number.

  kn n ==
    r : MATRIX INT := new(n,n,1)
    for i in 1..n repeat
      r.i.i := 0
    r

Here are some derangement numbers, which you see grow quite fast.

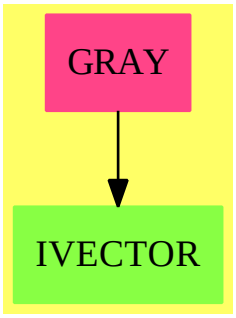
  permanent(kn(5) :: SQMATRIX(5,INT))

  [permanent(kn(n) :: SQMATRIX(n,INT)) for n in 1..13]

See Also:
o )show Permanent
```

———-

17.30.1 Permanent (PERMAN)



Exports:
permanent

— package PERMAN Permanent —

```
)abbrev package PERMAN Permanent
++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ Basic Operations: permanent
++ Related Constructors: GrayCode
++ Also See: MatrixLinearAlgebraFunctions
++ AMS Classifications:
++ Keywords: permanent
++ References:
++  Henryk Minc: Evaluation of Permanents,
++    Proc. of the Edinburgh Math. Soc.(1979), 22/1 pp 27-32.
++  Nijenhuis and Wilf : Combinatorical Algorithms, Academic
++    Press, New York 1978.
++  S.G.Williamson, Combinatorics for Computer Science,
++    Computer Science Press, 1985.
++ Description:
++ Permanent implements the functions permanent, the
++ permanent for square matrices.

Permanent(n : PositiveInteger, R : Ring with commutative("*")):
 public == private where
  I  ==> Integer
  L  ==> List
  V  ==> Vector
  SM  ==> SquareMatrix(n,R)
  VECTPKG1 ==> VectorPackage1(I)
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  GRAY ==> GrayCode
```

```
public ==> with

  permanent:  SM  -> R
    ++ permanent(x) computes the permanent of a square matrix x.
    ++ The permanent is equivalent to
    ++ the \spadfun{determinant} except that coefficients have
    ++ no change of sign. This function
    ++ is much more difficult to compute than the
    ++ determinant. The formula used is by H.J. Ryser,
    ++ improved by [Nijenhuis and Wilf, Ch. 19].
    ++ Note that permanent(x) choose one of three algorithms, depending
    ++ on the underlying ring R and on n, the number of rows (and
    ++ columns) of x:\br
    ++ if 2 has an inverse in R we can use the algorithm of
    ++ [Nijenhuis and Wilf, ch.19,p.158]; if 2 has no inverse,
    ++ some modifications are necessary:\br
    ++ if n > 6 and R is an integral domain with characteristic
    ++ different from 2 (the algorithm works if and only 2 is not a
    ++ zero-divisor of R and characteristic()$R ^= 2,
    ++ but how to check that for any given R ?),
    ++ the local function permanent2 is called;\br
    ++ else, the local function permanent3 is called
    ++ (works for all commutative rings R).

private ==> add

  -- local functions:

  permanent2:  SM  -> R

  permanent3:  SM  -> R

  x : SM
  a,b : R
  i,j,k,l : I

  permanent3(x) ==
    -- This algorithm is based upon the principle of inclusion-
    -- exclusion. A Gray-code is used to generate the subsets of
    -- 1,... ,n. This reduces the number of additions needed in
    -- every step.
    sgn : R := 1
    k : R
    a := 0$R
    vv : V V I := firstSubsetGray(n)$GRAY
      -- For the meaning of the elements of vv, see GRAY.
    w : V R := new(n,0$R)
    j := 1   -- Will be the number of the element changed in subset
    while j ^= (n+1) repeat  -- we sum over all subsets of (1,...,n)
```

```
    sgn := -sgn
    b := sgn
    if vv.1.j = 1 then k := -1
    else k := 1  -- was that element deleted(k=-1) or added(k=1)?
    for i in 1..(n::I) repeat
      w.i :=  w.i +$R k *$R  x(i,j)
      b := b *$R w.i
    a := a +$R b
    vv := nextSubsetGray(vv,n)$GRAY
    j := vv.2.1
  if odd?(n) then a := -a
  a


permanent(x) ==
  -- If 2 has an inverse in R, we can spare half of the calcu-
  -- lation needed in "permanent3": This is the algorithm of
  -- [Nijenhuis and Wilf, ch.19,p.158]
  n = 1 => x(1,1)
  two : R := (2:I) :: R
  half : Union(R,"failed") := recip(two)
  if (half case "failed") then
    if n < 7 then return permanent3(x)
    else return permanent2(x)
  sgn : R := 1
  a := 0$R
  w : V R := new(n,0$R)
  -- w.i will be at first x.i and later lambda.i in
  -- [Nijenhuis and Wilf, p.158, (24a) resp.(26)].
  rowi : V R := new(n,0$R)
  for i in 1..n repeat
    rowi := row(x,i) :: V R
    b := 0$R
    for j in 1..n repeat
      b := b + rowi.j
    w.i := rowi(n) - (half*b)$R
  vv : V V I := firstSubsetGray((n-1): PI)$GRAY
   -- For the meaning of the elements of vv, see GRAY.
  n :: I
  b := 1
  for i in 1..n repeat
    b := b * w.i
  a := a+b
  j := 1   -- Will be the number of the element changed in subset
  while j ^= n repeat  -- we sum over all subsets of (1,...,n-1)
    sgn := -sgn
    b := sgn
    if vv.1.j = 1 then k := -1
    else k := 1  -- was that element deleted(k=-1) or added(k=1)?
    for i in 1..n repeat
```

```
      w.i :=  w.i +$R k *$R  x(i,j)
      b := b *$R w.i
    a := a +$R b
    vv := nextSubsetGray(vv,(n-1) : PI)$GRAY
    j := vv.2.1
  if not odd?(n) then a := -a
  two * a

permanent2(x) ==
  c : R := 0
  sgn : R := 1
  if (not (R has IntegralDomain))
    -- or (characteristic()$R = (2:NNI))
    -- compiler refuses to compile the line above !!
    or  (sgn + sgn = c)
  then return permanent3(x)
  -- This is a slight modification of permanent which is
  -- necessary if 2 is not zero or a zero-divisor in R, but has
  -- no inverse in R.
  n = 1 => x(1,1)
  two : R := (2:I) :: R
  a := 0$R
  w : V R := new(n,0$R)
  -- w.i will be at first x.i and later lambda.i in
  -- [Nijenhuis and Wilf, p.158, (24a) resp.(26)].
  rowi : V R := new(n,0$R)
  for i in 1..n repeat
    rowi := row(x,i) :: V R
    b := 0$R
    for j in 1..n repeat
      b := b + rowi.j
    w.i := (two*(rowi(n)))$R - b
  vv : V V I := firstSubsetGray((n-1): PI)$GRAY
  n :: I
  b := 1
  for i in 1..n repeat
    b := b *$R w.i
  a := a +$R b
  j := 1   -- Will be the number of the element changed in subset
  while j ^= n repeat  -- we sum over all subsets of (1,...,n-1)
    sgn := -sgn
    b := sgn
    if vv.1.j = 1 then k := -1
    else k := 1  -- was that element deleted(k=-1) or added(k=1)?
    c := k * two
    for i in 1..n repeat
      w.i :=  w.i +$R c *$R x(i,j)
      b := b *$R w.i
    a := a +$R b
    vv := nextSubsetGray(vv,(n-1) : PI)$GRAY
```

```
      j := vv.2.1
    if not odd?(n) then a := -a
    b := two ** ((n-1):NNI)
    (a exquo b) :: R
```

———-

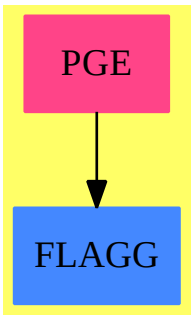
— **PERMAN.dotabb** —

```
"PERMAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PERMAN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"PERMAN" -> "IVECTOR"
```

———-

17.31 package PGE PermutationGroupExamples

17.31.1 PermutationGroupExamples (PGE)



Exports:

abelianGroup	alternatingGroup	cyclicGroup	dihedralGroup	janko2
mathieu11	mathieu12	mathieu22	mathieu23	mathieu24
rubiksGroup	symmetricGroup	youngGroup		

— **package PGE PermutationGroupExamples** —

```
)abbrev package PGE PermutationGroupExamples
++ Authors: M. Weller, G. Schneider, J. Grabmeier
++ Date Created: 20 February 1990
++ Date Last Updated: 09 June 1990
++ Basic Operations:
++ Related Constructors:
```

```
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++  J. Conway, R. Curtis, S. Norton, R. Parker, R. Wilson:
++   Atlas of Finite Groups, Oxford, Clarendon Press, 1987
++ Description:
++ PermutationGroupExamples provides permutation groups for
++ some classes of groups: symmetric, alternating, dihedral, cyclic,
++ direct products of cyclic, which are in fact the finite abelian groups
++ of symmetric groups called Young subgroups.
++ Furthermore, Rubik's group as permutation group of 48 integers and a list
++ of sporadic simple groups derived from the atlas of finite groups.

PermutationGroupExamples():public == private where

    L          ==> List
    I          ==> Integer
    PI         ==> PositiveInteger
    NNI        ==> NonNegativeInteger
    PERM       ==> Permutation
    PERMGRP   ==> PermutationGroup

    public ==> with

      symmetricGroup:       PI        -> PERMGRP I
        ++ symmetricGroup(n) constructs the symmetric group Sn
        ++ acting on the integers 1,...,n, generators are the
        ++ n-cycle (1,...,n) and the 2-cycle (1,2).
      symmetricGroup:       L I       -> PERMGRP I
        ++ symmetricGroup(li) constructs the symmetric group acting on
        ++ the integers in the list li, generators are the
        ++ cycle given by li and the 2-cycle (li.1,li.2).
        ++ Note that duplicates in the list will be removed.
      alternatingGroup:     PI        -> PERMGRP I
        ++ alternatingGroup(n) constructs the alternating group An
        ++ acting on the integers 1,...,n,  generators are in general the
        ++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
        ++ if n is odd and the product of the 2-cycle (1,2) with
        ++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
        ++ if n is even.
      alternatingGroup:     L I       -> PERMGRP I
        ++ alternatingGroup(li) constructs the alternating group acting
        ++ on the integers in the list li, generators are in general the
        ++ n-2-cycle (li.3,...,li.n) and the 3-cycle
        ++ (li.1,li.2,li.3), if n is odd and
        ++ product of the 2-cycle (li.1,li.2) with
        ++ n-2-cycle (li.3,...,li.n) and the 3-cycle
        ++ (li.1,li.2,li.3), if n is even.
        ++ Note that duplicates in the list will be removed.
```

```
abelianGroup:         L PI      -> PERMGRP I
  ++ abelianGroup([n1,...,nk]) constructs the abelian group that
  ++ is the direct product of cyclic groups with order ni.
cyclicGroup:          PI        -> PERMGRP I
  ++ cyclicGroup(n) constructs the cyclic group of order n acting
  ++ on the integers 1,...,n.
cyclicGroup:          L I       -> PERMGRP I
  ++ cyclicGroup([i1,...,ik]) constructs the cyclic group of
  ++ order k acting on the integers i1,...,ik.
  ++ Note that duplicates in the list will be removed.
dihedralGroup:        PI        -> PERMGRP I
  ++ dihedralGroup(n) constructs the dihedral group of order 2n
  ++ acting on integers 1,...,N.
dihedralGroup:        L I       -> PERMGRP I
  ++ dihedralGroup([i1,...,ik]) constructs the dihedral group of
  ++ order 2k acting on the integers out of i1,...,ik.
  ++ Note that duplicates in the list will be removed.
mathieu11:            L I       -> PERMGRP I
  ++ mathieu11(li) constructs the mathieu group acting on the 11
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ error, if li has less or more than 11 different entries.
mathieu11:            ()        -> PERMGRP I
  ++ mathieu11 constructs the mathieu group acting on the
  ++ integers 1,...,11.
mathieu12:            L I       -> PERMGRP I
  ++ mathieu12(li) constructs the mathieu group acting on the 12
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed
  ++ Error: if li has less or more than 12 different entries.
mathieu12:            ()        -> PERMGRP I
  ++ mathieu12 constructs the mathieu group acting on the
  ++ integers 1,...,12.
mathieu22:            L I       -> PERMGRP I
  ++ mathieu22(li) constructs the mathieu group acting on the 22
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ Error: if li has less or more than 22 different entries.
mathieu22:            ()        -> PERMGRP I
  ++ mathieu22 constructs the mathieu group acting on the
  ++ integers 1,...,22.
mathieu23:            L I       -> PERMGRP I
  ++ mathieu23(li) constructs the mathieu group acting on the 23
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ Error: if li has less or more than 23 different entries.
mathieu23:            ()        -> PERMGRP I
  ++ mathieu23 constructs the mathieu group acting on the
  ++ integers 1,...,23.
mathieu24:            L I       -> PERMGRP I
```

```
  ++ mathieu24(li) constructs the mathieu group acting on the 24
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ Error: if li has less or more than 24 different entries.
mathieu24:            ()        -> PERMGRP I
  ++ mathieu24 constructs the mathieu group acting on the
  ++ integers 1,...,24.
janko2:               L I       -> PERMGRP I
  ++ janko2(li) constructs the janko group acting on the 100
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ Error: if li has less or more than 100 different entries
janko2:               ()        -> PERMGRP I
  ++ janko2 constructs the janko group acting on the
  ++ integers 1,...,100.
rubiksGroup:          ()        -> PERMGRP I
  ++ rubiksGroup constructs the permutation group representing
  ++ Rubic's Cube acting on integers 10*i+j for
  ++ 1 <= i <= 6, 1 <= j <= 8.
  ++ The faces of Rubik's Cube are labelled in the obvious way
  ++ Front, Right, Up, Down, Left, Back and numbered from 1 to 6
  ++ in this given ordering, the pieces on each face
  ++ (except the unmoveable center piece) are clockwise numbered
  ++ from 1 to 8 starting with the piece in the upper left
  ++ corner. The moves of the cube are represented as permutations
  ++ on these pieces, represented as a two digit
  ++ integer ij where i is the numer of theface (1 to 6)
  ++ and j is the number of the piece on this face.
  ++ The remaining ambiguities are resolved by looking
  ++ at the 6 generators, which represent a 90 degree turns of the
  ++ faces, or from the following pictorial description.
  ++ Permutation group representing Rubic's Cube acting on integers
  ++ 10*i+j for 1 <= i <= 6, 1 <= j <=8.
  ++
  ++ \begin{verbatim}
  ++ Rubik's Cube:   +-----+ +-- B   where: marks Side # :
  ++                / U   /|/
  ++               /     / |         F(ront)    <->    1
  ++       L -->  +-----+ R|         R(ight)    <->    2
  ++              |     |  +         U(p)       <->    3
  ++              |  F  | /          D(own)     <->    4
  ++              |     |/           L(eft)     <->    5
  ++              +-----+            B(ack)     <->    6
  ++                 ^
  ++                 |
  ++                 D
  ++
  ++ The Cube's surface:
  ++                                The pieces on each side
  ++             +---+              (except the unmoveable center
```

```
    ++             |567|             piece) are clockwise numbered
    ++             |4U8|             from 1 to 8 starting with the
    ++             |321|             piece in the upper left
    ++         +---+---+---+         corner (see figure on the
    ++         |781|123|345|         left).  The moves of the cube
    ++         |6L2|8F4|2R6|         are represented as
    ++         |543|765|187|         permutations on these pieces.
    ++         +---+---+---+         Each of the pieces is
    ++             |123|             represented as a two digit
    ++             |8D4|             integer ij where i is the
    ++             |765|             # of the side ( 1 to 6 for
    ++             +---+             F to B (see table above ))
    ++             |567|             and j is the # of the piece.
    ++             |4B8|
    ++             |321|
    ++             +---+
    ++ \end{verbatim}
  youngGroup:           L I      -> PERMGRP I
    ++ youngGroup([n1,...,nk]) constructs the direct product of the
    ++ symmetric groups Sn1,...,Snk.
  youngGroup:    Partition        -> PERMGRP I
    ++ youngGroup(lambda) constructs the direct product of the symmetric
    ++ groups given by the parts of the partition lambda.

private ==> add

  -- import the permutation and permutation group domains:

  import PERM I
  import PERMGRP I

  -- import the needed map function:

  import ListFunctions2(L L I,PERM I)
  -- the internal functions:

  llli2gp(l:L L L I):PERMGRP I ==
    --++ Converts an list of permutations each represented by a list
    --++ of cycles ( each of them represented as a list of Integers )
    --++ to the permutation group generated by these permutations.
    (map(cycles,l))::PERMGRP I

  li1n(n:I):L I ==
    --++ constructs the list of integers from 1 to n
    [i for i in 1..n]

  -- definition of the exported functions:
  youngGroup(l:L I):PERMGRP I ==
    gens:= nil()$(L L L I)
    element:I:= 1
```

```
  for n in l | n > 1 repeat
    gens:=cons(list [i for i in element..(element+n-1)], gens)
    if n >= 3 then gens := cons([[element,element+1]],gens)
    element:=element+n
  llli2gp
    #gens = 0 => [[[1]]]
    gens

youngGroup(lambda : Partition):PERMGRP I ==
  youngGroup(convert(lambda)$Partition)

rubiksGroup():PERMGRP I ==
  -- each generator represents a 90 degree turn of the appropriate
  -- side.
  f:L L I:=
   [[11,13,15,17],[12,14,16,18],[51,31,21,41],[53,33,23,43],[52,32,22,42]]
  r:L L I:=
   [[21,23,25,27],[22,24,26,28],[13,37,67,43],[15,31,61,45],[14,38,68,44]]
  u:L L I:=
   [[31,33,35,37],[32,34,36,38],[13,51,63,25],[11,57,61,23],[12,58,62,24]]
  d:L L I:=
   [[41,43,45,47],[42,44,46,48],[17,21,67,55],[15,27,65,53],[16,28,66,54]]
  l:L L I:=
   [[51,53,55,57],[52,54,56,58],[11,41,65,35],[17,47,63,33],[18,48,64,34]]
  b:L L I:=
   [[61,63,65,67],[62,64,66,68],[45,25,35,55],[47,27,37,57],[46,26,36,56]]
  llli2gp [f,r,u,d,l,b]

mathieu11(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 11 => error "Exactly 11 integers for mathieu11 needed !"
  a:L L I:=[[l.1,l.10],[l.2,l.8],[l.3,l.11],[l.5,l.7]]
  llli2gp [a,[[l.1,l.4,l.7,l.6],[l.2,l.11,l.10,l.9]]]

mathieu11():PERMGRP I == mathieu11 li1n 11

mathieu12(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 12 => error "Exactly 12 integers for mathieu12 needed !"
  a:L L I:=
    [[l.1,l.2,l.3,l.4,l.5,l.6,l.7,l.8,l.9,l.10,l.11]]
  llli2gp [a,[[l.1,l.6,l.5,l.8,l.3,l.7,l.4,l.2,l.9,l.10],[l.11,l.12]]]

mathieu12():PERMGRP I == mathieu12 li1n 12

mathieu22(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
```

```
  #l ^= 22 => error "Exactly 22 integers for mathieu22 needed !"
  a:L L I:=[[l.1,l.2,l.4,l.8,l.16,l.9,l.18,l.13,l.3,l.6,l.12],   _
    [l.5,l.10,l.20,l.17,l.11,l.22,l.21,l.19,l.15,l.7,l.14]]
  b:L L I:= [[l.1,l.2,l.6,l.18],[l.3,l.15],[l.5,l.8,l.21,l.13],   _
    [l.7,l.9,l.20,l.12],[l.10,l.16],[l.11,l.19,l.14,l.22]]
  llli2gp [a,b]

mathieu22():PERMGRP I == mathieu22 li1n 22

mathieu23(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 23 => error "Exactly 23 integers for mathieu23 needed !"
  a:L L I:= [[l.1,l.2,l.3,l.4,l.5,l.6,l.7,l.8,l.9,l.10,l.11,l.12,l.13,l.14,_
             l.15,l.16,l.17,l.18,l.19,l.20,l.21,l.22,l.23]]
  b:L L I:= [[l.2,l.16,l.9,l.6,l.8],[l.3,l.12,l.13,l.18,l.4],             _
             [l.7,l.17,l.10,l.11,l.22],[l.14,l.19,l.21,l.20,l.15]]
  llli2gp [a,b]

mathieu23():PERMGRP I == mathieu23 li1n 23

mathieu24(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 24 => error "Exactly 24 integers for mathieu24 needed !"
  a:L L I:= [[l.1,l.16,l.10,l.22,l.24],[l.2,l.12,l.18,l.21,l.7],          _
             [l.4,l.5,l.8,l.6,l.17],[l.9,l.11,l.13,l.19,l.15]]
  b:L L I:= [[l.1,l.22,l.13,l.14,l.6,l.20,l.3,l.21,l.8,l.11],[l.2,l.10],  _
             [l.4,l.15,l.18,l.17,l.16,l.5,l.9,l.19,l.12,l.7],[l.23,l.24]]
  llli2gp [a,b]

mathieu24():PERMGRP I == mathieu24 li1n 24

janko2(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
  l:=removeDuplicates l
  #l ^= 100 => error "Exactly 100 integers for janko2 needed !"
  a:L L I:=[                                                            _
           [l.2,l.3,l.4,l.5,l.6,l.7,l.8],                               _
           [l.9,l.10,l.11,l.12,l.13,l.14,l.15],                         _
           [l.16,l.17,l.18,l.19,l.20,l.21,l.22],                        _
           [l.23,l.24,l.25,l.26,l.27,l.28,l.29],                        _
           [l.30,l.31,l.32,l.33,l.34,l.35,l.36],                        _
           [l.37,l.38,l.39,l.40,l.41,l.42,l.43],                        _
           [l.44,l.45,l.46,l.47,l.48,l.49,l.50],                        _
           [l.51,l.52,l.53,l.54,l.55,l.56,l.57],                        _
           [l.58,l.59,l.60,l.61,l.62,l.63,l.64],                        _
           [l.65,l.66,l.67,l.68,l.69,l.70,l.71],                        _
           [l.72,l.73,l.74,l.75,l.76,l.77,l.78],                        _
           [l.79,l.80,l.81,l.82,l.83,l.84,l.85],                        _
```

```
           [1.86,1.87,1.88,1.89,1.90,1.91,1.92],                    _
           [1.93,1.94,1.95,1.96,1.97,1.98,1.99] ]
  b:L L I:=[
          [1.1,1.74,1.83,1.21,1.36,1.77,1.44,1.80,1.64,1.2,1.34,1.75,1.48,1.17,1.100],_
          [1.3,1.15,1.31,1.52,1.19,1.11,1.73,1.79,1.26,1.56,1.41,1.99,1.39,1.84,1.90],_
          [1.4,1.57,1.86,1.63,1.85,1.95,1.82,1.97,1.98,1.81,1.8,1.69,1.38,1.43,1.58],_
          [1.5,1.66,1.49,1.59,1.61],_
          [1.6,1.68,1.89,1.94,1.92,1.20,1.13,1.54,1.24,1.51,1.87,1.27,1.76,1.23,1.67],_
          [1.7,1.72,1.22,1.35,1.30,1.70,1.47,1.62,1.45,1.46,1.40,1.28,1.65,1.93,1.42],_
          [1.9,1.71,1.37,1.91,1.18,1.55,1.96,1.60,1.16,1.53,1.50,1.25,1.32,1.14,1.33],_
          [1.10,1.78,1.88,1.29,1.12] ]
  llli2gp [a,b]

janko2():PERMGRP I == janko2 li1n 100

abelianGroup(l:L PI):PERMGRP I ==
  gens:= nil()$(L L L I)
  element:I:= 1
  for n in l | n > 1 repeat
    gens:=cons( list [i for i in element..(element+n-1) ], gens )
    element:=element+n
  llli2gp
    #gens = 0 => [[[1]]]
    gens

alternatingGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 =>
    error "Cannot construct alternating group on empty set"
  #l < 3 => llli2gp [[[l.1]]]
  #l = 3 => llli2gp [[[l.1,l.2,l.3]]]
  tmp:= [l.i for i in 3..(#l)]
  gens:L L L I:=[[tmp],[[l.1,l.2,l.3]]]
  odd?(#l) => llli2gp gens
  gens.1 := cons([l.1,l.2],gens.1)
  llli2gp gens

alternatingGroup(n:PI):PERMGRP I == alternatingGroup li1n n

symmetricGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 => error "Cannot construct symmetric group on empty set !"
  #l < 3 => llli2gp [[l]]
  llli2gp [[l],[[l.1,l.2]]]

symmetricGroup(n:PI):PERMGRP I == symmetricGroup li1n n

cyclicGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l = 0 => error "Cannot construct cyclic group on empty set"
```

```
    llli2gp [[l]]

cyclicGroup(n:PI):PERMGRP I == cyclicGroup li1n n

dihedralGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #l < 3 => error "in dihedralGroup: Minimum of 3 elements needed !"
  tmp := [[l.i, l.(#l-i+1) ] for i in 1..(#l quo 2)]
  llli2gp [ [ l ], tmp ]

dihedralGroup(n:PI):PERMGRP I ==
  n = 1 => symmetricGroup (2::PI)
  n = 2 => llli2gp [[[1,2]],[[3,4]]]
  dihedralGroup li1n n
```

———-

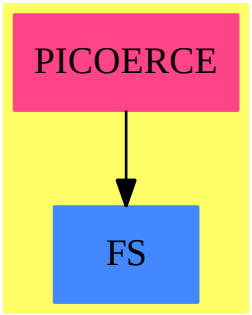
— **PGE.dotabb** —

```
"PGE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PGE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PGE" -> "FLAGG"
```

———-

17.32 package PICOERCE PiCoercions

17.32.1 PiCoercions (PICOERCE)



Exports:
coerce

— **package PICOERCE PiCoercions** —

```
)abbrev package PICOERCE PiCoercions
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 21 Feb 1990
++ Description:
++ Provides a coercion from the symbolic fractions in %pi with
++ integer coefficients to any Expression type.

PiCoercions(R:Join(OrderedSet, IntegralDomain)): with
  coerce: Pi -> Expression R
    ++ coerce(f) returns f as an Expression(R).
 == add
  p2e: SparseUnivariatePolynomial Integer -> Expression R

  coerce(x:Pi):Expression(R) ==
    f := convert(x)@Fraction(SparseUnivariatePolynomial Integer)
    p2e(numer f) / p2e(denom f)

  p2e p ==
    map((x1:Integer):Expression(R) +-> x1::Expression(R), p)_
      $SparseUnivariatePolynomialFunctions2(Integer, Expression R)_
       (pi()$Expression(R))
```

———-

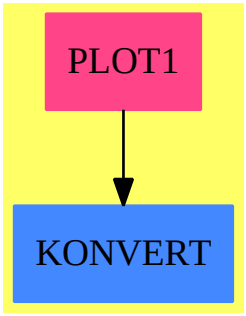
— PICOERCE.dotabb —

```
"PICOERCE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PICOERCE"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PICOERCE" -> "FS"
```

———-

17.33 package PLOT1 PlotFunctions1

17.33.1 PlotFunctions1 (PLOT1)



Exports:
plotPolar plot

— package PLOT1 PlotFunctions1 —

```
)abbrev package PLOT1 PlotFunctions1
++ Authors: R.T.M. Bronstein, C.J. Williamson
++ Date Created: Jan 1989
++ Date Last Updated: 4 Mar 1990
++ Basic Operations: plot, plotPolar
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ PlotFunctions1 provides facilities for plotting curves
++ where functions SF -> SF are specified by giving an expression

PlotFunctions1(S:ConvertibleTo InputForm): with
    plot : (S, Symbol, Segment DoubleFloat) -> Plot
      ++ plot(fcn,x,seg) plots the graph of \spad{y = f(x)} on a interval
    plot : (S, S, Symbol, Segment DoubleFloat) -> Plot
      ++ plot(f,g,t,seg) plots the graph of \spad{x = f(t)}, \spad{y = g(t)}
      ++ as t ranges over an interval.
    plotPolar : (S, Symbol, Segment DoubleFloat) -> Plot
      ++ plotPolar(f,theta,seg) plots the graph of \spad{r = f(theta)} as
      ++ theta ranges over an interval
    plotPolar : (S, Symbol) -> Plot
      ++ plotPolar(f,theta) plots the graph of \spad{r = f(theta)} as
      ++ theta ranges from 0 to 2 pi
  == add
```

```
    import MakeFloatCompiledFunction(S)

    plot(f, x, xRange) == plot(makeFloatFunction(f, x), xRange)
    plotPolar(f,theta) == plotPolar(makeFloatFunction(f,theta))
    plot(f1, f2, t, tRange) ==
      plot(makeFloatFunction(f1, t), makeFloatFunction(f2, t), tRange)
    plotPolar(f,theta,thetaRange) ==
      plotPolar(makeFloatFunction(f,theta),thetaRange)
```

———-

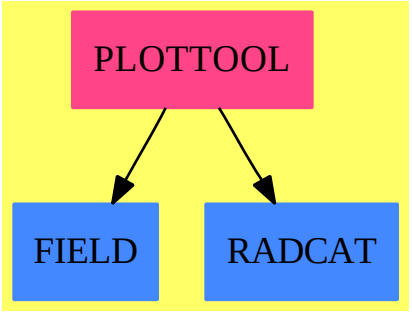
— **PLOT1.dotabb** —

```
"PLOT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLOT1"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"PLOT1" -> "KONVERT"
```

———-

17.34 package PLOTTOOL PlotTools

17.34.1 PlotTools (PLOTTOOL)



Exports:
calcRanges

— **package PLOTTOOL PlotTools** —

```
)abbrev package PLOTTOOL PlotTools
++ Author:
++ Date Created:
++ Date Last Updated:
```

```
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package exports plotting tools

PlotTools(): Exports == Implementation where
  L   ==> List
--  Pt  ==> TwoDimensionalPoint
  SEG ==> Segment
  SF  ==> DoubleFloat
  Pt ==> Point(SF)
  PLOT ==> Plot
  DROP ==> DrawOption
  S    ==> String
  VIEW2D ==> TwoDimensionalViewport

  Exports ==> with
    calcRanges: L L Pt             -> L SEG SF
      ++ calcRanges(l) \undocumented

  Implementation ==> add
    import GraphicsDefaults
    import PLOT
    import TwoDimensionalPlotClipping
    import DrawOptionFunctions0
    import ViewportPackage
    import POINT
    import PointPackage(SF)

    --%Local functions
    xRange0: L Pt -> SEG SF
    xRange: L L Pt -> SEG SF
    yRange0: L Pt -> SEG SF
    yRange: L L Pt -> SEG SF
    drawToScaleRanges: (SEG SF,SEG SF) -> L SEG SF

    drawToScaleRanges(xVals,yVals) ==
      xDiff := (xHi := hi xVals) - (xLo := lo xVals)
      yDiff := (yHi := hi yVals) - (yLo := lo yVals)
      pad := abs(yDiff - xDiff)/2
      yDiff > xDiff => [segment(xLo - pad,xHi + pad),yVals]
      [xVals,segment(yLo - pad,yHi + pad)]

    select : (L Pt,Pt -> SF,(SF,SF) -> SF) -> SF
    select(l,f,g) ==
      m := f first l
      for p in rest l repeat m := g(m,f p)
      m
```

```
    xRange0(list:L Pt) == select(list,xCoord,min) .. select(list,xCoord,max)
    yRange0(list:L Pt) == select(list,yCoord,min) .. select(list,yCoord,max)

    select2: (L L Pt,L Pt -> SF,(SF,SF) -> SF) -> SF
    select2(l,f,g) ==
      m := f first l
      for p in rest l repeat m := g(m,f p)
      m

    xRange(list:L L Pt) ==
      select2(list,(u1:L(Pt)):SF +-> lo(xRange0(u1)),min) _
         .. select2(list,(v1:L(Pt)):SF +-> hi(xRange0(v1)),max)

    yRange(list:L L Pt) ==
      select2(list,(u1:L(Pt)):SF +-> lo(yRange0(u1)),min) _
         .. select2(list,(v1:L(Pt)):SF +-> hi(yRange0(v1)),max)

  --%Exported Functions
    calcRanges(llp) ==
      drawToScale() => drawToScaleRanges(xRange llp, yRange llp)
      [xRange llp, yRange llp]
```

———-

— PLOTTOOL.dotabb —

```
"PLOTTOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLOTTOOL"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"PLOTTOOL" -> "FIELD"
"PLOTTOOL" -> "RADCAT"
```

———-

PRJALGPK ProjectiveAlgebraicSetPackage

17.35 package PRJALGPK ProjectiveAlgebraicSetPackage

— ProjectiveAlgebraicSetPackage.input —

```
)set break resume
)sys rm -f ProjectiveAlgebraicSetPackage.output
)spool ProjectiveAlgebraicSetPackage.output
```

```
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectiveAlgebraicSetPackage
--R
--R ProjectiveAlgebraicSetPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,O
--R Abbreviation for ProjectiveAlgebraicSetPackage is PRJALGPK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRJALGPK
--R
--R------------------------------ Operations ------------------------------
--R algebraicSet : List(PolyRing) -> List(ProjPt)
--R rationalPoints : (PolyRing,PositiveInteger) -> List(ProjPt)
--R singularPoints : PolyRing -> List(ProjPt)
--R singularPointsWithRestriction : (PolyRing,List(PolyRing)) -> List(ProjPt)
--R
--E 1

)spool
)lisp (bye)
```

———-

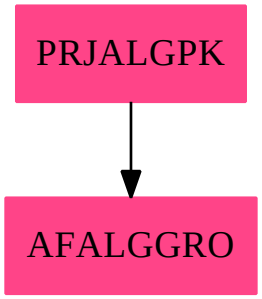
— ProjectiveAlgebraicSetPackage.help —

```
=====================================================================
ProjectiveAlgebraicSetPackage examples
=====================================================================

See Also:
o )show ProjectiveAlgebraicSetPackage
```

———-

17.35.1 ProjectiveAlgebraicSetPackage (PRJALGPK)



Exports:
algebraicSet rationalPoints singularPoints singularPointsWithRestriction

— package PRJALGPK ProjectiveAlgebraicSetPackage —

```
)abbrev package PRJALGPK ProjectiveAlgebraicSetPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt):_
 Exports == Implementation where
  K   : Field
  symb: List(Symbol)
  OV  ==> OrderedVariableList(symb)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt   : ProjectiveSpaceCategory(K)
  PCS      : LocalPowerSeriesCategory(K)

  OF     ==> OutputForm
  PI     ==> PositiveInteger
  RFP    ==> RootsFindingPackage
  SUP    ==> SparseUnivariatePolynomial
  PPFC1  ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
  SPWRES ==> AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt)
  SPWGRO ==> AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt)

  Exports ==>  with

    singularPointsWithRestriction: (PolyRing,List(PolyRing)) -> List(ProjPt)
      ++ return the singular points that anhilate

    singularPoints: PolyRing -> List(ProjPt)
```

```
      ++ singularPoints retourne les points singulier

    algebraicSet: List(PolyRing) -> List(ProjPt)
      ++ algebraicSet returns the algebraic set if finite (dimension 0).

    rationalPoints: (PolyRing,PI) -> List(ProjPt)
      ++ \axiom{rationalPoints(f,d)} returns all points on the curve \axiom{f}
      ++ in the extension of the ground field of degree \axiom{d}.
      ++ For \axiom{d > 1} this only works if \axiom{K} is a
      ++ \axiomType{LocallyAlgebraicallyClosedField}

  Implementation ==>  add

    import PPFC1
    import PolyRing
    import ProjPt

    listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]
    polyToX10 : PolyRing -> SUP(K)

    --fonctions de resolution de sys. alg. de dim 0
    singularPoints(crb)==
      F:=crb
      Fx:=differentiate(F,index(1)$OV)
      Fy:=differentiate(F,index(2)$OV)
      Fz:=differentiate(F,index(3)$OV)
      idealT:List PolyRing:=[F,Fx,Fy,Fz]
      idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT]
      recOfZerosX10:= distinguishedCommonRootsOf(idealToX10,1)$RFP(K)
      listOfExtDeg:List Integer:=[recOfZerosX10.extDegree]
      degExt:=lcm listOfExtDeg
      zero?(degExt) =>
        error("------- Infinite number of points ------")
      ^one?(degExt) =>
        print(("You need an extension of degree")::OF)
        print(degExt::OF)
        error("-------------Have a nice day-------------")
      listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
      tempL:= affineSingularPoints(crb)$SPWRES
      if tempL case "failed" then
        print(("failed with resultant")::OF)
        print("The singular points will be computed using grobner basis"::OF)
        tempL := affineSingularPoints(crb)$SPWGRO
      tempL case "Infinite" =>
        error("------- Infinite number of points ------")
      tempL case Integer =>
        print(("You need an extension of degree")::OF)
        print(tempL ::OF)
        error("-------------Have a nice day-------------")
      listPtsIdl2:List(ProjPt)
```

```
      if tempL case List(ProjPt) then
        listPtsIdl2:= ( tempL :: List(ProjPt))
      else
        error" From ProjectiveAlgebraicSetPackage: this should not happen"
      listPtsIdl := concat( listPtsIdl , listPtsIdl2)
      if  pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
        listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
      listPtsIdl

    algebraicSet(idealT:List(PolyRing)) ==
      idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT]
      recOfZerosX10:= distinguishedCommonRootsOf(idealToX10,1)$RFP(K)
      listOfExtDeg:List Integer:=[recOfZerosX10.extDegree]
      degExt:=lcm listOfExtDeg
      zero?(degExt) =>
        error("------- Infinite number of points ------")
      ^one?(degExt) =>
        print(("You need an extension of degree")::OF)
        print(degExt::OF)
        error("-------------Have a nice day-------------")
      listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
      tempL:= affineAlgSet( idealT )$SPWRES
      if tempL case "failed" then
        print("failed with resultant"::OF)
        print("The finte alg. set  will be computed using grobner basis"::OF)
        tempL := affineAlgSet( idealT )$SPWGRO
      tempL case "Infinite" =>
        error("------- Infinite number of points ------")
      tempL case Integer =>
        print(("You need an extension of degree")::OF)
        print(tempL ::OF)
        error("-------------Have a nice day-------------")
      listPtsIdl2:List(ProjPt)
      if tempL case List(ProjPt) then
        listPtsIdl2:= ( tempL :: List(ProjPt) )
      else
        error" From ProjectiveAlgebraicSetPackage: this should not hapen"
      listPtsIdl := concat( listPtsIdl , listPtsIdl2)
      if  pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
        listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
      listPtsIdl

    if K has FiniteFieldCategory then

      rationalPoints(crv:PolyRing,extdegree:PI):List(ProjPt) ==
--The code of this is almost the same as for algebraicSet
--We could just construct the ideal and call algebraicSet
--Should we do that? This might be a bit faster.
listPtsIdl:List(ProjPt):= empty()
        x:= monomial(1,1)$SUP(K)
```

```
if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
  setTower!(1$K)$K
        q:= size()$K
        px:= x**(q**extdegree) - x
crvX10:= polyToX10 crv
recOfZerosX10:=distinguishedCommonRootsOf([crvX10,px],1$K)$RFP(K)
        listPtsIdl:=[projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
        --now we got all of the projective points where z = 0 and y ^= 0
        ratXY1 : List ProjPt:= affineRationalPoints( crv, extdegree )$SPWGRO
listPtsIdl:= concat(ratXY1,listPtsIdl)
        if  pointInIdeal?([crv],projectivePoint([1,0,0]))$PPFC1 then
          listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
        listPtsIdl

    polyToX10(pol)==
      zero?(pol) => 0
      dd:= degree pol
      lc:= leadingCoefficient pol
      pp:= parts dd
      lp:= last pp
      ^zero?(lp) => polyToX10 reductum pol
      e1:= pp.1
      monomial(lc,e1)$SUP(K) + polyToX10 reductum pol

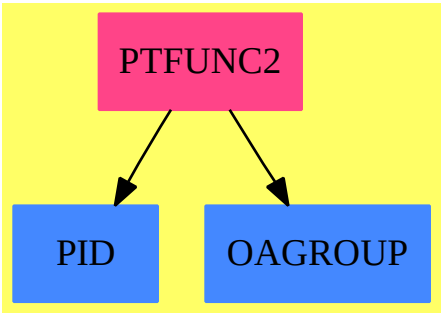
    singularPointsWithRestriction(F,lstPol)==
      Fx:=differentiate(F,index(1)$OV)
      Fy:=differentiate(F,index(2)$OV)
      Fz:=differentiate(F,index(3)$OV)
      idealSingulier:List(PolyRing):=concat([F,Fx,Fy,Fz],lstPol)
      algebraicSet(idealSingulier)
```

— PRJALGPK.dotabb —

```
"PRJALGPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRJALGPK"]
"AFALGGRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGGRO"]
"PRJALGPK" -> "AFALGGRO"
```

17.36 package PTFUNC2 PointFunctions2

17.36.1 PointFunctions2 (PTFUNC2)



Exports:
map

— package PTFUNC2 PointFunctions2 —

```
)abbrev package PTFUNC2 PointFunctions2
++ Description:
++ This package has no description

PointFunctions2(R1:Ring,R2:Ring):Exports == Implementation where

  Exports == with
    map : ((R1->R2),Point(R1)) -> Point(R2)
      ++ map(f,p) \undocumented

  Implementation ==> add
    import Point(R1)
    import Point(R2)

    map(mapping,p) ==
      point([mapping p.(i::PositiveInteger) for i in minIndex(p)..maxIndex(p)])$Point(R2)
```

———-

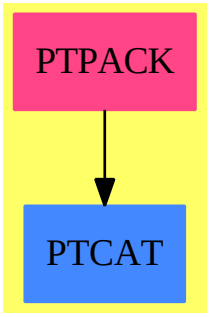
— PTFUNC2.dotabb —

```
"PTFUNC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PTFUNC2"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PTFUNC2" -> "PID"
```

```
"PTFUNC2" -> "OAGROUP"
```

17.37 package PTPACK PointPackage

17.37.1 PointPackage (PTPACK)



Exports:

color	hue	phiCoord	rCoord	shade
thetaCoord	xCoord	yCoord	zCoord	

— package PTPACK PointPackage —

```
)abbrev package PTPACK PointPackage
++ Description:
++ This package has no description

PointPackage(R:Ring):Exports == Implementation where

  POINT ==> Point(R)
  I    ==> Integer
  PI   ==> PositiveInteger
  NNI  ==> NonNegativeInteger
  L    ==> List
  B    ==> Boolean

  Exports == with
    xCoord       : POINT -> R
      ++ xCoord(pt) returns the first element of the point, pt,
      ++ although no assumptions are made as to the coordinate
      ++ system being used.  This function is defined for the
      ++ convenience of the user dealing with a Cartesian
      ++ coordinate system.
```

```
yCoord       : POINT -> R
  ++ yCoord(pt) returns the second element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used.  This function is defined for the
  ++ convenience of the user dealing with a Cartesian
  ++ coordinate system.
zCoord       : POINT -> R
  ++ zCoord(pt) returns the third element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used.  This function is defined for the
  ++ convenience of the user dealing with a Cartesian
  ++ or a cylindrical coordinate system.
rCoord       : POINT -> R
  ++ rCoord(pt) returns the first element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used.  This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ or a cylindrical coordinate system.
thetaCoord   : POINT -> R
  ++ thetaCoord(pt) returns the second element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used.  This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ or a cylindrical coordinate system.
phiCoord     : POINT -> R
  ++ phiCoord(pt) returns the third element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used.  This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ coordinate system.
color        : POINT -> R
  ++ color(pt) returns the fourth element of the point, pt,
  ++ although no assumptions are made with regards as to
  ++ how the components of higher dimensional points are
  ++ interpreted.  This function is defined for the
  ++ convenience of the user using specifically, color
  ++ to express a fourth dimension.
hue : POINT -> R
  ++ hue(pt) returns the third element of the two dimensional point, pt,
  ++ although no assumptions are made with regards as to how the
  ++ components of higher dimensional points are interpreted. This
  ++ function is defined for the convenience of the user using
  ++ specifically, hue to express a third dimension.
shade : POINT -> R
  ++ shade(pt) returns the fourth element of the two dimensional
  ++ point, pt, although no assumptions are made with regards as to
  ++ how the components of higher dimensional points are interpreted.
  ++ This function is defined for the convenience of the user using
  ++ specifically, shade to express a fourth dimension.
```

```
      -- 2D and 3D extraction of data
  Implementation ==> add

    xCoord p == elt(p,1)
    yCoord p == elt(p,2)
    zCoord p == elt(p,3)
    rCoord p == elt(p,1)
    thetaCoord p == elt(p,2)
    phiCoord p == elt(p,3)
    color p ==
      #p > 3 => p.4
      p.3
    hue p == elt(p,3)
      -- 4D points in 2D using extra dimensions for palette information
    shade p == elt(p,4)
      -- 4D points in 2D using extra dimensions for palette information
```

———-

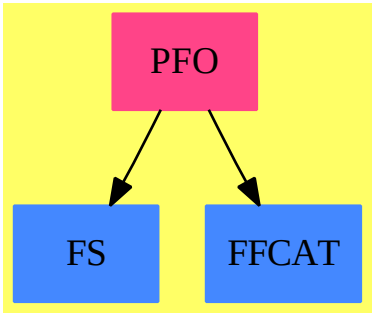
— **PTPACK.dotabb** —

```
"PTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PTPACK"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"PTPACK" -> "PTCAT"
```

———-

17.38 package PFO PointsOfFiniteOrder

17.38.1 PointsOfFiniteOrder (PFO)



Exports:
order torsion? torsionIfCan

— package PFO PointsOfFiniteOrder —

```
)abbrev package PFO PointsOfFiniteOrder
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 July 1998
++ Keywords: divisor, algebraic, curve.
++ Description:
++ This package provides function for testing whether a divisor on a
++ curve is a torsion divisor.

PointsOfFiniteOrder(R0, F, UP, UPUP, R): Exports == Implementation where
  R0   : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F    : FunctionSpace R0
  UP   : UnivariatePolynomialCategory F
  UPUP : UnivariatePolynomialCategory Fraction UP
  R    : FunctionFieldCategory(F, UP, UPUP)

  PI  ==> PositiveInteger
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Integer
  UPF ==> SparseUnivariatePolynomial F
  UPQ ==> SparseUnivariatePolynomial Q
  QF  ==> Fraction UP
  UPUPQ ==> SparseUnivariatePolynomial Fraction UPQ
  UP2 ==> SparseUnivariatePolynomial UPQ
  UP3 ==> SparseUnivariatePolynomial UP2
  FD  ==> FiniteDivisor(F, UP, UPUP, R)
  K   ==> Kernel F
  REC ==> Record(ncurve:UP3, disc:Z, dfpoly:UPQ)
  RC0 ==> Record(ncurve:UPUPQ, disc:Z)
  ID  ==> FractionalIdeal(UP, QF, UPUP, R)
  SMP ==> SparseMultivariatePolynomial(R0,K)
  ALGOP  ==> "%alg"

  Exports ==> with
    order        : FD -> Union(N, "failed")
      ++ order(f) \undocumented
    torsion?     : FD -> Boolean
      ++ torsion?(f) \undocumented
    torsionIfCan : FD -> Union(Record(order:N, function:R), "failed")
      ++ torsionIfCan(f)\ undocumented

  Implementation ==> add
    import IntegerPrimesPackage(Z)
    import PointsOfFiniteOrderTools(UPQ, UPUPQ)
    import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)
```

```
cmult: List SMP -> SMP
raise         : (UPQ, K) -> F
raise2        : (UP2, K) -> UP
qmod          : F     -> Q
fmod          : UPF   -> UPQ
rmod          : UP    -> UPQ
pmod          : UPUP  -> UPUPQ
kqmod         : (F,    K) -> UPQ
krmod         : (UP,   K) -> UP2
kpmod         : (UPUP, K) -> UP3
selectIntegers: K   -> REC
selIntegers:    ()  -> RC0
possibleOrder : FD -> N
ratcurve      : (FD, RC0)    -> N
algcurve      : (FD, REC, K) -> N
kbad3Num      : (UP3, UPQ) -> Z
kbadBadNum    : (UP2, UPQ) -> Z
kgetGoodPrime : (REC, UPQ, UP3, UP2,UP2) -> Record(prime:PI,poly:UPQ)
goodRed       : (REC, UPQ, UP3, UP2, UP2, PI) -> Union(UPQ, "failed")
good?         : (UPQ, UP3, UP2, UP2, PI, UPQ) -> Boolean
klist         : UP -> List K
aklist        : R  -> List K
alglist       : FD -> List K
notIrr?       : UPQ -> Boolean
rat           : (UPUP, FD, PI) -> N
toQ1          : (UP2, UPQ) -> UP
toQ2          : (UP3, UPQ) -> R
Q2F           : Q -> F
Q2UPUP        : UPUPQ -> UPUP

q := FunctionSpaceReduce(R0, F)

torsion? d == order(d) case N
Q2F x      == numer(x)::F / denom(x)::F
qmod x     == bringDown(x)$q
kqmod(x,k) == bringDown(x, k)$q
fmod p     == map(qmod, p)$SparseUnivariatePolynomialFunctions2(F, Q)
pmod p     == map(qmod, p)$MultipleMap(F, UP, UPUP, Q, UPQ, UPUPQ)
Q2UPUP p   == map(Q2F, p)$MultipleMap(Q, UPQ, UPUPQ, F, UP, UPUP)
klist d    == "setUnion"/[kernels c for c in coefficients d]
notIrr? d  == #(factors factor(d)$RationalFactorize(UPQ)) > 1
kbadBadNum(d, m) == mix [badNum(c rem m) for c in coefficients d]
kbad3Num(h, m)   == lcm [kbadBadNum(c, m) for c in coefficients h]

torsionIfCan d ==
  zero?(n := possibleOrder(d := reduce d)) => "failed"
  (g := generator reduce(n::Z * d)) case "failed" => "failed"
  [n, g::R]

UPQ2F(p:UPQ, k:K):F ==
```

```
    map(Q2F, p)$UnivariatePolynomialCategoryFunctions2(Q, UPQ, F, UP) (k::F)

  UP22UP(p:UP2, k:K):UP ==
    map((p1:UPQ):F +-> UPQ2F(p1, k), p)_
      $UnivariatePolynomialCategoryFunctions2(UPQ,UP2,F,UP)

  UP32UPUP(p:UP3, k:K):UPUP ==
    map((p1:UP2):QF +-> UP22UP(p1,k)::QF,p)_
      $UnivariatePolynomialCategoryFunctions2(UP2, UP3, QF, UPUP)

  if R0 has GcdDomain then
     cmult(l:List SMP):SMP == lcm l
  else
     cmult(l:List SMP):SMP == */l

  doubleDisc(f:UP3):Z ==
    d := discriminant f
    g := gcd(d, differentiate d)
    d := (d exquo g)::UP2
    zero?(e := discriminant d) => 0
    gcd [retract(c)@Z for c in coefficients e]

  commonDen(p:UP):SMP ==
    l1:List F := coefficients p
    l2:List SMP := [denom c for c in l1]
    cmult l2

  polyred(f:UPUP):UPUP ==
    cmult([commonDen(retract(c)@UP) for c in coefficients f])::F::UP::QF * f

  aklist f ==
    (r := retractIfCan(f)@Union(QF, "failed")) case "failed" =>
      "setUnion"/[klist(retract(c)@UP) for c in coefficients lift f]
    klist(retract(r::QF)@UP)

  alglist d ==
    n := numer(i := ideal d)
    select_!((k1:K):Boolean +-> has?(operator k1, ALGOP),
        setUnion(klist denom i,
          "setUnion"/[aklist qelt(n,i) for i in minIndex n..maxIndex n]))

  krmod(p,k) ==
     map(z1 +-> kqmod(z1, k),
         p)$UnivariatePolynomialCategoryFunctions2(F, UP, UPQ, UP2)

  rmod p ==
     map(qmod, p)$UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)

  raise(p, k) ==
    (map(Q2F, p)$SparseUnivariatePolynomialFunctions2(Q, F)) (k::F)
```

```
    raise2(p, k) ==
      map(z1 +-> raise(z1, k),
          p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)

    algcurve(d, rc, k) ==
      mn := minIndex(n := numer(i := minimize ideal d))
      h  := kpmod(lift(hh := n(mn + 1)), k)
      b2 := primitivePart
                 raise2(b := krmod(retract(retract(n.mn)@QF)@UP, k), k)
      s  := kqmod(resultant(primitivePart separate(raise2(krmod(
                    retract(norm hh)@UP, k), k), b2).primePart, b2), k)
      pr := kgetGoodPrime(rc, s, h, b, dd := krmod(denom i, k))
      p  := pr.prime
      pp := UP32UPUP(rc.ncurve, k)
      mm := pr.poly
      gf := InnerPrimeField p
      m  := map((z1:Q):gf +-> retract(z1)@Z :: gf,mm)_
              $SparseUnivariatePolynomialFunctions2(Q, gf)
--      one? degree m =>
      (degree m = 1) =>
        alpha := - coefficient(m, 0) / leadingCoefficient m
        order(d, pp,
           (z1:F):gf +-> (map((q1:Q):gf +-> numer(q1)::gf / denom(q1)::gf,
            kqmod(z1,k))$SparseUnivariatePolynomialFunctions2(Q,gf))(alpha)
                                  )$ReducedDivisor(F, UP, UPUP, R, gf)
        -- d1 := toQ1(dd, mm)
        -- rat(pp, divisor ideal([(toQ1(b, mm) / d1)::QF::R,
                                      -- inv(d1::QF) * toQ2(h,mm)])$ID, p)
        sae:= SimpleAlgebraicExtension(gf,SparseUnivariatePolynomial gf,m)
        order(d, pp,
          (z1:F):sae +-> reduce(map((q1:Q):gf +-> numer(q1)::gf / denom(q1)::gf,
              kqmod(z1,k))$SparseUnivariatePolynomialFunctions2(Q,gf))$sae
                                  )$ReducedDivisor(F, UP, UPUP, R, sae)

-- returns the potential order of d, 0 if d is of infinite order
    ratcurve(d, rc) ==
      mn  := minIndex(nm := numer(i := minimize ideal d))
      h   := pmod lift(hh := nm(mn + 1))
      b   := rmod(retract(retract(nm.mn)@QF)@UP)
      s   := separate(rmod(retract(norm hh)@UP), b).primePart
      bd  := badNum rmod denom i
      r   := resultant(s, b)
      bad := lcm [rc.disc, numer r, denom r, bd.den*bd.gcdnum, badNum h]$List(Z)
      pp  := Q2UPUP(rc.ncurve)
      n   := rat(pp, d, p := getGoodPrime bad)
-- if n > 1 then it is cheaper to compute the order modulo a second prime,
-- since computing n * d could be very expensive
--      one? n => n
      (n = 1) => n
```

```
      m   := rat(pp, d, getGoodPrime(p * bad))
      n = m => n
      0

-- returns the order of d mod p
    rat(pp, d, p) ==
      gf := InnerPrimeField p
      order(d, pp,
        (q1:F):gf +-> (qq := qmod q1;numer(qq)::gf / denom(qq)::gf)
                                    )$ReducedDivisor(F, UP, UPUP, R, gf)

-- returns the potential order of d, 0 if d is of infinite order
    possibleOrder d ==
--      zero?(genus()) or one?(#(numer ideal d)) => 1
      zero?(genus()) or (#(numer ideal d) = 1) => 1
      empty?(la := alglist d) => ratcurve(d, selIntegers())
      not(empty? rest la) =>
           error "PFO::possibleOrder: more than 1 algebraic constant"
      algcurve(d, selectIntegers first la, first la)

    selIntegers():RC0 ==
      f := definingPolynomial()$R
      while zero?(d := doubleDisc(r := polyred pmod f)) repeat newReduc()$q
      [r, d]

    selectIntegers(k:K):REC ==
      g := polyred(f := definingPolynomial()$R)
      p := minPoly k
      while zero?(d := doubleDisc(r := kpmod(g, k))) or (notIrr? fmod p)
          repeat newReduc()$q
      [r, d, splitDenominator(fmod p).num]

    toQ1(p, d) ==
      map((p1:UPQ):F +-> Q2F(retract(p1 rem d)@Q),
          p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)

    toQ2(p, d) ==
      reduce map((p1:UP2):QF +-> toQ1(p1, d)::QF,
        p)$UnivariatePolynomialCategoryFunctions2(UP2, UP3, QF, UPUP)

    kpmod(p, k) ==
      map((p1:QF):UP2 +-> krmod(retract(p1)@UP, k),
        p)$UnivariatePolynomialCategoryFunctions2(QF, UPUP, UP2, UP3)

    order d ==
      zero?(n := possibleOrder(d := reduce d)) => "failed"
      principal? reduce(n::Z * d) => n
      "failed"

    kgetGoodPrime(rec, res, h, b, d) ==
```

```
      p:PI := 3
      while (u := goodRed(rec, res, h, b, d, p)) case "failed" repeat
        p := nextPrime(p::Z)::PI
      [p, u::UPQ]

    goodRed(rec, res, h, b, d, p) ==
      zero?(rec.disc rem p) => "failed"
      gf := InnerPrimeField p
      l  := [f.factor for f in factors
             factor(map((z1:Q):gf +-> retract(z1)@Z :: gf,
               rec.dfpoly)$SparseUnivariatePolynomialFunctions2(Q,
                 gf))$DistinctDegreeFactorize(gf,
--                   SparseUnivariatePolynomial gf) | one?(f.exponent)]
                    SparseUnivariatePolynomial gf) | (f.exponent = 1)]
      empty? l => "failed"
      mdg := first l
      for ff in rest l repeat
        if degree(ff) < degree(mdg) then mdg := ff
      md := map((z1:gf):Q +-> convert(z1)@Z :: Q,
                 mdg)$SparseUnivariatePolynomialFunctions2(gf, Q)
      good?(res, h, b, d, p, md) => md
      "failed"

    good?(res, h, b, d, p, m) ==
      bd := badNum(res rem m)
      not (zero?(bd.den rem p) or zero?(bd.gcdnum rem p) or
        zero?(kbadBadNum(b,m) rem p) or zero?(kbadBadNum(d,m) rem p)
              or zero?(kbad3Num(h, m) rem p))
```

———-

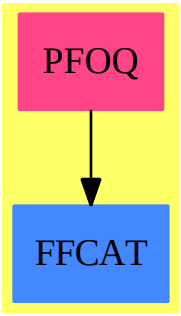
— PFO.dotabb —

```
"PFO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFO"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"PFO" -> "FS"
"PFO" -> "FFCAT"
```

———-

17.39 package PFOQ PointsOfFiniteOrderRational

17.39.1 PointsOfFiniteOrderRational (PFOQ)



Exports:
order torsion? torsionIfCan

— package PFOQ PointsOfFiniteOrderRational —

```
)abbrev package PFOQ PointsOfFiniteOrderRational
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1988
++ Date Last Updated: 3 August 1993
++ Keywords: divisor, algebraic, curve.
++ Description:
++ This package provides function for testing whether a divisor on a
++ curve is a torsion divisor.

PointsOfFiniteOrderRational(UP, UPUP, R): Exports == Implementation where
  UP   : UnivariatePolynomialCategory Fraction Integer
  UPUP : UnivariatePolynomialCategory Fraction UP
  R    : FunctionFieldCategory(Fraction Integer, UP, UPUP)

  PI  ==> PositiveInteger
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Integer
  FD  ==> FiniteDivisor(Q, UP, UPUP, R)

  Exports ==> with
    order       : FD -> Union(N, "failed")
      ++ order(f) \undocumented
    torsion?    : FD -> Boolean
      ++ torsion?(f) \undocumented
    torsionIfCan: FD -> Union(Record(order:N, function:R), "failed")
      ++ torsionIfCan(f) \undocumented
```

```
  Implementation ==> add
    import PointsOfFiniteOrderTools(UP, UPUP)

    possibleOrder: FD -> N
    ratcurve     : (FD, UPUP, Z) -> N
    rat          : (UPUP, FD, PI) -> N

    torsion? d  == order(d) case N

-- returns the potential order of d, 0 if d is of infinite order
    ratcurve(d, modulus, disc) ==
      mn  := minIndex(nm := numer(i := ideal d))
      h   := lift(hh := nm(mn + 1))
      s   := separate(retract(norm hh)@UP,
                b := retract(retract(nm.mn)@Fraction(UP))@UP).primePart
      bd  := badNum denom i
      r   := resultant(s, b)
      bad := lcm [disc, numer r, denom r, bd.den * bd.gcdnum, badNum h]$List(Z)
      n   := rat(modulus, d, p := getGoodPrime bad)
-- if n > 1 then it is cheaper to compute the order modulo a second prime,
-- since computing n * d could be very expensive
--      one? n => n
      (n = 1) => n
      m   := rat(modulus, d, getGoodPrime(p * bad))
      n = m => n
      0

    rat(pp, d, p) ==
      gf := InnerPrimeField p
      order(d, pp,
       (z1:Q):gf +->
        numer(z1)::gf / denom(z1)::gf)$ReducedDivisor(Q, UP, UPUP, R, gf)

-- returns the potential order of d, 0 if d is of infinite order
    possibleOrder d ==
--      zero?(genus()) or one?(#(numer ideal d)) => 1
      zero?(genus()) or (#(numer ideal d) = 1) => 1
      r := polyred definingPolynomial()$R
      ratcurve(d, r, doubleDisc r)

    order d ==
      zero?(n := possibleOrder(d := reduce d)) => "failed"
      principal? reduce(n::Z * d) => n
      "failed"

    torsionIfCan d ==
      zero?(n := possibleOrder(d := reduce d)) => "failed"
      (g := generator reduce(n::Z * d)) case "failed" => "failed"
      [n, g::R]
```

———-

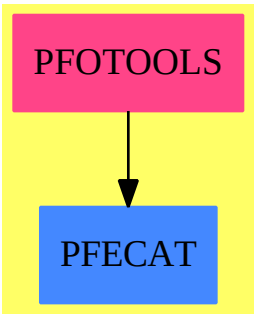
— **PFOQ.dotabb** —

```
"PFOQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFOQ"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"PFOQ" -> "FFCAT"
```

———-

17.40 package PFOTOOLS PointsOfFiniteOrderTools

17.40.1 PointsOfFiniteOrderTools (PFOTOOLS)



Exports:
badNum doubleDisc getGoodPrime mix polyred

— **package PFOTOOLS PointsOfFiniteOrderTools** —

```
)abbrev package PFOTOOLS PointsOfFiniteOrderTools
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Utilities for PFOQ and PFO

PointsOfFiniteOrderTools(UP, UPUP): Exports == Implementation where
  UP   : UnivariatePolynomialCategory Fraction Integer
  UPUP : UnivariatePolynomialCategory Fraction UP

  PI  ==> PositiveInteger
```

```
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Integer

  Exports ==> with
    getGoodPrime : Z -> PI
      ++ getGoodPrime n returns the smallest prime not dividing n
    badNum       : UP   -> Record(den:Z, gcdnum:Z)
      ++ badNum(p) \undocumented
    badNum       : UPUP -> Z
      ++ badNum(u) \undocumented
    mix          : List Record(den:Z, gcdnum:Z) -> Z
      ++ mix(l) \undocumented
    doubleDisc   : UPUP -> Z
      ++ doubleDisc(u) \undocumented
    polyred      : UPUP -> UPUP
      ++ polyred(u) \undocumented

  Implementation ==> add
    import IntegerPrimesPackage(Z)
    import UnivariatePolynomialCommonDenominator(Z, Q, UP)

    mix l          == lcm(lcm [p.den for p in l], gcd [p.gcdnum for p in l])
    badNum(p:UPUP) == mix [badNum(retract(c)@UP) for c in coefficients p]

    polyred r ==
      lcm [commonDenominator(retract(c)@UP) for c in coefficients r] * r

    badNum(p:UP) ==
      cd := splitDenominator p
      [cd.den, gcd [retract(c)@Z for c in coefficients(cd.num)]]

    getGoodPrime n ==
      p:PI := 3
      while zero?(n rem p) repeat
        p := nextPrime(p::Z)::PI
      p

    doubleDisc r ==
      d := retract(discriminant r)@UP
      retract(discriminant((d exquo gcd(d, differentiate d))::UP))@Z
```

———-

— PFOTOOLS.dotabb —

```
"PFOTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFOTOOLS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
```

```
"PFOTOOLS" -> "PFECAT"
```

———-

17.41 package PLPKCRV PolynomialPackageForCurve

— **PolynomialPackageForCurve.input** —

```
)set break resume
)sys rm -f PolynomialPackageForCurve.output
)spool PolynomialPackageForCurve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialPackageForCurve
--R
--R PolynomialPackageForCurve(K: Field,PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(dim,
--R Abbreviation for PolynomialPackageForCurve is PLPKCRV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLPKCRV
--R
--R------------------------------ Operations --------------------------------
--R eval : (PolyRing,ProjPt) -> K
--R minimalForm : (PolyRing,ProjPt) -> PolyRing
--R minimalForm : (PolyRing,ProjPt,Integer) -> PolyRing
--R multiplicity : (PolyRing,ProjPt) -> NonNegativeInteger
--R multiplicity : (PolyRing,ProjPt,Integer) -> NonNegativeInteger
--R pointInIdeal? : (List(PolyRing),ProjPt) -> Boolean
--R translateToOrigin : (PolyRing,ProjPt,Integer) -> PolyRing
--R translateToOrigin : (PolyRing,ProjPt) -> PolyRing
--R
--E 1

)spool
)lisp (bye)
```

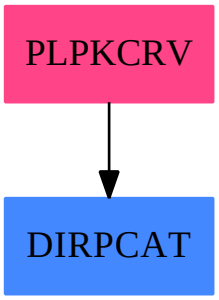
———-

— **PolynomialPackageForCurve.help** —

```
==========================================================================
PolynomialPackageForCurve examples
==========================================================================
```

```
See Also:
o )show PolynomialPackageForCurve
```

17.41.1 PolynomialPackageForCurve (PLPKCRV)



Exports:
eval minimalForm multiplicity pointInIdeal? translateToOrigin

— package PLPKCRV PolynomialPackageForCurve —

```
)abbrev package PLPKCRV PolynomialPackageForCurve
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PolynomialPackageForCurve(K,PolyRing,E,dim,ProjPt):Exp == Impl where
  K:Field
  dim:NonNegativeInteger
  E   :  DirectProductCategory(dim,NonNegativeInteger)
  PolyRing: FiniteAbelianMonoidRing(K,E) --   PolynomialCategory(K,E,OV)
  ProjPt : ProjectiveSpaceCategory(K)

  PackPoly ==> PackageForPoly(K,PolyRing,E,dim)

  Exp ==>  with

    pointInIdeal?: (List(PolyRing),ProjPt) -> Boolean
      ++ pointInIdeal? test if the given point is in the algebraic set
      ++ defined by the given list of polynomials.
```

```
    eval: (PolyRing,ProjPt) -> K
      ++ eval returns the value at given point.

    translateToOrigin: (PolyRing,ProjPt,Integer) -> PolyRing
      ++ translateToOrigin translate the polynomial from the given point
      ++ to the origin

    translateToOrigin: (PolyRing,ProjPt) -> PolyRing
      ++ translateToOrigin translate the polynomial from the given point
      ++ to the origin

    minimalForm: (PolyRing,ProjPt) -> PolyRing
      ++ minimalForm returns the minimal form after translation to
      ++ the origin.

    minimalForm: (PolyRing,ProjPt,Integer) -> PolyRing
      ++ minimalForm returns the minimal form after translation to
      ++ the origin.

    multiplicity: (PolyRing,ProjPt) -> NonNegativeInteger
      ++ multiplicity returns the multiplicity of the polynomial at
      ++ given point.

    multiplicity: (PolyRing,ProjPt,Integer) -> NonNegativeInteger
      ++ multiplicity returns the multiplicity of the polynomial at
      ++ given point.

  Impl ==> add

    import PolyRing
    import ProjPt
    import PackPoly

    translateToOrigin(pol,pt,nV)==
        zero?(pt.nV) => error "Impossible de translater"
pt:=homogenize(pt,nV)
lpt:List K:=list(pt)$ProjPt
translate(pol,lpt,nV)

    pointInIdeal?(lstPol,pt)==
      temp:Boolean:=true()$Boolean
      for pol in lstPol repeat
        temp:=(zero?(eval(pol,pt)) and temp)
      temp

    eval(f,pt)==
      zero? f => 0
      lpt:List(K) := list pt
      dd:List NonNegativeInteger := entries degree f
```

```
      lc:= leadingCoefficient f
      ee:= reduce( "*" , [ p**e for p in lpt for e in dd | ^zero?(e)], 1$K)
      lc * ee + eval( reductum f, pt)

    translateToOrigin(pol,pt)==
      translateToOrigin(pol,pt,lastNonNull(pt))

    multiplicity(crb,pt)==
      degreeOfMinimalForm(translateToOrigin(crb,pt))

    multiplicity(crb,pt,nV)==
      degreeOfMinimalForm(translateToOrigin(crb,pt,nV))

    minimalForm(crb,pt)==
      minimalForm(translateToOrigin(crb,pt))

    minimalForm(crb,pt,nV)==
      minimalForm(translateToOrigin(crb,pt,nV))
```

———-

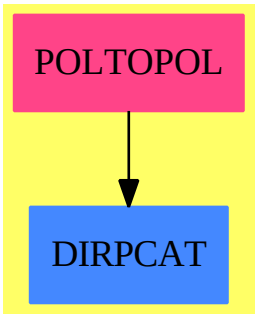
— PLPKCRV.dotabb —

```
"PLPKCRV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLPKCRV"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PLPKCRV" -> "DIRPCAT"
```

———-

17.42 package POLTOPOL PolToPol

17.42.1 PolToPol (POLTOPOL)



Exports:
dmpToHdmp dmpToP hdmpToDmp hdmpToP pToDmp pToHdmp

— package POLTOPOL PolToPol —

```
)abbrev package POLTOPOL PolToPol
++ Author : P.Gianni, Summer '88
++ Description:
++ Package with the conversion functions among different kind of polynomials

PolToPol(lv,R) : C == T

 where
  R      :    Ring
  lv     :    List Symbol
  NNI    ==>  NonNegativeInteger
  Ov     ==>  OrderedVariableList(lv)
  IES    ==>  IndexedExponents Symbol

  DP     ==>  DirectProduct(#lv,NonNegativeInteger)
  DPoly  ==>  DistributedMultivariatePolynomial(lv,R)

  HDP    ==>  HomogeneousDirectProduct(#lv,NonNegativeInteger)
  HDPoly ==>  HomogeneousDistributedMultivariatePolynomial(lv,R)
  P      ==>  Polynomial R
  VV     ==>  Vector NNI
  MPC3   ==>  MPolyCatFunctions3

  C == with
     dmpToHdmp    :    DPoly    -> HDPoly
       ++ dmpToHdmp(p) converts p from a \spadtype{DMP} to a \spadtype{HDMP}.
     hdmpToDmp    :   HDPoly    -> DPoly
       ++ hdmpToDmp(p) converts p from a \spadtype{HDMP} to a \spadtype{DMP}.
     pToHdmp      :     P       -> HDPoly
       ++ pToHdmp(p) converts p from a \spadtype{POLY} to a \spadtype{HDMP}.
     hdmpToP      :   HDPoly    -> P
       ++ hdmpToP(p) converts p from a \spadtype{HDMP} to a \spadtype{POLY}.
     dmpToP       :    DPoly    -> P
       ++ dmpToP(p) converts p from a \spadtype{DMP} to a \spadtype{POLY}.
     pToDmp       :     P       -> DPoly
       ++ pToDmp(p) converts p from a \spadtype{POLY} to a \spadtype{DMP}.
  T == add

    variable1(xx:Symbol):Ov == variable(xx)::Ov

   -- transform a P in a HDPoly --
    pToHdmp(pol:P) : HDPoly ==
      map(variable1,pol)$MPC3(Symbol,Ov,IES,HDP,R,P,HDPoly)
```

```
   -- transform an HDPoly in a P --
    hdmpToP(hdpol:HDPoly) : P ==
      map(convert,hdpol)$MPC3(Ov,Symbol,HDP,IES,R,HDPoly,P)

   -- transform an DPoly in a P --
    dmpToP(dpol:DPoly) : P ==
      map(convert,dpol)$MPC3(Ov,Symbol,DP,IES,R,DPoly,P)

   -- transform a P in a DPoly --
    pToDmp(pol:P) : DPoly ==
      map(variable1,pol)$MPC3(Symbol,Ov,IES,DP,R,P,DPoly)

   -- transform a DPoly in a HDPoly --
    dmpToHdmp(dpol:DPoly) : HDPoly ==
      dpol=0 => 0$HDPoly
      monomial(leadingCoefficient dpol,
               directProduct(degree(dpol)::VV)$HDP)$HDPoly+
                                                 dmpToHdmp(reductum dpol)

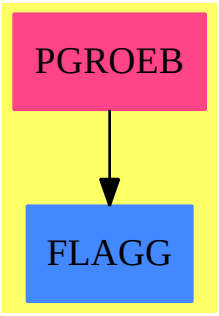
   -- transform a HDPoly in a DPoly --
    hdmpToDmp(hdpol:HDPoly) : DPoly ==
      hdpol=0 => 0$DPoly
      dd:DP:= directProduct((degree hdpol)::VV)$DP
      monomial(leadingCoefficient hdpol,dd)$DPoly+
               hdmpToDmp(reductum hdpol)
```

— POLTOPOL.dotabb —

```
"POLTOPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLTOPOL"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"POLTOPOL" -> "DIRPCAT"
```

17.43 package PGROEB PolyGroebner

17.43.1 PolyGroebner (PGROEB)



Exports:
lexGroebner totalGroebner

— package PGROEB PolyGroebner —

```
)abbrev package PGROEB PolyGroebner
++ Author: P. Gianni
++ Date Created: Summer 1988
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: GroebnerPackage
++ Also See:
++ AMS Classifications:
++ Keywords: groebner basis, polynomial ideals
++ References:
++ Description:
++ Groebner functions for P F
++ This package is an interface package to the groebner basis
++ package which allows you to compute groebner bases for polynomials
++ in either lexicographic ordering or total degree ordering refined
++ by reverse lex. The input is the ordinary polynomial type which
++ is internally converted to a type with the required ordering.
++ The resulting grobner basis is converted back to ordinary polynomials.
++ The ordering among the variables is controlled by an explicit list
++ of variables which is passed as a second argument. The coefficient
++ domain is allowed to be any gcd domain, but the groebner basis is
++ computed as if the polynomials were over a field.

PolyGroebner(F) : C == T

 where
  F      :   GcdDomain
```

```
NNI    ==>  NonNegativeInteger
P      ==>  Polynomial F
L      ==>  List
E      ==>  Symbol

C == with
   lexGroebner   :    (L P,L E)  ->  L P
     ++ lexGroebner(lp,lv) computes Groebner basis
     ++ for the list of polynomials lp in lexicographic order.
     ++ The variables are ordered by their position in the list lv.

   totalGroebner :    (L P, L E) ->  L P
     ++ totalGroebner(lp,lv) computes Groebner basis
     ++ for the list of polynomials lp with the terms
     ++ ordered first by total degree and then
     ++ refined by reverse lexicographic ordering.
     ++ The variables are ordered by their position in the list lv.

T == add
   lexGroebner(lp: L P,lv:L E) : L P ==
     PP:=  PolToPol(lv,F)
     DPoly := DistributedMultivariatePolynomial(lv,F)
     DP:=DirectProduct(#lv,NNI)
     OV:=OrderedVariableList lv
     b:L DPoly:=[pToDmp(pol)$PP for pol in lp]
     gb:L DPoly :=groebner(b)$GroebnerPackage(F,DP,OV,DPoly)
     [dmpToP(pp)$PP for pp in gb]

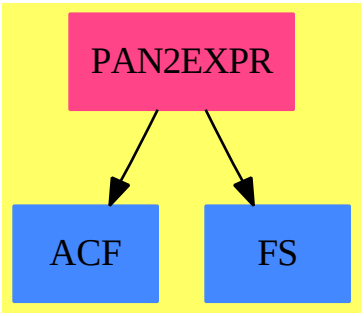
   totalGroebner(lp: L P,lv:L E) : L P ==
     PP:=  PolToPol(lv,F)
     HDPoly := HomogeneousDistributedMultivariatePolynomial(lv,F)
     HDP:=HomogeneousDirectProduct(#lv,NNI)
     OV:=OrderedVariableList lv
     b:L HDPoly:=[pToHdmp(pol)$PP for pol in lp]
     gb:=groebner(b)$GroebnerPackage(F,HDP,OV,HDPoly)
     [hdmpToP(pp)$PP for pp in gb]
```

— PGROEB.dotabb —

```
"PGROEB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PGROEB"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PGROEB" -> "FLAGG"
```

17.44 package PAN2EXPR PolynomialAN2Expression

17.44.1 PolynomialAN2Expression (PAN2EXPR)



Exports:
coerce

— package PAN2EXPR PolynomialAN2Expression —

```
)abbrev package PAN2EXPR PolynomialAN2Expression
++ Author: Barry Trager
++ Date Created: 8 Oct 1991
++ Description:
++ This package provides a coerce from polynomials over
++ algebraic numbers to \spadtype{Expression AlgebraicNumber}.

PolynomialAN2Expression():Target == Implementation where
  EXPR ==> Expression(Integer)
  AN ==> AlgebraicNumber
  PAN ==> Polynomial AN
  SY ==> Symbol
  Target ==> with
      coerce: Polynomial AlgebraicNumber -> Expression(Integer)
        ++ coerce(p) converts the polynomial \spad{p} with algebraic number
        ++ coefficients to \spadtype{Expression Integer}.
      coerce: Fraction Polynomial AlgebraicNumber -> Expression(Integer)
        ++ coerce(rf) converts \spad{rf}, a fraction of polynomial
        ++ \spad{p} with
        ++ algebraic number coefficients to \spadtype{Expression Integer}.
  Implementation ==> add
    coerce(p:PAN):EXPR ==
        map(x+->x::EXPR, y+->y::EXPR, p)$PolynomialCategoryLifting(
                                  IndexedExponents SY, SY, AN, PAN, EXPR)
    coerce(rf:Fraction PAN):EXPR ==
        numer(rf)::EXPR / denom(rf)::EXPR
```

———-

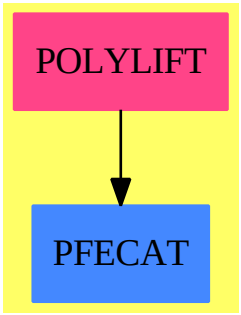
— PAN2EXPR.dotabb —

```
"PAN2EXPR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PAN2EXPR"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PAN2EXPR" -> "ACF"
"PAN2EXPR" -> "FS"
```

———-

17.45 package POLYLIFT PolynomialCategoryLifting

17.45.1 PolynomialCategoryLifting (POLYLIFT)



Exports:
map

— package POLYLIFT PolynomialCategoryLifting —

```
)abbrev package POLYLIFT PolynomialCategoryLifting
++ Author: Manuel Bronstein
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides a very general map function, which
```

```
++ given a set S and polynomials over R with maps from the
++ variables into S and the coefficients into S, maps polynomials
++ into S. S is assumed to support \spad{+}, \spad{*} and \spad{**}.

PolynomialCategoryLifting(E,Vars,R,P,S): Exports == Implementation where
  E   : OrderedAbelianMonoidSup
  Vars: OrderedSet
  R   : Ring
  P   : PolynomialCategory(R, E, Vars)
  S   : SetCategory with
          "+" : (%, %) -> %
          "*" : (%, %) -> %
          "**": (%, NonNegativeInteger) -> %

  Exports ==> with
    map: (Vars -> S, R -> S, P) -> S
      ++ map(varmap, coefmap, p) takes a
      ++ varmap, a mapping from the variables of polynomial p into S,
      ++ coefmap, a mapping from coefficients of p into S, and p, and
      ++ produces a member of S using the corresponding arithmetic.
      ++ in S

  Implementation ==> add
    map(fv, fc, p) ==
      (x1 := mainVariable p) case "failed" => fc leadingCoefficient p
      up := univariate(p, x1::Vars)
      t  := fv(x1::Vars)
      ans:= fc 0
      while not ground? up repeat
        ans := ans + map(fv,fc, leadingCoefficient up) * t ** (degree up)
        up  := reductum up
      ans + map(fv, fc, leadingCoefficient up)
```

———-

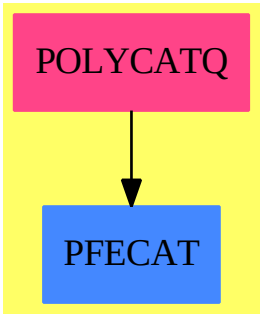
— POLYLIFT.dotabb —

```
"POLYLIFT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYLIFT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYLIFT" -> "PFECAT"
```

———-

17.46 package POLYCATQ PolynomialCategoryQuotient-Functions

17.46.1 PolynomialCategoryQuotientFunctions (POLYCATQ)



Exports:

variables	isExpt	isPlus	isPower	isTimes
mainVariable	multivariate	univariate		

— **package POLYCATQ PolynomialCategoryQuotientFunctions** —

```
)abbrev package POLYCATQ PolynomialCategoryQuotientFunctions
++ Author: Manuel Bronstein
++ Date Created: March 1988
++ Date Last Updated: 9 July 1990
++ Keywords: polynomial, fraction, transformation
++ Description:
++ Manipulations on polynomial quotients
++ This package transforms multivariate polynomials or fractions into
++ univariate polynomials or fractions, and back.

PolynomialCategoryQuotientFunctions(E, V, R, P, F):
 Exports == Implementation where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: Ring
  P: PolynomialCategory(R, E, V)
  F: Field with
    coerce: P -> %
    numer : % -> P
    denom : % -> P

  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP

  Exports ==> with
```

```
    variables   : F -> List V
      ++ variables(f) returns the list of variables appearing
      ++ in the numerator or the denominator of f.
    mainVariable: F -> Union(V, "failed")
      ++ mainVariable(f) returns the highest variable appearing
      ++ in the numerator or the denominator of f, "failed" if
      ++ f has no variables.
    univariate  : (F, V) -> RF
      ++ univariate(f, v) returns f viewed as a univariate
      ++ rational function in v.
    multivariate: (RF, V) -> F
      ++ multivariate(f, v) applies both the numerator and
      ++ denominator of f to v.
    univariate  : (F, V, UP) -> UP
      ++ univariate(f, x, p) returns f viewed as a univariate
      ++ polynomial in x, using the side-condition \spad{p(x) = 0}.
    isPlus      : F -> Union(List F, "failed")
      ++ isPlus(p) returns [m1,...,mn] if \spad{p = m1 + ... + mn} and
      ++ \spad{n > 1}, "failed" otherwise.
    isTimes     : F -> Union(List F, "failed")
      ++ isTimes(p) returns \spad{[a1,...,an]} if
      ++ \spad{p = a1 ... an} and \spad{n > 1},
      ++ "failed" otherwise.
    isExpt      : F -> Union(Record(var:V, exponent:Integer), "failed")
      ++ isExpt(p) returns \spad{[x, n]} if \spad{p = x**n} and \spad{n <> 0},
      ++ "failed" otherwise.
    isPower     : F -> Union(Record(val:F, exponent:Integer), "failed")
      ++ isPower(p) returns \spad{[x, n]} if \spad{p = x**n} and \spad{n <> 0},
      ++ "failed" otherwise.

  Implementation ==> add
    P2UP: (P, V) -> UP

    univariate(f, x) == P2UP(numer f, x) / P2UP(denom f, x)

    univariate(f, x, modulus) ==
      (bc := extendedEuclidean(P2UP(denom f, x), modulus, 1))
             case "failed" => error "univariate: denominator is 0 mod p"
      (P2UP(numer f, x) * bc.coef1) rem modulus

    multivariate(f, x) ==
      v := x::P::F
      ((numer f) v) / ((denom f) v)

    mymerge:(List V,List V) ->List V
    mymerge(l:List V,m:List V):List V==
         empty? l => m
         empty? m => l
         first l = first m => cons(first l,mymerge(rest l,rest m))
         first l > first m => cons(first l,mymerge(rest l,m))
```

```
        cons(first m,mymerge(l,rest m))

    variables f ==
      mymerge(variables numer f, variables denom f)

    isPower f ==
      (den := denom f) ^= 1 =>
        numer f ^= 1 => "failed"
        (ur := isExpt den) case "failed" => [den::F, -1]
        r := ur::Record(var:V, exponent:NonNegativeInteger)
        [r.var::P::F, - (r.exponent::Integer)]
      (ur := isExpt numer f) case "failed" => "failed"
      r := ur::Record(var:V, exponent:NonNegativeInteger)
      [r.var::P::F, r.exponent::Integer]

    isExpt f ==
      (ur := isExpt numer f) case "failed" =>
--        one? numer f =>
        (numer f) = 1 =>
          (ur := isExpt denom f) case "failed" => "failed"
          r := ur::Record(var:V, exponent:NonNegativeInteger)
          [r.var, - (r.exponent::Integer)]
        "failed"
      r := ur::Record(var:V, exponent:NonNegativeInteger)
--      one? denom f => [r.var, r.exponent::Integer]
      (denom f) = 1 => [r.var, r.exponent::Integer]
      "failed"

    isTimes f ==
      t := isTimes(num := numer f)
      l:Union(List F, "failed") :=
        t case "failed" => "failed"
        [x::F for x in t]
--      one?(den := denom f) => l
      ((den := denom f) = 1) => l
--      one? num => "failed"
      num = 1 => "failed"
      d := inv(den::F)
      l case "failed" => [num::F, d]
      concat_!(l::List(F), d)

    isPlus f ==
      denom f ^= 1 => "failed"
      (s := isPlus numer f) case "failed" => "failed"
      [x::F for x in s]

    mainVariable f ==
      a := mainVariable numer f
      (b := mainVariable denom f) case "failed" => a
      a case "failed" => b
```

```
      max(a::V, b::V)

    P2UP(p, x) ==
      map(z +-> z::F,
          univariate(p, x))$SparseUnivariatePolynomialFunctions2(P, F)
```

———-

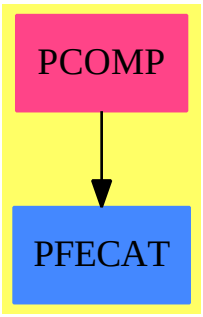
— **POLYCATQ.dotabb** —

```
"POLYCATQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYCATQ"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYCATQ" -> "PFECAT"
```

———-

17.47 package PCOMP PolynomialComposition

17.47.1 PolynomialComposition (PCOMP)



Exports:
compose

— **package PCOMP PolynomialComposition** —

```
)abbrev package PCOMP PolynomialComposition
++ References: Kozen and Landau, Cornell University  TR 86-773
++ Description:
++ Polynomial composition and decomposition functions\br
++ If f = g o h then g=leftFactor(f,h) and h=rightFactor(f,g)

PolynomialComposition(UP: UnivariatePolynomialCategory(R), R: Ring): with
```

```
        compose: (UP, UP) -> UP
          ++ compose(p,q) \undocumented
    == add
        compose(g, h) ==
            r: UP := 0
            while g ^= 0 repeat
                r := leadingCoefficient(g)*h**degree(g) + r
                g := reductum g
            r
```

———-

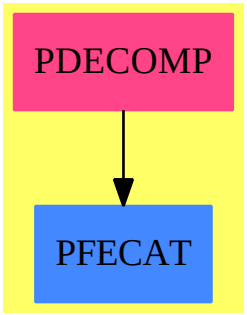
— **PCOMP.dotabb** —

```
"PCOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PCOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PCOMP" -> "PFECAT"
```

———-

17.48 package PDECOMP PolynomialDecomposition

17.48.1 PolynomialDecomposition (PDECOMP)



Exports:
decompose leftFactor rightFactorCandidate

— **package PDECOMP PolynomialDecomposition** —

```
)abbrev package PDECOMP PolynomialDecomposition
++ References: Kozen and Landau, Cornell University  TR 86-773
++ Description:
```

```
++ Polynomial composition and decomposition functions\br
++ If f = g o h then g=leftFactor(f,h) and h=rightFactor(f,g)

PolynomialDecomposition(UP, F): PDcat == PDdef where
    F:Field
    UP:UnivariatePolynomialCategory F
    NNI ==> NonNegativeInteger
    LR  ==> Record(left: UP, right: UP)

    PDcat == with
        decompose: UP -> List UP
          ++ decompose(up) \undocumented
        decompose: (UP, NNI, NNI) -> Union(LR, "failed")
          ++ decompose(up,m,n) \undocumented
        leftFactor: (UP, UP) -> Union(UP, "failed")
          ++ leftFactor(p,q) \undocumented
        rightFactorCandidate:  (UP, NNI) -> UP
          ++ rightFactorCandidate(p,n) \undocumented
    PDdef == add
        leftFactor(f, h) ==
             g: UP := 0
             for i in 0.. while f ^= 0 repeat
                 fr := divide(f, h)
                 f := fr.quotient; r := fr.remainder
                 degree r > 0 => return "failed"
                 g := g + r * monomial(1, i)
             g

        decompose(f, dg, dh) ==
            df := degree f
            dg*dh ^= df => "failed"
            h := rightFactorCandidate(f, dh)
            g := leftFactor(f, h)
            g case "failed" => "failed"
            [g::UP, h]

        decompose f ==
            df := degree f
            for dh in 2..df-1 | df rem dh = 0 repeat
                h := rightFactorCandidate(f, dh)
                g := leftFactor(f, h)
                g case UP => return
                    append(decompose(g::UP), decompose h)
            [f]
        rightFactorCandidate(f, dh) ==
            f  := f/leadingCoefficient f
            df := degree f
            dg := df quo dh
            h  := monomial(1, dh)
            for k in 1..dh repeat
```

```
        hdg:= h**dg
        c  := (coefficient(f,(df-k)::NNI)-coefficient(hdg,(df-k)::NNI))/(dg::F)
        h  := h + monomial(c, (dh-k)::NNI)
    h - monomial(coefficient(h, 0), 0) -- drop constant term
```

———-

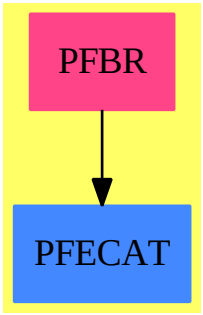
— **PDECOMP.dotabb** —

```
"PDECOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PDECOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PDECOMP" -> "PFECAT"
```

———-

17.49 package PFBR PolynomialFactorizationByRecursion

17.49.1 PolynomialFactorizationByRecursion (PFBR)



Exports:

randomR	bivariateSLPEBR	factorByRecursion
factorSFBRlcUnit	factorSquareFreeByRecursion	solveLinearPolynomialEquationByRecursion

— **package PFBR PolynomialFactorizationByRecursion** —

```
)abbrev package PFBR PolynomialFactorizationByRecursion
++ Description:
++ PolynomialFactorizationByRecursion(R,E,VarSet,S)
++ is used for factorization of sparse univariate polynomials over
++ a domain S of multivariate polynomials over R.
```

```
PolynomialFactorizationByRecursion(R,E, VarSet:OrderedSet, S): public ==
 private where
  R:PolynomialFactorizationExplicit
  E:OrderedAbelianMonoidSup
  S:PolynomialCategory(R,E,VarSet)
  PI ==> PositiveInteger
  SupR ==> SparseUnivariatePolynomial R
  SupSupR ==> SparseUnivariatePolynomial SupR
  SupS ==> SparseUnivariatePolynomial S
  SupSupS ==> SparseUnivariatePolynomial SupS
  LPEBFS ==> LinearPolynomialEquationByFractions(S)
  public ==  with
     solveLinearPolynomialEquationByRecursion: (List SupS, SupS)  ->
                                               Union(List SupS,"failed")
        ++ \spad{solveLinearPolynomialEquationByRecursion([p1,...,pn],p)}
        ++ returns the list of polynomials \spad{[q1,...,qn]}
        ++ such that \spad{sum qi/pi = p / prod pi}, a
        ++ recursion step for solveLinearPolynomialEquation
        ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
        ++ (see \spadfun{solveLinearPolynomialEquation}).
        ++ If no such list of qi exists, then "failed" is returned.
     factorByRecursion:  SupS -> Factored SupS
        ++ factorByRecursion(p) factors polynomial p. This function
        ++ performs the recursion step for factorPolynomial,
        ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
        ++ (see \spadfun{factorPolynomial})
     factorSquareFreeByRecursion:  SupS -> Factored SupS
        ++ factorSquareFreeByRecursion(p) returns the square free
        ++ factorization of p. This functions performs
        ++ the recursion step for factorSquareFreePolynomial,
        ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
        ++ (see \spadfun{factorSquareFreePolynomial}).
     randomR: -> R  -- has to be global, since has alternative definitions
        ++ randomR produces a random element of R
     bivariateSLPEBR: (List SupS, SupS,  VarSet)  -> Union(List SupS,"failed")
        ++ bivariateSLPEBR(lp,p,v) implements
        ++ the bivariate case of solveLinearPolynomialEquationByRecursion
        ++ its implementation depends on R
     factorSFBRlcUnit: (List VarSet, SupS) -> Factored SupS
        ++ factorSFBRlcUnit(p) returns the square free factorization of
        ++ polynomial p
        ++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivariate})
        ++ in the case where the leading coefficient of p
        ++ is a unit.
  private  == add
   supR: SparseUnivariatePolynomial R
   pp: SupS
   lpolys,factors: List SupS
   vv:VarSet
   lvpolys,lvpp: List VarSet
```

```
r:R
lr:List R
import FactoredFunctionUtilities(SupS)
import FactoredFunctions2(S,SupS)
import FactoredFunctions2(SupR,SupS)
import CommuteUnivariatePolynomialCategory(S,SupS, SupSupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupS,SupSupS)
import UnivariatePolynomialCategoryFunctions2(SupS,SupSupS,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,R,SupR)
import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupR,SupSupR)
import UnivariatePolynomialCategoryFunctions2(SupR,SupSupR,S,SupS)
hensel: (SupS,VarSet,R,List SupS) ->
        Union(Record(fctrs:List SupS),"failed")
chooseSLPEViableSubstitutions: (List VarSet,List SupS,SupS) ->
 Record(substnsField:List R,lpolysRField:List SupR,ppRField:SupR)
  --++ chooseSLPEViableSubstitutions(lv,lp,p) chooses substitutions
  --++ for the variables in first arg (which are all
  --++ the variables that exist) so that the polys in second argument don't
  --++ drop in degree and remain square-free, and third arg doesn't drop
  --++ drop in degree
chooseFSQViableSubstitutions: (List VarSet,SupS) ->
 Record(substnsField:List R,ppRField:SupR)
  --++ chooseFSQViableSubstitutions(lv,p) chooses substitutions for the variables in first ar
  --++ the variables that exist) so that the second argument poly doesn't
  --++ drop in degree and remains square-free
raise: SupR -> SupS
lower: SupS -> SupR
SLPEBR: (List SupS, List VarSet, SupS, List VarSet)  ->
                                      Union(List SupS,"failed")
factorSFBRlcUnitInner: (List VarSet, SupS,R) ->
                                      Union(Factored SupS,"failed")
hensel(pp,vv,r,factors) ==
   origFactors:=factors
   totdegree:Integer:=0
   proddegree:Integer:=
               "max"/[degree(u,vv) for u in coefficients pp]
   n:PI:=1
   prime:=vv::S - r::S
   foundFactors:List SupS:=empty()
   while (totdegree <= proddegree) repeat
       pn:=prime**n
       Ecart:=(pp-*/factors) exquo  pn
       Ecart case "failed" =>
             error "failed lifting in hensel in PFBR"
       zero? Ecart =>
          -- then we have all the factors
          return [append(foundFactors, factors)]
       step:=solveLinearPolynomialEquation(origFactors,
                                        map(z1 +-> eval(z1,vv,r),
```

```
                                              Ecart))
        step case "failed" => return "failed" -- must be a false split
        factors:=[a+b*pn for a in factors for b in step]
        for a in factors for c in origFactors repeat
            pp1:= pp exquo a
            pp1 case "failed" => "next"
            pp:=pp1
            proddegree := proddegree - "max"/[degree(u,vv)
                                             for u in coefficients a]
            factors:=remove(a,factors)
            origFactors:=remove(c,origFactors)
            foundFactors:=[a,:foundFactors]
        #factors < 2 =>
           return [(empty? factors => foundFactors;
                                  [pp,:foundFactors])]
        totdegree:= +/["max"/[degree(u,vv)
                              for u in coefficients u1]
                       for u1 in factors]
        n:=n+1
    "failed" -- must have been a false split

  factorSFBRlcUnitInner(lvpp,pp,r) ==
     -- pp is square-free as a Sup, and its coefficients have precisely
     -- the variables of lvpp. Furthermore, its LC is a unit
     -- returns "failed" if the substitution is bad, else a factorization
     ppR:=map(z1 +-> eval(z1,first lvpp,r),pp)
     degree ppR < degree pp => "failed"
     degree gcd(ppR,differentiate ppR) >0 => "failed"
     factors:=
        empty? rest lvpp =>
           fDown:=factorSquareFreePolynomial map(z1 +-> retract(z1)::R,ppR)
           [raise (unit fDown * factorList(fDown).first.fctr),
            :[raise u.fctr for u in factorList(fDown).rest]]
        fSame:=factorSFBRlcUnit(rest lvpp,ppR)
        [unit fSame * factorList(fSame).first.fctr,
         :[uu.fctr for uu in factorList(fSame).rest]]
     #factors = 1 => makeFR(1,[["irred",pp,1]])
     hen:=hensel(pp,first lvpp,r,factors)
     hen case "failed" => "failed"
     makeFR(1,[["irred",u,1] for u in hen.fctrs])
  if R has StepThrough then
    factorSFBRlcUnit(lvpp,pp) ==
      val:R := init()
      while true repeat
         tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
         not (tempAns case "failed") => return tempAns
         val1:=nextItem val
         val1 case "failed" =>
           error "at this point, we know we have a finite field"
         val:=val1
```

```
  else
    factorSFBRlcUnit(lvpp,pp) ==
      val:R := randomR()
      while true repeat
         tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
         not (tempAns case "failed") => return tempAns
         val := randomR()
  if R has random: -> R then
     randomR() == random()
  else randomR() == (random()$Integer)::R
  if R has FiniteFieldCategory then
    bivariateSLPEBR(lpolys,pp,v) ==
      lpolysR:List SupSupR:=[map(univariate,u) for u in lpolys]
      ppR: SupSupR:=map(univariate,pp)
      ans:=solveLinearPolynomialEquation(lpolysR,ppR)$SupR
      ans case "failed" => "failed"
      [map(z1 +-> multivariate(z1,v),w) for w in ans]
  else
    bivariateSLPEBR(lpolys,pp,v) ==
      solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS
  chooseFSQViableSubstitutions(lvpp,pp) ==
    substns:List R
    ppR: SupR
    while true repeat
       substns:= [randomR() for v in lvpp]
       zero? eval(leadingCoefficient pp,lvpp,substns ) => "next"
       ppR:=map(z1 +->(retract eval(z1,lvpp,substns))::R,pp)
       degree gcd(ppR,differentiate ppR)>0 => "next"
       leave
    [substns,ppR]
  chooseSLPEViableSubstitutions(lvpolys,lpolys,pp) ==
    substns:List R
    lpolysR:List SupR
    ppR: SupR
    while true repeat
       substns:= [randomR() for v in lvpolys]
       zero? eval(leadingCoefficient pp,lvpolys,substns ) => "next"
       "or"/[zero? eval(leadingCoefficient u,lvpolys,substns)
                   for u in lpolys] => "next"
       lpolysR:=[map(z1 +-> (retract eval(z1,lvpolys,substns))::R,u)
                 for u in lpolys]
       uu:=lpolysR
       while not empty? uu repeat
         "or"/[ degree(gcd(uu.first,v))>0 for v in uu.rest] => leave
         uu:=rest uu
       not empty? uu => "next"
       leave
    ppR:=map(z1 +-> (retract eval(z1,lvpolys,substns))::R,pp)
    [substns,lpolysR,ppR]
  raise(supR) == map(z1 +-> z1:R::S,supR)
```

```
  lower(pp) == map(z1 +-> retract(z1)::R,pp)
  SLPEBR(lpolys,lvpolys,pp,lvpp) ==
    not empty? (m:=setDifference(lvpp,lvpolys)) =>
        v:=first m
        lvpp:=remove(v,lvpp)
        pp1:SupSupS :=swap map(z1 +-> univariate(z1,v),pp)
        -- pp1 is mathematically equal to pp, but is in S[z][v]
        -- so we wish to operate on all of its coefficients
        ans:List SupSupS:= [0 for u in lpolys]
        for m in reverse_! monomials pp1 repeat
            ans1:=SLPEBR(lpolys,lvpolys,leadingCoefficient m,lvpp)
            ans1 case "failed" => return "failed"
            d:=degree m
            ans:=[monomial(a1,d)+a for a in ans for a1 in ans1]
        [map(z1 +-> multivariate(z1,v),swap pp1) for pp1 in ans]
    empty? lvpolys =>
        lpolysR:List SupR
        ppR:SupR
        lpolysR:=[map(retract,u) for u in lpolys]
        ppR:=map(retract,pp)
        ansR:=solveLinearPolynomialEquation(lpolysR,ppR)
        ansR case "failed" => return "failed"
        [map(z1 +-> z1::S,uu) for uu in ansR]
    cVS:=chooseSLPEViableSubstitutions(lvpolys,lpolys,pp)
    ansR:=solveLinearPolynomialEquation(cVS.lpolysRField,cVS.ppRField)
    ansR case "failed" => "failed"
    #lvpolys = 1 => bivariateSLPEBR(lpolys,pp, first lvpolys)
    solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS

  solveLinearPolynomialEquationByRecursion(lpolys,pp) ==
    lvpolys := removeDuplicates_!
                concat [ concat [variables z for z in coefficients u]
                                                for u in lpolys]
    lvpp := removeDuplicates_!
              concat [variables z for z in coefficients pp]
    SLPEBR(lpolys,lvpolys,pp,lvpp)

  factorByRecursion pp ==
    lv:List(VarSet) := removeDuplicates_!
                         concat [variables z for z in coefficients pp]
    empty? lv =>
        map(raise,factorPolynomial lower pp)
    c:=content pp
    unit? c => refine(squareFree pp,factorSquareFreeByRecursion)
    pp:=(pp exquo c)::SupS
    mergeFactors(refine(squareFree pp,factorSquareFreeByRecursion),
                 map(z1 +-> z1:S::SupS,factor(c)$S))
  factorSquareFreeByRecursion pp ==
    lv:List(VarSet) := removeDuplicates_!
                         concat [variables z for z in coefficients pp]
```

```
    empty? lv =>
        map(raise,factorPolynomial lower pp)
    unit? (lcpp := leadingCoefficient pp) => factorSFBRlcUnit(lv,pp)
    oldnfact:NonNegativeInteger:= 999999
                    -- I hope we never have to factor a polynomial
                    -- with more than this number of factors
    lcppPow:S
    while true repeat
      cVS:=chooseFSQViableSubstitutions(lv,pp)
      factorsR:=factorSquareFreePolynomial(cVS.ppRField)
      (nfact:=numberOfFactors factorsR) = 1 =>
                return makeFR(1,[["irred",pp,1]])
      -- OK, force all leading coefficients to be equal to the leading
      -- coefficient of the input
      nfact > oldnfact => "next"   -- can't be a good reduction
      oldnfact:=nfact
      factors:=[(lcpp exquo leadingCoefficient u.fctr)::S * raise u.fctr
                for u in factorList factorsR]
      ppAdjust:=(lcppPow:=lcpp**#(rest factors)) * pp
      lvppList:=lv
      OK:=true
      for u in lvppList for v in cVS.substnsField repeat
          hen:=hensel(ppAdjust,u,v,factors)
          hen case "failed" =>
              OK:=false
              "leave"
          factors:=hen.fctrs
      OK => leave
    factors:=[ (lc:=content w;
                lcppPow:=(lcppPow exquo lc)::S;
                 (w exquo lc)::SupS)
               for w in factors]
    not unit? lcppPow =>
        error "internal error in factorSquareFreeByRecursion"
    makeFR((recip lcppPow)::S::SupS,
            [["irred",w,1] for w in factors])
```

———-

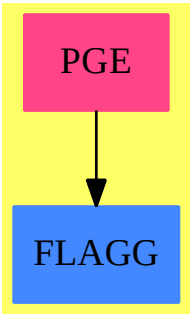
— PFBR.dotabb —

```
"PFBR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFBR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFBR" -> "PFECAT"
```

———-

17.50 package PFBRU PolynomialFactorizationByRecursionUnivariate

17.50.1 PolynomialFactorizationByRecursionUnivariate (PFBRU)



Exports:

randomR	factorByRecursion	factorSFBRlcUnit
factorSquareFreeByRecursion	solveLinearPolynomialEquationByRecursion	

— **package PFBRU PolynomialFactorizationByRecursionUnivariate** —

```
)abbrev package PFBRU PolynomialFactorizationByRecursionUnivariate
++ Description:
++ PolynomialFactorizationByRecursionUnivariate
++ R is a \spadfun{PolynomialFactorizationExplicit} domain,
++ S is univariate polynomials over R
++ We are interested in handling SparseUnivariatePolynomials over
++ S, is a variable we shall call z

PolynomialFactorizationByRecursionUnivariate(R, S): public == private where
  R:PolynomialFactorizationExplicit
  S:UnivariatePolynomialCategory(R)
  PI ==> PositiveInteger
  SupR ==> SparseUnivariatePolynomial R
  SupSupR ==> SparseUnivariatePolynomial SupR
  SupS ==> SparseUnivariatePolynomial S
  SupSupS ==> SparseUnivariatePolynomial SupS
  LPEBFS ==> LinearPolynomialEquationByFractions(S)
  public ==  with
     solveLinearPolynomialEquationByRecursion: (List SupS, SupS)  ->
                                               Union(List SupS,"failed")
        ++ \spad{solveLinearPolynomialEquationByRecursion([p1,...,pn],p)}
        ++ returns the list of polynomials \spad{[q1,...,qn]}
        ++ such that \spad{sum qi/pi = p / prod pi}, a
        ++ recursion step for solveLinearPolynomialEquation
```

```
       ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
       ++ (see \spadfun{solveLinearPolynomialEquation}).
       ++ If no such list of qi exists, then "failed" is returned.
    factorByRecursion:  SupS -> Factored SupS
       ++ factorByRecursion(p) factors polynomial p. This function
       ++ performs the recursion step for factorPolynomial,
       ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
       ++ (see \spadfun{factorPolynomial})
    factorSquareFreeByRecursion:  SupS -> Factored SupS
       ++ factorSquareFreeByRecursion(p) returns the square free
       ++ factorization of p. This functions performs
       ++ the recursion step for factorSquareFreePolynomial,
       ++ as defined in \spadfun{PolynomialFactorizationExplicit} category
       ++ (see \spadfun{factorSquareFreePolynomial}).
    randomR: -> R  -- has to be global, since has alternative definitions
       ++ randomR() produces a random element of R
    factorSFBRlcUnit: (SupS) -> Factored SupS
       ++ factorSFBRlcUnit(p) returns the square free factorization of
       ++ polynomial p
       ++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivaria
       ++ in the case where the leading coefficient of p
       ++ is a unit.
  private  == add
   supR: SparseUnivariatePolynomial R
   pp: SupS
   lpolys,factors: List SupS
   r:R
   lr:List R
   import FactoredFunctionUtilities(SupS)
   import FactoredFunctions2(SupR,SupS)
   import FactoredFunctions2(S,SupS)
   import UnivariatePolynomialCategoryFunctions2(S,SupS,R,SupR)
   import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
   -- local function declarations
   raise: SupR -> SupS
   lower: SupS -> SupR
   factorSFBRlcUnitInner: (SupS,R) -> Union(Factored SupS,"failed")
   hensel: (SupS,R,List SupS) ->
           Union(Record(fctrs:List SupS),"failed")
   chooseFSQViableSubstitutions: (SupS) ->
    Record(substnsField:R,ppRField:SupR)
     --++ chooseFSQViableSubstitutions(p), p is a sup
     --++ ("sparse univariate polynomial")
     --++ over a sup over R, returns a record
     --++ \spad{[substnsField: r, ppRField: q]} where r is a substitution point
     --++ q is a sup over R so that the (implicit) variable in q
     --++ does not drop in degree and remains square-free.
   -- here for the moment, until it compiles
   -- N.B., we know that R is NOT a FiniteField, since
   -- that is meant to have a special implementation, to break the
```

```
-- recursion
solveLinearPolynomialEquationByRecursion(lpolys,pp) ==
  lhsdeg:="max"/["max"/[degree v for v in coefficients u] for u in lpolys]
  rhsdeg:="max"/[degree v for v in coefficients pp]
  lhsdeg = 0 =>
    lpolysLower:=[lower u for u in lpolys]
    answer:List SupS := [0 for u in lpolys]
    for i in 0..rhsdeg repeat
      ppx:=map((z1:S):R +-> coefficient(z1,i),pp)
      zero? ppx => "next"
      recAns:= solveLinearPolynomialEquation(lpolysLower,ppx)
      recAns case "failed" => return "failed"
      answer:=[monomial(1,i)$S * raise c + d
                 for c in recAns for d in answer]
    answer
  solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS
-- local function definitions
hensel(pp,r,factors) ==
   -- factors is a relatively prime factorization of pp modulo the ideal
   -- (x-r), with suitably imposed leading coefficients.
   -- This is lifted, without re-combinations, to a factorization
   -- return "failed" if this can't be done
   origFactors:=factors
   totdegree:Integer:=0
   proddegree:Integer:=
               "max"/[degree(u) for u in coefficients pp]
   n:PI:=1
   pn:=prime:=monomial(1,1) - r::S
   foundFactors:List SupS:=empty()
   while (totdegree <= proddegree) repeat
       Ecart:=(pp-*/factors) exquo  pn
       Ecart case "failed" =>
             error "failed lifting in hensel in PFBRU"
       zero? Ecart =>
          -- then we have all the factors
          return [append(foundFactors, factors)]
       step:=solveLinearPolynomialEquation(origFactors,
                                           map(z1 +-> elt(z1,r::S),
                                              Ecart))
       step case "failed" => return "failed" -- must be a false split
       factors:=[a+b*pn for a in factors for b in step]
       for a in factors for c in origFactors repeat
           pp1:= pp exquo a
           pp1 case "failed" => "next"
           pp:=pp1
           proddegree := proddegree - "max"/[degree(u)
                                             for u in coefficients a]
           factors:=remove(a,factors)
           origFactors:=remove(c,origFactors)
           foundFactors:=[a,:foundFactors]
```

```
        #factors < 2 =>
           return [(empty? factors => foundFactors;
                                    [pp,:foundFactors])]
        totdegree:= +/["max"/[degree(u)
                              for u in coefficients u1]
                        for u1 in factors]
        n:=n+1
        pn:=pn*prime
    "failed" -- must have been a false split
  chooseFSQViableSubstitutions(pp) ==
    substns:R
    ppR: SupR
    while true repeat
       substns:= randomR()
       zero? elt(leadingCoefficient pp,substns ) => "next"
       ppR:=map(z1 +-> elt(z1,substns),pp)
       degree gcd(ppR,differentiate ppR)>0 => "next"
       leave
    [substns,ppR]
  raise(supR) == map(z1 +-> z1:R::S,supR)
  lower(pp) == map(z1 +-> retract(z1)::R,pp)
  factorSFBRlcUnitInner(pp,r) ==
     -- pp is square-free as a Sup, but the Up  variable occurs.
     -- Furthermore, its LC is a unit
     -- returns "failed" if the substitution is bad, else a factorization
     ppR:=map(z1 +-> elt(z1,r),pp)
     degree ppR < degree pp => "failed"
     degree gcd(ppR,differentiate ppR) >0 => "failed"
     factors:=
       fDown:=factorSquareFreePolynomial ppR
       [raise (unit fDown * factorList(fDown).first.fctr),
        :[raise u.fctr for u in factorList(fDown).rest]]
     #factors = 1 => makeFR(1,[["irred",pp,1]])
     hen:=hensel(pp,r,factors)
     hen case "failed" => "failed"
     makeFR(1,[["irred",u,1] for u in hen.fctrs])
  -- exported function definitions
  if R has StepThrough then
    factorSFBRlcUnit(pp) ==
      val:R := init()
      while true repeat
         tempAns:=factorSFBRlcUnitInner(pp,val)
         not (tempAns case "failed") => return tempAns
         val1:=nextItem val
         val1 case "failed" =>
           error "at this point, we know we have a finite field"
         val:=val1
  else
    factorSFBRlcUnit(pp) ==
      val:R := randomR()
```

```
       while true repeat
          tempAns:=factorSFBRlcUnitInner(pp,val)
          not (tempAns case "failed") => return tempAns
          val := randomR()
   if R has StepThrough then
      randomCount:R:= init()
      randomR() ==
        v:=nextItem(randomCount)
        v case "failed" =>
          SAY$Lisp "Taking another set of random values"
          randomCount:=init()
          randomCount
        randomCount:=v
        randomCount
   else if R has random: -> R then
      randomR() == random()
   else randomR() == (random()$Integer rem 100)::R
   factorByRecursion pp ==
     and/[zero? degree u for u in coefficients pp] =>
         map(raise,factorPolynomial lower pp)
     c:=content pp
     unit? c => refine(squareFree pp,factorSquareFreeByRecursion)
     pp:=(pp exquo c)::SupS
     mergeFactors(refine(squareFree pp,factorSquareFreeByRecursion),
                  map(z1 +-> z1:S::SupS,factor(c)$S))
   factorSquareFreeByRecursion pp ==
     and/[zero? degree u for u in coefficients pp] =>
        map(raise,factorSquareFreePolynomial lower pp)
     unit? (lcpp := leadingCoefficient pp) => factorSFBRlcUnit(pp)
     oldnfact:NonNegativeInteger:= 999999
                      -- I hope we never have to factor a polynomial
                      -- with more than this number of factors
     lcppPow:S
     while true repeat  -- a loop over possible false splits
       cVS:=chooseFSQViableSubstitutions(pp)
       newppR:=primitivePart cVS.ppRField
       factorsR:=factorSquareFreePolynomial(newppR)
       (nfact:=numberOfFactors factorsR) = 1 =>
                  return makeFR(1,[["irred",pp,1]])
       -- OK, force all leading coefficients to be equal to the leading
       -- coefficient of the input
       nfact > oldnfact => "next"   -- can't be a good reduction
       oldnfact:=nfact
       lcppR:=leadingCoefficient cVS.ppRField
       factors:=[raise((lcppR exquo leadingCoefficient u.fctr) ::R * u.fctr)
                  for u in factorList factorsR]
       -- factors now multiplies to give cVS.ppRField * lcppR^(#factors-1)
       -- Now change the leading coefficient to be lcpp
       factors:=[monomial(lcpp,degree u) + reductum u for u in factors]
--     factors:=[(lcpp exquo leadingCoefficient u.fctr)::S * raise u.fctr
```

```
--                for u in factorList factorsR]
       ppAdjust:=(lcppPow:=lcpp**#(rest factors)) * pp
       OK:=true
       hen:=hensel(ppAdjust,cVS.substnsField,factors)
       hen case "failed" => "next"
       factors:=hen.fctrs
       leave
     factors:=[ (lc:=content w;
                 lcppPow:=(lcppPow exquo lc)::S;
                  (w exquo lc)::SupS)
                for w in factors]
     not unit? lcppPow =>
         error "internal error in factorSquareFreeByRecursion"
     makeFR((recip lcppPow)::S::SupS,
            [["irred",w,1] for w in factors])
```

———-

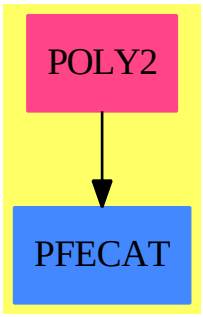
— **PFBRU.dotabb** —

```
"PFBRU" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFBRU"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFBRU" -> "PFECAT"
```

———-

17.51 package POLY2 PolynomialFunctions2

17.51.1 PolynomialFunctions2 (POLY2)



Exports:
map

— **package POLY2 PolynomialFunctions2** —

```
)abbrev package POLY2 PolynomialFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package takes a mapping between coefficient rings, and lifts
++ it to a mapping between polynomials over those rings.

PolynomialFunctions2(R:Ring, S:Ring): with
  map: (R -> S, Polynomial R) -> Polynomial S
    ++ map(f, p) produces a new polynomial as a result of applying
    ++ the function f to every coefficient of the polynomial p.
 == add
  map(f, p) == map(x1 +-> x1::Polynomial(S), x2 +-> f(x2)::Polynomial(S),
                  p)$PolynomialCategoryLifting(IndexedExponents Symbol,
                                  Symbol, R, Polynomial R, Polynomial S)
```

———-

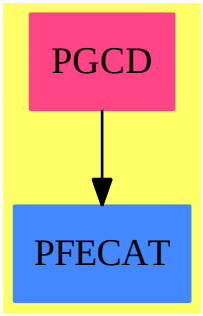
— **POLY2.dotabb** —

```
"POLY2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLY2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLY2" -> "PFECAT"
```

———-

17.52 package PGCD PolynomialGcdPackage

17.52.1 PolynomialGcdPackage (PGCD)



Exports:
gcd gcdPrimitive

— package PGCD PolynomialGcdPackage —

```
)abbrev package PGCD PolynomialGcdPackage
++ Author: Michael Lucks, P. Gianni
++ Date Created:
++ Date Last Updated: 17 June 1996
++ Fix History: Moved unitCanonicals for performance (BMT);
++              Fixed a problem with gcd(x,0) (Frederic Lehobey)
++ Basic Functions: gcd, content
++ Related Constructors: Polynomial
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package computes multivariate polynomial gcd's using
++ a hensel lifting strategy. The contraint on the coefficient
++ domain is imposed by the lifting strategy. It is assumed that
++ the coefficient domain has the property that almost all specializations
++ preserve the degree of the gcd.

PolynomialGcdPackage(E,OV,R,P):C == T where
    I        ==> Integer
    NNI      ==> NonNegativeInteger
    PI       ==> PositiveInteger
    R     :  EuclideanDomain
    P     :  PolynomialCategory(R,E,OV)
    OV    :  OrderedSet
    E     :  OrderedAbelianMonoidSup
```

```
SUPP      ==> SparseUnivariatePolynomial P

C == with
  gcd               :   (P,P)    -> P
    ++ gcd(p,q) computes the gcd of the two polynomials p and q.
  gcd               :   List P   -> P
    ++ gcd(lp) computes the gcd of the list of polynomials lp.
  gcd               :   (SUPP,SUPP)    -> SUPP
    ++ gcd(p,q) computes the gcd of the two polynomials p and q.
  gcd               :   List SUPP   -> SUPP
    ++ gcd(lp) computes the gcd of the list of polynomials lp.
  gcdPrimitive      :   (P,P)    -> P
    ++ gcdPrimitive(p,q) computes the gcd of the primitive polynomials
    ++ p and q.
  gcdPrimitive      :   (SUPP,SUPP)    -> SUPP
    ++ gcdPrimitive(p,q) computes the gcd of the primitive polynomials
    ++ p and q.
  gcdPrimitive      :   List P   -> P
    ++ gcdPrimitive lp computes the gcd of the list of primitive
    ++ polynomials lp.

T == add

  SUP      ==> SparseUnivariatePolynomial R

  LGcd     ==> Record(locgcd:SUPP,goodint:List List R)
  UTerm    ==> Record(lpol:List SUP,lint:List List R,mpol:SUPP)
  pmod:R   :=  (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

  import MultivariateLifting(E,OV,R,P)
  import FactoringUtilities(E,OV,R,P)

            --------  Local  Functions  --------

  myran           :    Integer   -> Union(R,"failed")
  better          :    (P,P)     -> Boolean
  failtest        :   (SUPP,SUPP,SUPP)    -> Boolean
  monomContent    :   (SUPP)   -> SUPP
  gcdMonom        :  (SUPP,SUPP)    -> SUPP
  gcdTermList     :    (P,P)     -> P
  good            :  (SUPP,List OV,List List R) -> Record(upol:SUP,inval:List List R)

  chooseVal       :  (SUPP,SUPP,List OV,List List R) -> Union(UTerm,"failed")
  localgcd        :  (SUPP,SUPP,List OV,List List R) -> LGcd
  notCoprime      :  (SUPP,SUPP, List NNI,List OV,List List R)   -> SUPP
  imposelc        : (List SUP,List OV,List R,List P) -> List SUP

  lift? :(SUPP,SUPP,UTerm,List NNI,List OV) -> Union(s:SUPP,failed:"failed",notCoprime:"notCoprime")
  lift  :(SUPP,SUP,SUP,P,List OV,List NNI,List R) -> Union(SUPP,"failed")
```

```
                ---- Local  functions ----
-- test if something wrong happened in the gcd
  failtest(f:SUPP,p1:SUPP,p2:SUPP) : Boolean ==
    (p1 exquo f) case "failed" or (p2 exquo f) case "failed"

-- Choose the integers
  chooseVal(p1:SUPP,p2:SUPP,lvr:List OV,ltry:List List R):Union(UTerm,"failed") ==
    d1:=degree(p1)
    d2:=degree(p2)
    dd:NNI:=0$NNI
    nvr:NNI:=#lvr
    lval:List R :=[]
    range:I:=8
    repeat
      range:=2*range
      lval:=[ran(range) for i in 1..nvr]
      member?(lval,ltry) => "new point"
      ltry:=cons(lval,ltry)
      uf1:SUP:=completeEval(p1,lvr,lval)
      degree uf1 ^= d1 => "new point"
      uf2:SUP:= completeEval(p2,lvr,lval)
      degree uf2 ^= d2 => "new point"
      u:=gcd(uf1,uf2)
      du:=degree u
     --the univariate gcd is 1
      if du=0 then return [[1$SUP],ltry,0$SUPP]$UTerm

      ugcd:List SUP:=[u,(uf1 exquo u)::SUP,(uf2 exquo u)::SUP]
      uterm:=[ugcd,ltry,0$SUPP]$UTerm
      dd=0 => dd:=du

    --the degree is not changed
      du=dd =>

       --test if one of the polynomials is the gcd
        dd=d1 =>
          if ^((f:=p2 exquo p1) case "failed") then
            return [[u],ltry,p1]$UTerm
          if dd^=d2 then dd:=(dd-1)::NNI

        dd=d2 =>
          if ^((f:=p1 exquo p2) case "failed") then
            return [[u],ltry,p2]$UTerm
          dd:=(dd-1)::NNI
        return uterm

      --the new gcd has degree less
       du<dd => dd:=du
```

```
    good(f:SUPP,lvr:List OV,ltry:List List R):Record(upol:SUP,inval:List List R) ==
      nvr:NNI:=#lvr
      range:I:=1
      while true repeat
        range:=2*range
        lval:=[ran(range) for i in 1..nvr]
        member?(lval,ltry) => "new point"
        ltry:=cons(lval,ltry)
        uf:=completeEval(f,lvr,lval)
        if degree gcd(uf,differentiate uf)=0 then return [uf,ltry]

    -- impose the right lc
    imposelc(lipol:List SUP,
             lvar:List OV,lval:List R,leadc:List P):List SUP ==
      result:List SUP :=[]
      for pol in lipol for leadpol in leadc repeat
         p1:= univariate eval(leadpol,lvar,lval) * pol
         result:= cons((p1 exquo leadingCoefficient pol)::SUP,result)
      reverse result

  --Compute the gcd between not coprime polynomials
    notCoprime(g:SUPP,p2:SUPP,ldeg:List NNI,lvar1:List OV,ltry:List List R) : SUPP ==
      g1:=gcd(g,differentiate g)
      l1 := (g exquo g1)::SUPP
      lg:LGcd:=localgcd(l1,p2,lvar1,ltry)
      (l,ltry):=(lg.locgcd,lg.goodint)
      lval:=ltry.first
      p2l:=(p2 exquo l)::SUPP
      (gd1,gd2):=(l,l)
      ul:=completeEval(l,lvar1,lval)
      dl:=degree ul
      if degree gcd(ul,differentiate ul) ^=0 then
        newchoice:=good(l,lvar1,ltry)
        ul:=newchoice.upol
        ltry:=newchoice.inval
        lval:=ltry.first
      ug1:=completeEval(g1,lvar1,lval)
      ulist:=[ug1,completeEval(p2l,lvar1,lval)]
      lcpol:List P:=[leadingCoefficient g1, leadingCoefficient p2]
      while true repeat
        d:SUP:=gcd(cons(ul,ulist))
        if degree d =0 then return gd1
        lquo:=(ul exquo d)::SUP
        if degree lquo ^=0 then
          lgcd:=gcd(cons(leadingCoefficient l,lcpol))
          (gdl:=lift(l,d,lquo,lgcd,lvar1,ldeg,lval)) case "failed" =>
             return notCoprime(g,p2,ldeg,lvar1,ltry)
          l:=gd2:=gdl::SUPP
          ul:=completeEval(l,lvar1,lval)
          dl:=degree ul
```

```
          gd1:=gd1*gd2
          ulist:=[(uf exquo d)::SUP for uf in ulist]

      gcdPrimitive(p1:SUPP,p2:SUPP) : SUPP ==
        if (d1:=degree(p1)) > (d2:=degree(p2)) then
            (p1,p2):= (p2,p1)
            (d1,d2):= (d2,d1)
        degree p1 = 0 =>
            p1 = 0 => unitCanonical p2
            unitCanonical p1
        lvar:List OV:=
          sort((a:OV,b:OV):Boolean+->a>b,setUnion(variables p1,variables p2))
        empty? lvar =>
           raisePolynomial(gcd(lowerPolynomial p1,lowerPolynomial p2))
        (p2 exquo p1) case SUPP => unitCanonical p1
        ltry:List List R:=empty()
        totResult:=localgcd(p1,p2,lvar,ltry)
        result: SUPP:=totResult.locgcd
        -- special cases
        result=1 => 1$SUPP
        while failtest(result,p1,p2) repeat
--        SAY$Lisp  "retrying gcd"
          ltry:=totResult.goodint
          totResult:=localgcd(p1,p2,lvar,ltry)
          result:=totResult.locgcd
        result

    --local function for the gcd : it returns the evaluation point too
      localgcd(p1:SUPP,p2:SUPP,lvar:List(OV),ltry:List List R) : LGcd ==
        uterm:=chooseVal(p1,p2,lvar,ltry)::UTerm
        ltry:=uterm.lint
        listpol:= uterm.lpol
        ud:=listpol.first
        dd:= degree ud

        --the univariate gcd is 1
        dd=0 => [1$SUPP,ltry]$LGcd

        --one of the polynomials is the gcd
        dd=degree(p1) or dd=degree(p2) =>
                        [uterm.mpol,ltry]$LGcd
        ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))

       -- if there is a polynomial g s.t. g/gcd and gcd are coprime ...
        -- I can lift
        (h:=lift?(p1,p2,uterm,ldeg,lvar)) case notCoprime =>
          [notCoprime(p1,p2,ldeg,lvar,ltry),ltry]$LGcd
        h case failed => localgcd(p1,p2,lvar,ltry) -- skip bad values?
        [h.s,ltry]$LGcd
```

```
-- content, internal functions return the poly if it is a monomial
    monomContent(p:SUPP):SUPP ==
      degree(p)=0 => 1
      md:= minimumDegree(p)
      monomial(gcd sort(better,coefficients p),md)

-- Ordering for gcd purposes
    better(p1:P,p2:P):Boolean ==
      ground? p1 => true
      ground? p2 => false
      degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

-- Gcd between polynomial p1 and p2 with
-- mainVariable p1 < x=mainVariable p2
    gcdTermList(p1:P,p2:P) : P ==
      termList:=sort(better,
         cons(p1,coefficients univariate(p2,(mainVariable p2)::OV)))
      q:P:=termList.first
      for term in termList.rest until q = 1$P repeat q:= gcd(q,term)
      q

-- Gcd between polynomials with the same mainVariable
    gcd(p1:SUPP,p2:SUPP): SUPP ==
      if degree(p1) > degree(p2) then (p1,p2):= (p2,p1)
      degree p1 = 0 =>
         p1 = 0 => unitCanonical p2
         p1 = 1 => unitCanonical p1
         gcd(leadingCoefficient p1, content p2)::SUPP
      reductum(p1)=0 => gcdMonom(p1,monomContent p2)
      c1:= monomContent(p1)
      reductum(p2)=0 => gcdMonom(c1,p2)
      c2:= monomContent(p2)
      p1:= (p1 exquo c1)::SUPP
      p2:= (p2 exquo c2)::SUPP
      gcdPrimitive(p1,p2) * gcdMonom(c1,c2)

 -- gcd between 2 monomials
    gcdMonom(m1:SUPP,m2:SUPP):SUPP ==
      monomial(gcd(leadingCoefficient(m1),leadingCoefficient(m2)),
               min(degree(m1),degree(m2)))


  --If there is a pol s.t. pol/gcd and gcd are coprime I can lift
    lift?(p1:SUPP,p2:SUPP,uterm:UTerm,ldeg:List NNI,
                  lvar:List OV) : Union(s:SUPP,failed:"failed",notCoprime:"notCoprime") ==
      leadpol:Boolean:=false
      (listpol,lval):=(uterm.lpol,uterm.lint.first)
      d:=listpol.first
      listpol:=listpol.rest
```

```
      nolift:Boolean:=true
      for uf in listpol repeat
            --note uf and d not necessarily primitive
        degree gcd(uf,d) =0 => nolift:=false
      nolift => ["notCoprime"]
      f:SUPP:=([p1,p2]$List(SUPP)).(position(uf,listpol))
      lgcd:=gcd(leadingCoefficient p1, leadingCoefficient p2)
      (l:=lift(f,d,uf,lgcd,lvar,ldeg,lval)) case "failed" => ["failed"]
      [l :: SUPP]

 -- interface with the general "lifting" function
    lift(f:SUPP,d:SUP,uf:SUP,lgcd:P,lvar:List OV,
                     ldeg:List NNI,lval:List R):Union(SUPP,"failed") ==
      leadpol:Boolean:=false
      lcf:P
      lcf:=leadingCoefficient f
      df:=degree f
      leadlist:List(P):=[]

      if lgcd^=1 then
        leadpol:=true
        f:=lgcd*f
        ldeg:=[n0+n1 for n0 in ldeg for n1 in degree(lgcd,lvar)]
        lcd:R:=leadingCoefficient d
        if degree(lgcd)=0 then d:=((retract lgcd) *d exquo lcd)::SUP
        else d:=(retract(eval(lgcd,lvar,lval)) * d exquo lcd)::SUP
        uf:=lcd*uf
      leadlist:=[lgcd,lcf]
      lg:=imposelc([d,uf],lvar,lval,leadlist)
      (pl:=lifting(f,lvar,lg,lval,leadlist,ldeg,pmod)) case "failed" =>
             "failed"
      plist := pl :: List SUPP
      (p0:SUPP,p1:SUPP):=(plist.first,plist.2)
      if completeEval(p0,lvar,lval) ^= lg.first then
         (p0,p1):=(p1,p0)
      ^leadpol => p0
      p0 exquo content(p0)

  -- Gcd for two multivariate polynomials
     gcd(p1:P,p2:P) : P ==
       ground? p1 =>
         p1 := unitCanonical p1
         p1 = 1$P => p1
         p1 = 0$P => unitCanonical p2
         ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
         gcdTermList(p1,p2)
       ground? p2 =>
         p2 := unitCanonical p2
         p2 = 1$P => p2
         p2 = 0$P => unitCanonical p1
```

```
        gcdTermList(p2,p1)
      (p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
      mv1:= mainVariable(p1)::OV
      mv2:= mainVariable(p2)::OV
      mv1 = mv2 => multivariate(gcd(univariate(p1,mv1),
                                    univariate(p2,mv1)),mv1)
      mv1 < mv2 => gcdTermList(p1,p2)
      gcdTermList(p2,p1)

 -- Gcd for a list of multivariate polynomials
    gcd(listp:List P) : P ==
      lf:=sort(better,listp)
      f:=lf.first
      for g in lf.rest repeat
        f:=gcd(f,g)
        if f=1$P then return f
      f

    gcd(listp:List SUPP) : SUPP ==
      lf:=sort((z1:SUPP,z2:SUPP):Boolean +-> degree(z1)<degree(z2),listp)
      f:=lf.first
      for g in lf.rest repeat
        f:=gcd(f,g)
        if f=1 then return f
      f


 -- Gcd for primitive polynomials
    gcdPrimitive(p1:P,p2:P):P ==
      (p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
      ground? p1 =>
        ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
        p1 = 0$P => p2
        1$P
      ground? p2 =>
        p2 = 0$P => p1
        1$P
      mv1:= mainVariable(p1)::OV
      mv2:= mainVariable(p2)::OV
      mv1 = mv2 =>
        md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv2))
        mp:=1$P
        if md>1 then
          mp:=(mv1::P)**md
          p1:=(p1 exquo mp)::P
          p2:=(p2 exquo mp)::P
        up1 := univariate(p1,mv1)
        up2 := univariate(p2,mv2)
        mp*multivariate(gcdPrimitive(up1,up2),mv1)
      1$P
```

```
  -- Gcd for a list of primitive multivariate polynomials
      gcdPrimitive(listp:List P) : P ==
        lf:=sort(better,listp)
        f:=lf.first
        for g in lf.rest repeat
          f:=gcdPrimitive(f,g)
          if f=1$P then return f
        f
```

———-

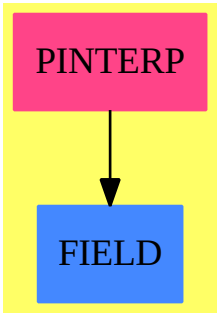
— **PGCD.dotabb** —

```
"PGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PGCD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PGCD" -> "PFECAT"
```

———-

17.53 package PINTERP PolynomialInterpolation

17.53.1 PolynomialInterpolation (PINTERP)



Exports:
interpolate

— **package PINTERP PolynomialInterpolation** —

```
)abbrev package PINTERP PolynomialInterpolation
++ Description:
++ This package exports interpolation algorithms
```

```
PolynomialInterpolation(xx, F): Cat == Body   where
    xx: Symbol
    F:  Field
    UP  ==> UnivariatePolynomial
    SUP ==> SparseUnivariatePolynomial

    Cat ==> with
        interpolate: (UP(xx,F), List F, List F) -> UP(xx,F)
          ++ interpolate(u,lf,lg) \undocumented
        interpolate: (List F, List F)           -> SUP F
          ++ interpolate(lf,lg) \undocumented

    Body ==> add
        PIA ==> PolynomialInterpolationAlgorithms

        interpolate(qx, lx, ly) ==
            px := LagrangeInterpolation(lx, ly)$PIA(F, UP(xx, F))
            elt(px, qx)

        interpolate(lx, ly) ==
            LagrangeInterpolation(lx, ly)$PIA(F, SUP F)
```

———-

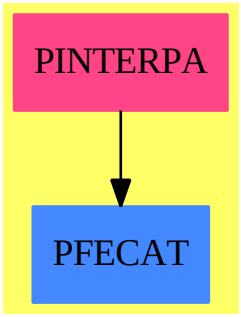
— **PINTERP.dotabb** —

```
"PINTERP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PINTERP"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"PINTERP" -> "FIELD"
```

———-

17.54 package PINTERPA PolynomialInterpolationAlgorithms

17.54.1 PolynomialInterpolationAlgorithms (PINTERPA)



Exports:
LagrangeInterpolation

— **package PINTERPA PolynomialInterpolationAlgorithms** —

```
)abbrev package PINTERPA PolynomialInterpolationAlgorithms
++ Description:
++ This package exports interpolation algorithms

PolynomialInterpolationAlgorithms(F, P): Cat == Body   where
    F: Field
    P: UnivariatePolynomialCategory(F)

    Cat ==> with
        LagrangeInterpolation: (List F, List F) -> P
          ++ LagrangeInterpolation(l1,l2) \undocumented

    Body ==> add
        LagrangeInterpolation(lx, ly) ==
            #lx ^= #ly =>
                error "Different number of points and values."
            ip: P := 0
            for xi in lx for yi in ly for i in 0.. repeat
                pp: P := 1
                xp: F := 1
                for xj in lx for j in 0.. | i ^= j repeat
                    pp := pp * (monomial(1,1) - monomial(xj,0))
                    xp := xp * (xi - xj)
                ip := ip + (yi/xp) * pp
            ip
```

———-

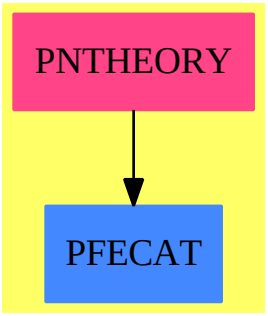
— **PINTERPA.dotabb** —

```
"PINTERPA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PINTERPA"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PINTERPA" -> "PFECAT"
```

———-

17.55 package PNTHEORY PolynomialNumberTheory-Functions

17.55.1 PolynomialNumberTheoryFunctions (PNTHEORY)



Exports:

bernoulli	chebyshevT	chebyshevU	cyclotomic	euler
fixedDivisor	hermite	laguerre	legendre	

— **package PNTHEORY PolynomialNumberTheoryFunctions** —

```
)abbrev package PNTHEORY PolynomialNumberTheoryFunctions
++ Author: Michael Monagan, Clifton J. Williamson
++ Date Created: June 1987
++ Date Last Updated: 10 November 1996 (Claude Quitte)
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: polynomial, number theory
++ Examples:
```

```
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various polynomial number theoretic functions
++ over the integers.

PolynomialNumberTheoryFunctions(): Exports == Implementation where
 I ==> Integer
 RN ==> Fraction I
 SUP ==> SparseUnivariatePolynomial
 NNI ==> NonNegativeInteger

 Exports ==> with
  bernoulli : I -> SUP RN
    ++ bernoulli(n) returns the nth Bernoulli polynomial \spad{B[n](x)}.
    ++ Bernoulli polynomials denoted \spad{B(n,x)} computed by solving the
    ++ differential equation  \spad{differentiate(B(n,x),x) = n B(n-1,x)} where
    ++ \spad{B(0,x) = 1} and initial condition comes from \spad{B(n) = B(n,0)}.
  chebyshevT: I -> SUP I
    ++ chebyshevT(n) returns the nth Chebyshev polynomial \spad{T[n](x)}.
    ++ Note that Chebyshev polynomials of the first kind,
    ++ denoted \spad{T[n](x)},
    ++ computed from the two term recurrence.  The generating function
    ++ \spad{(1-t*x)/(1-2*t*x+t**2) = sum(T[n](x)*t**n, n=0..infinity)}.
  chebyshevU: I -> SUP I
    ++ chebyshevU(n) returns the nth Chebyshev polynomial \spad{U[n](x)}.
    ++ Note that Chebyshev polynomials of the second kind,
    ++ denoted \spad{U[n](x)},
    ++ computed from the two term recurrence.  The generating function
    ++ \spad{1/(1-2*t*x+t**2) = sum(T[n](x)*t**n, n=0..infinity)}.
  cyclotomic: I -> SUP I
    ++ cyclotomic(n) returns the nth cyclotomic polynomial \spad{phi[n](x)}.
    ++ Note that \spad{phi[n](x)} is the factor of \spad{x**n - 1} whose roots
    ++ are the primitive nth roots of unity.
  euler     : I -> SUP RN
    ++ euler(n) returns the nth Euler polynomial \spad{E[n](x)}.
    ++ Note that Euler polynomials denoted \spad{E(n,x)} computed by solving
    ++ the differential equation
    ++ \spad{differentiate(E(n,x),x) = n E(n-1,x)} where
    ++ \spad{E(0,x) = 1} and initial condition comes
    ++ from \spad{E(n) = 2**n E(n,1/2)}.
  fixedDivisor: SUP I -> I
    ++ fixedDivisor(a) for \spad{a(x)} in \spad{Z[x]} is the largest integer
    ++ f such that f divides \spad{a(x=k)} for all integers k.
    ++ Note that fixed divisor of \spad{a} is
    ++ \spad{reduce(gcd,[a(x=k) for k in 0..degree(a)])}.
  hermite   : I -> SUP I
    ++ hermite(n) returns the nth Hermite polynomial \spad{H[n](x)}.
    ++ Note that Hermite polynomials, denoted \spad{H[n](x)}, are computed from
    ++ the two term recurrence.  The generating function is:
    ++ \spad{exp(2*t*x-t**2) = sum(H[n](x)*t**n/n!, n=0..infinity)}.
```

```
 laguerre  : I -> SUP I
   ++ laguerre(n) returns the nth Laguerre polynomial \spad{L[n](x)}.
   ++ Note that Laguerre polynomials, denoted \spad{L[n](x)}, are computed
   ++ from the two term recurrence.  The generating function is:
   ++ \spad{exp(x*t/(t-1))/(1-t) = sum(L[n](x)*t**n/n!, n=0..infinity)}.
 legendre  : I -> SUP RN
   ++ legendre(n) returns the nth Legendre polynomial \spad{P[n](x)}.
   ++ Note that Legendre polynomials, denoted \spad{P[n](x)}, are computed
   ++ from the two term recurrence.  The generating function is:
   ++ \spad{1/sqrt(1-2*t*x+t**2) = sum(P[n](x)*t**n, n=0..infinity)}.
Implementation ==> add
 import IntegerPrimesPackage(I)

 x := monomial(1,1)$SUP(I)
 y := monomial(1,1)$SUP(RN)

 -- For functions computed via a fixed term recurrence we record
 -- previous values so that the next value can be computed directly

 E : Record(En:I, Ev:SUP(RN)) := [0,1]
 B : Record( Bn:I, Bv:SUP(RN) ) := [0,1]
 H : Record( Hn:I, H1:SUP(I), H2:SUP(I) ) := [0,1,x]
 L : Record( Ln:I, L1:SUP(I), L2:SUP(I) ) := [0,1,x]
 P : Record( Pn:I, P1:SUP(RN), P2:SUP(RN) ) := [0,1,y]
 CT : Record( Tn:I, T1:SUP(I), T2:SUP(I) ) := [0,1,x]
 U : Record( Un:I, U1:SUP(I), U2:SUP(I) ) := [0,1,0]

 MonicQuotient: (SUP(I),SUP(I)) -> SUP(I)
 MonicQuotient (a,b) ==
   leadingCoefficient(b) ^= 1 => error "divisor must be monic"
   b = 1 => a
   da := degree a
   db := degree b            -- assertion: degree b > 0
   q:SUP(I) := 0
   while da >= db repeat
     t := monomial(leadingCoefficient a, (da-db)::NNI)
     a := a - b * t
     q := q + t
     da := degree a
   q

 cyclotomic n ==
   --++ cyclotomic polynomial denoted phi[n](x)
   p:I; q:I; r:I; s:I; m:NNI; c:SUP(I); t:SUP(I)
   n < 0 => error "cyclotomic not defined for negative integers"
   n = 0 => x
   k := n; s := p := 1
   c := x - 1
   while k > 1 repeat
     p := nextPrime p
```

```
    (q,r) := divide(k, p)
    if r = 0 then
      while r = 0 repeat (k := q; (q,r) := divide(k,p))
      t := multiplyExponents(c,p::NNI)
      c := MonicQuotient(t,c)
      s := s * p
  m := (n quo s) :: NNI
  multiplyExponents(c,m)

euler n ==
  p : SUP(RN); t : SUP(RN); c : RN; s : I
  n < 0 => error "euler not defined for negative integers"
  if n < E.En then (s,p) := (0$I,1$SUP(RN)) else (s,p) := E
  -- (s,p) := if n < E.En then (0,1) else E
  for i in s+1 .. n repeat
    t := (i::RN) * integrate p
    c := euler(i)$IntegerNumberTheoryFunctions / 2**(i::NNI) - t(1/2)
    p := t + c::SUP(RN)
  E.En := n
  E.Ev := p
  p

bernoulli n ==
  p : SUP RN; t : SUP RN; c : RN; s : I
  n < 0 => error "bernoulli not defined for negative integers"
  if n < B.Bn then (s,p) := (0$I,1$SUP(RN)) else (s,p) := B
  -- (s,p) := if n < B.Bn then (0,1) else B
  for i in s+1 .. n repeat
    t := (i::RN) * integrate p
    c := bernoulli(i)$IntegerNumberTheoryFunctions
    p := t + c::SUP(RN)
  B.Bn := n
  B.Bv := p
  p

fixedDivisor a ==
  g:I; d:NNI; SUP(I)
  d := degree a
  g := coefficient(a, minimumDegree a)
  for k in 1..d while g > 1 repeat g := gcd(g,a k)
  g

hermite n ==
  s : I; p : SUP(I); q : SUP(I)
  n < 0 => error "hermite not defined for negative integers"
  -- (s,p,q) := if n < H.Hn then (0,1,x) else H
  if n < H.Hn then (s := 0; p := 1; q := x) else (s,p,q) := H
  for k in s+1 .. n repeat (p,q) := (2*x*p-2*(k-1)*q,p)
  H.Hn := n
  H.H1 := p
```

```
  H.H2 := q
  p

legendre n ==
  s:I; t:I; p:SUP(RN); q:SUP(RN)
  n < 0 => error "legendre not defined for negative integers"
  -- (s,p,q) := if n < P.Pn then (0,1,y) else P
  if n < P.Pn then (s := 0; p := 1; q := y) else (s,p,q) := P
  for k in s+1 .. n repeat
    t := k-1
    (p,q) := ((k+t)$I/k*y*p - t/k*q,p)
  P.Pn := n
  P.P1 := p
  P.P2 := q
  p

laguerre n ==
  k:I; s:I; t:I; p:SUP(I); q:SUP(I)
  n < 0 => error "laguerre not defined for negative integers"
  -- (s,p,q) := if n < L.Ln then (0,1,x) else L
  if n < L.Ln then (s := 0; p := 1; q := x) else (s,p,q) := L
  for k in s+1 .. n repeat
    t := k-1
    (p,q) := ((((k+t)$I)::SUP(I)-x)*p-t**2*q,p)
  L.Ln := n
  L.L1 := p
  L.L2 := q
  p

chebyshevT n ==
  s : I; p : SUP(I); q : SUP(I)
  n < 0 => error "chebyshevT not defined for negative integers"
  -- (s,p,q) := if n < CT.Tn then (0,1,x) else CT
  if n < CT.Tn then (s := 0; p := 1; q := x) else (s,p,q) := CT
  for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
  CT.Tn := n
  CT.T1 := p
  CT.T2 := q
  p

chebyshevU n ==
  s : I; p : SUP(I); q : SUP(I)
  n < 0 => error "chebyshevU not defined for negative integers"
  if n < U.Un then (s := 0; p := 1; q := 0) else (s,p,q) := U
  for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
  U.Un := n
  U.U1 := p
  U.U2 := q
  p
```

———-

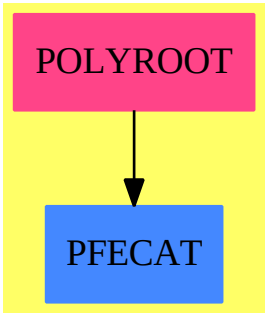
— PNTHEORY.dotabb —

```
"PNTHEORY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PNTHEORY"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PNTHEORY" -> "PFECAT"
```

———-

17.56 package POLYROOT PolynomialRoots

17.56.1 PolynomialRoots (POLYROOT)



Exports:
froot nthr qroot rroot

— package POLYROOT PolynomialRoots —

```
)abbrev package POLYROOT PolynomialRoots
++ Author: Manuel Bronstein
++ Date Created: 15 July 1988
++ Date Last Updated: 10 November 1993
++ Description:
++ Computes n-th roots of quotients of multivariate polynomials
-- not visible to the user

PolynomialRoots(E, V, R, P, F):Exports == Implementation where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: IntegralDomain
  P: PolynomialCategory(R, E, V)
  F: Field with
```

```
    numer : $ -> P
      ++ numer(x) \undocumented
    denom : $ -> P
      ++ denom(x) \undocumented
    coerce: P -> $
      ++ coerce(p) \undocumented

  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Z
  REC ==> Record(exponent:N, coef:F, radicand:F)

  Exports ==> with
    rroot: (R, N) -> REC
      ++ rroot(f, n) returns \spad{[m,c,r]} such
      ++ that \spad{f**(1/n) = c * r**(1/m)}.
    qroot : (Q, N) -> REC
      ++ qroot(f, n) returns \spad{[m,c,r]} such
      ++ that \spad{f**(1/n) = c * r**(1/m)}.
    if R has GcdDomain then froot: (F, N) -> REC
      ++ froot(f, n) returns \spad{[m,c,r]} such
      ++ that \spad{f**(1/n) = c * r**(1/m)}.
    nthr: (P, N) -> Record(exponent:N,coef:P,radicand:List P)
      ++ nthr(p,n) should be local but conditional

  Implementation ==> add
    import FactoredFunctions Z
    import FactoredFunctions P

    rsplit: List P -> Record(coef:R, poly:P)
    zroot : (Z, N) -> Record(exponent:N, coef:Z, radicand:Z)

    zroot(x, n) ==
--      zero? x or one? x => [1, x, 1]
      zero? x or (x = 1) => [1, x, 1]
      s := nthRoot(squareFree x, n)
      [s.exponent, s.coef, */s.radicand]

    if R has imaginary: () -> R then
      czroot: (Z, N) -> REC

      czroot(x, n) ==
        rec := zroot(x, n)
        rec.exponent = 2 and rec.radicand < 0 =>
          [rec.exponent, rec.coef * imaginary()::P::F, (-rec.radicand)::F]
        [rec.exponent, rec.coef::F, rec.radicand::F]

      qroot(x, n) ==
        sn := czroot(numer x, n)
        sd := czroot(denom x, n)
```

```
        m  := lcm(sn.exponent, sd.exponent)::N
        [m, sn.coef / sd.coef,
                    (sn.radicand ** (m quo sn.exponent)) /
                                (sd.radicand ** (m quo sd.exponent))]
    else
      qroot(x, n) ==
        sn := zroot(numer x, n)
        sd := zroot(denom x, n)
        m  := lcm(sn.exponent, sd.exponent)::N
        [m, sn.coef::F / sd.coef::F,
                    (sn.radicand ** (m quo sn.exponent))::F /
                                (sd.radicand ** (m quo sd.exponent))::F]

    if R has RetractableTo Fraction Z then
      rroot(x, n) ==
        (r := retractIfCan(x)@Union(Fraction Z,"failed")) case "failed"
          => [n, 1, x::P::F]
        qroot(r::Q, n)

    else
      if R has RetractableTo Z then
        rroot(x, n) ==
          (r := retractIfCan(x)@Union(Z,"failed")) case "failed"
            => [n, 1, x::P::F]
          qroot(r::Z::Q, n)
      else
        rroot(x, n) == [n, 1, x::P::F]

    rsplit l ==
      r := 1$R
      p := 1$P
      for q in l repeat
        if (u := retractIfCan(q)@Union(R, "failed")) case "failed"
          then p := p * q
          else r := r * u::R
      [r, p]

    if R has GcdDomain then
      if R has RetractableTo Z then
        nthr(x, n) ==
          (r := retractIfCan(x)@Union(Z,"failed")) case "failed"
             => nthRoot(squareFree x, n)
          rec := zroot(r::Z, n)
          [rec.exponent, rec.coef::P, [rec.radicand::P]]
      else nthr(x, n) == nthRoot(squareFree x, n)

      froot(x, n) ==
--        zero? x or one? x => [1, x, 1]
        zero? x or (x = 1) => [1, x, 1]
        sn := nthr(numer x, n)
```

```
        sd := nthr(denom x, n)
        pn := rsplit(sn.radicand)
        pd := rsplit(sd.radicand)
        rn := rroot(pn.coef, sn.exponent)
        rd := rroot(pd.coef, sd.exponent)
        m := lcm([rn.exponent, rd.exponent, sn.exponent, sd.exponent])::N
        [m, (sn.coef::F / sd.coef::F) * (rn.coef / rd.coef),
             ((rn.radicand ** (m quo rn.exponent)) /
                    (rd.radicand ** (m quo rd.exponent))) *
                           (pn.poly ** (m quo sn.exponent))::F /
                                    (pd.poly ** (m quo sd.exponent))::F]
```

———-

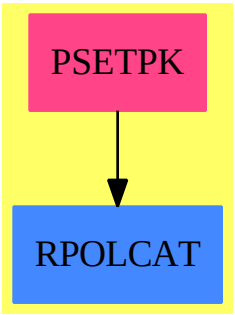
— **POLYROOT.dotabb** —

```
"POLYROOT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYROOT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYROOT" -> "PFECAT"
```

———-

17.57 package PSETPK PolynomialSetUtilitiesPackage

17.57.1 PolynomialSetUtilitiesPackage (PSETPK)



Exports:

bivariate?	crushedSet
interReduce	linear?
univariate?	bivariatePolynomials
certainlySubVariety?	irreducibleFactors
lazyIrreducibleFactors	linearPolynomials
possiblyNewVariety?	probablyZeroDim?
quasiMonicPolynomials	removeIrreducibleRedundantFactors
removeRedundantFactors	removeRedundantFactorsInContents
removeRedundantFactorsInPols	removeRoughlyRedundantFactorsInContents
removeRoughlyRedundantFactorsInPol	removeRoughlyRedundantFactorsInPols
removeSquaresIfCan	rewriteIdealWithQuasiMonicGenerators
rewriteSetByReducingWithParticularGenerators	roughBasicSet
selectAndPolynomials	selectOrPolynomials
selectPolynomials	squareFreeFactors
univariatePolynomials	univariatePolynomialsGcds
unprotectedRemoveRedundantFactors	

— package PSETPK PolynomialSetUtilitiesPackage —

```
)abbrev package PSETPK PolynomialSetUtilitiesPackage
++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 12/01/1995
++ Date Last Updated: 12/15/1998
++ SPARC Version
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides modest routines for polynomial system solving.
++ The aim of many of the operations of this package is to remove certain
++ factors in some polynomials in order to avoid unnecessary computations
++ in algorithms involving splitting techniques by partial factorization.

PolynomialSetUtilitiesPackage (R,E,V,P) : Exports == Implementation where

  R : IntegralDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LP ==> List P
  FP ==> Factored P
```

```
T ==> GeneralTriangularSet(R,E,V,P)
RRZ ==> Record(factor: P,exponent: Integer)
RBT ==> Record(bas:T,top:LP)
RUL ==> Record(chs:Union(T,"failed"),rfs:LP)
GPS ==> GeneralPolynomialSet(R,E,V,P)
pf ==> MultivariateFactorize(V, E, R, P)

Exports ==  with

   removeRedundantFactors: LP -> LP
      ++ \axiom{removeRedundantFactors(lp)} returns \axiom{lq} such that if
      ++ \axiom{lp = [p1,...,pn]} and \axiom{lq = [q1,...,qm]}
      ++ then the product \axiom{p1*p2*...*pn} vanishes iff the product \axiom{q1*q2*...*qm} vanishes,
      ++ and the product of degrees of the \axiom{qi} is not greater than
      ++ the one of the \axiom{pj}, and no polynomial in \axiom{lq}
      ++ divides another polynomial in \axiom{lq}. In particular,
      ++ polynomials lying in the base ring \axiom{R} are removed.
      ++ Moreover, \axiom{lq} is sorted w.r.t \axiom{infRittWu?}.
      ++ Furthermore, if R is gcd-domain, the polynomials in \axiom{lq} are
      ++ pairwise without common non trivial factor.
   removeRedundantFactors: (P,P) -> LP
      ++ \axiom{removeRedundantFactors(p,q)} returns the same as
      ++ \axiom{removeRedundantFactors([p,q])}
   removeSquaresIfCan : LP -> LP
      ++ \axiom{removeSquaresIfCan(lp)} returns
      ++ \axiom{removeDuplicates [squareFreePart(p)$P for p in lp]}
      ++ if \axiom{R} is gcd-domain else returns \axiom{lp}.
   unprotectedRemoveRedundantFactors: (P,P) -> LP
      ++ \axiom{unprotectedRemoveRedundantFactors(p,q)} returns the same as
      ++ \axiom{removeRedundantFactors(p,q)} but does assume that neither
      ++ \axiom{p} nor \axiom{q} lie in the base ring \axiom{R} and assumes that
      ++ \axiom{infRittWu?(p,q)} holds. Moreover, if \axiom{R} is gcd-domain,
      ++ then \axiom{p} and \axiom{q} are assumed to be square free.
   removeRedundantFactors: (LP,P) -> LP
      ++ \axiom{removeRedundantFactors(lp,q)} returns the same as
      ++ \axiom{removeRedundantFactors(cons(q,lp))} assuming
      ++ that \axiom{removeRedundantFactors(lp)} returns \axiom{lp}
      ++ up to replacing some polynomial \axiom{pj} in \axiom{lp}
      ++ by some some polynomial \axiom{qj} associated to \axiom{pj}.
   removeRedundantFactors : (LP,LP) -> LP
      ++ \axiom{removeRedundantFactors(lp,lq)} returns the same as
      ++ \axiom{removeRedundantFactors(concat(lp,lq))} assuming
      ++ that \axiom{removeRedundantFactors(lp)} returns \axiom{lp}
      ++ up to replacing some polynomial \axiom{pj} in \axiom{lp}
      ++ by some polynomial \axiom{qj} associated to \axiom{pj}.
   removeRedundantFactors : (LP,LP,(LP -> LP)) -> LP
      ++ \axiom{removeRedundantFactors(lp,lq,remOp)} returns the same as
      ++ \axiom{concat(remOp(removeRoughlyRedundantFactorsInPols(lp,lq)),lq)}
      ++ assuming that \axiom{remOp(lq)} returns \axiom{lq} up to similarity.
   certainlySubVariety? : (LP,LP) -> B
```

```
   ++ \axiom{certainlySubVariety?(newlp,lp)} returns true iff for every \axiom{p}
   ++ in \axiom{lp} the remainder of \axiom{p} by \axiom{newlp} using the division algorith
   ++ of Groebner techniques is zero.
possiblyNewVariety? : (LP, List LP) -> B
   ++ \axiom{possiblyNewVariety?(newlp,llp)} returns true iff for every \axiom{lp}
   ++ in \axiom{llp} certainlySubVariety?(newlp,lp) does not hold.
probablyZeroDim?: LP -> B
   ++ \axiom{probablyZeroDim?(lp)} returns true iff the number of polynomials
   ++ in \axiom{lp} is not smaller than the number of variables occurring
   ++ in these polynomials.
selectPolynomials : ((P -> B),LP) -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{selectPolynomials(pred?,ps)} returns \axiom{gps,bps} where
   ++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
   ++ such that \axiom{pred?(p)} holds and \axiom{bps} are the other ones.
selectOrPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{selectOrPolynomials(lpred?,ps)} returns \axiom{gps,bps} where
   ++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
   ++ such that \axiom{pred?(p)} holds for some \axiom{pred?} in \axiom{lpred?}
   ++ and \axiom{bps} are the other ones.
selectAndPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{selectAndPolynomials(lpred?,ps)} returns \axiom{gps,bps} where
   ++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
   ++ such that \axiom{pred?(p)} holds for every \axiom{pred?} in \axiom{lpred?}
   ++ and \axiom{bps} are the other ones.
quasiMonicPolynomials : LP -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{quasiMonicPolynomials(lp)} returns \axiom{qmps,nqmps} where
   ++ \axiom{qmps} is a list of the quasi-monic polynomials in \axiom{lp}
   ++ and \axiom{nqmps} are the other ones.
univariate? : P -> B
   ++ \axiom{univariate?(p)} returns true iff \axiom{p} involves one and
   ++ only one variable.
univariatePolynomials : LP -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{univariatePolynomials(lp)} returns \axiom{ups,nups} where
   ++ \axiom{ups} is a list of the univariate polynomials,
   ++ and \axiom{nups} are the other ones.
linear? : P -> B
   ++ \axiom{linear?(p)} returns true iff \axiom{p} does not lie
   ++ in the base ring \axiom{R} and has main degree \axiom{1}.
linearPolynomials : LP -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{linearPolynomials(lp)} returns \axiom{lps,nlps} where
   ++ \axiom{lps} is a list of the linear polynomials in lp,
   ++ and  \axiom{nlps} are the other ones.
bivariate? : P -> B
   ++ \axiom{bivariate?(p)} returns true iff \axiom{p} involves two and
   ++ only two variables.
bivariatePolynomials : LP -> Record(goodPols:LP,badPols:LP)
   ++ \axiom{bivariatePolynomials(lp)} returns \axiom{bps,nbps} where
   ++ \axiom{bps} is a list of the bivariate polynomials,
   ++ and \axiom{nbps} are the other ones.
removeRoughlyRedundantFactorsInPols : (LP, LP) -> LP
```

```
      ++ \axiom{removeRoughlyRedundantFactorsInPols(lp,lf)} returns
      ++ \axiom{newlp}where \axiom{newlp} is obtained from \axiom{lp}
      ++ by removing in every polynomial \axiom{p} of \axiom{lp}
      ++ any occurence of a polynomial \axiom{f} in \axiom{lf}.
      ++ This may involve a lot of exact-quotients computations.
    removeRoughlyRedundantFactorsInPols : (LP, LP,B) -> LP
      ++ \axiom{removeRoughlyRedundantFactorsInPols(lp,lf,opt)} returns
      ++ the same as \axiom{removeRoughlyRedundantFactorsInPols(lp,lf)}
      ++ if \axiom{opt} is \axiom{false} and if the previous operation
      ++ does not return any non null and constant polynomial,
      ++ else return \axiom{[1]}.
    removeRoughlyRedundantFactorsInPol : (P,LP) -> P
      ++ \axiom{removeRoughlyRedundantFactorsInPol(p,lf)} returns the same as
      ++ removeRoughlyRedundantFactorsInPols([p],lf,true)
    interReduce: LP -> LP
      ++ \axiom{interReduce(lp)} returns \axiom{lq} such that \axiom{lp}
      ++ and \axiom{lq} generate the same ideal and no polynomial
      ++ in \axiom{lq} is reducuble by the others in the sense
      ++ of Groebner bases. Since no assumptions are required
      ++ the result may depend on the ordering the reductions are
      ++ performed.
    roughBasicSet: LP -> Union(Record(bas:T,top:LP),"failed")
      ++ \axiom{roughBasicSet(lp)} returns the smallest (with Ritt-Wu
      ++ ordering) triangular set contained in \axiom{lp}.
    crushedSet: LP -> LP
      ++ \axiom{crushedSet(lp)} returns \axiom{lq} such that \axiom{lp} and
      ++ and \axiom{lq} generate the same ideal and no rough basic
      ++ sets reduce (in the sense of Groebner bases) the other
      ++ polynomials in \axiom{lq}.
    rewriteSetByReducingWithParticularGenerators : (LP,(P->B),((P,P)->B),((P,P)->P)) -> LP
      ++ \axiom{rewriteSetByReducingWithParticularGenerators(lp,pred?,redOp?,redOp)}
      ++ returns \axiom{lq} where \axiom{lq} is computed by the following
      ++ algorithm. Chose a basic set w.r.t. the reduction-test \axiom{redOp?}
      ++ among the polynomials satisfying property \axiom{pred?},
      ++ if it is empty then leave, else reduce the other polynomials by
      ++ this basic set w.r.t. the reduction-operation \axiom{redOp}.
      ++ Repeat while another basic set with smaller rank can be computed.
      ++ See code. If \axiom{pred?} is \axiom{quasiMonic?} the ideal is unchanged.
    rewriteIdealWithQuasiMonicGenerators : (LP,((P,P)->B),((P,P)->P)) -> LP
      ++ \axiom{rewriteIdealWithQuasiMonicGenerators(lp,redOp?,redOp)} returns
      ++ \axiom{lq} where \axiom{lq} and \axiom{lp} generate
      ++ the same ideal in \axiom{R^(-1) P} and \axiom{lq}
      ++ has rank not higher than the one of \axiom{lp}.
      ++ Moreover, \axiom{lq} is computed by reducing \axiom{lp}
      ++ w.r.t. some basic set of the ideal generated by
      ++ the quasi-monic polynomials in \axiom{lp}.
    if R has GcdDomain
    then
      squareFreeFactors : P -> LP
        ++ \axiom{squareFreeFactors(p)} returns the square-free factors of \axiom{p}
```

```
      ++ over \axiom{R}
    univariatePolynomialsGcds : LP -> LP
      ++ \axiom{univariatePolynomialsGcds(lp)} returns \axiom{lg} where
      ++ \axiom{lg} is a list of the gcds of every pair in \axiom{lp}
      ++ of univariate polynomials in the same main variable.
    univariatePolynomialsGcds : (LP,B) -> LP
      ++ \axiom{univariatePolynomialsGcds(lp,opt)} returns the same as
      ++ \axiom{univariatePolynomialsGcds(lp)} if \axiom{opt} is
      ++ \axiom{false} and if the previous operation does not return
      ++ any non null and constant polynomial, else return \axiom{[1]}.
    removeRoughlyRedundantFactorsInContents : (LP, LP) -> LP
      ++ \axiom{removeRoughlyRedundantFactorsInContents(lp,lf)} returns
      ++ \axiom{newlp}where \axiom{newlp} is obtained from \axiom{lp}
      ++ by removing in the content of every polynomial of \axiom{lp}
      ++ any occurence of a polynomial \axiom{f} in \axiom{lf}. Moreover,
      ++ squares over \axiom{R} are first removed in the content
      ++ of every polynomial of \axiom{lp}.
    removeRedundantFactorsInContents : (LP, LP) -> LP
      ++ \axiom{removeRedundantFactorsInContents(lp,lf)} returns \axiom{newlp}
      ++ where \axiom{newlp} is obtained from \axiom{lp} by removing
      ++ in the content of every polynomial of \axiom{lp} any non trivial
      ++ factor of any polynomial \axiom{f} in \axiom{lf}. Moreover,
      ++ squares over \axiom{R} are first removed in the content
      ++ of every polynomial of \axiom{lp}.
    removeRedundantFactorsInPols : (LP, LP) -> LP
      ++ \axiom{removeRedundantFactorsInPols(lp,lf)} returns \axiom{newlp}
      ++ where \axiom{newlp} is obtained from \axiom{lp} by removing
      ++ in every polynomial \axiom{p} of \axiom{lp} any non trivial
      ++ factor of any polynomial \axiom{f} in \axiom{lf}. Moreover,
      ++ squares over \axiom{R} are first removed in every
      ++ polynomial \axiom{lp}.
  if (R has EuclideanDomain) and (R has CharacteristicZero)
  then
    irreducibleFactors : LP -> LP
      ++ \axiom{irreducibleFactors(lp)} returns \axiom{lf} such that if
      ++ \axiom{lp = [p1,...,pn]} and \axiom{lf = [f1,...,fm]} then
      ++ \axiom{p1*p2*...*pn=0} means \axiom{f1*f2*...*fm=0}, and the \axiom{fi}
      ++ are irreducible over \axiom{R} and are pairwise distinct.
    lazyIrreducibleFactors : LP -> LP
      ++ \axiom{lazyIrreducibleFactors(lp)} returns \axiom{lf} such that if
      ++ \axiom{lp = [p1,...,pn]} and \axiom{lf = [f1,...,fm]} then
      ++ \axiom{p1*p2*...*pn=0} means \axiom{f1*f2*...*fm=0}, and the \axiom{fi}
      ++ are irreducible over \axiom{R} and are pairwise distinct.
      ++ The algorithm tries to avoid factorization into irreducible
      ++ factors as far as possible and makes previously use of gcd
      ++ techniques over \axiom{R}.
    removeIrreducibleRedundantFactors : (LP, LP) -> LP
      ++ \axiom{removeIrreducibleRedundantFactors(lp,lq)} returns the same
      ++ as \axiom{irreducibleFactors(concat(lp,lq))} assuming
      ++ that \axiom{irreducibleFactors(lp)} returns \axiom{lp}
```

```
        ++ up to replacing some polynomial \axiom{pj} in \axiom{lp}
        ++ by some polynomial \axiom{qj} associated to \axiom{pj}.

Implementation ==  add

   autoRemainder: T -> List(P)

   removeAssociates (lp:LP):LP ==
     removeDuplicates [primPartElseUnitCanonical(p) for p in lp]

   selectPolynomials  (pred?,ps) ==
     gps : LP := []
     bps : LP := []
     while not empty? ps repeat
       p := first ps
       ps := rest ps
       if pred?(p)
         then
           gps := cons(p,gps)
         else
           bps := cons(p,bps)
     gps := sort(infRittWu?,gps)
     bps := sort(infRittWu?,bps)
     [gps,bps]

   selectOrPolynomials (lpred?,ps) ==
     gps : LP := []
     bps : LP := []
     while not empty? ps repeat
       p := first ps
       ps := rest ps
       clpred? :=  lpred?
       while (not empty? clpred?) and (not (first clpred?)(p)) repeat
         clpred? :=  rest clpred?
       if not empty?(clpred?)
         then
           gps := cons(p,gps)
         else
           bps := cons(p,bps)
     gps := sort(infRittWu?,gps)
     bps := sort(infRittWu?,bps)
     [gps,bps]

   selectAndPolynomials (lpred?,ps) ==
     gps : LP := []
     bps : LP := []
     while not empty? ps repeat
       p := first ps
       ps := rest ps
       clpred? :=  lpred?
```

```
        while (not empty? clpred?) and ((first clpred?)(p)) repeat
          clpred? :=  rest clpred?
        if empty?(clpred?)
          then
            gps := cons(p,gps)
          else
            bps := cons(p,bps)
      gps := sort(infRittWu?,gps)
      bps := sort(infRittWu?,bps)
      [gps,bps]

    linear? p ==
      ground? p => false
--      one?(mdeg(p))
      (mdeg(p) = 1)

    linearPolynomials  ps ==
      selectPolynomials(linear?,ps)

    univariate? p ==
      ground? p => false
      not(ground?(init(p))) => false
      tp := tail(p)
      ground?(tp) => true
      not (mvar(p) = mvar(tp)) => false
      univariate?(tp)

    univariatePolynomials ps ==
      selectPolynomials(univariate?,ps)

    bivariate? p ==
      ground? p => false
      ground? tail(p) => univariate?(init(p))
      vp := mvar(p)
      vtp := mvar(tail(p))
      ((ground? init(p)) and (vp = vtp)) => bivariate? tail(p)
      ((ground? init(p)) and (vp > vtp)) => univariate? tail(p)
      not univariate?(init(p)) => false
      vip := mvar(init(p))
      vip > vtp => false
      vip = vtp => univariate? tail(p)
      vtp < vp => false
      zero? degree(tail(p),vip) => univariate? tail(p)
      bivariate? tail(p)

    bivariatePolynomials ps ==
      selectPolynomials(bivariate?,ps)

    quasiMonicPolynomials ps ==
      selectPolynomials(quasiMonic?,ps)
```

```
removeRoughlyRedundantFactorsInPols (lp,lf,opt) ==
  empty? lp => lp
  newlp : LP := []
  stop : B := false
  lp := remove(zero?,lp)
  lf := sort(infRittWu?,lf)
  test : Union(P,"failed")
  while (not empty? lp) and (not stop) repeat
    p := first lp
    lp := rest lp
    copylf := lf
    while (not empty? copylf) and (not ground? p) and (not (mvar(p) < mvar(first copylf))) repeat
      f := first copylf
      copylf := rest copylf
      while (((test := p exquo$P f)) case P) repeat
        p := test::P
    stop := opt and ground?(p)
    newlp := cons(unitCanonical(p),newlp)
  stop => [1$P]
  newlp

removeRoughlyRedundantFactorsInPol(p,lf) ==
  zero? p => p
  lp : LP := [p]
  first removeRoughlyRedundantFactorsInPols (lp,lf,true()$B)

removeRoughlyRedundantFactorsInPols (lp,lf) ==
  removeRoughlyRedundantFactorsInPols (lp,lf,false()$B)

possiblyNewVariety?(newlp,llp) ==
  while (not empty? llp) and _
   (not certainlySubVariety?(newlp,first(llp))) repeat
    llp := rest llp
  empty? llp

certainlySubVariety?(lp,lq) ==
  gs := construct(lp)$GPS
  while (not empty? lq) and _
   (zero? (remainder(first(lq),gs)$GPS).polnum) repeat
    lq := rest lq
  empty? lq

probablyZeroDim?(lp: List P) : Boolean ==
  m := #lp
  lv : List V := variables(first lp)
  while not empty? (lp := rest lp) repeat
    lv := concat(variables(first lp),lv)
  n := #(removeDuplicates lv)
  not (n > m)
```

```
interReduce(lp: LP): LP ==
  ps := lp
  rs: List(P) := []
  repeat
    empty? ps => return rs
    ps := sort(supRittWu?, ps)
    p := first ps
    ps := rest ps
    r := remainder(p,[ps]$GPS).polnum
    zero? r => "leave"
    ground? r => return []
    associates?(r,p) => rs := cons(r,rs)
    ps := concat(ps,cons(r,rs))
    rs := []

roughRed?(p:P,q:P):B ==
  ground? p => false
  ground? q => true
  mvar(p) > mvar(q)

roughBasicSet(lp) == basicSet(lp,roughRed?)$T

autoRemainder(ts:T): List(P) ==
  empty? ts => members(ts)
  lp := sort(infRittWu?, reverse members(ts))
  newlp : List(P) := [primPartElseUnitCanonical first(lp)]
  lp := rest(lp)
  while not empty? lp repeat
    p := (remainder(first(lp),construct(newlp)$GPS)$GPS).polnum
    if not zero? p
      then
        if ground? p
          then
            newlp := [1$P]
            lp := []
          else
            newlp := cons(p,newlp)
            lp := rest(lp)
      else
        lp := rest(lp)
  newlp

crushedSet(lp) ==
  rec := roughBasicSet(lp)
  contradiction := (rec case "failed")@B
  finished : B := false
  while (not finished) and (not contradiction) repeat
    bs := (rec::RBT).bas
    rs := (rec::RBT).top
```

```
        rs :=  rewriteIdealWithRemainder(rs,bs)$T
--        contradiction := ((not empty? rs) and (one? first(rs)))
        contradiction := ((not empty? rs) and (first(rs) = 1))
        if not contradiction
          then
            rs := concat(rs,autoRemainder(bs))
            rec := roughBasicSet(rs)
            contradiction := (rec case "failed")@B
            not contradiction => finished := not infRittWu?((rec::RBT).bas,bs)
      contradiction => [1$P]
      rs

    rewriteSetByReducingWithParticularGenerators (ps,pred?,redOp?,redOp) ==
      rs : LP := remove(zero?,ps)
      any?(ground?,rs) => [1$P]
      contradiction : B := false
      bs1 : T := empty()$T
      rec : Union(RBT,"failed")
      ar : Union(T,List(P))
      stop : B := false
      while (not contradiction) and (not stop) repeat
        rec := basicSet(rs,pred?,redOp?)$T
        bs2 : T := (rec::RBT).bas
        rs := (rec::RBT).top
        -- ar := autoReduce(bs2,lazyPrem,reduced?)@Union(T,List(P))
        ar := bs2::Union(T,List(P))
        if (ar case T)@B
          then
            bs2 := ar::T
            if infRittWu?(bs2,bs1)
              then
                rs := rewriteSetWithReduction(rs,bs2,redOp,redOp?)$T
                bs1 := bs2
              else
                stop := true
            rs := concat(members(bs2),rs)
          else
            rs := concat(ar::LP,rs)
      if any?(ground?,rs)
        then
          contradiction := true
          rs := [1$P]
      rs

    removeRedundantFactors (lp:LP,lq :LP, remOp : (LP -> LP)) ==
      -- ASSUME remOp(lp) returns lp up to similarity
      lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
      lq := remOp lq
      sort(infRittWu?,concat(lp,lq))
```

```
removeRedundantFactors (lp:LP,lq :LP) ==
  lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
  lq := removeRedundantFactors lq
  sort(infRittWu?,concat(lp,lq))

if (R has EuclideanDomain) and (R has CharacteristicZero)
then
  irreducibleFactors lp ==
    newlp : LP := []
    lrrz : List RRZ
    rrz : RRZ
    fp : FP
    while not empty? lp repeat
      p := first lp
      lp := rest lp
      fp := factor(p)$pf
      lrrz := factors(fp)$FP
      lf := remove(ground?,[rrz.factor for rrz in lrrz])
      newlp := concat(lf,newlp)
    removeDuplicates newlp

  lazyIrreducibleFactors lp ==
    lp := removeRedundantFactors(lp)
    newlp : LP := []
    lrrz : List RRZ
    rrz : RRZ
    fp : FP
    while not empty? lp repeat
      p := first lp
      lp := rest lp
      fp := factor(p)$pf
      lrrz := factors(fp)$FP
      lf := remove(ground?,[rrz.factor for rrz in lrrz])
      newlp := concat(lf,newlp)
    newlp

  removeIrreducibleRedundantFactors (lp:LP,lq :LP) ==
    -- ASSUME lp only contains irreducible factors over R
    lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
    lq := irreducibleFactors lq
    sort(infRittWu?,concat(lp,lq))

if R has GcdDomain
then

  squareFreeFactors(p:P) ==
    sfp: Factored P := squareFree(p)$P
    lsf: List P := [foo.factor for foo in factors(sfp)]
    lsf
```

```
univariatePolynomialsGcds (ps,opt) ==
  lg : LP := []
  pInV : LP
  stop : B := false
  ps := sort(infRittWu?,ps)
  p,g : P
  v : V
  while (not empty? ps) and (not stop) repeat
    while (not empty? ps) and (not univariate?((p := first(ps)))) repeat
      ps := rest ps
    if not empty? ps
      then
        v := mvar(p)$P
        pInV := [p]
        while (not empty? ps) and (mvar((p := first(ps))) = v) repeat
          if (univariate?(p))
            then
              pInV := cons(p,pInV)
          ps := rest ps
        g := gcd(pInV)$P
        stop := opt and (ground? g)
        lg := cons(g,lg)
  stop => [1$P]
  lg

univariatePolynomialsGcds ps ==
  univariatePolynomialsGcds (ps,false)

removeSquaresIfCan lp ==
  empty? lp => lp
  removeDuplicates [squareFreePart(p)$P for p in lp]

rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp) ==
  ups := removeSquaresIfCan(univariatePolynomialsGcds(ps,true))
  ps := removeDuplicates concat(ups,ps)
  rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp)

removeRoughlyRedundantFactorsInContents (ps,lf) ==
  empty? ps => ps
  newps : LP := []
  p,newp,cp,newcp,f,g : P
  test : Union(P,"failed")
  copylf : LP
  while not empty? ps repeat
    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newcp
```

```
      then
        copylf := [f for f in lf | mvar(f) <= mvar(newcp)]
        while (not empty? copylf) and (not ground? newcp) repeat
          f := first copylf
          copylf := rest copylf
          test := (newcp exquo$P f)
          if (test case P)@B
            then
              newcp := test::P
    if ground? newcp
      then
        newp := unitCanonical(newp)
      else
        newp := unitCanonical(newp * newcp)
    newps := cons(newp,newps)
  newps

removeRedundantFactorsInContents (ps,lf) ==
  empty? ps => ps
  newps : LP := []
  p,newp,cp,newcp,f,g : P
  while not empty? ps repeat
    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newcp
      then
        copylf := lf
        while (not empty? copylf) and (not ground? newcp) repeat
          f := first copylf
          copylf := rest copylf
          g := gcd(newcp,f)$P
          if not ground? g
            then
              newcp := (newcp exquo$P g)::P
    if ground? newcp
      then
        newp := unitCanonical(newp)
      else
        newp := unitCanonical(newp * newcp)
    newps := cons(newp,newps)
  newps

removeRedundantFactorsInPols (ps,lf) ==
  empty? ps => ps
  newps : LP := []
  p,newp,cp,newcp,f,g : P
  while not empty? ps repeat
```

```
      p := first ps
      ps := rest ps
      cp := mainContent(p)$P
      newcp := squareFreePart(cp)$P
      newp := (p exquo$P cp)::P
      newp := squareFreePart(newp)$P
      copylf := lf
      while not empty? copylf repeat
        f := first copylf
        copylf := rest copylf
        if not ground? newcp
          then
            g := gcd(newcp,f)$P
            if not ground? g
              then
                newcp := (newcp exquo$P g)::P
        if not ground? newp
          then
            g := gcd(newp,f)$P
            if not ground? g
              then
                newp := (newp exquo$P g)::P
      if ground? newcp
        then
          newp := unitCanonical(newp)
        else
          newp := unitCanonical(newp * newcp)
      newps := cons(newp,newps)
    newps

  removeRedundantFactors (a:P,b:P) : LP ==
    a := primPartElseUnitCanonical(squareFreePart(a))
    b := primPartElseUnitCanonical(squareFreePart(b))
    if not infRittWu?(a,b)
      then
       (a,b) := (b,a)
    if ground? a
      then
        if ground? b
          then
            return([])
          else
            return([b])
      else
        if ground? b
          then
            return([a])
          else
            return(unprotectedRemoveRedundantFactors(a,b))
```

```
    unprotectedRemoveRedundantFactors (a,b) ==
      c := b exquo$P a
      if (c case P)@B
        then
          d : P := c::P
          if ground? d
            then
              return([a])
            else
              return([a,d])
        else
          g : P := gcd(a,b)$P
          if ground? g
            then
              return([a,b])
            else
              return([g,(a exquo$P g)::P,(b exquo$P g)::P])

  else

    removeSquaresIfCan lp ==
      lp

    rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp) ==
      rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp)

    removeRedundantFactors (a:P,b:P) ==
      a := primPartElseUnitCanonical(a)
      b := primPartElseUnitCanonical(b)
      if not infRittWu?(a,b)
        then
         (a,b) := (b,a)
      if ground? a
        then
          if ground? b
            then
              return([])
            else
              return([b])
        else
          if ground? b
            then
              return([a])
            else
              return(unprotectedRemoveRedundantFactors(a,b))

    unprotectedRemoveRedundantFactors (a,b) ==
      c := b exquo$P a
      if (c case P)@B
        then
```

```
        d : P := c::P
        if ground? d
          then
            return([a])
          else
            if infRittWu?(d,a) then (a,d) := (d,a)
            return(unprotectedRemoveRedundantFactors(a,d))
       else
         return([a,b])

removeRedundantFactors (lp:LP) ==
  lp := remove(ground?, lp)
  lp := removeDuplicates [primPartElseUnitCanonical(p) for p in lp]
  lp := removeSquaresIfCan lp
  lp := removeDuplicates [unitCanonical(p) for p in lp]
  empty? lp => lp
  size?(lp,1$N)$(List P) => lp
  lp := sort(infRittWu?,lp)
  p : P := first lp
  lp := rest lp
  base : LP := unprotectedRemoveRedundantFactors(p,first lp)
  top : LP := rest lp
  while not empty? top repeat
    p := first top
    base := removeRedundantFactors(base,p)
    top := rest top
  base

removeRedundantFactors (lp:LP,a:P) ==
  lp := remove(ground?, lp)
  lp := sort(infRittWu?, lp)
  ground? a => lp
  empty? lp => [a]
  toSee : LP := lp
  toSave : LP := []
  while not empty? toSee repeat
    b := first toSee
    toSee := rest toSee
    if not infRittWu?(b,a)
      then
        (c,d) := (a,b)
      else
        (c,d) := (b,a)
    rrf := unprotectedRemoveRedundantFactors(c,d)
    empty? rrf => error"in removeRedundantFactors : (LP,P) -> LP from PSETPK"
    c := first rrf
    rrf := rest rrf
    if empty? rrf
      then
        if associates?(c,b)
```

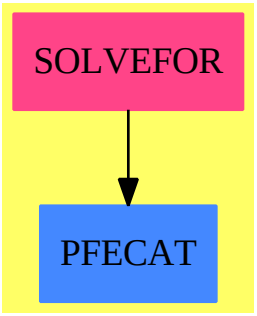
```
            then
              toSave := concat(toSave,toSee)
              a := b
              toSee := []
            else
              a := c
              toSee := concat(toSave,toSee)
              toSave := []
        else
          d := first rrf
          rrf := rest rrf
          if empty? rrf
            then
              if associates?(c,b)
                then
                  toSave := concat(toSave,[b])
                  a := d
                else
                  if associates?(d,b)
                    then
                      toSave := concat(toSave,[b])
                      a := c
                    else
                      toSave := removeRedundantFactors(toSave,c)
                      a := d
            else
              e := first rrf
              not empty? rest(rrf) => error"in removeRedundantFactors:(LP,P)->LP from PSETPK"
              -- ASSUME that neither c, nor d, nor e may be associated to b
              toSave := removeRedundantFactors(toSave,c)
              toSave := removeRedundantFactors(toSave,d)
              a := e
      if empty? toSee
        then
          toSave := sort(infRittWu?,cons(a,toSave))
    toSave
```

— PSETPK.dotabb —

```
"PSETPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSETPK"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"PSETPK" -> "RPOLCAT"
```

17.58 package SOLVEFOR PolynomialSolveByFormulas

17.58.1 PolynomialSolveByFormulas (SOLVEFOR)



Exports:

aCubic	aLinear	aQuadratic	aQuartic	cubic
linear	particularSolution	quadratic	quartic	solve
mapSolve				

— package SOLVEFOR PolynomialSolveByFormulas —

```
)abbrev package SOLVEFOR PolynomialSolveByFormulas
--  Current fields with "**": (%, RationalNumber) -> % are
--     ComplexFloat, RadicalExtension(K) and RationalRadical
--  SMW June 86, BMT Sept 93
++ Description:
++ This package factors the formulas out of the general solve code,
++ allowing their recursive use over different domains.
++ Care is taken to introduce few radicals so that radical extension
++ domains can more easily simplify the results.

PolynomialSolveByFormulas(UP, F): PSFcat == PSFdef where

    UP: UnivariatePolynomialCategory F
    F:  Field with "**": (%, Fraction Integer) -> %

    L  ==> List

    PSFcat == with
        solve:      UP -> L F
          ++ solve(u) \undocumented
        particularSolution:  UP -> F
          ++ particularSolution(u) \undocumented
        mapSolve:   (UP, F -> F) -> Record(solns: L F,
                                           maps: L Record(arg:F,res:F))
          ++ mapSolve(u,f) \undocumented
```

```
        linear:     UP -> L F
          ++ linear(u) \undocumented
        quadratic:  UP -> L F
          ++ quadratic(u) \undocumented
        cubic:      UP -> L F
          ++ cubic(u) \undocumented
        quartic:    UP -> L F
          ++ quartic(u) \undocumented

        -- Arguments give coefs from high to low degree.
        linear:     (F, F)          -> L F
          ++ linear(f,g) \undocumented
        quadratic:  (F, F, F)       -> L F
          ++ quadratic(f,g,h) \undocumented
        cubic:      (F, F, F, F)    -> L F
          ++ cubic(f,g,h,i) \undocumented
        quartic:    (F, F, F, F, F) -> L F
          ++ quartic(f,g,h,i,j) \undocumented

        aLinear:    (F, F)          -> F
          ++ aLinear(f,g) \undocumented
        aQuadratic: (F, F, F)       -> F
          ++ aQuadratic(f,g,h) \undocumented
        aCubic:     (F, F, F, F)    -> F
          ++ aCubic(f,g,h,j) \undocumented
        aQuartic:   (F, F, F, F, F) -> F
          ++ aQuartic(f,g,h,i,k) \undocumented

    PSFdef == add

        ----------------------------------------------------------------
        -- Stuff for mapSolve
        ----------------------------------------------------------------
        id ==> (IDENTITY$Lisp)

        maplist: List Record(arg: F, res: F) := []
        mapSolving?: Boolean := false
        -- map: F -> F := id #1    replaced with line below
        map: Boolean := false

        mapSolve(p, fn) ==
            -- map := fn #1   replaced with line below
            locmap: F -> F := x +-> fn x; map := id locmap
            mapSolving? := true;  maplist := []
            slist := solve p
            mapSolving? := false;
            -- map := id #1   replaced with line below
            locmap := x +-> id x; map := id locmap
            [slist, maplist]
```

```
part(s: F): F ==
    not mapSolving? => s
    -- t := map s     replaced with line below
    t: F := SPADCALL(s, map)$Lisp
    t = s => s
    maplist := cons([t, s], maplist)
    t

------------------------------------------------------------------
-- Entry points and error handling
------------------------------------------------------------------
cc ==> coefficient

-- local intsolve
intsolve(u:UP):L(F) ==
    u := (factors squareFree u).1.factor
    n := degree u
    n=1 => linear    (cc(u,1), cc(u,0))
    n=2 => quadratic (cc(u,2), cc(u,1), cc(u,0))
    n=3 => cubic     (cc(u,3), cc(u,2), cc(u,1), cc(u,0))
    n=4 => quartic   (cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))
    error "All sqfr factors of polynomial must be of degree < 5"

solve u ==
    ls := nil$L(F)
    for f in factors squareFree u repeat
       lsf := intsolve f.factor
       for i in 1..(f.exponent) repeat ls := [:lsf,:ls]
    ls

particularSolution u ==
    u := (factors squareFree u).1.factor
    n := degree u
    n=1 => aLinear    (cc(u,1), cc(u,0))
    n=2 => aQuadratic (cc(u,2), cc(u,1), cc(u,0))
    n=3 => aCubic     (cc(u,3), cc(u,2), cc(u,1), cc(u,0))
    n=4 => aQuartic   (cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))
    error "All sqfr factors of polynomial must be of degree < 5"

needDegree(n: Integer, u: UP): Boolean ==
    degree u = n => true
    error concat("Polynomial must be of degree ", n::String)

needLcoef(cn: F): Boolean ==
    cn ^= 0 => true
    error "Leading coefficient must not be 0."

needChar0(): Boolean ==
    characteristic()$F = 0 => true
    error "Formula defined only for fields of characteristic 0."
```

```
linear u ==
    needDegree(1, u)
    linear (coefficient(u,1), coefficient(u,0))

quadratic u ==
    needDegree(2, u)
    quadratic (coefficient(u,2), coefficient(u,1),
               coefficient(u,0))

cubic u ==
    needDegree(3, u)
    cubic (coefficient(u,3), coefficient(u,2),
           coefficient(u,1), coefficient(u,0))

quartic u ==
    needDegree(4, u)
    quartic (coefficient(u,4),coefficient(u,3),
             coefficient(u,2),coefficient(u,1),coefficient(u,0))

------------------------------------------------------------------
-- The formulas
------------------------------------------------------------------

-- local function for testing equality of radicals.
--  This function is necessary to detect at least some of the
--  situations like sqrt(9)-3 = 0 --> false.
equ(x:F,y:F):Boolean ==
    ( (recip(x-y)) case "failed" ) => true
    false

linear(c1, c0) ==
    needLcoef c1
    [- c0/c1 ]

aLinear(c1, c0) ==
    first linear(c1,c0)

quadratic(c2, c1, c0) ==
    needLcoef c2; needChar0()
    (c0 = 0) => [0$F,:linear(c2, c1)]
    (c1 = 0) => [(-c0/c2)**(1/2),-(-c0/c2)**(1/2)]
    D := part(c1**2 - 4*c2*c0)**(1/2)
    [(-c1+D)/(2*c2), (-c1-D)/(2*c2)]

aQuadratic(c2, c1, c0) ==
    needLcoef c2; needChar0()
    (c0 = 0) => 0$F
    (c1 = 0) => (-c0/c2)**(1/2)
    D := part(c1**2 - 4*c2*c0)**(1/2)
```

```
    (-c1+D)/(2*c2)

w3: F := (-1 + (-3::F)**(1/2)) / 2::F

cubic(c3, c2, c1, c0) ==
    needLcoef c3; needChar0()

    -- case one root = 0, not necessary but keeps result small
    (c0 = 0) => [0$F,:quadratic(c3, c2, c1)]
    a1 := c2/c3;  a2 := c1/c3;  a3 := c0/c3

    -- case x**3-a3 = 0, not necessary but keeps result small
    (a1 = 0 and a2 = 0) =>
        [ u*(-a3)**(1/3) for u in [1, w3, w3**2 ] ]

    -- case x**3 + a1*x**2 + a1**2*x/3 + a3 = 0, the general for-
    --   mula is not valid in this case, but solution is easy.
    P := part(-a1/3::F)
    equ(a1**2,3*a2) =>
      S := part((- a3 + (a1**3)/27::F)**(1/3))
      [ P + S*u for u in [1,w3,w3**2] ]

    -- general case
    Q := part((3*a2 - a1**2)/9::F)
    R := part((9*a1*a2 - 27*a3 - 2*a1**3)/54::F)
    D := part(Q**3 + R**2)**(1/2)
    S := part(R + D)**(1/3)
    -- S = 0 is done in the previous case
    [ P + S*u - Q/(S*u) for u in [1, w3, w3**2] ]

aCubic(c3, c2, c1, c0) ==
    needLcoef c3; needChar0()
    (c0 = 0) => 0$F
    a1 := c2/c3;  a2 := c1/c3;  a3 := c0/c3
    (a1 = 0 and a2 = 0) => (-a3)**(1/3)
    P := part(-a1/3::F)
    equ(a1**2,3*a2) =>
      S := part((- a3 + (a1**3)/27::F)**(1/3))
      P + S
    Q := part((3*a2 - a1**2)/9::F)
    R := part((9*a1*a2 - 27*a3 - 2*a1**3)/54::F)
    D := part(Q**3 + R**2)**(1/2)
    S := part(R + D)**(1/3)
    P + S - Q/S

quartic(c4, c3, c2, c1, c0) ==
    needLcoef c4; needChar0()

    -- case one root = 0, not necessary but keeps result small
    (c0 = 0) => [0$F,:cubic(c4, c3, c2, c1)]
```

```
    -- Make monic:
    a1 := c3/c4; a2 := c2/c4; a3 := c1/c4; a4 := c0/c4

    -- case x**4 + a4 = 0 <=> (x**2-sqrt(-a4))*(x**2+sqrt(-a4))
    -- not necessary but keeps result small.
    (a1 = 0 and a2 = 0 and a3 = 0) =>
        append( quadratic(1, 0, (-a4)**(1/2)),_
                quadratic(1 ,0, -((-a4)**(1/2))) )

    -- Translate w = x+a1/4 to eliminate a1:  w**4+p*w**2+q*w+r
    p := part(a2-3*a1*a1/8::F)
    q := part(a3-a1*a2/2::F + a1**3/8::F)
    r := part(a4-a1*a3/4::F + a1**2*a2/16::F - 3*a1**4/256::F)
    -- t0 := the cubic resolvent of x**3-p*x**2-4*r*x+4*p*r-q**2
    -- The roots of the translated polynomial are those of
    -- two quadratics. (What about rt=0 ?)
    -- rt=0 can be avoided by picking a root ^= p of the cubic
    -- polynomial above. This is always possible provided that
    -- the input is squarefree. In this case the two other roots
    -- are +(-) 2*r**(1/2).
    if equ(q,0)            -- this means p is a root
      then t0 := part(2*(r**(1/2)))
      else t0 := aCubic(1, -p, -4*r, 4*p*r - q**2)
    rt    := part(t0 - p)**(1/2)
    slist := append( quadratic( 1,  rt, (-q/rt + t0)/2::F ),
                     quadratic( 1, -rt, ( q/rt + t0)/2::F ))
    -- Translate back:
    [s - a1/4::F for s in slist]

aQuartic(c4, c3, c2, c1, c0) ==
    needLcoef c4; needChar0()
    (c0 = 0) => 0$F
    a1 := c3/c4; a2 := c2/c4; a3 := c1/c4; a4 := c0/c4
    (a1 = 0 and a2 = 0 and a3 = 0) => (-a4)**(1/4)
    p  := part(a2-3*a1*a1/8::F)
    q  := part(a3-a1*a2/2::F + a1**2*a1/8::F)
    r  := part(a4-a1*a3/4::F + a1**2*a2/16::F - 3*a1**4/256::F)
    if equ(q,0)
      then t0 := part(2*(r**(1/2)))
      else t0 := aCubic(1, -p, -4*r, 4*p*r - q**2)
    rt := part(t0 - p)**(1/2)
    s  := aQuadratic( 1,  rt, (-q/rt + t0)/2::F )
    s - a1/4::F
```

———-

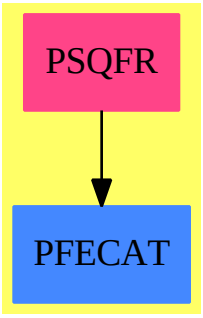
— SOLVEFOR.dotabb —

```
"SOLVEFOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVEFOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SOLVEFOR" -> "PFECAT"
```

———-

17.59 package PSQFR PolynomialSquareFree

17.59.1 PolynomialSquareFree (PSQFR)



Exports:
squareFree

— **package PSQFR PolynomialSquareFree** —

```
)abbrev package PSQFR PolynomialSquareFree
++ Author:
++ Date Created:
++ Date Last Updated: November 1993, (P.Gianni)
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package computes square-free decomposition of multivariate
++ polynomials over a coefficient ring which is an arbitrary gcd domain.
++ The requirement on the coefficient domain guarantees that the
++ \spadfun{content} can be
++ removed so that factors will be primitive as well as square-free.
++ Over an infinite ring of finite characteristic,it may not be possible to
++ guarantee that the factors are square-free.
```

```
PolynomialSquareFree(VarSet:OrderedSet,E,RC:GcdDomain,P):C == T where
  E:OrderedAbelianMonoidSup
  P:PolynomialCategory(RC,E,VarSet)

  C == with
    squareFree : P -> Factored P
      ++ squareFree(p) returns the square-free factorization of the
      ++ polynomial p.  Each factor has no repeated roots, and the
      ++ factors are pairwise relatively prime.

  T == add
    SUP    ==> SparseUnivariatePolynomial(P)
    NNI    ==> NonNegativeInteger
    fUnion ==> Union("nil", "sqfr", "irred", "prime")
    FF     ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

    finSqFr : (P,List VarSet) -> Factored P
    pthPower : P -> Factored P
    pPolRoot : P -> P
    putPth   : P -> P

    chrc:=characteristic$RC

    if RC has CharacteristicNonZero then
    -- find the p-th root of a polynomial
      pPolRoot(f:P) : P ==
        lvar:=variables f
        empty? lvar => f
        mv:=first lvar
        uf:=univariate(f,mv)
        uf:=divideExponents(uf,chrc)::SUP
        uf:=map(pPolRoot,uf)
        multivariate(uf,mv)

    -- substitute variables with their p-th power
      putPth(f:P) : P ==
        lvar:=variables f
        empty? lvar => f
        mv:=first lvar
        uf:=univariate(f,mv)
        uf:=multiplyExponents(uf,chrc)::SUP
        uf:=map(putPth,uf)
        multivariate(uf,mv)

    -- the polynomial is a perfect power
      pthPower(f:P) : Factored P ==
        proot : P := 0
        isSq  : Boolean := false
        if (g:=charthRoot f) case "failed" then proot:=pPolRoot(f)
        else
```

```
      proot := g :: P
      isSq  := true
    psqfr:=finSqFr(proot,variables f)
    isSq  =>
      makeFR((unit psqfr)**chrc,[[u.flg,u.fctr,
       (u.xpnt)*chrc] for u in factorList psqfr])
    makeFR((unit psqfr),[["nil",putPth u.fctr,u.xpnt]
                         for u in factorList psqfr])

-- compute the square free decomposition, finite characteristic case
  finSqFr(f:P,lvar:List VarSet) : Factored P ==
     empty? lvar => pthPower(f)
     mv:=first lvar
     lvar:=lvar.rest
     differentiate(f,mv)=0 => finSqFr(f,lvar)
     uf:=univariate(f,mv)
     cont := content uf
     cont1:P:=1
     uf := (uf exquo cont)::SUP
     squf := squareFree(uf)$UnivariatePolynomialSquareFree(P,SUP)
     pfaclist:List FF :=[]
     for u in factorList squf repeat
       uexp:NNI:=(u.xpnt):NNI
       u.flg = "sqfr" =>  -- the square free factor is OK
         pfaclist:= cons([u.flg,multivariate(u.fctr,mv),uexp],
                         pfaclist)
       --listfin1:= finSqFr(multivariate(u.fctr,mv),lvar)
       listfin1:= squareFree multivariate(u.fctr,mv)
       flistfin1:=[[uu.flg,uu.fctr,uu.xpnt*uexp]
                   for uu in factorList listfin1]
       cont1:=cont1*((unit listfin1)**uexp)
       pfaclist:=append(flistfin1,pfaclist)
     cont:=cont*cont1
     cont ^= 1 =>
       sqp := squareFree cont
       pfaclist:= append (factorList sqp,pfaclist)
       makeFR(unit(sqp)*coefficient(unit squf,0),pfaclist)
     makeFR(coefficient(unit squf,0),pfaclist)

squareFree(p:P) ==
   mv:=mainVariable p
   mv case "failed" => makeFR(p,[])$Factored(P)
   characteristic$RC ^=0 => finSqFr(p,variables p)
   up:=univariate(p,mv)
   cont := content up
   up := (up exquo cont)::SUP
   squp := squareFree(up)$UnivariatePolynomialSquareFree(P,SUP)
   pfaclist:List FF :=
     [[u.flg,multivariate(u.fctr,mv),u.xpnt]
                                       for u in factorList squp]
```

```
        cont ^= 1 =>
          sqp := squareFree cont
          makeFR(unit(sqp)*coefficient(unit squp,0),
               append(factorList sqp, pfaclist))
        makeFR(coefficient(unit squp,0),pfaclist)
```

———-

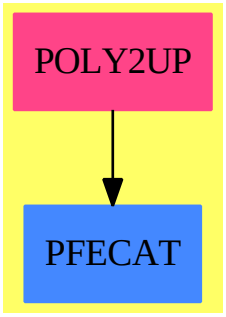
— **PSQFR.dotabb** —

```
"PSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PSQFR" -> "PFECAT"
```

———-

17.60 package POLY2UP PolynomialToUnivariatePolynomial

17.60.1 PolynomialToUnivariatePolynomial (POLY2UP)



Exports:
univariate

— **package POLY2UP PolynomialToUnivariatePolynomial** —

```
)abbrev package POLY2UP PolynomialToUnivariatePolynomial
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
```

```
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package is primarily to help the interpreter do coercions.
++ It allows you to view a polynomial as a
++ univariate polynomial in one of its variables with
++ coefficients which are again a polynomial in all the
++ other variables.

PolynomialToUnivariatePolynomial(x:Symbol, R:Ring): with
  univariate: (Polynomial R, Variable x) ->
                                  UnivariatePolynomial(x, Polynomial R)
     ++ univariate(p, x) converts the polynomial p to a one of type
     ++ \spad{UnivariatePolynomial(x,Polynomial(R))}, ie. as a member of \spad{R[...][x]}.
 == add
  univariate(p, y) ==
    q:SparseUnivariatePolynomial(Polynomial R) := univariate(p, x)
    map(x1+->x1, q)$UnivariatePolynomialCategoryFunctions2(Polynomial R,
                  SparseUnivariatePolynomial Polynomial R, Polynomial R,
                      UnivariatePolynomial(x, Polynomial R))
```

———-

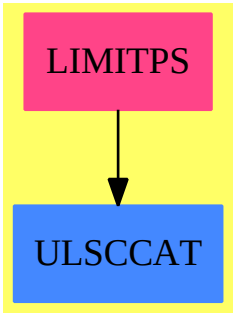
— **POLY2UP.dotabb** —

```
"POLY2UP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLY2UP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLY2UP" -> "PFECAT"
```

———-

17.61 package LIMITPS PowerSeriesLimitPackage

17.61.1 PowerSeriesLimitPackage (LIMITPS)



Exports:
complexLimit limit

— package LIMITPS PowerSeriesLimitPackage —

```
)abbrev package LIMITPS PowerSeriesLimitPackage
++ Author: Clifton J. Williamson
++ Date Created: 21 March 1989
++ Date Last Updated: 30 March 1994
++ Basic Operations:
++ Related Domains: UnivariateLaurentSeries(FE,x,a),
++   UnivariatePuiseuxSeries(FE,x,a),ExponentialExpansion(R,FE,x,a)
++ Also See:
++ AMS Classifications:
++ Keywords: limit, functional expression, power series
++ Examples:
++ References:
++ Description:
++ PowerSeriesLimitPackage implements limits of expressions
++ in one or more variables as one of the variables approaches a
++ limiting value.  Included are two-sided limits, left- and right-
++ hand limits, and limits at plus or minus infinity.

PowerSeriesLimitPackage(R,FE): Exports == Implementation where
  R  : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
            LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
            FunctionSpace R)
  Z       ==> Integer
  RN      ==> Fraction Integer
  RF      ==> Fraction Polynomial R
  OFE     ==> OrderedCompletion FE
```

```
OPF     ==> OnePointCompletion FE
SY      ==> Symbol
EQ      ==> Equation
LF      ==> LiouvillianFunction
UTS     ==> UnivariateTaylorSeries
ULS     ==> UnivariateLaurentSeries
UPXS    ==> UnivariatePuiseuxSeries
EFULS   ==> ElementaryFunctionsUnivariateLaurentSeries
EFUPXS  ==> ElementaryFunctionsUnivariatePuiseuxSeries
FS2UPS  ==> FunctionSpaceToUnivariatePowerSeries
FS2EXPXP ==> FunctionSpaceToExponentialExpansion
Problem ==> Record(func:String,prob:String)
RESULT  ==> Union(OFE,"failed")
TwoSide ==> Record(leftHandLimit:RESULT,rightHandLimit:RESULT)
U       ==> Union(OFE,TwoSide,"failed")
SIGNEF  ==> ElementaryFunctionSign(R,FE)

Exports ==> with

  limit: (FE,EQ OFE) -> U
    ++ limit(f(x),x = a) computes the real limit \spad{lim(x -> a,f(x))}.

  complexLimit: (FE,EQ OPF) -> Union(OPF, "failed")
    ++ complexLimit(f(x),x = a) computes the complex limit
    ++ \spad{lim(x -> a,f(x))}.

  limit: (FE,EQ FE,String) -> RESULT
    ++ limit(f(x),x=a,"left") computes the left hand real limit
    ++ \spad{lim(x -> a-,f(x))};
    ++ \spad{limit(f(x),x=a,"right")} computes the right hand real limit
    ++ \spad{lim(x -> a+,f(x))}.

Implementation ==> add
  import ToolsForSign(R)
  import ElementaryFunctionStructurePackage(R,FE)

  zeroFE:FE := 0
  anyRootsOrAtrigs?   : FE -> Boolean
  complLimit  : (FE,SY) -> Union(OPF,"failed")
  okProblem?  : (String,String) -> Boolean
  realLimit   : (FE,SY) -> U
  xxpLimit    : (FE,SY) -> RESULT
  limitPlus   : (FE,SY) -> RESULT
  localsubst  : (FE,Kernel FE,Z,FE) -> FE
  locallimit  : (FE,SY,OFE) -> U
  locallimitcomplex : (FE,SY,OPF) -> Union(OPF,"failed")
  poleLimit:(RN,FE,SY) -> U
  poleLimitPlus:(RN,FE,SY) -> RESULT

  noX?: (FE,SY) -> Boolean
```

```
noX?(fcn,x) == not member?(x,variables fcn)

constant?: FE -> Boolean
constant? fcn == empty? variables fcn

firstNonLogPtr: (FE,SY) -> List Kernel FE
firstNonLogPtr(fcn,x) ==
  -- returns a pointer to the first element of kernels(fcn) which
  -- has 'x' as a variable, which is not a logarithm, and which is
  -- not simply 'x'
  list := kernels fcn
  while not empty? list repeat
    ker := first list
    not is?(ker,"log" :: Symbol) and member?(x,variables(ker::FE)) _
           and not(x = name(ker)) =>
      return list
    list := rest list
  empty()

finiteValueAtInfinity?: Kernel FE -> Boolean
finiteValueAtInfinity? ker ==
  is?(ker,"erf" :: Symbol) => true
  is?(ker,"sech" :: Symbol) => true
  is?(ker,"csch" :: Symbol) => true
  is?(ker,"tanh" :: Symbol) => true
  is?(ker,"coth" :: Symbol) => true
  is?(ker,"atan" :: Symbol) => true
  is?(ker,"acot" :: Symbol) => true
  is?(ker,"asec" :: Symbol) => true
  is?(ker,"acsc" :: Symbol) => true
  is?(ker,"acsch" :: Symbol) => true
  is?(ker,"acoth" :: Symbol) => true
  is?(ker,"fresnelS" :: Symbol) => true
  is?(ker,"fresnelC" :: Symbol) => true
  error "finiteValueAtInfinity? true, but unknown value at infinity"

knownValueAtInfinity?: Kernel FE -> Boolean
knownValueAtInfinity? ker ==
  is?(ker,"exp" :: Symbol) => true
  is?(ker,"sinh" :: Symbol) => true
  is?(ker,"cosh" :: Symbol) => true
  false

leftOrRight: (FE,SY,FE) -> SingleInteger
leftOrRight(fcn,x,limVal) ==
  -- function is called when limitPlus(fcn,x) = limVal
  -- determines whether the limiting value is approached
  -- from the left or from the right
  (value := limitPlus(inv(fcn - limVal),x)) case "failed" => 0
  (inf := whatInfinity(val := value :: OFE)) = 0 =>
```

```
      error "limit package: internal error"
    inf

  specialLimit1: (FE,SY) -> RESULT
  specialLimitKernel: (Kernel FE,SY) -> RESULT
  specialLimitNormalize: (FE,SY) -> RESULT
  specialLimit: (FE, SY) -> RESULT

  specialLimit(fcn, x) ==
    xkers := [k for k in kernels fcn | member?(x,variables(k::FE))]
    #xkers = 1 => specialLimit1(fcn,x)
    num := numerator fcn
    den := denominator fcn
    for k in xkers repeat
      (fval := limitPlus(k::FE,x)) case "failed" =>
          return specialLimitNormalize(fcn,x)
      whatInfinity(val := fval::OFE) ^= 0 =>
          return specialLimitNormalize(fcn,x)
      (valu := retractIfCan(val)@Union(FE,"failed")) case "failed" =>
          return specialLimitNormalize(fcn,x)
      finVal := valu :: FE
      num := eval(num, k, finVal)
      den := eval(den, k, finVal)
      den = 0 => return specialLimitNormalize(fcn,x)
    (num/den) :: OFE :: RESULT

  specialLimitNormalize(fcn,x) == -- tries to normalize result first
    nfcn := normalize(fcn)
    fcn ^= nfcn => limitPlus(nfcn,x)
    xkers := [k for k in tower fcn | member?(x,variables(k::FE))]
    # xkers ^= 2 => "failed"
    expKers := [k for k in xkers | is?(k, "exp" :: Symbol)]
    # expKers ^= 1 => "failed"
  -- fcn is a rational function of x and exp(g(x)) for some rational function g
    expKer := first expKers
    (fval := limitPlus(expKer::FE,x)) case "failed" => "failed"
    vv := new()$SY; eq : EQ FE := equation(expKer :: FE,vv :: FE)
    cc := eval(fcn,eq)
    expKerLim := fval :: OFE
      -- following test for "failed" is needed due to compiler bug
      -- limVal case OFE generates EQCAR(limVal, 1) which fails on atom "failed"
    (limVal := locallimit(cc,vv,expKerLim)) case "failed" => "failed"
    limVal case OFE =>
       limm := limVal :: OFE
       (lim := retractIfCan(limm)@Union(FE,"failed")) case "failed" =>
             "failed" -- need special handling for directions at infinity
       limitPlus(lim, x)
    "failed"

  -- limit of expression having only 1 kernel involving x
```

```
specialLimit1(fcn,x) ==
  -- find the first interesting kernel in tower(fcn)
  xkers := [k for k in kernels fcn | member?(x,variables(k::FE))]
  #xkers ^= 1 => "failed"
  ker := first xkers
  vv := new()$SY; eq : EQ FE := equation(ker :: FE,vv :: FE)
  cc := eval(fcn,eq)
  member?(x,variables cc) => "failed"
  (lim := specialLimitKernel(ker, x)) case "failed" => lim
  argLim : OFE := lim :: OFE
  (limVal := locallimit(cc,vv,argLim)) case "failed" => "failed"
  limVal case OFE => limVal :: OFE
  "failed"

-- limit of single kernel involving x
specialLimitKernel(ker,x) ==
  is?(ker,"log" :: Symbol) =>
      args := argument ker
      empty? args => "failed" -- error "No argument"
      not empty? rest args => "failed" -- error "Too many arugments"
      arg := first args
      -- compute limit(x -> 0+,arg)
      (limm := limitPlus(arg,x)) case "failed" => "failed"
      lim := limm :: OFE
      (inf := whatInfinity lim) = -1 => "failed"
      argLim : OFE :=
        -- log(+infinity) = +infinity
        inf = 1 => lim
        -- now 'lim' must be finite
        (li := retractIfCan(lim)@Union(FE,"failed") :: FE) = 0 =>
          -- log(0) = -infinity
          leftOrRight(arg,x,0) = 1 => minusInfinity()
          return "failed"
        log(li) :: OFE
  -- kernel should be a function of one argument f(arg)
  args := argument(ker)
  empty? args => "failed"  -- error "No argument"
  not empty? rest args => "failed" -- error "Too many arugments"
  arg := first args
  -- compute limit(x -> 0+,arg)
  (limm := limitPlus(arg,x)) case "failed" => "failed"
  lim := limm :: OFE
  f := elt(operator ker,(var := new()$SY) :: FE)
  -- compute limit(x -> 0+,f(arg))
  -- case where 'lim' is finite
  (inf := whatInfinity lim) = 0 =>
     is?(ker,"erf" :: Symbol) => erf(retract(lim)@FE)$LF(R,FE) :: OFE
     (kerValue := locallimit(f,var,lim)) case "failed" => "failed"
     kerValue case OFE => kerValue :: OFE
     "failed"
```

```
    -- case where 'lim' is plus infinity
    inf = 1 =>
      finiteValueAtInfinity? ker =>
        val : FE :=
          is?(ker,"erf" :: Symbol) => 1
          is?(ker,"sech" :: Symbol) => 0
          is?(ker,"csch" :: Symbol) => 0
          is?(ker,"tanh" :: Symbol) => 0
          is?(ker,"coth" :: Symbol) => 0
          is?(ker,"atan" :: Symbol) => pi()/(2 :: FE)
          is?(ker,"acot" :: Symbol) => 0
          is?(ker,"asec" :: Symbol) => pi()/(2 :: FE)
          is?(ker,"acsc" :: Symbol) => 0
          is?(ker,"acsch" :: Symbol) => 0
          is?(ker,"fresnelS" :: Symbol) => -sqrt(pi()/(8::FE))
          is?(ker,"fresnelC" :: Symbol) => -sqrt(pi()/(8::FE))
          error "finiteValueAtInfinity? true, but unknown value at infinity"
          -- ker must be acoth
          0
        val :: OFE
      knownValueAtInfinity? ker =>
        lim -- limit(exp, cosh, sinh ,x=inf) = inf
      "failed"
    -- case where 'lim' is minus infinity
    finiteValueAtInfinity? ker =>
      val : FE :=
        is?(ker,"erf" :: Symbol) => -1
        is?(ker,"sech" :: Symbol) => 0
        is?(ker,"csch" :: Symbol) => 0
        is?(ker,"tanh" :: Symbol) => 0
        is?(ker,"coth" :: Symbol) => 0
        is?(ker,"atan" :: Symbol) => -pi()/(2 :: FE)
        is?(ker,"acot" :: Symbol) => pi()
        is?(ker,"asec" :: Symbol) => -pi()/(2 :: FE)
        is?(ker,"acsc" :: Symbol) => -pi()
        is?(ker,"acsch" :: Symbol) => 0
        -- ker must be acoth
        0
      val :: OFE
    knownValueAtInfinity? ker =>
      is?(ker,"exp" :: Symbol) => (0@FE) :: OFE
      is?(ker,"sinh" :: Symbol) => lim
      is?(ker,"cosh" :: Symbol) => plusInfinity()
      "failed"
    "failed"

  logOnlyLimit: (FE,SY) -> RESULT
  logOnlyLimit(coef,x) ==
    -- this function is called when the 'constant' coefficient involves
    -- the variable 'x'. Its purpose is to compute a right hand limit
```

```
    -- of an expression involving log x. Here log x is replaced by -1/v,
    -- where v is a new variable. If the new expression no longer involves
    -- x, then take the right hand limit as v -> 0+
    vv := new()$SY
    eq : EQ FE := equation(log(x :: FE),-inv(vv :: FE))
    member?(x,variables(cc := eval(coef,eq))) => "failed"
    limitPlus(cc,vv)

  locallimit(fcn,x,a) ==
    -- Here 'fcn' is a function f(x) = f(x,...) in 'x' and possibly
    -- other variables, and 'a' is a limiting value.  The function
    -- computes lim(x -> a,f(x)).
    xK := retract(x::FE)@Kernel(FE)
    (n := whatInfinity a) = 0 =>
      realLimit(localsubst(fcn,xK,1,retract(a)@FE),x)
    (u := limitPlus(eval(fcn,xK,n * inv(xK::FE)),x))
                                            case "failed" => "failed"
    u::OFE

  localsubst(fcn, k, n, a) ==
    a = 0 and n = 1 => fcn
    eval(fcn,k,n * (k::FE) + a)

  locallimitcomplex(fcn,x,a) ==
    xK := retract(x::FE)@Kernel(FE)
    (g := retractIfCan(a)@Union(FE,"failed")) case FE =>
      complLimit(localsubst(fcn,xK,1,g::FE),x)
    complLimit(eval(fcn,xK,inv(xK::FE)),x)

  limit(fcn,eq,str) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    xK := retract(x::FE)@Kernel(FE)
    limitPlus(localsubst(fcn,xK,direction str,a),x)

  anyRootsOrAtrigs? fcn ==
    -- determines if 'fcn' has any kernels which are roots
    -- or if 'fcn' has any kernels which are inverse trig functions
    -- which could produce series expansions with fractional exponents
    for kernel in tower fcn repeat
      is?(kernel,"nthRoot" :: Symbol) => return true
      is?(kernel,"asin" :: Symbol) => return true
      is?(kernel,"acos" :: Symbol) => return true
      is?(kernel,"asec" :: Symbol) => return true
      is?(kernel,"acsc" :: Symbol) => return true
    false

  complLimit(fcn,x) ==
    -- computes lim(x -> 0,fcn) using a Puiseux expansion of fcn,
```

```
  -- if fcn is an expression involving roots, and using a Laurent
  -- expansion of fcn otherwise
  lim : FE :=
    anyRootsOrAtrigs? fcn =>
      ppack := FS2UPS(R,FE,RN,_
          UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_
          EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
      pseries := exprToUPS(fcn,false,"complex")$ppack
      pseries case %problem => return "failed"
      if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
      pole? upxs => return infinity()
      coefficient(upxs,0)
    lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
             EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
    lseries := exprToUPS(fcn,false,"complex")$lpack
    lseries case %problem => return "failed"
    if pole?(uls := lseries.%series) then uls := map(normalize,uls)
    pole? uls => return infinity()
    coefficient(uls,0)
  -- can the following happen?
  member?(x,variables lim) =>
    member?(x,variables(answer := normalize lim)) =>
      error "limit: can't evaluate limit"
    answer :: OPF
  lim :: FE :: OPF

okProblem?(function,problem) ==
  (function = "log") or (function = "nth root") =>
    (problem = "series of non-zero order") or _
           (problem = "negative leading coefficient")
  (function = "atan") => problem = "branch problem"
  (function = "erf") => problem = "unknown kernel"
  problem = "essential singularity"

poleLimit(order,coef,x) ==
  -- compute limit for function with pole
  not member?(x,variables coef) =>
    (s := sign(coef)$SIGNEF) case Integer =>
      rtLim := (s :: Integer) * plusInfinity()
      even? numer order => rtLim
      even? denom order => ["failed",rtLim]$TwoSide
      [-rtLim,rtLim]$TwoSide
    -- infinite limit, but cannot determine sign
    "failed"
  error "limit: can't evaluate limit"

poleLimitPlus(order,coef,x) ==
  -- compute right hand limit for function with pole
  not member?(x,variables coef) =>
    (s := sign(coef)$SIGNEF) case Integer =>
```

```
        (s :: Integer) * plusInfinity()
      -- infinite limit, but cannot determine sign
      "failed"
    (clim := specialLimit(coef,x)) case "failed" => "failed"
    zero? (lim := clim :: OFE) =>
      -- in this event, we need to determine if the limit of
      -- the coef is 0+ or 0-
      (cclim := specialLimit(inv coef,x)) case "failed" => "failed"
      ss := whatInfinity(cclim :: OFE) :: Z
      zero? ss =>
        error "limit: internal error"
      ss * plusInfinity()
    t := whatInfinity(lim :: OFE) :: Z
    zero? t =>
      (tt := sign(coef)$SIGNEF) case Integer =>
        (tt :: Integer) * plusInfinity()
      -- infinite limit, but cannot determine sign
      "failed"
    t * plusInfinity()

  realLimit(fcn,x) ==
    -- computes lim(x -> 0,fcn) using a Puiseux expansion of fcn,
    -- if fcn is an expression involving roots, and using a Laurent
    -- expansion of fcn otherwise
    lim : Union(FE,"failed") :=
      anyRootsOrAtrigs? fcn =>
        ppack := FS2UPS(R,FE,RN,_
             UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_
             EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
        pseries := exprToUPS(fcn,true,"real: two sides")$ppack
        pseries case %problem =>
          trouble := pseries.%problem
          function := trouble.func; problem := trouble.prob
          okProblem?(function,problem) =>
            left :=
              xK : Kernel FE := kernel x
              fcn0 := eval(fcn,xK,-(xK :: FE))
              limitPlus(fcn0,x)
            right := limitPlus(fcn,x)
            (left case "failed") and (right case "failed") =>
              return "failed"
            if (left case OFE) and (right case OFE) then
              (left :: OFE) = (right :: OFE) => return (left :: OFE)
            return([left,right]$TwoSide)
          return "failed"
        if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
        pole? upxs =>
          cp := coefficient(upxs,ordp := order upxs)
          return poleLimit(ordp,cp,x)
        coefficient(upxs,0)
```

```
      lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
               EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
      lseries := exprToUPS(fcn,true,"real: two sides")$lpack
      lseries case %problem =>
        trouble := lseries.%problem
        function := trouble.func; problem := trouble.prob
        okProblem?(function,problem) =>
          left :=
            xK : Kernel FE := kernel x
            fcn0 := eval(fcn,xK,-(xK :: FE))
            limitPlus(fcn0,x)
          right := limitPlus(fcn,x)
          (left case "failed") and (right case "failed") =>
            return "failed"
          if (left case OFE) and (right case OFE) then
            (left :: OFE) = (right :: OFE) => return (left :: OFE)
          return([left,right]$TwoSide)
        return "failed"
      if pole?(uls := lseries.%series) then uls := map(normalize,uls)
      pole? uls =>
        cl := coefficient(uls,ordl := order uls)
        return poleLimit(ordl :: RN,cl,x)
      coefficient(uls,0)
    lim case "failed" => "failed"
    member?(x,variables(lim :: FE)) =>
      member?(x,variables(answer := normalize(lim :: FE))) =>
        error "limit: can't evaluate limit"
      answer :: OFE
    lim :: FE :: OFE

  xxpLimit(fcn,x) ==
    -- computes lim(x -> 0+,fcn) using an exponential expansion of fcn
    xpack := FS2EXPXP(R,FE,x,zeroFE)
    xxp := exprToXXP(fcn,true)$xpack
    xxp case %problem => "failed"
    limitPlus(xxp.%expansion)

  limitPlus(fcn,x) ==
    -- computes lim(x -> 0+,fcn) using a generalized Puiseux expansion
    -- of fcn, if fcn is an expression involving roots, and using a
    -- generalized Laurent expansion of fcn otherwise
    lim : Union(FE,"failed") :=
      anyRootsOrAtrigs? fcn =>
        ppack := FS2UPS(R,FE,RN,_
             UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_
             EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
        pseries := exprToGenUPS(fcn,true,"real: right side")$ppack
        pseries case %problem =>
          trouble := pseries.%problem
          ff := trouble.func; pp := trouble.prob
```

```
          (pp = "negative leading coefficient") => return "failed"
          "failed"
        -- pseries case %problem => return "failed"
        if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
        pole? upxs =>
          cp := coefficient(upxs,ordp := order upxs)
          return poleLimitPlus(ordp,cp,x)
        coefficient(upxs,0)
      lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
               EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
      lseries := exprToGenUPS(fcn,true,"real: right side")$lpack
      lseries case %problem =>
        trouble := lseries.%problem
        ff := trouble.func; pp := trouble.prob
        (pp = "negative leading coefficient") => return "failed"
        "failed"
      -- lseries case %problem => return "failed"
      if pole?(uls := lseries.%series) then uls := map(normalize,uls)
      pole? uls =>
        cl := coefficient(uls,ordl := order uls)
        return poleLimitPlus(ordl :: RN,cl,x)
      coefficient(uls,0)
    lim case "failed" =>
      (xLim := xxpLimit(fcn,x)) case "failed" => specialLimit(fcn,x)
      xLim
    member?(x,variables(lim :: FE)) =>
      member?(x,variables(answer := normalize(lim :: FE))) =>
        (xLim := xxpLimit(answer,x)) case "failed" => specialLimit(answer,x)
        xLim
      answer :: OFE
    lim :: FE :: OFE

  limit(fcn:FE,eq:EQ OFE) ==
    (f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
    (xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    locallimit(fcn,x,a)

  complexLimit(fcn:FE,eq:EQ OPF) ==
    (f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
    (xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
      error "limit:left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    locallimitcomplex(fcn,x,a)
```

———-

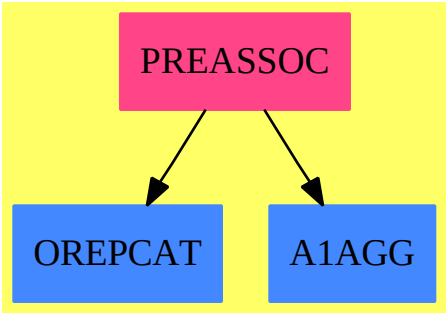
— **LIMITPS.dotabb** —

```
"LIMITPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIMITPS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"LIMITPS" -> "ULSCCAT"
```

———-

17.62 package PREASSOC PrecomputedAssociatedEquations

17.62.1 PrecomputedAssociatedEquations (PREASSOC)



Exports:
firstUncouplingMatrix

— **package PREASSOC PrecomputedAssociatedEquations** —

```
)abbrev package PREASSOC PrecomputedAssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 13 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{PrecomputedAssociatedEquations} stores some generic
++ precomputations which speed up the computations of the
++ associated equations needed for factoring operators.

PrecomputedAssociatedEquations(R, L): Exports == Implementation where
  R: IntegralDomain
  L: LinearOrdinaryDifferentialOperatorCategory R

  PI  ==> PositiveInteger
```

```
N   ==> NonNegativeInteger
A   ==> PrimitiveArray R
U   ==> Union(Matrix R, "failed")

Exports ==> with
  firstUncouplingMatrix: (L, PI) -> U
    ++ firstUncouplingMatrix(op, m) returns the matrix A such that
    ++ \spad{A w = (W',W'',...,W^N)} in the corresponding associated
    ++ equations for right-factors of order m of op.
    ++ Returns "failed" if the matrix A has not been precomputed for
    ++ the particular combination \spad{degree(L), m}.

Implementation ==> add
  A32:  L -> U
  A42:  L -> U
  A425: (A, A, A) -> List R
  A426: (A, A, A) -> List R
  makeMonic: L -> Union(A, "failed")

  diff:L := D()

  firstUncouplingMatrix(op, m) ==
    n := degree op
    n = 3 and m = 2 => A32 op
    n = 4 and m = 2 => A42 op
    "failed"

  makeMonic op ==
    lc := leadingCoefficient op
    a:A := new(n := degree op, 0)
    for i in 0..(n-1)::N repeat
      (u := coefficient(op, i) exquo lc) case "failed" => return "failed"
      a.i := - (u::R)
    a

  A32 op ==
    (u := makeMonic op) case "failed" => "failed"
    a := u::A
    matrix [[0, 1, 0], [a.1, a.2, 1],
            [diff(a.1) + a.1 * a.2 - a.0, diff(a.2) + a.2**2 + a.1, 2 * a.2]]

  A42 op ==
    (u := makeMonic op) case "failed" => "failed"
    a := u::A
    a':A := new(4, 0)
    a'':A := new(4, 0)
    for i in 0..3 repeat
      a'.i := diff(a.i)
      a''.i := diff(a'.i)
    matrix [[0, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0], [a.1,a.2,0,a.3,2::R,0],
```

```
            [a'.1 + a.1 * a.3 - 2 * a.0, a'.2 + a.2 * a.3 + a.1, 3 * a.2,
             a'.3 + a.3 ** 2 + a.2, 3 * a.3, 2::R],
              A425(a, a', a''), A426(a, a', a'')]

  A425(a, a', a'') ==
    [a''.1 + 2 * a.1 * a'.3 + a.3 * a'.1 - 2 * a'.0 + a.1 * a.3 ** 2
     - 3 * a.0 * a.3 + a.1 * a.2,
      a''.2 + 2 * a.2 * a'.3 + a.3 * a'.2 + 2 * a'.1 + a.2 * a.3 ** 2
       + a.1 * a.3 + a.2 ** 2 - 4 * a.0,
        4 * a'.2 + 4 * a.2 * a.3 - a.1,
         a''.3 + 3 * a.3 * a'.3 + 2 * a'.2 + a.3 ** 3 + 2 * a.2 * a.3 + a.1,
          4 * a'.3 + 4 * a.3 ** 2 + 4 * a.2, 5 * a.3]

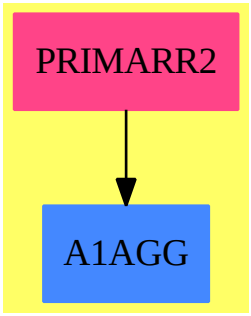
  A426(a, a', a'') ==
    [diff(a''.1) + 3 * a.1 * a''.3 + a.3 * a''.1 - 2 * a''.0
     + (3 * a'.1 + 5 * a.1 * a.3 - 7 * a.0) * a'.3 + 3 * a.1 * a'.2
      + (a.3 ** 2 + a.2) * a'.1 - 3 * a.3 * a'.0 + a.1 * a.3 ** 3
       - 4 * a.0 * a.3 ** 2 + 2 * a.1 * a.2 * a.3 - 4 * a.0 * a.2 + a.1 ** 2,
        diff(a''.2) + 3 * a.2 * a''.3 + a.3 * a''.2 + 3 * a''.1
         + (3*a'.2 + 5*a.2 * a.3 + 3 * a.1) * a'.3 + (a.3**2 + 4*a.2)*a'.2
          + 2 * a.3 * a'.1 - 6 * a'.0 + a.2 * a.3 ** 3 + a.1 * a.3 ** 2
           + (2 * a.2**2 - 8 * a.0) * a.3 + 2 * a.1 * a.2,
            5 * a''.2 + 10 * a.2 * a'.3 + 5 * a.3 * a'.2 + a'.1
             + 5 * a.2 * a.3 ** 2 - 4 * a.1 * a.3 + 5 * a.2**2 - 4 * a.0,
              diff(a''.3) + 4 * a.3 * a''.3 + 3*a''.2 + 3 * a'.3**2
               + (6 * a.3**2 + 4 * a.2) * a'.3 + 5 * a.3 * a'.2 + 3 * a'.1
                + a.3**4 + 3 * a.2 * a.3**2 + 2 * a.1 * a.3 + a.2**2 - 4*a.0,
                 5 * a''.3 + 15 * a.3 * a'.3 + 10 * a'.2 + 5 * a.3**3
                  + 10 * a.2 * a.3, 9 * a'.3 + 9 * a.3**2 + 4 * a.2]
```

— PREASSOC.dotabb —

```
"PREASSOC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PREASSOC"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"PREASSOC" -> "OREPCAT"
"PREASSOC" -> "A1AGG"
```

17.63 package PRIMARR2 PrimitiveArrayFunctions2

17.63.1 PrimitiveArrayFunctions2 (PRIMARR2)



Exports:
map reduce scan

— **package PRIMARR2 PrimitiveArrayFunctions2** —

```
)abbrev package PRIMARR2 PrimitiveArrayFunctions2
++ Description:
++ This package provides tools for operating on primitive arrays
++ with unary and binary functions involving different underlying types

PrimitiveArrayFunctions2(A, B): Exports == Implementation where
  A, B: Type

  VA ==> PrimitiveArray A
  VB ==> PrimitiveArray B
  O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)
  Exports ==> with
   scan : ((A, B) -> B, VA, B) -> VB
    ++ scan(f,a,r) successively applies
    ++ \spad{reduce(f,x,r)} to more and more leading sub-arrays
    ++ x of primitive array \spad{a}.
    ++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
    ++ \spad{scan(f,a,r)} returns
    ++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
    ++
    ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
    ++X adder(a:Integer,b:Integer):Integer == a+b
    ++X scan(adder,[i for i in 1..10],0)$T1

   reduce : ((A, B) -> B, VA, B) -> B
    ++ reduce(f,a,r) applies function f to each
    ++ successive element of the
```

```
    ++ primitive array \spad{a} and an accumulant initialized to r.
    ++ For example, \spad{reduce(_+$Integer,[1,2,3],0)}
    ++ does \spad{3+(2+(1+0))}. Note that third argument r
    ++ may be regarded as the identity element for the function f.
    ++
    ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
    ++X adder(a:Integer,b:Integer):Integer == a+b
    ++X reduce(adder,[i for i in 1..10],0)$T1

   map : (A -> B, VA) -> VB
    ++ map(f,a) applies function f to each member of primitive array
    ++ \spad{a} resulting in a new primitive array over a
    ++ possibly different underlying domain.
    ++
    ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
    ++X map(x+->x+2,[i for i in 1..10])$T1

  Implementation ==> add
    map(f, v)       == map(f, v)$O2
    scan(f, v, b)   == scan(f, v, b)$O2
    reduce(f, v, b) == reduce(f, v, b)$O2
```

———-

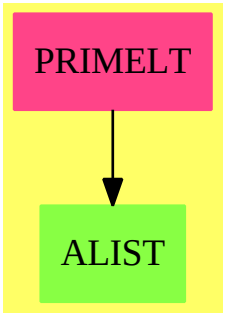
— **PRIMARR2.dotabb** —

```
"PRIMARR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMARR2"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"PRIMARR2" -> "A1AGG"
```

———-

17.64 package PRIMELT PrimitiveElement

17.64.1 PrimitiveElement (PRIMELT)



Exports:
primitiveElement

— package PRIMELT PrimitiveElement —

```
)abbrev package PRIMELT PrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Keywords: algebraic, extension, primitive.
++ Description:
++ PrimitiveElement provides functions to compute primitive elements
++ in algebraic extensions;

PrimitiveElement(F): Exports == Implementation where
  F : Join(Field, CharacteristicZero)

  SY  ==> Symbol
  P   ==> Polynomial F
  UP  ==> SparseUnivariatePolynomial F
  RC  ==> Record(coef1: Integer, coef2: Integer, prim:UP)
  REC ==> Record(coef: List Integer, poly:List UP, prim: UP)

  Exports ==> with
    primitiveElement: (P, SY, P, SY) -> RC
      ++ primitiveElement(p1, a1, p2, a2) returns \spad{[c1, c2, q]}
      ++ such that \spad{k(a1, a2) = k(a)}
      ++ where \spad{a = c1 a1 + c2 a2, and q(a) = 0}.
      ++ The pi's are the defining polynomials for the ai's.
      ++ The p2 may involve a1, but p1 must not involve a2.
      ++ This operation uses \spadfun{resultant}.
    primitiveElement: (List P, List SY) -> REC
```

```
      ++ primitiveElement([p1,...,pn], [a1,...,an]) returns
      ++ \spad{[[c1,...,cn], [q1,...,qn], q]}
      ++ such that then \spad{k(a1,...,an) = k(a)},
      ++ where \spad{a = a1 c1 + ... + an cn},
      ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
      ++ The pi's are the defining polynomials for the ai's.
      ++ This operation uses the technique of
      ++ \spadglossSee{groebner bases}{Groebner basis}.
    primitiveElement: (List P, List SY, SY) -> REC
      ++ primitiveElement([p1,...,pn], [a1,...,an], a) returns
      ++ \spad{[[c1,...,cn], [q1,...,qn], q]}
      ++ such that then \spad{k(a1,...,an) = k(a)},
      ++ where \spad{a = a1 c1 + ... + an cn},
      ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
      ++ The pi's are the defining polynomials for the ai's.
      ++ This operation uses the technique of
      ++ \spadglossSee{groebner bases}{Groebner basis}.

  Implementation ==> add
    import PolyGroebner(F)

    multi     : (UP, SY) -> P
    randomInts: (NonNegativeInteger, NonNegativeInteger) -> List Integer
    findUniv  : (List P, SY, SY) -> Union(P, "failed")
    incl?     : (List SY, List SY) -> Boolean
    triangularLinearIfCan:(List P,List SY,SY) -> Union(List UP,"failed")
    innerPrimitiveElement: (List P, List SY, SY) -> REC

    multi(p, v)            == multivariate(map((f1:F):F +-> f1, p), v)
    randomInts(n, m)       == [symmetricRemainder(random()$Integer, m) for i in 1..n]
    incl?(a, b)            == every?((s1:SY):Boolean +-> member?(s1, b), a)
    primitiveElement(l, v) == primitiveElement(l, v, new()$SY)

    primitiveElement(p1, a1, p2, a2) ==
--      one? degree(p2, a1) => [0, 1, univariate resultant(p1, p2, a1)]
      (degree(p2, a1) = 1) => [0, 1, univariate resultant(p1, p2, a1)]
      u := (new()$SY)::P
      b := a2::P
      for i in 10.. repeat
        c := symmetricRemainder(random()$Integer, i)
        w := u - c * b
        r := univariate resultant(eval(p1, a1, w), eval(p2, a1, w), a2)
        not zero? r and r = squareFreePart r => return [1, c, r]

    findUniv(l, v, opt) ==
      for p in l repeat
        degree(p, v) > 0 and incl?(variables p, [v, opt]) => return p
      "failed"

    triangularLinearIfCan(l, lv, w) ==
```

```
      (u := findUniv(l, w, w)) case "failed" => "failed"
      pw := univariate(u::P)
      ll := nil()$List(UP)
      for v in lv repeat
        ((u := findUniv(l, v, w)) case "failed") or
          (degree(p := univariate(u::P, v)) ^= 1) => return "failed"
        (bc := extendedEuclidean(univariate leadingCoefficient p, pw,1))
           case "failed" => error "Should not happen"
        ll := concat(map((z1:F):F +-> z1,
                 (- univariate(coefficient(p,0)) * bc.coef1) rem pw), ll)
      concat(map((f1:F):F +-> f1, pw), reverse_! ll)

    primitiveElement(l, vars, uu) ==
      u    := uu::P
      vv   := [v::P for v in vars]
      elim := concat(vars, uu)
      w    := uu::P
      n    := #l
      for i in 10.. repeat
        cf := randomInts(n, i)
        (tt := triangularLinearIfCan(lexGroebner(
             concat(w - +/[c * t for c in cf for t in vv], l), elim),
                vars, uu)) case List(UP) =>
                   ltt := tt::List(UP)
                   return([cf, rest ltt, first ltt])
```

———-

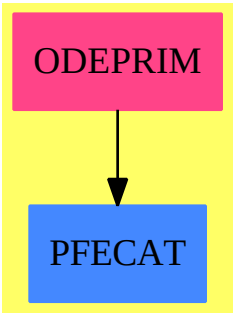
— PRIMELT.dotabb —

```
"PRIMELT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMELT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PRIMELT" -> "ALIST"
```

———-

17.65 package ODEPRIM PrimitiveRatDE

17.65.1 PrimitiveRatDE (ODEPRIM)



Exports:
indicialEquation denomLODE indicialEquations splitDenominator

— package ODEPRIM PrimitiveRatDE —

```
)abbrev package ODEPRIM PrimitiveRatDE
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{PrimitiveRatDE} provides functions for in-field solutions of linear
++ ordinary differential equations, in the transcendental case.
++ The derivation to use is given by the parameter \spad{L}.

PrimitiveRatDE(F, UP, L, LQ): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  L  : LinearOrdinaryDifferentialOperatorCategory UP
  LQ : LinearOrdinaryDifferentialOperatorCategory Fraction UP

  N   ==> NonNegativeInteger
  Z   ==> Integer
  RF  ==> Fraction UP
  UP2 ==> SparseUnivariatePolynomial UP
  REC ==> Record(center:UP, equation:UP)

  Exports ==> with
    denomLODE: (L, RF) -> Union(UP, "failed")
      ++ denomLODE(op, g) returns a polynomial d such that
      ++ any rational solution of \spad{op y = g}
      ++ is of the form \spad{p/d} for some polynomial p, and
      ++ "failed", if the equation has no rational solution.
```

```
  denomLODE: (L, List RF) -> UP
    ++ denomLODE(op, [g1,...,gm]) returns a polynomial
    ++ d such that any rational solution of \spad{op y = c1 g1 + ... + cm gm}
    ++ is of the form \spad{p/d} for some polynomial p.
  indicialEquations: L -> List REC
    ++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
    ++ the \spad{d_i}'s are the affine singularities of \spad{op},
    ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
  indicialEquations: (L, UP) -> List REC
    ++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
    ++ the \spad{d_i}'s are the affine singularities of \spad{op}
    ++ above the roots of \spad{p},
    ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
  indicialEquation: (L, F) -> UP
    ++ indicialEquation(op, a) returns the indicial equation of \spad{op}
    ++ at \spad{a}.
  indicialEquations: LQ -> List REC
    ++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
    ++ the \spad{d_i}'s are the affine singularities of \spad{op},
    ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
  indicialEquations: (LQ, UP) -> List REC
    ++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
    ++ the \spad{d_i}'s are the affine singularities of \spad{op}
    ++ above the roots of \spad{p},
    ++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.
  indicialEquation: (LQ, F) -> UP
    ++ indicialEquation(op, a) returns the indicial equation of \spad{op}
    ++ at \spad{a}.
  splitDenominator: (LQ, List RF) -> Record(eq:L, rh:List RF)
    ++ splitDenominator(op, [g1,...,gm]) returns \spad{op0, [h1,...,hm]}
    ++ such that the equations \spad{op y = c1 g1 + ... + cm gm} and
    ++ \spad{op0 y = c1 h1 + ... + cm hm} have the same solutions.

Implementation ==> add
  import BoundIntegerRoots(F, UP)
  import BalancedFactorisation(F, UP)
  import InnerCommonDenominator(UP, RF, List UP, List RF)
  import UnivariatePolynomialCategoryFunctions2(F, UP, UP, UP2)

  tau          : (UP, UP, UP, N) -> UP
  NPbound      : (UP, L, UP) -> N
  hdenom       : (L, UP, UP) -> UP
  denom0       : (Z, L, UP, UP, UP) -> UP
  indicialEq   : (UP, List N, List UP) -> UP
  separateZeros: (UP, UP) -> UP
  UPfact       : N -> UP
  UP2UP2       : UP -> UP2
  indeq        : (UP, L) -> UP
  NPmulambda   : (UP, L) -> Record(mu:Z, lambda:List N, func:List UP)
```

```
    diff := D()$L

    UP2UP2 p                    == map((f1:F):UP +->f1::UP, p)
    indicialEquations(op:L)     == indicialEquations(op, leadingCoefficient op)
    indicialEquation(op:L, a:F) == indeq(monomial(1, 1) - a::UP, op)

    splitDenominator(op, lg) ==
      cd := splitDenominator coefficients op
      f  := cd.den / gcd(cd.num)
      l:L := 0
      while op ^= 0 repeat
          l  := l + monomial(retract(f * leadingCoefficient op), degree op)
          op := reductum op
      [l, [f * g for g in lg]]

    tau(p, pp, q, n) ==
      ((pp ** n) * ((q exquo (p ** order(q, p)))::UP)) rem p

    indicialEquations(op:LQ) ==
      indicialEquations(splitDenominator(op, empty()).eq)

    indicialEquations(op:LQ, p:UP) ==
      indicialEquations(splitDenominator(op, empty()).eq, p)

    indicialEquation(op:LQ, a:F) ==
      indeq(monomial(1, 1) - a::UP, splitDenominator(op, empty()).eq)

-- returns z(z-1)...(z-(n-1))
    UPfact n ==
      zero? n => 1
      z := monomial(1, 1)$UP
      */[z - i::F::UP for i in 0..(n-1)::N]

    indicialEq(c, lamb, lf) ==
      cp := diff c
      cc := UP2UP2 c
      s:UP2 := 0
      for i in lamb for f in lf repeat
        s := s + (UPfact i) * UP2UP2 tau(c, cp, f, i)
      primitivePart resultant(cc, s)

    NPmulambda(c, l) ==
      lamb:List(N) := [d := degree l]
      lf:List(UP) := [a := leadingCoefficient l]
      mup := d::Z - order(a, c)
      while (l := reductum l) ^= 0 repeat
          a := leadingCoefficient l
          if (m := (d := degree l)::Z - order(a, c)) > mup then
              mup := m
              lamb := [d]
```

```
            lf := [a]
        else if (m = mup) then
            lamb := concat(d, lamb)
            lf := concat(a, lf)
      [mup, lamb, lf]

-- e = 0 means homogeneous equation
    NPbound(c, l, e) ==
      rec := NPmulambda(c, l)
      n := max(0, - integerBound indicialEq(c, rec.lambda, rec.func))
      zero? e => n::N
      max(n, order(e, c)::Z - rec.mu)::N

    hdenom(l, d, e) ==
      */[dd.factor ** NPbound(dd.factor, l, e)
                   for dd in factors balancedFactorisation(d, coefficients l)]

    denom0(n, l, d, e, h) ==
      hdenom(l, d, e) * */[hh.factor ** max(0, order(e, hh.factor) - n)::N
                                  for hh in factors balancedFactorisation(h, e)]

-- returns a polynomials whose zeros are the zeros of e which are not
-- zeros of d
    separateZeros(d, e) ==
      ((g := squareFreePart e) exquo gcd(g, squareFreePart d))::UP

    indeq(c, l) ==
      rec := NPmulambda(c, l)
      indicialEq(c, rec.lambda, rec.func)

    indicialEquations(op:L, p:UP) ==
      [[dd.factor, indeq(dd.factor, op)]
                  for dd in factors balancedFactorisation(p, coefficients op)]

-- cannot return "failed" in the homogeneous case
    denomLODE(l:L, g:RF) ==
      d := leadingCoefficient l
      zero? g => hdenom(l, d, 0)
      h := separateZeros(d, e := denom g)
      n := degree l
      (e exquo (h**(n + 1))) case "failed" => "failed"
      denom0(n, l, d, e, h)

    denomLODE(l:L, lg:List RF) ==
      empty? lg => denomLODE(l, 0)::UP
      d := leadingCoefficient l
      h := separateZeros(d, e := "lcm"/[denom g for g in lg])
      denom0(degree l, l, d, e, h)
```

———-

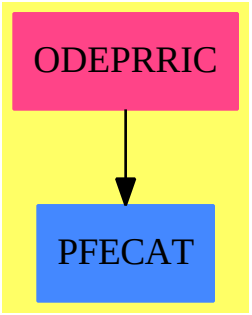
— **ODEPRIM.dotabb** —

```
"ODEPRIM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPRIM"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ODEPRIM" -> "PFECAT"
```

———-

17.66 package ODEPRRIC PrimitiveRatRicDE

17.66.1 PrimitiveRatRicDE (ODEPRRIC)



Exports:

changeVar	denomRicDE	constantCoefficientRicDE	leadingCoefficientRicDE
polyRicDE	singRicDE		

— **package ODEPRRIC PrimitiveRatRicDE** —

```
)abbrev package ODEPRRIC PrimitiveRatRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 2 February 1993
++ Description:
++ In-field solution of Riccati equations, primitive case.

PrimitiveRatRicDE(F, UP, L, LQ): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  L  : LinearOrdinaryDifferentialOperatorCategory UP
  LQ : LinearOrdinaryDifferentialOperatorCategory Fraction UP
```

```
N    ==> NonNegativeInteger
Z    ==> Integer
RF   ==> Fraction UP
UP2  ==> SparseUnivariatePolynomial UP
REC  ==> Record(deg:N, eq:UP)
REC2 ==> Record(deg:N, eq:UP2)
POL  ==> Record(poly:UP, eq:L)
FRC  ==> Record(frac:RF, eq:L)
CNT  ==> Record(constant:F, eq:L)
IJ   ==> Record(ij: List Z, deg:N)

Exports ==> with
  denomRicDE: L -> UP
    ++ denomRicDE(op) returns a polynomial \spad{d} such that any rational
    ++ solution of the associated Riccati equation of \spad{op y = 0} is
    ++ of the form \spad{p/d + q'/q + r} for some polynomials p and q
    ++ and a reduced r. Also, \spad{deg(p) < deg(d)} and {gcd(d,q) = 1}.
  leadingCoefficientRicDE:  L -> List REC
    ++ leadingCoefficientRicDE(op) returns
    ++ \spad{[[m1, p1], [m2, p2], ... , [mk, pk]]} such that the polynomial
    ++ part of any rational solution of the associated Riccati equation of
    ++ \spad{op y = 0} must have degree mj for some j, and its leading
    ++ coefficient is then a zero of pj. In addition,\spad{m1>m2> ... >mk}.
  constantCoefficientRicDE: (L, UP -> List F) -> List CNT
    ++ constantCoefficientRicDE(op, ric) returns
    ++ \spad{[[a1, L1], [a2, L2], ... , [ak, Lk]]} such that any rational
    ++ solution with no polynomial part of the associated Riccati equation of
    ++ \spad{op y = 0} must be one of the ai's in which case the equation for
    ++ \spad{z = y e^{-int ai}} is \spad{Li z = 0}.
    ++ \spad{ric} is a Riccati equation solver over \spad{F}, whose input
    ++ is the associated linear equation.
  polyRicDE: (L, UP -> List F) -> List POL
    ++ polyRicDE(op, zeros) returns
    ++ \spad{[[p1, L1], [p2, L2], ... , [pk, Lk]]} such that the polynomial
    ++ part of any rational solution of the associated Riccati equation of
    ++ \spad{op y=0} must be one of the pi's (up to the constant coefficient),
    ++ in which case the equation for \spad{z=y e^{-int p}} is \spad{Li z =0}.
    ++ \spad{zeros} is a zero finder in \spad{UP}.
  singRicDE: (L, (UP, UP2) -> List UP, UP -> Factored UP) -> List FRC
    ++ singRicDE(op, zeros, ezfactor) returns
    ++ \spad{[[f1, L1], [f2, L2], ... , [fk, Lk]]} such that the singular
    ++ part of any rational solution of the associated Riccati equation of
    ++ \spad{op y=0} must be one of the fi's (up to the constant coefficient),
    ++ in which case the equation for \spad{z=y e^{-int p}} is \spad{Li z=0}.
    ++ \spad{zeros(C(x),H(x,y))} returns all the \spad{P_i(x)}'s such that
    ++ \spad{H(x,P_i(x)) = 0 modulo C(x)}.
    ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
    ++ not necessarily into irreducibles.
  changeVar: (L, UP) -> L
    ++ changeVar(+/[ai D^i], a) returns the operator \spad{+/[ai (D+a)^i]}.
```

```
    changeVar: (L, RF) -> L
      ++ changeVar(+/[ai D^i], a) returns the operator \spad{+/[ai (D+a)^i]}.

  Implementation ==> add
    import PrimitiveRatDE(F, UP, L, LQ)
    import BalancedFactorisation(F, UP)

    bound             : (UP, L) -> N
    lambda            : (UP, L) -> List IJ
    infmax            : (IJ, L) -> List Z
    dmax              : (IJ, UP, L) -> List Z
    getPoly           : (IJ, L, List Z) -> UP
    getPol            : (IJ, UP, L, List Z) -> UP2
    innerlb           : (L, UP -> Z) -> List IJ
    innermax          : (IJ, L, UP -> Z) -> List Z
    tau0              : (UP, UP) -> UP
    poly1             : (UP, UP, Z) -> UP2
    getPol1           : (List Z, UP, L) -> UP2
    getIndices        : (N, List IJ) -> List Z
    refine            : (List UP, UP -> Factored UP) -> List UP
    polysol           : (L, N, Boolean, UP -> List F) -> List POL
    fracsol           : (L, (UP, UP2) -> List UP, List UP) -> List FRC
    padicsol      l   : (UP, L, N, Boolean, (UP, UP2) -> List UP) -> List FRC
    leadingDenomRicDE : (UP, L) -> List REC2
    factoredDenomRicDE: L -> List UP
    constantCoefficientOperator: (L, N) -> UP
    infLambda: L -> List IJ
      -- infLambda(op) returns
      -- \spad{[[[i,j], (\deg(a_i)-\deg(a_j))/(i-j) ]]} for all the pairs
      -- of indices \spad{i,j} such that \spad{(\deg(a_i)-\deg(a_j))/(i-j)} is
      -- an integer.

    diff  := D()$L
    diffq := D()$LQ

    lambda(c, l)       == innerlb(l, z +-> order(z, c)::Z)
    infLambda l        == innerlb(l, z +-> -(degree(z)::Z))
    infmax(rec,l)      == innermax(rec, l, z +-> degree(z)::Z)
    dmax(rec, c,l)     == innermax(rec, l, z +-> - order(z, c)::Z)
    tau0(p, q)         == ((q exquo (p ** order(q, p)))::UP) rem p
    poly1(c, cp,i)     == */[monomial(1,1)$UP2 - (j * cp)::UP2 for j in 0..i-1]
    getIndices(n,l)    == removeDuplicates_! concat [r.ij for r in l | r.deg=n]
    denomRicDE l       == */[c ** bound(c, l) for c in factoredDenomRicDE l]
    polyRicDE(l,zeros) == concat([0, l], polysol(l, 0, false, zeros))

-- refine([p1,...,pn], foo) refines the list of factors using foo
    refine(l, ezfactor) ==
      concat [[r.factor for r in factors ezfactor p] for p in l]

-- returns [] if the solutions of l have no p-adic component at c
```

```
    padicsol(c, op, b, finite?, zeros) ==
      ans:List(FRC) := empty()
      finite? and zero? b => ans
      lc := leadingDenomRicDE(c, op)
      if finite? then lc := select_!(z +-> z.deg <= b, lc)
      for rec in lc repeat
        for r in zeros(c, rec.eq) | r ^= 0 repeat
          rcn := r /$RF (c ** rec.deg)
          neweq := changeVar(op, rcn)
          sols := padicsol(c, neweq, (rec.deg-1)::N, true, zeros)
          ans :=
            empty? sols => concat([rcn, neweq], ans)
            concat_!([[rcn + sol.frac, sol.eq] for sol in sols], ans)
      ans

    leadingDenomRicDE(c, l) ==
      ind:List(Z)          -- to cure the compiler... (won't compile without)
      lb := lambda(c, l)
      done:List(N) := empty()
      ans:List(REC2) := empty()
      for rec in lb | (not member?(rec.deg, done)) and
        not(empty?(ind := dmax(rec, c, l))) repeat
          ans := concat([rec.deg, getPol(rec, c, l, ind)], ans)
          done := concat(rec.deg, done)
      sort_!((z1,z2) +-> z1.deg > z2.deg, ans)

    getPol(rec, c, l, ind) ==
--      one?(rec.deg) => getPol1(ind, c, l)
      (rec.deg = 1) => getPol1(ind, c, l)
      +/[monomial(tau0(c, coefficient(l, i::N)), i::N)$UP2 for i in ind]

    getPol1(ind, c, l) ==
      cp := diff c
      +/[tau0(c, coefficient(l, i::N)) * poly1(c, cp, i) for i in ind]

    constantCoefficientRicDE(op, ric) ==
      m := "max"/[degree p for p in coefficients op]
      [[a, changeVar(op,a::UP)] for a in ric constantCoefficientOperator(op,m)]

    constantCoefficientOperator(op, m) ==
      ans:UP := 0
      while op ^= 0 repeat
        if degree(p := leadingCoefficient op) = m then
          ans := ans + monomial(leadingCoefficient p, degree op)
        op := reductum op
      ans

    getPoly(rec, l, ind) ==
      +/[monomial(leadingCoefficient coefficient(l,i::N),i::N)$UP for i in ind]
```

```
-- returns empty() if rec is does not reach the max,
-- the list of indices (including rec) that reach the max otherwise
    innermax(rec, l, nu) ==
      n := degree l
      i := first(rec.ij)
      m := i * (d := rec.deg) + nu coefficient(l, i::N)
      ans:List(Z) := empty()
      for j in 0..n | (f := coefficient(l, j)) ^= 0 repeat
        if ((k := (j * d + nu f)) > m) then return empty()
        else if (k = m) then ans := concat(j, ans)
      ans

    leadingCoefficientRicDE l ==
      ind:List(Z)          -- to cure the compiler... (won't compile without)
      lb := infLambda l
      done:List(N) := empty()
      ans:List(REC) := empty()
      for rec in lb | (not member?(rec.deg, done)) and
        not(empty?(ind := infmax(rec, l))) repeat
          ans := concat([rec.deg, getPoly(rec, l, ind)], ans)
          done := concat(rec.deg, done)
      sort_!((z1,z2) +-> z1.deg > z2.deg, ans)

    factoredDenomRicDE l ==
      bd := factors balancedFactorisation(leadingCoefficient l, coefficients l)
      [dd.factor for dd in bd]

    changeVar(l:L, a:UP) ==
      dpa := diff + a::L            -- the operator (D + a)
      dpan:L := 1                   -- will accumulate the powers of (D + a)
      op:L := 0
      for i in 0..degree l repeat
        op   := op + coefficient(l, i) * dpan
        dpan := dpa * dpan
      primitivePart op

    changeVar(l:L, a:RF) ==
      dpa := diffq + a::LQ          -- the operator (D + a)
      dpan:LQ := 1                  -- will accumulate the powers of (D + a)
      op:LQ := 0
      for i in 0..degree l repeat
        op   := op + coefficient(l, i)::RF * dpan
        dpan := dpa * dpan
      splitDenominator(op, empty()).eq

    bound(c, l) ==
      empty?(lb := lambda(c, l)) => 1
      "max"/[rec.deg for rec in lb]

-- returns all the pairs [[i, j], n] such that
```

```
-- n = (nu(i) - nu(j)) / (i - j) is an integer
    innerlb(l, nu) ==
      lb:List(IJ) := empty()
      n := degree l
      for i in 0..n | (li := coefficient(l, i)) ^= 0repeat
        for j in i+1..n | (lj := coefficient(l, j)) ^= 0 repeat
          u := (nu li - nu lj) exquo (i-j)
          if (u case Z) and ((b := u::Z) > 0) then
            lb := concat([[i, j], b::N], lb)
      lb

    singRicDE(l, zeros, ezfactor) ==
      concat([0, l], fracsol(l, zeros, refine(factoredDenomRicDE l, ezfactor)))

-- returns [] if the solutions of l have no singular component
    fracsol(l, zeros, lc) ==
      ans:List(FRC) := empty()
      empty? lc => ans
      empty?(sols := padicsol(first lc, l, 0, false, zeros)) =>
        fracsol(l, zeros, rest lc)
      for rec in sols repeat
        neweq := changeVar(l, rec.frac)
        sols := fracsol(neweq, zeros, rest lc)
        ans :=
          empty? sols => concat(rec, ans)
          concat_!([[rec.frac + sol.frac, sol.eq] for sol in sols], ans)
      ans

-- returns [] if the solutions of l have no polynomial component
    polysol(l, b, finite?, zeros) ==
      ans:List(POL) := empty()
      finite? and zero? b => ans
      lc := leadingCoefficientRicDE l
      if finite? then lc := select_!(z +-> z.deg <= b, lc)
      for rec in lc repeat
        for a in zeros(rec.eq) | a ^= 0 repeat
          atn:UP := monomial(a, rec.deg)
          neweq := changeVar(l, atn)
          sols := polysol(neweq, (rec.deg - 1)::N, true, zeros)
          ans :=
            empty? sols => concat([atn, neweq], ans)
            concat_!([[atn + sol.poly, sol.eq] for sol in sols], ans)
      ans
```

———-

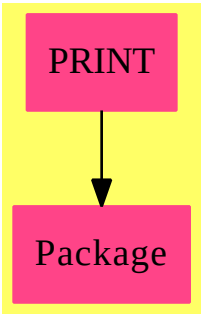
— ODEPRRIC.dotabb —

```
"ODEPRRIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPRRIC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ODEPRRIC" -> "PFECAT"
```

———-

17.67 package PRINT PrintPackage

17.67.1 PrintPackage (PRINT)



Exports:
print

— package PRINT PrintPackage —

```
)abbrev package PRINT PrintPackage
++ Author: Scott Morrison
++ Date Created: Aug. 1, 1990
++ Date Last Updated:
++ Basic Operations: print
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: print
++ References:
++ Description:
++ PrintPackage provides a print function for output forms.

PrintPackage(): with
    print : OutputForm ->  Void
      ++ print(o) writes the output form o on standard output using the
      ++ two-dimensional formatter.
 == add
    print(x) == print(x)$OutputForm
```

———-

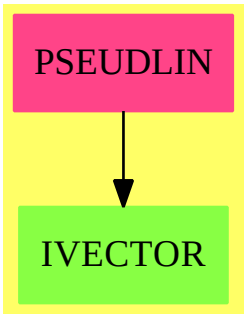
— **PRINT.dotabb** —

```
"PRINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRINT"]
"Package" [color="#FF4488"]
"PRINT" -> "Package"
```

———-

17.68 package PSEUDLIN PseudoLinearNormalForm

17.68.1 PseudoLinearNormalForm (PSEUDLIN)



Exports:
changeBase companionBlocks normalForm

— **package PSEUDLIN PseudoLinearNormalForm** —

```
)abbrev package PSEUDLIN PseudoLinearNormalForm
++ Author: Bruno Zuercher
++ Date Created: November 1993
++ Date Last Updated: 12 April 1994
++ Description:
++ PseudoLinearNormalForm provides a function for computing a block-companion
++ form for pseudo-linear operators.

PseudoLinearNormalForm(K:Field): Exports == Implementation where
  ER  ==> Record(C: Matrix K, g: Vector K)
  REC ==> Record(R: Matrix K, A: Matrix K, Ainv: Matrix K)
```

```
Exports ==> with
  normalForm: (Matrix K, Automorphism K, K -> K) -> REC
    ++ normalForm(M, sig, der) returns \spad{[R, A, A^{-1}]} such that
    ++ the pseudo-linear operator whose matrix in the basis \spad{y} is
    ++ \spad{M} had matrix \spad{R} in the basis \spad{z = A y}.
    ++ \spad{der} is a \spad{sig}-derivation.
  changeBase: (Matrix K, Matrix K, Automorphism K, K -> K) -> Matrix K
    ++ changeBase(M, A, sig, der): computes the new matrix of a pseudo-linear
    ++ transform given by the matrix M under the change of base A
  companionBlocks: (Matrix K, Vector K) -> List ER
    ++ companionBlocks(m, v) returns \spad{[[C_1, g_1],...,[C_k, g_k]]}
    ++ such that each \spad{C_i} is a companion block and
    ++ \spad{m = diagonal(C_1,...,C_k)}.

Implementation ==> add
  normalForm0: (Matrix K, Automorphism K, Automorphism K, K -> K) -> REC
  mulMatrix: (Integer, Integer, K) -> Matrix K
    -- mulMatrix(N, i, a): under a change of base with the resulting matrix of
    -- size N*N the following operations are performed:
    -- D1: column i will be multiplied by sig(a)
    -- D2: row i will be multiplied by 1/a
    -- D3: addition of der(a)/a to the element at position (i,i)
  addMatrix: (Integer, Integer, Integer, K) -> Matrix K
    -- addMatrix(N, i, k, a): under a change of base with the resulting matrix
    -- of size N*N the following operations are performed:
    -- C1: addition of column i multiplied by sig(a) to column k
    -- C2: addition of row k multiplied by -a to row i
    -- C3: addition of -a*der(a) to the element at position (i,k)
  permutationMatrix: (Integer, Integer, Integer) -> Matrix K
    -- permutationMatrix(N, i, k): under a change of base with the resulting
    -- permutation matrix of size N*N the following operations are performed:
    -- P1: columns i and k will be exchanged
    -- P2: rows i and k will be exchanged
  inv: Matrix K -> Matrix K
    -- inv(M): computes the inverse of a invertable matrix M.
    -- avoids possible type conflicts

  inv m                       == inverse(m) :: Matrix K
  changeBase(M, A, sig, der) ==
    inv(A) * (M * map((k1:K):K +-> sig k1, A) + map(der, A))
  normalForm(M, sig, der)    == normalForm0(M, sig, inv sig, der)

  companionBlocks(R, w) ==
    -- decomposes the rational matrix R into single companion blocks
    -- and the inhomogenity w as well
    i:Integer := 1
    n := nrows R
    l:List(ER) := empty()
    while i <= n repeat
      j := i
```

```
      while j+1 <= n and R(j,j+1) = 1 repeat j := j+1
      --split block now
      v:Vector K := new((j-i+1)::NonNegativeInteger, 0)
      for k in i..j repeat v(k-i+1) := w k
      l := concat([subMatrix(R,i,j,i,j), v], l)
      i := j+1
    l

  normalForm0(M, sig, siginv, der) ==
    -- the changes of base will be incremented in B and Binv,
    -- where B**(-1)=Binv; E defines an elementary matrix
    B, Binv, E    : Matrix K
    recOfMatrices : REC
    N := nrows M
    B := diagonalMatrix [1 for k in 1..N]
    Binv := copy B
    -- avoid unnecessary recursion
    if diagonal?(M) then return [M, B, Binv]
    i : Integer := 1
    while i < N repeat
      j := i + 1
      while j <= N and M(i, j) = 0 repeat  j := j + 1
      if j <= N then
        -- expand companionblock by lemma 5
        if j ^= i+1 then
          -- perform first a permutation
          E := permutationMatrix(N, i+1, j)
          M := changeBase(M, E, sig, der)
          B := B*E
          Binv := E*Binv
        -- now is M(i, i+1) ^= 0
        E := mulMatrix(N, i+1, siginv inv M(i,i+1))
        M := changeBase(M, E, sig, der)
        B := B*E
        Binv := inv(E)*Binv
        for j in 1..N repeat
          if j ^= i+1 then
            E := addMatrix(N, i+1, j, siginv(-M(i,j)))
            M := changeBase(M, E, sig, der)
            B := B*E
            Binv := inv(E)*Binv
        i := i + 1
      else
        -- apply lemma 6
        for j in i..2 by -1 repeat
          for k in (i+1)..N repeat
            E := addMatrix(N, k, j-1, M(k,j))
            M := changeBase(M, E, sig, der)
            B := B*E
            Binv := inv(E)*Binv
```

```
        j := i + 1
        while j <= N and M(j,1) = 0 repeat  j := j + 1
        if j <= N then
          -- expand companionblock by lemma 8
          E := permutationMatrix(N, 1, j)
          M := changeBase(M, E, sig, der)
          B := B*E
          Binv := E*Binv
          -- start again to establish rational form
          i := 1
        else
          -- split a direct factor
          recOfMatrices :=
            normalForm(subMatrix(M, i+1, N, i+1, N), sig, der)
          setsubMatrix!(M, i+1, i+1, recOfMatrices.R)
          E := diagonalMatrix [1 for k in 1..N]
          setsubMatrix!(E, i+1, i+1, recOfMatrices.A)
          B := B*E
          setsubMatrix!(E, i+1, i+1, recOfMatrices.Ainv)
          Binv := E*Binv
          -- M in blockdiagonalform, stop program
          i := N
      [M, B, Binv]

    mulMatrix(N, i, a) ==
      M : Matrix K := diagonalMatrix [1 for j in 1..N]
      M(i, i) := a
      M

    addMatrix(N, i, k, a) ==
      A : Matrix K := diagonalMatrix [1 for j in 1..N]
      A(i, k) := a
      A

    permutationMatrix(N, i, k) ==
      P : Matrix K := diagonalMatrix [1 for j in 1..N]
      P(i, i) := P(k, k) := 0
      P(i, k) := P(k, i) := 1
      P
```

———-

— PSEUDLIN.dotabb —

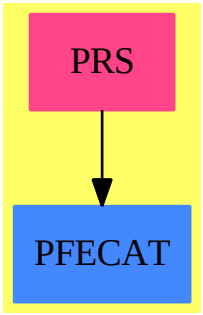
```
"PSEUDLIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSEUDLIN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"PSEUDLIN" -> "IVECTOR"
```

———-

17.69 package PRS PseudoRemainderSequence

The package constructor **PseudoRemainderSequence** provides efficient algorithms by Lionel Ducos (University of Poitiers, France) for computing sub-resultants. This leads to a speed up in many places in Axiom where sub-resultants are computed (polynomial system solving, algebraic factorization, integration).

17.69.1 PseudoRemainderSequence (PRS)



Exports:

discriminant	Lazard
Lazard2	chainSubResultants
degreeSubResultant	degreeSubResultantEuclidean
discriminantEuclidean	divide
exquo	gcd
indiceSubResultant	indiceSubResultantEuclidean
lastSubResultant	lastSubResultantEuclidean
nextsousResultant2	pseudoDivide
resultant	resultantEuclidean
resultantEuclideannaif	resultantnaif
resultantReduit	resultantReduitEuclidean
schema	semiDegreeSubResultantEuclidean
semiDiscriminantEuclidean	semiIndiceSubResultantEuclidean
semiLastSubResultantEuclidean	semiResultantEuclidean1
semiResultantEuclidean2	semiResultantEuclideannaif
semiResultantReduitEuclidean	semiSubResultantGcdEuclidean1
semiSubResultantGcdEuclidean2	subResultantGcd
subResultantGcdEuclidean	?*?

— **package PRS PseudoRemainderSequence** —

```
)abbrev package PRS PseudoRemainderSequence
++ Author: Ducos Lionel (Lionel.Ducos@mathlabo.univ-poitiers.fr)
++ Date Created: january 1995
++ Date Last Updated: 5 february 1999
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++ Lionel Ducos "Optimizations of the subresultant algorithm"
++ to appear in the Journal of Pure and Applied Algebra.
++ Description:
++ This package contains some functions: discriminant, resultant,
++ subResultantGcd, chainSubResultants, degreeSubResultant, lastSubResultant,
++ resultantEuclidean, subResultantGcdEuclidean, semiSubResultantGcdEuclidean1,
++ semiSubResultantGcdEuclidean2\br
++ These procedures come from improvements of the subresultants algorithm.

PseudoRemainderSequence(R, polR) : Specification == Implementation where
  R : IntegralDomain
  polR : UnivariatePolynomialCategory(R)
  NNI ==> NonNegativeInteger
  LC  ==> leadingCoefficient

  Specification == with
       resultant : (polR, polR) -> R
         ++ \axiom{resultant(P, Q)} returns the resultant
         ++ of \axiom{P} and \axiom{Q}

       resultantEuclidean : (polR, polR) ->
                            Record(coef1 : polR, coef2 : polR, resultant : R)
         ++ \axiom{resultantEuclidean(P,Q)} carries out the equality
         ++ \axiom{coef1*P + coef2*Q = resultant(P,Q)}

       semiResultantEuclidean2 : (polR, polR) ->
                            Record(coef2 : polR, resultant : R)
         ++ \axiom{semiResultantEuclidean2(P,Q)} carries out the equality
         ++ \axiom{...P + coef2*Q = resultant(P,Q)}.
         ++ Warning. \axiom{degree(P) >= degree(Q)}.

       semiResultantEuclidean1 : (polR, polR) ->
                            Record(coef1 : polR, resultant : R)
         ++ \axiom{semiResultantEuclidean1(P,Q)} carries out the equality
         ++ \axiom{coef1.P + ? Q = resultant(P,Q)}.

       indiceSubResultant : (polR, polR, NNI) -> polR
         ++ \axiom{indiceSubResultant(P, Q, i)} returns
         ++ the subresultant of indice \axiom{i}
```

```
indiceSubResultantEuclidean : (polR, polR, NNI) ->
                Record(coef1 : polR, coef2 : polR, subResultant : polR)
  ++ \axiom{indiceSubResultant(P, Q, i)} returns
  ++ the subresultant \axiom{S_i(P,Q)} and carries out the equality
  ++ \axiom{coef1*P + coef2*Q = S_i(P,Q)}

semiIndiceSubResultantEuclidean : (polR, polR, NNI) ->
                   Record(coef2 : polR, subResultant : polR)
  ++ \axiom{semiIndiceSubResultantEuclidean(P, Q, i)} returns
  ++ the subresultant \axiom{S_i(P,Q)} and carries out the equality
  ++ \axiom{...P + coef2*Q = S_i(P,Q)}
  ++ Warning. \axiom{degree(P) >= degree(Q)}.

degreeSubResultant : (polR, polR, NNI) -> polR
  ++ \axiom{degreeSubResultant(P, Q, d)} computes
  ++ a subresultant of degree \axiom{d}.

degreeSubResultantEuclidean : (polR, polR, NNI) ->
                Record(coef1 : polR, coef2 : polR, subResultant : polR)
  ++ \axiom{indiceSubResultant(P, Q, i)} returns
  ++ a subresultant \axiom{S} of degree \axiom{d}
  ++ and carries out the equality \axiom{coef1*P + coef2*Q = S_i}.

semiDegreeSubResultantEuclidean : (polR, polR, NNI) ->
                   Record(coef2 : polR, subResultant : polR)
  ++ \axiom{indiceSubResultant(P, Q, i)} returns
  ++ a subresultant \axiom{S} of degree \axiom{d}
  ++ and carries out the equality \axiom{...P + coef2*Q = S_i}.
  ++ Warning. \axiom{degree(P) >= degree(Q)}.

lastSubResultant : (polR, polR) -> polR
  ++ \axiom{lastSubResultant(P, Q)} computes
  ++ the last non zero subresultant of \axiom{P} and \axiom{Q}

lastSubResultantEuclidean : (polR, polR) ->
                Record(coef1 : polR, coef2 : polR, subResultant : polR)
  ++ \axiom{lastSubResultantEuclidean(P, Q)} computes
  ++ the last non zero subresultant \axiom{S}
  ++ and carries out the equality \axiom{coef1*P + coef2*Q = S}.

semiLastSubResultantEuclidean : (polR, polR) ->
                Record(coef2 : polR, subResultant : polR)
  ++ \axiom{semiLastSubResultantEuclidean(P, Q)} computes
  ++ the last non zero subresultant \axiom{S}
  ++ and carries out the equality \axiom{...P + coef2*Q = S}.
  ++ Warning. \axiom{degree(P) >= degree(Q)}.

subResultantGcd : (polR, polR) -> polR
  ++ \axiom{subResultantGcd(P, Q)} returns the gcd
  ++ of two primitive polynomials \axiom{P} and \axiom{Q}.
```

```
subResultantGcdEuclidean : (polR, polR)
             -> Record(coef1 : polR, coef2 : polR, gcd : polR)
  ++ \axiom{subResultantGcdEuclidean(P,Q)} carries out the equality
  ++ \axiom{coef1*P + coef2*Q = +/- S_i(P,Q)}
  ++ where the degree (not the indice)
  ++ of the subresultant \axiom{S_i(P,Q)} is the smaller as possible.

semiSubResultantGcdEuclidean2 : (polR, polR)
                           -> Record(coef2 : polR, gcd : polR)
  ++ \axiom{semiSubResultantGcdEuclidean2(P,Q)} carries out the equality
  ++ \axiom{...P + coef2*Q = +/- S_i(P,Q)}
  ++ where the degree (not the indice)
  ++ of the subresultant \axiom{S_i(P,Q)} is the smaller as possible.
  ++ Warning. \axiom{degree(P) >= degree(Q)}.

semiSubResultantGcdEuclidean1: (polR, polR)->Record(coef1: polR, gcd: polR)
  ++ \axiom{semiSubResultantGcdEuclidean1(P,Q)} carries out the equality
  ++ \axiom{coef1*P + ? Q = +/- S_i(P,Q)}
  ++ where the degree (not the indice)
  ++ of the subresultant \axiom{S_i(P,Q)} is the smaller as possible.

discriminant : polR -> R
  ++ \axiom{discriminant(P, Q)} returns the discriminant
  ++ of \axiom{P} and \axiom{Q}.

discriminantEuclidean : polR ->
                   Record(coef1 : polR, coef2 : polR, discriminant : R)
  ++ \axiom{discriminantEuclidean(P)} carries out the equality
  ++ \axiom{coef1 * P + coef2 * D(P) = discriminant(P)}.

semiDiscriminantEuclidean : polR ->
                   Record(coef2 : polR, discriminant : R)
  ++ \axiom{discriminantEuclidean(P)} carries out the equality
  ++ \axiom{...P + coef2 * D(P) = discriminant(P)}.
  ++ Warning. \axiom{degree(P) >= degree(Q)}.

chainSubResultants : (polR, polR) -> List(polR)
  ++ \axiom{chainSubResultants(P, Q)} computes the list
  ++ of non zero subresultants of \axiom{P} and \axiom{Q}.

schema : (polR, polR) -> List(NNI)
  ++ \axiom{schema(P,Q)} returns the list of degrees of
  ++ non zero subresultants of \axiom{P} and \axiom{Q}.

if R has GcdDomain then
  resultantReduit : (polR, polR) -> R
  ++ \axiom{resultantReduit(P,Q)} returns the "reduce resultant"
  ++ of \axiom{P} and \axiom{Q}.
```

```
  resultantReduitEuclidean : (polR, polR) ->
                Record(coef1 : polR, coef2 : polR, resultantReduit : R)
  ++ \axiom{resultantReduitEuclidean(P,Q)} returns
  ++ the "reduce resultant" and carries out the equality
  ++ \axiom{coef1*P + coef2*Q = resultantReduit(P,Q)}.

  semiResultantReduitEuclidean : (polR, polR) ->
                Record(coef2 : polR, resultantReduit : R)
  ++ \axiom{semiResultantReduitEuclidean(P,Q)} returns
  ++ the "reduce resultant" and carries out the equality
  ++ \axiom{...P + coef2*Q = resultantReduit(P,Q)}.

  gcd : (polR, polR) -> polR
  ++ \axiom{gcd(P, Q)} returns the gcd of \axiom{P} and \axiom{Q}.

-- sub-routines exported for convenience ----------------------------

"*" : (R, Vector(polR)) -> Vector(polR)
  ++ \axiom{r * v} computes the product of \axiom{r} and \axiom{v}

"exquo" : (Vector(polR), R) -> Vector(polR)
  ++ \axiom{v exquo r} computes
  ++ the exact quotient of \axiom{v} by \axiom{r}

pseudoDivide : (polR, polR) ->
                        Record(coef:R, quotient:polR, remainder:polR)
  ++ \axiom{pseudoDivide(P,Q)} computes the pseudoDivide
  ++ of \axiom{P} by \axiom{Q}.

divide : (polR, polR) -> Record(quotient : polR, remainder : polR)
  ++ \axiom{divide(F,G)} computes quotient and rest
  ++ of the exact euclidean division of \axiom{F} by \axiom{G}.

Lazard : (R, R, NNI) -> R
  ++ \axiom{Lazard(x, y, n)} computes \axiom{x**n/y**(n-1)}

Lazard2 : (polR, R, R, NNI) -> polR
  ++ \axiom{Lazard2(F, x, y, n)} computes  \axiom{(x/y)**(n-1) * F}

next_sousResultant2 : (polR, polR, polR, R) -> polR
  ++ \axiom{nextsousResultant2(P, Q, Z, s)} returns
  ++ the subresultant \axiom{S_{e-1}} where
  ++ \axiom{P ~ S_d,  Q = S_{d-1},  Z = S_e,  s = lc(S_d)}

resultant_naif : (polR, polR) -> R
  ++ \axiom{resultantEuclidean_naif(P,Q)} returns
  ++ the resultant of \axiom{P} and \axiom{Q} computed
  ++ by means of the naive algorithm.

resultantEuclidean_naif : (polR, polR) ->
```

```
                        Record(coef1 : polR, coef2 : polR, resultant : R)
      ++ \axiom{resultantEuclidean_naif(P,Q)} returns
      ++ the extended resultant of \axiom{P} and \axiom{Q} computed
      ++ by means of the naive algorithm.

    semiResultantEuclidean_naif : (polR, polR) ->
                        Record(coef2 : polR, resultant : R)
      ++ \axiom{resultantEuclidean_naif(P,Q)} returns
      ++ the semi-extended resultant of \axiom{P} and \axiom{Q} computed
      ++ by means of the naive algorithm.

Implementation == add
  X : polR := monomial(1$R,1)

  r : R * v : Vector(polR) == r::polR * v
            -- the instruction  map(r * #1, v) is slower !?

  v : Vector(polR) exquo r : R ==
    map((p1:polR):polR +-> (p1 exquo r)::polR, v)

  pseudoDivide(P : polR, Q : polR) :
                              Record(coef:R,quotient:polR,remainder:polR) ==
  -- computes the pseudoDivide of P by Q
     zero?(Q) => error("PseudoDivide$PRS : division by 0")
     zero?(P) => construct(1, 0, P)
     lcQ : R := LC(Q)
     (degP, degQ) := (degree(P), degree(Q))
     degP < degQ => construct(1, 0, P)
     Q := reductum(Q)
     i : NNI := (degP - degQ + 1)::NNI
     co : R := lcQ**i
     quot : polR := 0$polR
     while (delta : Integer := degree(P) - degQ) >= 0 repeat
       i := (i - 1)::NNI
       mon := monomial(LC(P), delta::NNI)$polR
       quot := quot + lcQ**i * mon
       P := lcQ * reductum(P) - mon * Q
     P := lcQ**i * P
     return construct(co, quot, P)

  divide(F : polR, G : polR) : Record(quotient : polR, remainder : polR)==
  -- computes quotient and rest of the exact euclidean division of F by G
       lcG : R := LC(G)
       degG : NNI := degree(G)
       zero?(degG) => ( F := (F exquo lcG)::polR; return construct(F, 0))
       G : polR := reductum(G)
       quot : polR := 0
       while (delta := degree(F) - degG) >= 0 repeat
          mon : polR := monomial((LC(F) exquo lcG)::R, delta::NNI)
          quot := quot + mon
```

```
        F := reductum(F) - mon * G
      return construct(quot, F)

resultant_naif(P : polR, Q : polR) : R ==
-- valid over a field
   a : R := 1
   repeat
      zero?(Q) => return 0
      (degP, degQ) := (degree(P), degree(Q))
      if odd?(degP) and odd?(degQ) then a := - a
      zero?(degQ) => return (a * LC(Q)**degP)
      U : polR := divide(P, Q).remainder
      a := a * LC(Q)**(degP - degree(U))::NNI
      (P, Q) := (Q, U)

resultantEuclidean_naif(P : polR, Q : polR) :
                  Record(coef1 : polR, coef2 : polR, resultant : R) ==
-- valid over a field.
   a : R := 1
   old_cf1 : polR := 1 ; cf1 : polR := 0
   old_cf2 : polR := 0 ; cf2 : polR := 1
   repeat
      zero?(Q) => construct(0::polR, 0::polR, 0::R)
      (degP, degQ) := (degree(P), degree(Q))
      if odd?(degP) and odd?(degQ) then a := -a
      if zero?(degQ) then
         a := a * LC(Q)**(degP-1)::NNI
         return construct(a*cf1, a*cf2, a*LC(Q))
      divid := divide(P,Q)
      a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
      (P, Q) := (Q, divid.remainder)
      (old_cf1, old_cf2, cf1, cf2) := (cf1, cf2,
            old_cf1 - divid.quotient * cf1, old_cf2 - divid.quotient * cf2)

semiResultantEuclidean_naif(P : polR, Q : polR) :
                  Record(coef2 : polR, resultant : R) ==
-- valid over a field
   a : R := 1
   old_cf2 : polR := 0 ; cf2 : polR := 1
   repeat
      zero?(Q) => construct(0::polR, 0::R)
      (degP, degQ) := (degree(P), degree(Q))
      if odd?(degP) and odd?(degQ) then a := -a
      if zero?(degQ) then
         a := a * LC(Q)**(degP-1)::NNI
         return construct(a*cf2, a*LC(Q))
      divid := divide(P,Q)
      a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
      (P, Q) := (Q, divid.remainder)
      (old_cf2, cf2) := (cf2, old_cf2 - divid.quotient * cf2)
```

```
    Lazard(x : R, y : R, n : NNI) : R ==
       zero?(n) => error("Lazard$PRS : n = 0")
--       one?(n) => x
       (n = 1) => x
       a : NNI := 1
       while n >= (b := 2*a) repeat a := b
       c : R := x
       n := (n - a)::NNI
       repeat                    --  c = x**i / y**(i-1),  i=n_0 quo a,  a=2**?
--          one?(a) => return c
          (a = 1) => return c
          a := a quo 2
          c := ((c * c) exquo y)::R
          if n >= a then ( c := ((c * x) exquo y)::R ; n := (n - a)::NNI )

    Lazard2(F : polR, x : R, y : R, n : NNI) : polR ==
       zero?(n) => error("Lazard2$PRS : n = 0")
--       one?(n) => F
       (n = 1) => F
       x := Lazard(x, y, (n-1)::NNI)
       return ((x * F) exquo y)::polR

    Lazard3(V : Vector(polR), x : R, y : R, n : NNI) : Vector(polR) ==
    -- computes x**(n-1) * V / y**(n-1)
       zero?(n) => error("Lazard2$prs : n = 0")
--       one?(n) => V
       (n = 1) => V
       x := Lazard(x, y, (n-1)::NNI)
       return ((x * V) exquo y)

    next_sousResultant2(P : polR, Q : polR, Z : polR, s : R) : polR ==
       (lcP, c, se) := (LC(P), LC(Q), LC(Z))
       (d, e) := (degree(P), degree(Q))
       (P, Q, H) := (reductum(P), reductum(Q), - reductum(Z))
       A : polR := coefficient(P, e) * H
       for i in e+1..d-1 repeat
          H := if degree(H) = e-1 then
                  X * reductum(H) - ((LC(H) * Q) exquo c)::polR
               else
                  X * H
          -- H = s_e * X^i mod S_d-1
          A := coefficient(P, i) * H + A
       while degree(P) >= e repeat P := reductum(P)
       A := A + se * P            --  A = s_e * reductum(P_0)       mod S_d-1
       A := (A exquo lcP)::polR   --  A = s_e * reductum(S_d) / s_d mod S_d-1
       A := if degree(H) = e-1 then
               c * (X * reductum(H) + A) - LC(H) * Q
            else
               c * (X * H + A)
```

```
      A := (A exquo s)::polR                    -- A = +/- S_e-1
      return (if odd?(d-e) then A else - A)

   next_sousResultant3(VP : Vector(polR), VQ : Vector(polR), s : R, ss : R) :
                                                    Vector(polR) ==
   -- P ~ S_d,  Q = S_d-1,  s = lc(S_d),  ss = lc(S_e)
      (P, Q) := (VP.1, VQ.1)
      (lcP, c) := (LC(P), LC(Q))
      e : NNI := degree(Q)
--      if one?(delta := degree(P) - e) then                  -- algo_new
      if ((delta := degree(P) - e) = 1) then                  -- algo_new
        VP := c * VP - coefficient(P, e) * VQ
        VP := VP exquo lcP
        VP := c * (VP - X * VQ) + coefficient(Q, (e-1)::NNI) * VQ
        VP := VP exquo s
      else                                    -- algorithm of Lickteig - Roy
        (r, rr) := (s * lcP, ss * c)
        divid := divide(rr * P, Q)
        VP.1 := (divid.remainder exquo r)::polR
        for i in 2..#VP repeat
          VP.i := rr * VP.i - VQ.i * divid.quotient
          VP.i := (VP.i exquo r)::polR
      return (if odd?(delta) then VP else - VP)

   algo_new(P : polR, Q : polR) : R ==
      delta : NNI := (degree(P) - degree(Q))::NNI
      s : R := LC(Q)**delta
      (P, Q) := (Q, pseudoRemainder(P, -Q))
      repeat
         -- P = S_c-1 (except the first turn : P ~ S_c-1),
         -- Q = S_d-1,  s = lc(S_d)
         zero?(Q) => return 0
         delta := (degree(P) - degree(Q))::NNI
         Z : polR := Lazard2(Q, LC(Q), s, delta)
         -- Z = S_e ~ S_d-1
         zero?(degree(Z)) => return LC(Z)
         (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
         s := LC(Z)

   resultant(P : polR, Q : polR) : R ==
      zero?(Q) or zero?(P) => 0
      if degree(P) < degree(Q) then
         (P, Q) := (Q, P)
         if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
      zero?(degree(Q)) => LC(Q)**degree(P)
      -- degree(P) >= degree(Q) > 0
      R has Finite => resultant_naif(P, Q)
      return algo_new(P, Q)

   subResultantEuclidean(P : polR, Q : polR) :
```

```
                    Record(coef1 : polR, coef2 : polR, resultant : R) ==
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 0::polR, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d)
      --  S_{c-1} = VP.2 P_0 + VP.3 Q_0,  S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) => return construct(0::polR, 0::polR, 0::R)
      e : NNI := degree(Q)
      delta : NNI := (degree(P) - e)::NNI
      if zero?(e) then
         l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
         return construct(l.2, l.3, LC(l.1))
      ss : R := Lazard(LC(Q), s, delta)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

resultantEuclidean(P : polR, Q : polR) :
                  Record(coef1 : polR, coef2 : polR, resultant : R) ==
   zero?(P) or zero?(Q) => construct(0::polR, 0::polR, 0::R)
   if degree(P) < degree(Q) then
      e : Integer := if odd?(degree(P)) and odd?(degree(Q)) then -1 else 1
      l := resultantEuclidean(Q, e * P)
      return construct(e * l.coef2, l.coef1, l.resultant)
   if zero?(degree(Q)) then
      degP : NNI := degree(P)
      zero?(degP) => error("resultantEuclidean$PRS : constant polynomials")
      s : R := LC(Q)**(degP-1)::NNI
      return construct(0::polR, s::polR, s * LC(Q))
   R has Finite => resultantEuclidean_naif(P, Q)
   return subResultantEuclidean(P,Q)

semiSubResultantEuclidean(P : polR, Q : polR) :
                  Record(coef2 : polR, resultant : R) ==
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d)
      --  S_{c-1} = ...P_0 + VP.3 Q_0,  S_{d-1} = ...P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) => return construct(0::polR, 0::R)
      e : NNI := degree(Q)
      delta : NNI := (degree(P) - e)::NNI
      if zero?(e) then
         l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
         return construct(l.2, LC(l.1))
```

```
      ss : R := Lazard(LC(Q), s, delta)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

  semiResultantEuclidean2(P : polR, Q : polR) :
                    Record(coef2 : polR, resultant : R) ==
    zero?(P) or zero?(Q) => construct(0::polR, 0::R)
    degree(P) < degree(Q) => error("semiResultantEuclidean2 : bad degrees")
    if zero?(degree(Q)) then
      degP : NNI := degree(P)
      zero?(degP) => error("semiResultantEuclidean2 : constant polynomials")
      s : R := LC(Q)**(degP-1)::NNI
      return construct(s::polR, s * LC(Q))
    R has Finite => semiResultantEuclidean_naif(P, Q)
    return semiSubResultantEuclidean(P,Q)

  semiResultantEuclidean1(P : polR, Q : polR) :
                    Record(coef1 : polR, resultant : R) ==
    result := resultantEuclidean(P,Q)
    [result.coef1, result.resultant]

  indiceSubResultant(P : polR, Q : polR, i : NNI) : polR ==
    zero?(Q) or zero?(P) => 0
    if degree(P) < degree(Q) then
      (P, Q) := (Q, P)
      if odd?(degree(P)-i) and odd?(degree(Q)-i) then Q := - Q
    if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) => error("indiceSubResultant$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return s*Q
    i > degree(Q) => 0
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    (P, Q) := (Q, pseudoRemainder(P, -Q))
    repeat
      -- P = S_{c-1} ~ S_d , Q = S_{d-1},  s = lc(S_d),  i < d
      (degP, degQ) := (degree(P), degree(Q))
      i = degP-1 => return Q
      zero?(Q) or (i > degQ) => return 0
      Z : polR := Lazard2(Q, LC(Q), s, (degP - degQ)::NNI)
      --  Z = S_e ~ S_d-1
      i = degQ => return Z
      (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
      s := LC(Z)

  indiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
                 Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
    if degree(P) < degree(Q) then
      e := if odd?(degree(P)-i) and odd?(degree(Q)-i) then -1 else 1
```

```
      l := indiceSubResultantEuclidean(Q, e * P, i)
      return construct(e * l.coef2, l.coef1, l.subResultant)
   if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) =>
                  error("indiceSubResultantEuclidean$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return construct(0::polR, s::polR, s * Q)
   i > degree(Q) => construct(0::polR, 0::polR, 0::polR)
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 0::polR, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d),  i < d
      --  S_{c-1} = VP.2 P_0 + VP.3 Q_0,  S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) => return construct(0::polR, 0::polR, 0::polR)
      (degP, degQ) := (degree(P), degree(Q))
      i = degP-1 => return construct(VQ.2, VQ.3, VQ.1)
      (i > degQ) => return construct(0::polR, 0::polR, 0::polR)
      VZ := Lazard3(VQ, LC(Q), s, (degP - degQ)::NNI)
      i = degQ => return construct(VZ.2, VZ.3, VZ.1)
      ss : R := LC(VZ.1)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

semiIndiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
               Record(coef2 : polR, subResultant : polR) ==
   zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
   degree(P) < degree(Q) =>
              error("semiIndiceSubResultantEuclidean$PRS : bad degrees")
   if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) =>
              error("semiIndiceSubResultantEuclidean$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return construct(s::polR, s * Q)
   i > degree(Q) => construct(0::polR, 0::polR)
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s = lc(S_d),  i < d
      --  S_{c-1} = ...P_0 + VP.2 Q_0,  S_{d-1} = ...P_0 + ...Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) => return construct(0::polR, 0::polR)
      (degP, degQ) := (degree(P), degree(Q))
      i = degP-1 => return construct(VQ.2, VQ.1)
```

```
      (i > degQ) => return construct(0::polR, 0::polR)
      VZ := Lazard3(VQ, LC(Q), s, (degP - degQ)::NNI)
      i = degQ => return construct(VZ.2, VZ.1)
      ss : R := LC(VZ.1)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

degreeSubResultant(P : polR, Q : polR, i : NNI) : polR ==
   zero?(Q) or zero?(P) => 0
   if degree(P) < degree(Q) then (P, Q) := (Q, P)
   if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) => error("degreeSubResultant$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return s*Q
   i > degree(Q) => 0
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   (P, Q) := (Q, pseudoRemainder(P, -Q))
   repeat
      -- P = S_{c-1},  Q = S_{d-1},  s = lc(S_d)
      zero?(Q) or (i > degree(Q)) => return 0
      i = degree(Q) => return Q
      Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
      --  Z = S_e ~ S_d-1
      (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
      s := LC(Z)

degreeSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
                Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
   zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
   if degree(P) < degree(Q) then
      l := degreeSubResultantEuclidean(Q, P, i)
      return construct(l.coef2, l.coef1, l.subResultant)
   if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) =>
                 error("degreeSubResultantEuclidean$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return construct(0::polR, s::polR, s * Q)
   i > degree(Q) => construct(0::polR, 0::polR, 0::polR)
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 0::polR, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d)
      --  S_{c-1} = ...P_0 + VP.3 Q_0,  S_{d-1} = ...P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) or (i > degree(Q)) =>
           return construct(0::polR, 0::polR, 0::polR)
```

```
      i = degree(Q) => return construct(VQ.2, VQ.3, VQ.1)
      ss : R := Lazard(LC(Q), s, (degree(P)-degree(Q))::NNI)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

  semiDegreeSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
                  Record(coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
    degree(P) < degree(Q) =>
              error("semiDegreeSubResultantEuclidean$PRS : bad degrees")
    if i = degree(Q) then
      delta : NNI := (degree(P)-degree(Q))::NNI
      zero?(delta) =>
              error("semiDegreeSubResultantEuclidean$PRS : bad degrees")
      s : R := LC(Q)**(delta-1)::NNI
      return construct(s::polR, s * Q)
    i > degree(Q) => construct(0::polR, 0::polR)
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 1::polR]
    pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
    repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d)
      --  S_{c-1} = ...P_0 + VP.3 Q_0,  S_{d-1} = ...P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) or (i > degree(Q)) =>
           return construct(0::polR, 0::polR)
      i = degree(Q) => return construct(VQ.2, VQ.1)
      ss : R := Lazard(LC(Q), s, (degree(P)-degree(Q))::NNI)
      (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

  lastSubResultant(P : polR, Q : polR) : polR ==
    zero?(Q) or zero?(P) => 0
    if degree(P) < degree(Q) then (P, Q) := (Q, P)
    zero?(degree(Q)) => (LC(Q)**degree(P))::polR
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    (P, Q) := (Q, pseudoRemainder(P, -Q))
    Z : polR := P
    repeat
      -- Z = S_d  (except the first turn : Z = P)
      -- P = S_{c-1} ~ S_d,  Q = S_{d-1},  s = lc(S_d)
      zero?(Q) => return Z
      Z := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
      -- Z = S_e ~ S_{d-1}
      zero?(degree(Z)) => return Z
      (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
      s := LC(Z)

  lastSubResultantEuclidean(P : polR, Q : polR) :
```

```
                Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
    if degree(P) < degree(Q) then
       l := lastSubResultantEuclidean(Q, P)
       return construct(l.coef2, l.coef1, l.subResultant)
    if zero?(degree(Q)) then
       degP : NNI := degree(P)
       zero?(degP) =>
           error("lastSubResultantEuclidean$PRS : constant polynomials")
       s : R := LC(Q)**(degP-1)::NNI
       return construct(0::polR, s::polR, s * Q)
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 0::polR, 1::polR]
    pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
    VZ : Vector(polR) := copy(VP)
    repeat
       --  VZ.1 = S_d,  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s = lc(S_d)
       --  S_{c-1} = VP.2 P_0 + VP.3 Q_0
       --  S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
       --  S_d     = VZ.2 P_0 + VZ.3 Q_0
       (Q, Z) := (VQ.1, VZ.1)
       zero?(Q) => return construct(VZ.2, VZ.3, VZ.1)
       VZ := Lazard3(VQ, LC(Q), s, (degree(Z) - degree(Q))::NNI)
       zero?(degree(Q)) => return construct(VZ.2, VZ.3, VZ.1)
       ss : R := LC(VZ.1)
       (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
       s := ss

 semiLastSubResultantEuclidean(P : polR, Q : polR) :
                Record(coef2 : polR, subResultant : polR) ==
    zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
    degree(P) < degree(Q) =>
           error("semiLastSubResultantEuclidean$PRS : bad degrees")
    if zero?(degree(Q)) then
       degP : NNI := degree(P)
       zero?(degP) =>
           error("semiLastSubResultantEuclidean$PRS : constant polynomials")
       s : R := LC(Q)**(degP-1)::NNI
       return construct(s::polR, s * Q)
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 1::polR]
    pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
    VZ : Vector(polR) := copy(VP)
    repeat
       --  VZ.1 = S_d,  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s = lc(S_d)
       --  S_{c-1} = ... P_0 + VP.2 Q_0
       --  S_{d-1} = ... P_0 + VQ.2 Q_0
       --  S_d     = ... P_0 + VZ.2 Q_0
```

```
        (Q, Z) := (VQ.1, VZ.1)
        zero?(Q) => return construct(VZ.2, VZ.1)
        VZ := Lazard3(VQ, LC(Q), s, (degree(Z) - degree(Q))::NNI)
        zero?(degree(Q)) => return construct(VZ.2, VZ.1)
        ss : R := LC(VZ.1)
        (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
        s := ss

    chainSubResultants(P : polR, Q : polR) : List(polR) ==
       zero?(Q) or zero?(P) => []
       if degree(P) < degree(Q) then
          (P, Q) := (Q, P)
          if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
       L : List(polR) := []
       zero?(degree(Q)) => L
       L := [Q]
       s : R := LC(Q)**(degree(P) - degree(Q))::NNI
       (P, Q) := (Q, pseudoRemainder(P, -Q))
       repeat
          -- P = S_{c-1},  Q = S_{d-1},  s = lc(S_d)
          -- L = [S_d,....,S_{q-1}]
          zero?(Q) => return L
          L := concat(Q, L)
          -- L = [S_{d-1},....,S_{q-1}]
          delta : NNI := (degree(P) - degree(Q))::NNI
          Z : polR := Lazard2(Q, LC(Q), s, delta)            -- Z = S_e ~ S_d-1
          if delta > 1 then L := concat(Z, L)
          -- L = [S_e,....,S_{q-1}]
          zero?(degree(Z)) => return L
          (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
          s := LC(Z)

    schema(P : polR, Q : polR) : List(NNI) ==
       zero?(Q) or zero?(P) => []
       if degree(P) < degree(Q) then (P, Q) := (Q, P)
       zero?(degree(Q)) => [0]
       L : List(NNI) := []
       s : R := LC(Q)**(degree(P) - degree(Q))::NNI
       (P, Q) := (Q, pseudoRemainder(P, Q))
       repeat
          -- P = S_{c-1} ~ S_d,  Q = S_{d-1},  s = lc(S_d)
          zero?(Q) => return L
          e : NNI := degree(Q)
          L := concat(e, L)
          delta : NNI := (degree(P) - e)::NNI
          Z : polR := Lazard2(Q, LC(Q), s, delta)            -- Z = S_e ~ S_d-1
          if delta > 1 then L := concat(e, L)
          zero?(e) => return L
          (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
          s := LC(Z)
```

```
subResultantGcd(P : polR, Q : polR) : polR ==
   zero?(P) and zero?(Q) => 0
   zero?(P) => Q
   zero?(Q) => P
   if degree(P) < degree(Q) then (P, Q) := (Q, P)
   zero?(degree(Q)) => 1$polR
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   (P, Q) := (Q, pseudoRemainder(P, -Q))
   repeat
      -- P = S_{c-1},  Q = S_{d-1},  s = lc(S_d)
      zero?(Q) => return P
      zero?(degree(Q)) => return 1$polR
      Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
      -- Z = S_e ~ S_d-1
      (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
      s := LC(Z)

subResultantGcdEuclidean(P : polR, Q : polR) :
               Record(coef1 : polR, coef2 : polR, gcd : polR) ==
   zero?(P) and zero?(Q) => construct(0::polR, 0::polR, 0::polR)
   zero?(P) => construct(0::polR, 1::polR, Q)
   zero?(Q) => construct(1::polR, 0::polR, P)
   if degree(P) < degree(Q) then
      l := subResultantGcdEuclidean(Q, P)
      return construct(l.coef2, l.coef1, l.gcd)
   zero?(degree(Q)) => construct(0::polR, 1::polR, Q)
   s : R := LC(Q)**(degree(P) - degree(Q))::NNI
   VP : Vector(polR) := [Q, 0::polR, 1::polR]
   pdiv := pseudoDivide(P, -Q)
   VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
   repeat
      --  VP.1 = S_{c-1},  VQ.1 = S_{d-1},  s=lc(S_d)
      --  S_{c-1} = VP.2 P_0 + VP.3 Q_0,  S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
      (P, Q) := (VP.1, VQ.1)
      zero?(Q) => return construct(VP.2, VP.3, P)
      e : NNI := degree(Q)
      zero?(e) => return construct(VQ.2, VQ.3, Q)
      ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
      (VP,VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
      s := ss

semiSubResultantGcdEuclidean2(P : polR, Q : polR) :
                              Record(coef2 : polR, gcd : polR) ==
   zero?(P) and zero?(Q) => construct(0::polR, 0::polR)
   zero?(P) => construct(1::polR, Q)
   zero?(Q) => construct(0::polR, P)
   degree(P) < degree(Q) =>
                  error("semiSubResultantGcdEuclidean2$PRS : bad degrees")
   zero?(degree(Q)) => construct(1::polR, Q)
```

```
      s : R := LC(Q)**(degree(P) - degree(Q))::NNI
      VP : Vector(polR) := [Q, 1::polR]
      pdiv := pseudoDivide(P, -Q)
      VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
      repeat
         --  P=S_{c-1},  Q=S_{d-1},  s=lc(S_d)
         --  S_{c-1} = ? P_0 + old_cf2 Q_0,  S_{d-1} = ? P_0 + cf2 Q_0
         (P, Q) := (VP.1, VQ.1)
         zero?(Q) => return construct(VP.2, P)
         e : NNI := degree(Q)
         zero?(e) => return construct(VQ.2, Q)
         ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
         (VP,VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
         s := ss

   semiSubResultantGcdEuclidean1(P : polR, Q : polR) :
                     Record(coef1 : polR, gcd : polR) ==
      result := subResultantGcdEuclidean(P,Q)
      [result.coef1, result.gcd]

   discriminant(P : polR) : R ==
      d : Integer := degree(P)
      zero?(d) => error "cannot take discriminant of constants"
      a : Integer := (d * (d-1)) quo 2
      a := (-1)**a::NonNegativeInteger
      dP : polR := differentiate P
      r : R := resultant(P, dP)
      d := d - degree(dP) - 1
      return (if zero?(d) then a * (r exquo LC(P))::R
              else a * r * LC(P)**(d-1)::NNI)

   discriminantEuclidean(P : polR) :
                     Record(coef1 : polR, coef2 : polR, discriminant : R) ==
      d : Integer := degree(P)
      zero?(d) => error "cannot take discriminant of constants"
      a : Integer := (d * (d-1)) quo 2
      a := (-1)**a::NonNegativeInteger
      dP : polR := differentiate P
      rE := resultantEuclidean(P, dP)
      d := d - degree(dP) - 1
      if zero?(d) then
         c1 : polR := a * (rE.coef1 exquo LC(P))::polR
         c2 : polR := a * (rE.coef2 exquo LC(P))::polR
         cr : R := a * (rE.resultant exquo LC(P))::R
      else
         c1 : polR := a * rE.coef1 * LC(P)**(d-1)::NNI
         c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
         cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
      return construct(c1, c2, cr)
```

```
    semiDiscriminantEuclidean(P : polR) :
                            Record(coef2 : polR, discriminant : R) ==
       d : Integer := degree(P)
       zero?(d) => error "cannot take discriminant of constants"
       a : Integer := (d * (d-1)) quo 2
       a := (-1)**a::NonNegativeInteger
       dP : polR := differentiate P
       rE := semiResultantEuclidean2(P, dP)
       d := d - degree(dP) - 1
       if zero?(d) then
          c2 : polR := a * (rE.coef2 exquo LC(P))::polR
          cr : R := a * (rE.resultant exquo LC(P))::R
       else
          c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
          cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
       return construct(c2, cr)

    if R has GcdDomain then
       resultantReduit(P : polR, Q : polR) : R ==
          UV := subResultantGcdEuclidean(P, Q)
          UVs : polR := UV.gcd
          degree(UVs) > 0 => 0
          l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
          return (LC(UVs) exquo gcd(l))::R

       resultantReduitEuclidean(P : polR, Q : polR) :
                     Record(coef1 : polR, coef2 : polR, resultantReduit : R) ==
          UV := subResultantGcdEuclidean(P, Q)
          UVs : polR := UV.gcd
          degree(UVs) > 0 => construct(0::polR, 0::polR, 0::R)
          l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
          gl : R := gcd(l)
          c1 : polR := (UV.coef1 exquo gl)::polR
          c2 : polR := (UV.coef2 exquo gl)::polR
          rr : R := (LC(UVs) exquo gl)::R
          return construct(c1, c2, rr)

       semiResultantReduitEuclidean(P : polR, Q : polR) :
                                  Record(coef2 : polR, resultantReduit : R) ==
          UV := subResultantGcdEuclidean(P, Q)
          UVs : polR := UV.gcd
          degree(UVs) > 0 => construct(0::polR, 0::R)
          l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
          gl : R := gcd(l)
          c2 : polR := (UV.coef2 exquo gl)::polR
          rr : R := (LC(UVs) exquo gl)::R
          return construct(c2, rr)

       gcd_naif(P : polR, Q : polR) : polR ==
       -- valid over a field
```

```
            zero?(P) => (Q exquo LC(Q))::polR
            repeat
               zero?(Q) => return (P exquo LC(P))::polR
               zero?(degree(Q)) => return 1$polR
               (P, Q) := (Q, divide(P, Q).remainder)

         gcd(P : polR, Q : polR) : polR ==
            R has Finite => gcd_naif(P,Q)
            zero?(P) => Q
            zero?(Q) => P
            cP : R := content(P)
            cQ : R := content(Q)
            P := (P exquo cP)::polR
            Q := (Q exquo cQ)::polR
            G : polR := subResultantGcd(P, Q)
            return gcd(cP,cQ) * primitivePart(G)
```

———-

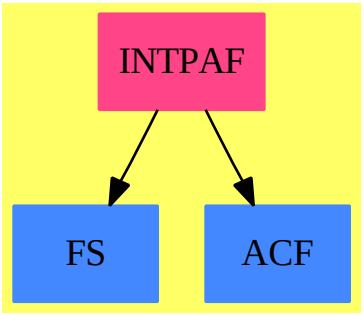
— **PRS.dotabb** —

```
"PRS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PRS" -> "PFECAT"
```

———-

17.70 package INTPAF PureAlgebraicIntegration

17.70.1 PureAlgebraicIntegration (INTPAF)



Exports:
palgLODE palgRDE palgextint palgint palglimint

— package INTPAF PureAlgebraicIntegration —

```
)abbrev package INTPAF PureAlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 27 May 1988
++ Date Last Updated: 24 June 1994
++ Description:
++ Integration of pure algebraic functions;
++ This package provides functions for integration, limited integration,
++ extended integration and the risch differential equation for
++ pure algebraic integrands;

PureAlgebraicIntegration(R, F, L): Exports == Implementation where
  R: Join(GcdDomain,RetractableTo Integer,OrderedSet, CharacteristicZero,
          LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, AlgebraicallyClosedField,
          TranscendentalFunctionCategory)
  L: SetCategory

  SY  ==> Symbol
  N   ==> NonNegativeInteger
  K   ==> Kernel F
  P   ==> SparseMultivariatePolynomial(R, K)
  UP  ==> SparseUnivariatePolynomial F
  RF  ==> Fraction UP
  UPUP==> SparseUnivariatePolynomial RF
  IR  ==> IntegrationResult F
  IR2 ==> IntegrationResultFunctions2(curve, F)
  ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
  LDALG ==> LinearOrdinaryDifferentialOperator1 curve
  RDALG ==> PureAlgebraicLODE(F, UP, UPUP, curve)
  LOG ==> Record(coeff:F, logand:F)
  REC ==> Record(particular:U1, basis:List F)
  CND ==> Record(left:UP, right:UP)
  CHV ==> Record(int:UPUP, left:UP, right:UP, den:RF, deg:N)
  U1  ==> Union(F, "failed")
  U2  ==> Union(Record(ratpart:F, coeff:F),"failed")
  U3  ==> Union(Record(mainpart:F, limitedlogs:List LOG), "failed")
  FAIL==> error "failed - cannot handle that integrand"

  Exports ==> with
    palgint   : (F, K, K)    -> IR
      ++ palgint(f, x, y) returns the integral of \spad{f(x,y)dx}
      ++ where y is an algebraic function of x.
    palgextint: (F, K, K, F) -> U2
      ++ palgextint(f, x, y, g) returns functions \spad{[h, c]} such that
      ++ \spad{dh/dx = f(x,y) - c g}, where y is an algebraic function of x;
      ++ returns "failed" if no such functions exist.
    palglimint: (F, K, K, List F) -> U3
```

```
    ++ palglimint(f, x, y, [u1,...,un]) returns functions
    ++ \spad{[h,[[ci, ui]]]} such that the ui's are among \spad{[u1,...,un]}
    ++ and \spad{d(h + sum(ci log(ui)))/dx = f(x,y)} if such functions exist,
    ++ "failed" otherwise;
    ++ y is an algebraic function of x.
  palgRDE   : (F, F, F, K, K, (F, F, SY) -> U1) -> U1
    ++ palgRDE(nfp, f, g, x, y, foo) returns a function \spad{z(x,y)}
    ++ such that \spad{dz/dx + n * df/dx z(x,y) = g(x,y)} if such a z exists,
    ++ "failed" otherwise;
    ++ y is an algebraic function of x;
    ++ \spad{foo(a, b, x)} is a function that solves
    ++ \spad{du/dx + n * da/dx u(x) = u(x)}
    ++ for an unknown \spad{u(x)} not involving y.
    ++ \spad{nfp} is \spad{n * df/dx}.
  if L has LinearOrdinaryDifferentialOperatorCategory F then
    palgLODE: (L, F, K, K, SY) -> REC
      ++ palgLODE(op, g, kx, y, x) returns the solution of \spad{op f = g}.
      ++ y is an algebraic function of x.

Implementation ==> add
  import IntegrationTools(R, F)
  import RationalIntegration(F, UP)
  import GenusZeroIntegration(R, F, L)
  import ChangeOfVariable(F, UP, UPUP)
  import IntegrationResultFunctions2(F, F)
  import IntegrationResultFunctions2(RF, F)
  import SparseUnivariatePolynomialFunctions2(F, RF)
  import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K,
                                             K, R, P, F)

  quadIfCan      : (K, K) -> Union(Record(coef:F, poly:UP), "failed")
  linearInXIfCan : (K, K) -> Union(Record(xsub:F, dxsub:RF), "failed")
  prootintegrate : (F, K, K) -> IR
  prootintegrate1: (UPUP, K, K, UPUP) -> IR
  prootextint    : (F, K, K, F) -> U2
  prootlimint    : (F, K, K, List F) -> U3
  prootRDE       : (F, F, F, K, K, (F, F, SY) -> U1) -> U1
  palgRDE1       : (F, F, K, K) -> U1
  palgLODE1      : (List F, F, K, K, SY) -> REC
  palgintegrate  : (F, K, K) -> IR
  palgext        : (F, K, K, F) -> U2
  palglim        : (F, K, K, List F) -> U3
  UPUP2F1        : (UPUP, RF, RF, K, K) -> F
  UPUP2F0        : (UPUP, K, K) -> F
  RF2UPUP        : (RF, UPUP) -> UPUP
  algaddx        : (IR, F) -> IR
  chvarIfCan     : (UPUP, RF, UP, RF) -> Union(UPUP, "failed")
  changeVarIfCan : (UPUP, RF, N) -> Union(CHV, "failed")
  rationalInt    : (UPUP, N, UP) -> IntegrationResult RF
```

```
    chv            : (UPUP, N, F, F) -> RF
    chv0           : (UPUP, N, F, F) -> F
    candidates     : UP -> List CND

    dummy := new()$SY
    dumk  := kernel(dummy)@K

    UPUP2F1(p, t, cf, kx, k) == UPUP2F0(eval(p, t, cf), kx, k)
    UPUP2F0(p, kx, k)        == multivariate(p, kx, k::F)
    chv(f, n, a, b)          == univariate(chv0(f, n, a, b), dumk)

    RF2UPUP(f, modulus) ==
      bc := extendedEuclidean(map((z1:F):RF+->z1::UP::RF, denom f), modulus,
                                      1)::Record(coef1:UPUP, coef2:UPUP)
      (map((x1:F):RF+->x1::UP::RF, numer f) * bc.coef1) rem modulus

-- returns "failed", or (xx, c) such that f(x, y)dx = f(xx, y) c dy
-- if p(x, y) = 0 is linear in x
    linearInXIfCan(x, y) ==
      a := b := 0$UP
      p := clearDenominator lift(minPoly y, x)
      while p ^= 0 repeat
        degree(q := numer leadingCoefficient p) > 1 => return "failed"
        a := a + monomial(coefficient(q, 1), d := degree p)
        b := b - monomial(coefficient(q, 0), d)
        p := reductum p
      xx:RF := b / a
      [xx(dumk::F), differentiate(xx, differentiate)]

-- return Int(f(x,y)dx) where y is an n^th root of a rational function in x
    prootintegrate(f, x, y) ==
      modulus := lift(p := minPoly y, x)
      rf      := reductum(ff := univariate(f, x, y, p))
      ((r := retractIfCan(rf)@Union(RF,"failed")) case RF) and rf ^= 0 =>
            -- in this case, ff := lc(ff) y^i + r so we integrate both terms
            -- separately to gain time
            map(f1+->f1(x::F), integrate(r::RF)) +
                 prootintegrate1(leadingMonomial ff, x, y, modulus)
      prootintegrate1(ff, x, y, modulus)

    prootintegrate1(ff, x, y, modulus) ==
      chv:CHV
      r := radPoly(modulus)::Record(radicand:RF, deg:N)
      (uu := changeVarIfCan(ff, r.radicand, r.deg)) case CHV =>
        chv := uu::CHV
        newalg := nthRoot((chv.left)(dumk::F), chv.deg)
        kz := retract(numer newalg)@K
        newf := multivariate(chv.int, ku := dumk, newalg)
        vu := (chv.right)(x::F)
        vz := (chv.den)(x::F) * (y::F) * denom(newalg)::F
```

```
        map(x1+->eval(x1, [ku, kz], [vu, vz]), palgint(newf, ku, kz))
      cv     := chvar(ff, modulus)
      r      := radPoly(cv.poly)::Record(radicand:RF, deg:N)
      qprime := differentiate(q := retract(r.radicand)@UP)::RF
      not zero? qprime and
       ((u := chvarIfCan(cv.func, 1, q, inv qprime)) case UPUP) =>
         m := monomial(1, r.deg)$UPUP - q::RF::UPUP
         map(x1+->UPUP2F1(RF2UPUP(x1, m), cv.c1, cv.c2, x, y),
            rationalInt(u::UPUP, r.deg, monomial(1, 1)))
      curve  := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
      algaddx(map(x1+->UPUP2F1(lift x1, cv.c1, cv.c2, x, y),
        palgintegrate(reduce(cv.func), differentiate$UP)$ALG)$IR2, x::F)

-- Do the rationalizing change of variable
-- Int(f(x, y) dx) --> Int(n u^(n-1) f((u^n - b)/a, u) / a du) where
-- u^n = y^n = g(x) = a x + b
-- returns the integral as an integral of a rational function in u
    rationalInt(f, n, g) ==
--      not one? degree g => error "rationalInt: radicand must be linear"
      not ((degree g) = 1) => error "rationalInt: radicand must be linear"
      a := leadingCoefficient g
      integrate(n * monomial(inv a, (n-1)::N)$UP
                  * chv(f, n, a, leadingCoefficient reductum g))

-- Do the rationalizing change of variable f(x,y) --> f((u^n - b)/a, u) where
-- u = y = (a x + b)^(1/n).
-- Returns f((u^n - b)/a,u) as an element of F
    chv0(f, n, a, b) ==
      d := dumk::F
      (f (d::UP::RF)) ((d ** n - b) / a)

-- candidates(p) returns a list of pairs [g, u] such that p(x) = g(u(x)),
-- those u's are candidates for change of variables
-- currently uses a dumb heuristic where the candidates u's are p itself
-- and all the powers x^2, x^3, ..., x^{deg(p)},
-- will use polynomial decomposition in smarter days   MB 8/93
    candidates p ==
      l:List(CND) := empty()
      ground? p => l
      for i in 2..degree p repeat
        if (u := composite(p, xi := monomial(1, i))) case UP then
          l := concat([u::UP, xi], l)
      concat([monomial(1, 1), p], l)

-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x,
-- z = d y for some polynomial d, and z^m = g(u)
-- returns either [r(u, z), g, u, d, m] or "failed"
-- we have y^n = radi
    changeVarIfCan(p, radi, n) ==
```

```
      rec := rootPoly(radi, n)
      for cnd in candidates(rec.radicand) repeat
        (u := chvarIfCan(p, rec.coef, cnd.right,
              inv(differentiate(cnd.right)::RF))) case UPUP =>
                 return [u::UPUP, cnd.left, cnd.right, rec.coef, rec.exponent]
      "failed"

-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x and z = d y
-- we have y^n = a(x)/d(x)
-- returns either "failed" or r(u, z)
    chvarIfCan(p, d, u, u1) ==
      ans:UPUP := 0
      while p ^= 0 repeat
        (v := composite(u1 * leadingCoefficient(p) / d ** degree(p), u))
          case "failed" => return "failed"
        ans := ans + monomial(v::RF, degree p)
        p   := reductum p
      ans

    algaddx(i, xx) ==
      elem? i => i
      mkAnswer(ratpart i, logpart i,
                [[- ne.integrand / (xx**2), xx] for ne in notelem i])

    prootRDE(nfp, f, g, x, k, rde) ==
      modulus := lift(p := minPoly k, x)
      r       := radPoly(modulus)::Record(radicand:RF, deg:N)
      rec     := rootPoly(r.radicand, r.deg)
      dqdx    := inv(differentiate(q := rec.radicand)::RF)
      ((uf := chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,1)) case UPUP) and
        ((ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP) =>
          (u := rde(chv0(uf::UPUP, rec.exponent, 1, 0), rec.exponent *
                    (dumk::F) ** (rec.exponent * (rec.exponent - 1))
                      * chv0(ug::UPUP, rec.exponent, 1, 0),
                       symbolIfCan(dumk)::SY)) case "failed" => "failed"
          eval(u::F, dumk, k::F)
--      one?(rec.coef) =>
      ((rec.coef) = 1) =>
        curve  := RadicalFunctionField(F, UP, UPUP, q::RF, rec.exponent)
        rc := algDsolve(D()$LDALG + reduce(univariate(nfp, x, k, p))::LDALG,
                        reduce univariate(g, x, k, p))$RDALG
        rc.particular case "failed" => "failed"
        UPUP2F0(lift((rc.particular)::curve), x, k)
      palgRDE1(nfp, g, x, k)

    prootlimint(f, x, k, lu) ==
      modulus := lift(p := minPoly k, x)
      r       := radPoly(modulus)::Record(radicand:RF, deg:N)
      rec     := rootPoly(r.radicand, r.deg)
```

```
    dqdx    := inv(differentiate(q := rec.radicand)::RF)
    (uf:=chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP =>
      l := empty()$List(RF)
      n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
      for u in lu repeat
       if ((v:=chvarIfCan(uu:=univariate(u,x,k,p),rec.coef,q,dqdx))case UPUP)
          then l := concat(n * chv(v::UPUP,rec.exponent, 1, 0), l) else FAIL
      m := monomial(1, rec.exponent)$UPUP - q::RF::UPUP
      map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
          limitedint(n * chv(uf::UPUP, rec.exponent, 1, 0), reverse_! l))
    cv     := chvar(ff, modulus)
    r      := radPoly(cv.poly)::Record(radicand:RF, deg:N)
    dqdx   := inv(differentiate(q := retract(r.radicand)@UP)::RF)
    curve  := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
    (ui := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
      case "failed" => FAIL
    [UPUP2F1(lift(ui::curve), cv.c1, cv.c2, x, k), empty()]

  prootextint(f, x, k, g) ==
    modulus := lift(p := minPoly k, x)
    r       := radPoly(modulus)::Record(radicand:RF, deg:N)
    rec     := rootPoly(r.radicand, r.deg)
    dqdx    := inv(differentiate(q := rec.radicand)::RF)
    ((uf:=chvarIfCan(ff:=univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP) and
     ((ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP) =>
        m := monomial(1, rec.exponent)$UPUP - q::RF::UPUP
        n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
        map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
            extendedint(n * chv(uf::UPUP, rec.exponent, 1, 0),
                        n * chv(ug::UPUP, rec.exponent, 1, 0)))
    cv     := chvar(ff, modulus)
    r      := radPoly(cv.poly)::Record(radicand:RF, deg:N)
    dqdx   := inv(differentiate(q := retract(r.radicand)@UP)::RF)
    curve  := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
    (u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
      case "failed" => FAIL
    [UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

  palgRDE1(nfp, g, x, y) ==
    palgLODE1([nfp, 1], g, x, y, symbolIfCan(x)::SY).particular

  palgLODE1(eq, g, kx, y, x) ==
    modulus:= lift(p := minPoly y, kx)
    curve  := AlgebraicFunctionField(F, UP, UPUP, modulus)
    neq:LDALG := 0
    for f in eq for i in 0.. repeat
        neq := neq + monomial(reduce univariate(f, kx, y, p), i)
    empty? remove_!(y, remove_!(kx, varselect(kernels g, x))) =>
      rec := algDsolve(neq, reduce univariate(g, kx, y, p))$RDALG
      bas:List(F) := [UPUP2F0(lift h, kx, y) for h in rec.basis]
```

```
      rec.particular case "failed" => ["failed", bas]
      [UPUP2F0(lift((rec.particular)::curve), kx, y), bas]
    rec := algDsolve(neq, 0)
    ["failed", [UPUP2F0(lift h, kx, y) for h in rec.basis]]

  palgintegrate(f, x, k) ==
    modulus:= lift(p := minPoly k, x)
    cv     := chvar(univariate(f, x, k, p), modulus)
    curve  := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
    knownInfBasis(cv.deg)
    algaddx(map(x1+->UPUP2F1(lift x1, cv.c1, cv.c2, x, k),
      palgintegrate(reduce(cv.func), differentiate$UP)$ALG)$IR2, x::F)

  palglim(f, x, k, lu) ==
    modulus:= lift(p := minPoly k, x)
    cv     := chvar(univariate(f, x, k, p), modulus)
    curve  := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
    knownInfBasis(cv.deg)
    (u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
      case "failed" => FAIL
    [UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), empty()]

  palgext(f, x, k, g) ==
    modulus:= lift(p := minPoly k, x)
    cv     := chvar(univariate(f, x, k, p), modulus)
    curve  := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
    knownInfBasis(cv.deg)
    (u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
      case "failed" => FAIL
    [UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

  palgint(f, x, y) ==
    (v := linearInXIfCan(x, y)) case "failed" =>
      (u := quadIfCan(x, y)) case "failed" =>
        is?(y, "nthRoot"::SY) => prootintegrate(f, x, y)
        is?(y,  "rootOf"::SY) => palgintegrate(f, x, y)
        FAIL
      palgint0(f, x, y, u.coef, u.poly)
    palgint0(f, x, y, dumk, v.xsub, v.dxsub)

  palgextint(f, x, y, g) ==
    (v := linearInXIfCan(x, y)) case "failed" =>
      (u := quadIfCan(x, y)) case "failed" =>
        is?(y, "nthRoot"::SY) => prootextint(f, x, y, g)
        is?(y,  "rootOf"::SY) => palgext(f, x, y, g)
        FAIL
      palgextint0(f, x, y, g, u.coef, u.poly)
    palgextint0(f, x, y, g, dumk, v.xsub, v.dxsub)

  palglimint(f, x, y, lu) ==
```

```
      (v := linearInXIfCan(x, y)) case "failed" =>
        (u := quadIfCan(x, y)) case "failed" =>
          is?(y, "nthRoot"::SY) => prootlimint(f, x, y, lu)
          is?(y,  "rootOf"::SY) => palglim(f, x, y, lu)
          FAIL
        palglimint0(f, x, y, lu, u.coef, u.poly)
      palglimint0(f, x, y, lu, dumk, v.xsub, v.dxsub)

    palgRDE(nfp, f, g, x, y, rde) ==
      (v := linearInXIfCan(x, y)) case "failed" =>
        (u := quadIfCan(x, y)) case "failed" =>
          is?(y, "nthRoot"::SY) => prootRDE(nfp, f, g, x, y, rde)
          palgRDE1(nfp, g, x, y)
        palgRDE0(f, g, x, y, rde, u.coef, u.poly)
      palgRDE0(f, g, x, y, rde, dumk, v.xsub, v.dxsub)

    -- returns "failed", or (d, P) such that (dy)**2 = P(x)
    -- and degree(P) = 2
    quadIfCan(x, y) ==
      (degree(p := minPoly y) = 2) and zero?(coefficient(p, 1)) =>
        d := denom(ff :=
                 univariate(- coefficient(p, 0) / coefficient(p, 2), x))
        degree(radi := d * numer ff) = 2 => [d(x::F), radi]
        "failed"
      "failed"

    if L has LinearOrdinaryDifferentialOperatorCategory F then
      palgLODE(eq, g, kx, y, x) ==
        (v := linearInXIfCan(kx, y)) case "failed" =>
          (u := quadIfCan(kx, y)) case "failed" =>
            palgLODE1([coefficient(eq, i) for i in 0..degree eq], g, kx, y, x)
          palgLODE0(eq, g, kx, y, u.coef, u.poly)
        palgLODE0(eq, g, kx, y, dumk, v.xsub, v.dxsub)
```

———-

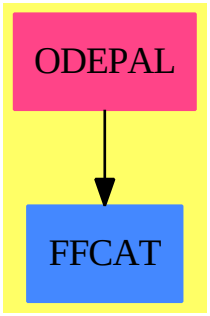
— **INTPAF.dotabb** —

```
"INTPAF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPAF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"INTPAF" -> "FS"
"INTPAF" -> "ACF"
```

———-

17.71 package ODEPAL PureAlgebraicLODE

17.71.1 PureAlgebraicLODE (ODEPAL)



Exports:
algDsolve

— package ODEPAL PureAlgebraicLODE —

```
)abbrev package ODEPAL PureAlgebraicLODE
++ Author: Manuel Bronstein
++ Date Created: 21 August 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ In-field solution of an linear ordinary differential equation,
++ pure algebraic case.

PureAlgebraicLODE(F, UP, UPUP, R): Exports == Implementation where
  F   : Join(Field, CharacteristicZero,
             RetractableTo Integer, RetractableTo Fraction Integer)
  UP  : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R   : FunctionFieldCategory(F, UP, UPUP)

  RF  ==> Fraction UP
  V   ==> Vector RF
  U   ==> Union(R, "failed")
  REC ==> Record(particular: Union(RF, "failed"), basis: List RF)
  L   ==> LinearOrdinaryDifferentialOperator1 R
  LQ  ==> LinearOrdinaryDifferentialOperator1 RF

  Exports ==> with
    algDsolve: (L, R) -> Record(particular: U, basis: List R)
      ++ algDsolve(op, g) returns \spad{["failed", []]} if the equation
      ++ \spad{op y = g} has no solution in \spad{R}. Otherwise, it returns
      ++ \spad{[f, [y1,...,ym]]} where \spad{f} is a particular rational
```

```
      ++ solution and the \spad{y_i's} form a basis for the solutions in
      ++ \spad{R} of the homogeneous equation.

  Implementation ==> add
    import RationalLODE(F, UP)
    import SystemODESolver(RF, LQ)
    import ReduceLODE(RF, LQ, UPUP, R, L)

    algDsolve(l, g) ==
      rec := reduceLODE(l, g)
      sol := solveInField(rec.mat, rec.vec, ratDsolve)
      bas:List(R) := [represents v for v in sol.basis]
      (u := sol.particular) case V => [represents(u::V), bas]
      ["failed", bas]
```

———-

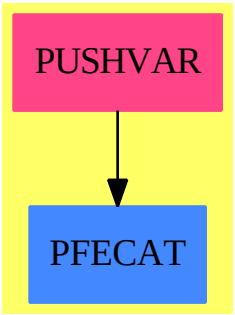
— **ODEPAL.dotabb** —

```
"ODEPAL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPAL"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"ODEPAL" -> "FFCAT"
```

———-

17.72 package PUSHVAR PushVariables

17.72.1 PushVariables (PUSHVAR)



Exports:

— **package PUSHVAR PushVariables** —

```
)abbrev package PUSHVAR PushVariables
++ Description:
++ This package has no description

PushVariables(R,E,OV,PPR):C == T where
   E : OrderedAbelianMonoidSup
   OV: OrderedSet with
        convert: % -> Symbol
          ++ convert(x) converts x to a symbol
        variable: Symbol -> Union(%, "failed")
          ++ variable(s) makes an element from symbol s or fails
   R  : Ring
   PR ==> Polynomial R
   PPR: PolynomialCategory(PR,E,OV)
   SUP ==> SparseUnivariatePolynomial
   C == with
     pushdown : (PPR, OV) -> PPR
       ++ pushdown(p,v) \undocumented{}
     pushdown : (PPR, List OV) -> PPR
       ++ pushdown(p,lv) \undocumented{}
     pushup   : (PPR, OV) -> PPR
       ++ pushup(p,v) \undocumented{}
     pushup   : (PPR, List OV) -> PPR
       ++ pushup(p,lv) \undocumented{}
     map      : ((PR -> PPR), PPR) -> PPR
       ++ map(f,p) \undocumented{}

   T == add
     pushdown(g:PPR,x:OV) : PPR ==
       eval(g,x,monomial(1,convert x,1)$PR)

     pushdown(g:PPR, lv:List OV) : PPR ==
       vals:=[monomial(1,convert x,1)$PR for x in lv]
       eval(g,lv,vals)

     map(f:(PR -> PPR), p: PPR) : PPR ==
       ground? p => f(retract p)
       v:=mainVariable(p)::OV
       multivariate(map((x:PPR):PPR+->map(f,x),univariate(p,v)),v)

              ----  push back the variable  ----
     pushupCoef(c:PR, lv:List OV): PPR ==
       ground? c => c::PPR
       v:=mainVariable(c)::Symbol
       v2 := variable(v)$OV
       uc := univariate(c,v)
       ppr : PPR := 0
       v2 case OV =>
          while not zero? uc repeat
             ppr := ppr + monomial(1,v2,degree(uc))$PPR *
```

```
                          pushupCoef(leadingCoefficient uc, lv)
            uc := reductum uc
          ppr
       while not zero? uc repeat
          ppr := ppr + monomial(1,v,degree(uc))$PR *
                          pushupCoef(leadingCoefficient uc, lv)
          uc := reductum uc
       ppr

     pushup(f:PPR,x:OV) :PPR ==
       map(y +-> pushupCoef(y,[x]), f)

     pushup(g:PPR, lv:List OV) : PPR ==
       map(y +-> pushupCoef(y, lv), g)
```

———-

— **PUSHVAR.dotabb** —

```
"PUSHVAR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PUSHVAR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PUSHVAR" -> "PFECAT"
```

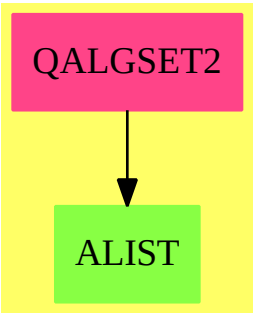
———-

Chapter 18

Chapter Q

18.1 package QALGSET2 QuasiAlgebraicSet2

18.1.1 QuasiAlgebraicSet2 (QALGSET2)



Exports:
radicalSimplify

— package QALGSET2 QuasiAlgebraicSet2 —

```
)abbrev package QALGSET2 QuasiAlgebraicSet2
++ Author:  William Sit
++ Date Created: March 13, 1992
++ Date Last Updated: June 12, 1992
++ Basic Operations:
++ Related Constructors:GroebnerPackage, IdealDecompositionPackage,
++                       PolynomialIdeals
++ See Also: QuasiAlgebraicSet
++ AMS Classifications:
```

```
++ Keywords: Zariski closed sets, quasi-algebraic sets
++ References:William Sit, "An Algorithm for Parametric Linear Systems"
++            J. Sym. Comp., April, 1992
++ Description:
++ \spadtype{QuasiAlgebraicSet2} adds a function \spadfun{radicalSimplify}
++ which uses \spadtype{IdealDecompositionPackage} to simplify
++ the representation of a quasi-algebraic set.  A quasi-algebraic set
++ is the intersection of a Zariski
++ closed set, defined as the common zeros of a given list of
++ polynomials (the defining polynomials for equations), and a principal
++ Zariski open set, defined as the complement of the common
++ zeros of a polynomial f (the defining polynomial for the inequation).
++ Quasi-algebraic sets are implemented in the domain
++ \spadtype{QuasiAlgebraicSet}, where two simplification routines are
++ provided:
++ \spadfun{idealSimplify} and \spadfun{simplify}.
++ The function
++ \spadfun{radicalSimplify} is added
++ for comparison study only.  Because the domain
++ \spadtype{IdealDecompositionPackage} provides facilities for
++ computing with radical ideals, it is necessary to restrict
++ the ground ring to the domain \spadtype{Fraction Integer},
++ and the polynomial ring to be of type
++ \spadtype{DistributedMultivariatePolynomial}.
++ The routine \spadfun{radicalSimplify} uses these to compute groebner
++ basis of radical ideals and
++ is inefficient and restricted when compared to the
++ two in \spadtype{QuasiAlgebraicSet}.

QuasiAlgebraicSet2(vl,nv) : C == T where
   vl       :   List Symbol
   nv       :   NonNegativeInteger
   R        ==> Integer
   F        ==> Fraction R
   Var      ==> OrderedVariableList vl
   NNI      ==> NonNegativeInteger
   Expon    ==> DirectProduct(nv,NNI)
   Dpoly    ==> DistributedMultivariatePolynomial(vl,F)
   QALG     ==> QuasiAlgebraicSet(F, Var, Expon, Dpoly)
   newExpon ==> DirectProduct(#newvl, NNI)
   newPoly  ==> DistributedMultivariatePolynomial(newvl,F)
   newVar   ==> OrderedVariableList newvl
   Status   ==> Union(Boolean,"failed") -- empty or not, or don't know

   C == with
     radicalSimplify:QALG -> QALG
       ++ radicalSimplify(s) returns a different and presumably simpler
       ++ representation of s with the defining polynomials for the
       ++ equations
       ++ forming a groebner basis, and the defining polynomial for the
```

```
    ++ inequation reduced with respect to the basis, using
    ++ using groebner basis of radical ideals
T  == add
            ----  Local Functions  ----
  ts:=new()$Symbol
  newvl:=concat(ts, vl)
  tv:newVar:=(variable ts)::newVar
  npoly         :     Dpoly            ->  newPoly
  oldpoly       :     newPoly          ->  Union(Dpoly,"failed")
  f             :     Var              ->  newPoly
  g             :     newVar           ->  Dpoly

  import PolynomialIdeals(F,newExpon,newVar,newPoly)
  import GroebnerPackage(F,Expon,Var,Dpoly)
  import GroebnerPackage(F,newExpon,newVar,newPoly)
  import IdealDecompositionPackage(newvl,#newvl)
  import QuasiAlgebraicSet(F, Var, Expon, Dpoly)
  import PolynomialCategoryLifting(Expon,Var,F,Dpoly,newPoly)
  import PolynomialCategoryLifting(newExpon,newVar,F,newPoly,Dpoly)
  f(v:Var):newPoly ==
    variable((convert v)@Symbol)@Union(newVar,"failed")::newVar
      ::newPoly
  g(v:newVar):Dpoly ==
    v = tv => 0
    variable((convert v)@Symbol)@Union(Var,"failed")::Var::Dpoly

  npoly(p:Dpoly) : newPoly ==  map(z1 +-> f z1, z2 +-> z2::newPoly, p)

  oldpoly(q:newPoly) : Union(Dpoly,"failed") ==
    (x:=mainVariable q) case "failed" => (leadingCoefficient q)::Dpoly
    (x::newVar = tv) => "failed"
    map(z1 +-> g z1, z2 +-> z2::Dpoly, q)

  radicalSimplify x ==
    status(x)$QALG = true => x     -- x is empty
    z0:=definingEquations x
    n0:=definingInequation x
    t:newPoly:= coerce(tv)$newPoly
    tp:newPoly:= t * (npoly n0) - 1$newPoly
    gen:List newPoly:= concat(tp, [npoly g for g in z0])
    id:=ideal gen
    ngb:=generators radical(id)
    member? (1$newPoly, ngb) => empty()$QALG
    gb:List Dpoly:=nil
    while not empty? ngb repeat
      if ((k:=oldpoly ngb.first) case Dpoly) then gb:=concat(k, gb)
      ngb:=ngb.rest
    y:=quasiAlgebraicSet(gb, primitivePart normalForm(n0, gb))
    setStatus(y,false::Status)
```

———-

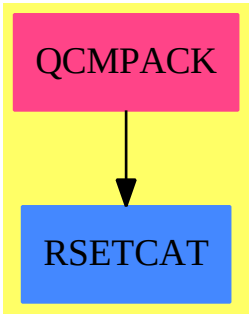
— QALGSET2.dotabb —

```
"QALGSET2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QALGSET2"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"QALGSET2" -> "ALIST"
```

———-

18.2 package QCMPACK QuasiComponentPackage

18.2.1 QuasiComponentPackage (QCMPACK)



Exports:

algebraicSort	branchIfCan
infRittWu?	internalInfRittWu?
internalSubPolSet?	internalSubQuasiComponent?
moreAlgebraic?	prepareDecompose
removeSuperfluousCases	removeSuperfluousQuasiComponents
startTable!	subCase?
subPolSet?	subQuasiComponent?
subQuasiComponent?	subTriSet?
supDimElseRittWu?	

— package QCMPACK QuasiComponentPackage —

```
)abbrev package QCMPACK QuasiComponentPackage
++ Author: Marc Moreno Maza
++   marc@nag.co.uk
++ Date Created: 08/30/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
```

```
++ Related Constructors:
++ Also See: 'tosedom.spad'
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] D. LAZARD "A new method for solving algebraic systems of
++      positive dimension" Discr. App. Math. 33:147-160,1991
++  [2] M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++      d'extensions simples et resolution des systemes d'equations
++      algebriques" These, Universite P.etM. Curie, Paris, 1997.
++  [3] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A package for removing redundant quasi-components and redundant
++ branches when decomposing a variety by means of quasi-components
++ of regular triangular sets.

QuasiComponentPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : RegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  LP ==> List P
  PtoP ==> P -> P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : TS)
  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Branch ==> Record(eq: List P, tower: TS, ineq: List P)
  UBF ==> Union(Branch,"failed")
  Split ==> List TS
  Key ==>  Record(left:TS, right:TS)
  Entry ==> Boolean
  H ==> TabulatedComputationPackage(Key, Entry)
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)

  Exports ==  with
     startTable!: (S,S,S) -> Void
         ++ \axiom{startTableGcd!(s1,s2,s3)}
         ++ is an internal subroutine, exported only for developement.
     stopTable!: () -> Void
         ++ \axiom{stopTableGcd!()}
         ++ is an internal subroutine, exported only for developement.
     supDimElseRittWu?: (TS,TS) -> Boolean
```

```
    ++ \axiom{supDimElseRittWu(ts,us)} returns true iff \axiom{ts}
    ++ has less elements than \axiom{us} otherwise if \axiom{ts}
    ++ has higher rank than \axiom{us} w.r.t. Riit and Wu ordering.
algebraicSort: Split -> Split
    ++ \axiom{algebraicSort(lts)} sorts \axiom{lts} w.r.t
    ++ supDimElseRittWu?
moreAlgebraic?: (TS,TS) -> Boolean
    ++ \axiom{moreAlgebraic?(ts,us)} returns false iff \axiom{ts}
    ++ and \axiom{us} are both empty, or \axiom{ts}
    ++ has less elements than \axiom{us}, or some variable is
    ++ algebraic w.r.t. \axiom{us} and is not w.r.t. \axiom{ts}.
subTriSet?: (TS,TS) -> Boolean
    ++ \axiom{subTriSet?(ts,us)} returns true iff \axiom{ts} is
    ++ a sub-set of \axiom{us}.
subPolSet?: (LP, LP) -> Boolean
    ++ \axiom{subPolSet?(lp1,lp2)} returns true iff \axiom{lp1} is
    ++ a sub-set of \axiom{lp2}.
internalSubPolSet?: (LP, LP) -> Boolean
    ++ \axiom{internalSubPolSet?(lp1,lp2)} returns true iff \axiom{lp1} is
    ++ a sub-set of \axiom{lp2} assuming that these lists are sorted
    ++ increasingly w.r.t.
    ++ infRittWu? from RecursivePolynomialCategory.
internalInfRittWu?: (LP, LP) -> Boolean
    ++ \axiom{internalInfRittWu?(lp1,lp2)}
    ++ is an internal subroutine, exported only for developement.
infRittWu?: (LP, LP) -> Boolean
    ++ \axiom{infRittWu?(lp1,lp2)}
    ++ is an internal subroutine, exported only for developement.
internalSubQuasiComponent?: (TS,TS) -> Union(Boolean,"failed")
    ++ \axiom{internalSubQuasiComponent?(ts,us)} returns a
    ++ boolean \spad{b} value if the fact that the regular
    ++ zero set of \axiom{us} contains that of
    ++ \axiom{ts} can be decided (and in that case \axiom{b} gives this
    ++ inclusion) otherwise returns \axiom{"failed"}.
subQuasiComponent?: (TS,TS) -> Boolean
    ++ \axiom{subQuasiComponent?(ts,us)} returns true iff
    ++ internalSubQuasiComponent?
    ++ returs true.
subQuasiComponent?: (TS,Split) -> Boolean
    ++ \axiom{subQuasiComponent?(ts,lus)} returns true iff
    ++ \axiom{subQuasiComponent?(ts,us)} holds for one \spad{us}
    ++ in \spad{lus}.
removeSuperfluousQuasiComponents: Split -> Split
    ++ \axiom{removeSuperfluousQuasiComponents(lts)} removes
    ++ from \axiom{lts} any \spad{ts} such that
    ++ \axiom{subQuasiComponent?(ts,us)} holds for
    ++ another \spad{us} in \axiom{lts}.
subCase?: (LpWT,LpWT) -> Boolean
    ++ \axiom{subCase?(lpwt1,lpwt2)}
    ++ is an internal subroutine, exported only for developement.
```

```
    removeSuperfluousCases: List LpWT -> List LpWT
        ++ \axiom{removeSuperfluousCases(llpwt)}
        ++ is an internal subroutine, exported only for developement.
    prepareDecompose: (LP, List(TS),B,B) -> List Branch
        ++ \axiom{prepareDecompose(lp,lts,b1,b2)}
        ++ is an internal subroutine, exported only for developement.
    branchIfCan: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
        ++ \axiom{branchIfCan(leq,ts,lineq,b1,b2,b3,b4,b5)}
        ++ is an internal subroutine, exported only for developement.

  Implementation == add

    squareFreeFactors(lp: LP): LP ==
      lsflp: LP := []
      for p in lp repeat
        lsfp := squareFreeFactors(p)$polsetpack
        lsflp := concat(lsfp,lsflp)
      sort(infRittWu?,removeDuplicates lsflp)

    startTable!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$H
      if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
      if (not empty? domainName) then startStats!(domainName)$H
      void()

    stopTable!(): Void ==
      if makingStats?()$H then printStats!()$H
      clearTable!()$H

    supDimElseRittWu? (ts:TS,us:TS): Boolean ==
      #ts < #us => true
      #ts > #us => false
      lp1 :LP := members(ts)
      lp2 :LP := members(us)
      while (not empty? lp1)
       and (not infRittWu?(first(lp2),first(lp1))) repeat
        lp1 := rest lp1
        lp2 := rest lp2
      not empty? lp1

    algebraicSort (lts:Split): Split ==
      lts := removeDuplicates lts
      sort(supDimElseRittWu?,lts)

    moreAlgebraic?(ts:TS,us:TS): Boolean  ==
      empty? ts => empty? us
      empty? us => true
      #ts < #us => false
      for p in (members us) repeat
         not algebraic?(mvar(p),ts) => return false
```

```
  true

subTriSet?(ts:TS,us:TS): Boolean  ==
  empty? ts => true
  empty? us => false
  mvar(ts) > mvar(us) => false
  mvar(ts) < mvar(us) => subTriSet?(ts,rest(us)::TS)
  first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS,rest(us)::TS)
  false

internalSubPolSet?(lp1: LP, lp2: LP): Boolean  ==
  empty? lp1 => true
  empty? lp2 => false
  associates?(first lp1, first lp2) =>
    internalSubPolSet?(rest lp1, rest lp2)
  infRittWu?(first lp1, first lp2) => false
  internalSubPolSet?(lp1, rest lp2)

subPolSet?(lp1: LP, lp2: LP): Boolean  ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1,lp2)

infRittWu?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1,lp2)

internalInfRittWu?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?(first lp1, first lp2)$P => true
  infRittWu?(first lp2, first lp1)$P => false
  infRittWu?(rest lp1, rest lp2)$$

subCase? (lpwt1:LpWT,lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false
  subQuasiComponent?(lpwt1.tower,lpwt2.tower)

internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
  -- "failed" is false iff saturate(us) is radical
  subTriSet?(us,ts) => true
  not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
  for p in (members us) repeat
    mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
      return("failed"::Union(Boolean,"failed"))
  for p in (members us) repeat
    not zero? initiallyReduce(p,ts) =>
      return("failed"::Union(Boolean,"failed"))
```

```
    lsfp := squareFreeFactors(initials us)
    for p in lsfp repeat
      not invertible?(p,ts)@B =>
        return(false::Union(Boolean,"failed"))
    true::Union(Boolean,"failed")

  subQuasiComponent?(ts:TS,us:TS): Boolean ==
    k: Key := [ts, us]
    e := extractIfCan(k)$H
    e case Entry => e::Entry
    ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts,us)
    b: Boolean := (ubf case Boolean) and (ubf::Boolean)
    insert!(k,b)$H
    b

  subQuasiComponent?(ts:TS,lus:Split): Boolean ==
    for us in lus repeat
       subQuasiComponent?(ts,us)@B => return true
    false

  removeSuperfluousCases (cases:List LpWT) ==
    #cases < 2 => cases
    toSee :=
      sort((x:LpWT,y:LpWT):Boolean +->
            supDimElseRittWu?(x.tower,y.tower),cases)
    lpwt1,lpwt2 : LpWT
    toSave,headmaxcases,maxcases,copymaxcases : List LpWT
    while not empty? toSee repeat
      lpwt1 := first toSee
      toSee := rest toSee
      toSave := []
      for lpwt2 in toSee repeat
         if subCase?(lpwt1,lpwt2)
           then
             lpwt1 := lpwt2
           else
             if not subCase?(lpwt2,lpwt1)
               then
                 toSave := cons(lpwt2,toSave)
      if empty? maxcases
        then
          headmaxcases := [lpwt1]
          maxcases := headmaxcases
        else
          copymaxcases := maxcases
          while (not empty? copymaxcases) and _
            (not subCase?(lpwt1,first(copymaxcases))) repeat
              copymaxcases := rest copymaxcases
          if empty? copymaxcases
            then
```

```
            setrest!(headmaxcases,[lpwt1])
            headmaxcases := rest headmaxcases
    toSee := reverse toSave
  maxcases

removeSuperfluousQuasiComponents(lts: Split): Split ==
  lts := removeDuplicates lts
  #lts < 2 => lts
  toSee := algebraicSort lts
  toSave,headmaxlts,maxlts,copymaxlts : Split
  while not empty? toSee repeat
    ts := first toSee
    toSee := rest toSee
    toSave := []
    for us in toSee repeat
       if subQuasiComponent?(ts,us)@B
         then
           ts := us
         else
           if not subQuasiComponent?(us,ts)@B
             then
               toSave := cons(us,toSave)
    if empty? maxlts
      then
        headmaxlts := [ts]
        maxlts := headmaxlts
      else
        copymaxlts := maxlts
        while (not empty? copymaxlts) and _
          (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
            copymaxlts := rest copymaxlts
        if empty? copymaxlts
          then
            setrest!(headmaxlts,[ts])
            headmaxlts := rest headmaxlts
    toSee := reverse toSave
  algebraicSort maxlts

removeAssociates (lp:LP):LP ==
  removeDuplicates [primitivePart(p) for p in lp]

branchIfCan(leq: LP,ts: TS,lineq: LP, b1:B,b2:B,b3:B,b4:B,b5:B):UBF ==
   -- ASSUME pols in leq are squarefree and mainly primitive
   -- if b1 then CLEAN UP leq
   -- if b2 then CLEAN UP lineq
   -- if b3 then SEARCH for ZERO in lineq with leq
   -- if b4 then SEARCH for ZERO in lineq with ts
   -- if b5 then SEARCH for ONE in leq with lineq
   if b1
     then
```

```
        leq := removeAssociates(leq)
        leq := remove(zero?,leq)
        any?(ground?,leq) =>
          return("failed"::Union(Branch,"failed"))
      if b2
        then
          any?(zero?,lineq) =>
            return("failed"::Union(Branch,"failed"))
          lineq := removeRedundantFactors(lineq)$polsetpack
      if b3
        then
          ps: PS := construct(leq)$PS
          for q in lineq repeat
            zero? remainder(q,ps).polnum =>
              return("failed"::Union(Branch,"failed"))
      (empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch)::UBF
      if b4
        then
          for q in lineq repeat
            zero? initiallyReduce(q,ts) =>
              return("failed"::Union(Branch,"failed"))
      if b5
        then
          newleq: LP := []
          for p in leq repeat
            for q in lineq repeat
              if mvar(p) = mvar(q)
                then
                  g := gcd(p,q)
                  newp := (p exquo g)::P
                  ground? newp =>
                    return("failed"::Union(Branch,"failed"))
                  newleq := cons(newp,newleq)
                else
                  newleq := cons(p,newleq)
          leq := newleq
      leq := sort(infRittWu?, removeDuplicates leq)
      ([leq, ts, lineq]$Branch)::UBF

    prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
      -- if b1 then REMOVE REDUNDANT COMPONENTS in lts
      -- if b2 then SPLIT the input system with squareFree
      lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))
      any?(ground?,lp) => []
      empty? lts => []
      if b1 then lts := removeSuperfluousQuasiComponents lts
      not b2 =>
        [[lp,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
      toSee: List Branch
      lq: LP := []
```

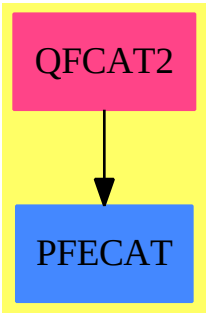
```
        toSee := [[lq,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
        empty? lp => toSee
        for p in lp repeat
          lsfp := squareFreeFactors(p)$polsetpack
          branches: List Branch := []
          lq := []
          for f in lsfp repeat
            for branch in toSee repeat
              leq : LP := branch.eq
              ts := branch.tower
              lineq : LP := branch.ineq
              ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true)@UBF
              ubf1 case "failed" => "leave"
              ubf2: UBF :=
                branchIfCan([f],ts,lineq,false,false,true,true,true)@UBF
              ubf2 case "failed" => "leave"
              leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))
              lineq :=
                sort(infRittWu?,removeDuplicates concat(ubf1.ineq,ubf2.ineq))
              newBranch :=
                branchIfCan(leq,ts,lineq,false,false,false,false,false)
              branches:= cons(newBranch::Branch,branches)
            lq := cons(f,lq)
          toSee := branches
        sort((x,y) +-> supDimElseRittWu?(x.tower,y.tower),toSee)
```

— QCMPACK.dotabb —

```
"QCMPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QCMPACK"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
"QCMPACK" -> "RSETCAT"
```

18.3 package QFCAT2 QuotientFieldCategoryFunctions2

18.3.1 QuotientFieldCategoryFunctions2 (QFCAT2)



Exports:
map

— package QFCAT2 QuotientFieldCategoryFunctions2 —

```
)abbrev package QFCAT2 QuotientFieldCategoryFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package extends a function between integral domains
++ to a mapping between their quotient fields.

QuotientFieldCategoryFunctions2(A, B, R, S): Exports == Impl where
  A, B: IntegralDomain
  R   : QuotientFieldCategory(A)
  S   : QuotientFieldCategory(B)

  Exports ==> with
    map: (A -> B, R) -> S
      ++ map(func,frac) applies the function func to the numerator
      ++ and denominator of frac.

  Impl ==> add
    map(f, r) == f(numer r) / f(denom r)
```

———-

— **QFCAT2.dotabb** —

```
"QFCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QFCAT2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"QFCAT2" -> "PFECAT"
```

———-

18.4 package QUATCT2 QuaternionCategoryFunctions2

— **QuaternionCategoryFunctions2.input** —

```
)set break resume
)spool QuaternionCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 4
q := quatern(2/11,-8,3/4,1)
--R
--R
--R         2        3
--R   (1)  -- - 8i + - j + k
--R        11        4
--R                                          Type: Quaternion(Fraction(Integer))
--E 1

--S 2 of 4
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer
--R
--R   Function declaration f : Fraction(Integer) -> Complex(Fraction(
--R      Integer)) has been added to workspace.
--R                                                                   Type: Void
--E 3

--S 3 of 4
map(f,q)
--R
--R   Compiling function f with type Fraction(Integer) -> Complex(Fraction
--R      (Integer))
--R
--R         2        3
```

```
--R   (3)  -- - 8i + - j + k
--R        11        4
--R                                  Type: Quaternion(Complex(Fraction(Integer)))
--E 3

--S 4 of 4
)show QuaternionCategoryFunctions2
--R
--R QuaternionCategoryFunctions2(QR: QuaternionCategory(R),R: CommutativeRing,QS: QuaternionCategory(S),S: C
--R Abbreviation for QuaternionCategoryFunctions2 is QUATCT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for QUATCT2
--R
--R------------------------------ Operations --------------------------------
--R map : ((R -> S),QR) -> QS
--R
--E 4

)spool
)lisp (bye)
```

———-

— QuaternionCategoryFunctions2.help —

```
=====================================================================
QuaternionCategoryFunctions2 examples
=====================================================================

The package QuaternionCategoryFunctions2 provides a function map
to convert an expression in Quaternion(R) to an expression in
Quaternion(S) using the function f.

q := quatern(2/11,-8,3/4,1)

         2        3
        -- - 8i + - j + k
        11        4
                                      Type: Quaternion Fraction Integer

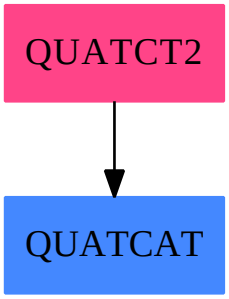
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer
   Function declaration f : Fraction Integer -> Complex Fraction
      Integer has been added to workspace.
                                                                  Type: Void

map(f,q)
   Compiling function f with type Fraction Integer -> Complex Fraction
      Integer
```

```
 2        3
-- - 8i + - j + k
11        4
                                  Type: Quaternion Complex Fraction Integer
```

```
See Also:
o )show QuaternionCategoryFunctions2
```

18.4.1 QuaternionCategoryFunctions2 (QUATCT2)



Exports:
map

— package QUATCT2 QuaternionCategoryFunctions2 —

```
)abbrev package QUATCT2 QuaternionCategoryFunctions2
++ Author: Robert S. Sutor
++ Date Created: 23 May 1990
++ Change History:
++   23 May 1990
++ Basic Operations: map
++ Related Constructors: QuaternionCategory, Quaternion
++ Also See:
++ AMS Classifications: 11R52
++ Keywords: quaternions, division ring, map
++ Description:
++ \spadtype{QuaternionCategoryFunctions2} implements functions between
++ two quaternion domains.  The function \spadfun{map} is used by
++ the system interpreter to coerce between quaternion types.
```

```
QuaternionCategoryFunctions2(QR,R,QS,S) : Exports ==
  Implementation where
    R  : CommutativeRing
    S  : CommutativeRing
    QR : QuaternionCategory R
    QS : QuaternionCategory S
    Exports == with
      map:     (R -> S, QR) -> QS
        ++ map(f,u) maps f onto the component parts of the quaternion u.
        ++ to convert an expression in Quaterion(R) to Quaternion(S)
        ++
        ++X f(a:FRAC(INT)):COMPLEX(FRAC(INT)) == a::COMPLEX(FRAC(INT))
        ++X q:=quatern(2/11,-8,3/4,1)
        ++X map(f,q)

    Implementation == add
      map(fn : R -> S, u : QR): QS ==
        quatern(fn real u, fn imagI u, fn imagJ u, fn imagK u)$QS
```

———-

— QUATCT2.dotabb —

```
"QUATCT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QUATCT2"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=QUATCAT"]
"QUATCT2" -> "QUATCAT"
```

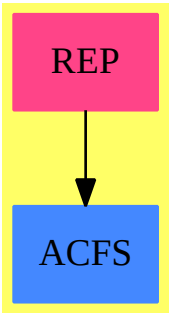
———-

Chapter 19

Chapter R

19.1 package REP RadicalEigenPackage

19.1.1 RadicalEigenPackage (REP)



Exports:

eigenMatrix	gramschmidt	normalise	orthonormalBasis
radicalEigenvalues	radicalEigenvectors		

— package REP RadicalEigenPackage —

```
)abbrev package REP RadicalEigenPackage
++ Author: P.Gianni
++ Date Created: Summer 1987
++ Date Last Updated: October 1992
++ Basic Functions:
++ Related Constructors: EigenPackage, RadicalSolve
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ Package for the computation of eigenvalues and eigenvectors.
++ This package works for matrices with coefficients which are
++ rational functions over the integers.
++ (see \spadtype{Fraction Polynomial Integer}).
++ The eigenvalues and eigenvectors are expressed in terms of radicals.

RadicalEigenPackage() : C == T
 where
   R     ==> Integer
   P     ==> Polynomial R
   F     ==> Fraction P
   RE    ==> Expression R
   SE    ==> Symbol()
   M     ==> Matrix(F)
   MRE   ==> Matrix(RE)
   ST    ==> SuchThat(SE,P)
   NNI   ==> NonNegativeInteger

   EigenForm    ==> Record(eigval:Union(F,ST),eigmult:NNI,eigvec:List(M))
   RadicalForm  ==> Record(radval:RE,radmult:Integer,radvect:List(MRE))



   C == with
     radicalEigenvectors    :  M     ->  List(RadicalForm)
       ++ radicalEigenvectors(m) computes
       ++ the eigenvalues and the corresponding eigenvectors of the
       ++ matrix m;
       ++ when possible, values are expressed in terms of radicals.

     radicalEigenvector       :  (RE,M)     ->  List(MRE)
       ++ radicalEigenvector(c,m) computes the eigenvector(s) of the
       ++ matrix m corresponding to the eigenvalue c;
       ++ when possible, values are
       ++ expressed in terms of radicals.

     radicalEigenvalues     :  M     ->  List RE
       ++ radicalEigenvalues(m) computes the eigenvalues of the matrix m;
       ++ when possible, the eigenvalues are expressed in terms of radicals.

     eigenMatrix       :    M        ->  Union(MRE,"failed")
       ++ eigenMatrix(m) returns the matrix b
       ++ such that \spad{b*m*(inverse b)} is diagonal,
       ++ or "failed" if no such b exists.

     normalise         :    MRE      ->  MRE
       ++ normalise(v) returns the column
       ++ vector v
```

```
      ++ divided by its euclidean norm;
      ++ when possible, the vector v is expressed in terms of radicals.

    gramschmidt      : List(MRE)   ->  List(MRE)
      ++ gramschmidt(lv) converts the list of column vectors lv into
      ++ a set of orthogonal column vectors
      ++ of euclidean length 1 using the Gram-Schmidt algorithm.

    orthonormalBasis  :    M        ->  List(MRE)
      ++ orthonormalBasis(m)  returns the orthogonal matrix b such that
      ++ \spad{b*m*(inverse b)} is diagonal.
      ++ Error: if m is not a symmetric matrix.

  T == add
    PI       ==> PositiveInteger
    RSP := RadicalSolvePackage R
    import EigenPackage R

                ----  Local  Functions  ----
    evalvect         :  (M,RE,SE)  ->  MRE
    innerprod        :   (MRE,MRE)   ->  RE

        ----  eval a vector of F in a radical expression  ----
    evalvect(vect:M,alg:RE,x:SE) : MRE ==
      n:=nrows vect
      xx:=kernel(x)$Kernel(RE)
      w:MRE:=zero(n,1)$MRE
      for i in 1..n repeat
        v:=eval(vect(i,1) :: RE,xx,alg)
        setelt(w,i,1,v)
      w
                    ---- inner product ----
    innerprod(v1:MRE,v2:MRE): RE == (((transpose v1)* v2)::MRE)(1,1)

                ----  normalization of a vector  ----
    normalise(v:MRE) : MRE ==
      normv:RE := sqrt(innerprod(v,v))
      normv = 0$RE => v
      (1/normv)*v

              ----  Eigenvalues of the matrix A  ----
    radicalEigenvalues(A:M): List(RE) ==
      x:SE :=new()$SE
      pol:= characteristicPolynomial(A,x) :: F
      radicalRoots(pol,x)$RSP

     ----  Eigenvectors belonging to a given eigenvalue  ----
           ----  expressed in terms of radicals ----
    radicalEigenvector(alpha:RE,A:M) : List(MRE) ==
      n:=nrows A
```

```
  B:MRE := zero(n,n)$MRE
  for i in 1..n repeat
    for j in 1..n repeat B(i,j):=(A(i,j))::RE
    B(i,i):= B(i,i) - alpha
  [v::MRE  for v in nullSpace B]

        ----  eigenvectors and eigenvalues  ----
radicalEigenvectors(A:M) : List(RadicalForm) ==
  leig:List EigenForm := eigenvectors A
  n:=nrows A
  sln:List RadicalForm := empty()
  veclist: List MRE
  for eig in leig repeat
    eig.eigval case F =>
      veclist := empty()
      for ll in eig.eigvec repeat
        m:MRE:=zero(n,1)
        for i in 1..n repeat m(i,1):=(ll(i,1))::RE
        veclist:=cons(m,veclist)
      sln:=cons([(eig.eigval)::F::RE,eig.eigmult,veclist]$RadicalForm,sln)
    sym := eig.eigval :: ST
    xx:= lhs sym
    lval : List RE := radicalRoots((rhs sym) :: F ,xx)$RSP
    for alg in lval repeat
      nsl:=[alg,eig.eigmult,
            [evalvect(ep,alg,xx) for ep in eig.eigvec]]$RadicalForm
      sln:=cons(nsl,sln)
  sln

       ----  orthonormalization of a list of vectors  ----
             ----  Grahm - Schmidt process  ----

gramschmidt(lvect:List(MRE)) : List(MRE) ==
  lvect=[]  => []
  v:=lvect.first
  n := nrows v
  RMR:=RectangularMatrix(n:PI,1,RE)
  orth:List(MRE):=[(normalise v)]
  for v in lvect.rest repeat
    pol:=((v:RMR)-(+/[(innerprod(w,v)*w):RMR for w in orth])):MRE
    orth:=cons(normalise pol,orth)
  orth


        ----  The matrix of eigenvectors  ----

eigenMatrix(A:M) : Union(MRE,"failed") ==
  lef:List(MRE):=[:eiv.radvect  for eiv in radicalEigenvectors(A)]
  n:=nrows A
  #lef <n => "failed"
```

```
      d:MRE:=copy(lef.first)
      for v in lef.rest repeat d:=(horizConcat(d,v))::MRE
      d

        ----  orthogonal basis for a symmetric matrix  ----

    orthonormalBasis(A:M):List(MRE) ==
      ^symmetric?(A) => error "the matrix is not symmetric"
      basis:List(MRE):=[]
      lvec:List(MRE) := []
      alglist:List(RadicalForm):=radicalEigenvectors(A)
      n:=nrows A
      for alterm in alglist repeat
        if (lvec:=alterm.radvect)=[] then error "sorry "
        if #(lvec)>1  then
          lvec:= gramschmidt(lvec)
          basis:=[:lvec,:basis]
        else basis:=[normalise(lvec.first),:basis]
      basis
```

———-

— **REP.dotabb** —

```
"REP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"REP" -> "ACFS"
```

———-

19.2 package SOLVERAD RadicalSolvePackage

— **RadicalSolvePackage.input** —

```
)set break resume
)sys rm -f RadicalSolvePackage.output
)spool RadicalSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 14
b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
--R
```

```
--R
--R          3
--R        3x  + 7
--R   (1)  --------
--R          2
--R        5x  - 13
--R                                        Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 14
radicalSolve(b,x)
--R
--R
--R   (2)
--R       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
--R       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
--R   [x= ------,x= ------------------------,x= --------------------------]
--R        3+-+               3+-+                          3+-+
--R        \|3               2\|3                          2\|3
--R                                   Type: List(Equation(Expression(Integer)))
--E 2

--S 3 of 14
radicalSolve(b)
--R
--R
--R   (3)
--R       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
--R       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
--R   [x= ------,x= ------------------------,x= --------------------------]
--R        3+-+               3+-+                          3+-+
--R        \|3               2\|3                          2\|3
--R                                   Type: List(Equation(Expression(Integer)))
--E 3

--S 4 of 14
radicalSolve(b=0,x)
--R
--R
--R   (4)
--R       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
--R       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
--R   [x= ------,x= ------------------------,x= --------------------------]
--R        3+-+               3+-+                          3+-+
--R        \|3               2\|3                          2\|3
--R                                   Type: List(Equation(Expression(Integer)))
--E 4

--S 5 of 14
radicalSolve(b=0)
```

```
--R
--R
--R   (5)
--R       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
--R       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
--R   [x= ------,x= ------------------------,x= --------------------------]
--R        3+-+               3+-+                          3+-+
--R        \|3               2\|3                          2\|3
--R                                  Type: List(Equation(Expression(Integer)))
--E 5

--S 6 of 14
radicalRoots(b,x)
--R
--R
--R         3+---+ 3+---+ +---+ +-+   3+---+   3+---+ +---+ +-+   3+---+
--R         \|- 7  \|- 7 \|- 1 \|3  - \|- 7  - \|- 7 \|- 1 \|3  - \|- 7
--R   (6)  [------,------------------------,--------------------------]
--R          3+-+             3+-+                       3+-+
--R          \|3             2\|3                       2\|3
--R                                          Type: List(Expression(Integer))
--E 6

--S 7 of 14
contractSolve(b=0,x)
--R
--R
--R                 +---+ +-+     3+--+     +---+ +-+     3+--+
--R         3+--+ (\|- 1 \|3  - 1)\|%A  (- \|- 1 \|3  - 1)\|%A           7
--R   (7)  [\|%A ,---------------------,-----------------------] | [%A= - -]
--R                        2                     2                     3
--RType: SuchThat(List(Expression(Integer)),List(Equation(Expression(Integer))))
--E 7

--S 8 of 14
contractSolve(b,x)
--R
--R
--R                 +---+ +-+     3+--+     +---+ +-+     3+--+
--R         3+--+ (\|- 1 \|3  - 1)\|%B  (- \|- 1 \|3  - 1)\|%B           7
--R   (8)  [\|%B ,---------------------,-----------------------] | [%B= - -]
--R                        2                     2                     3
--RType: SuchThat(List(Expression(Integer)),List(Equation(Expression(Integer))))
--E 8

--S 9 of 14
c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
--R
--R
--R         2
```

```
--R        y  + 4
--R   (9)  ------
--R         y + 1
--R                                          Type: Fraction(Polynomial(Integer))
--E 9

--S 10 of 14
radicalSolve([b,c],[x,y])
--R
--R
--R   (10)
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7         +---+
--R   [[x= --------------------------,y= - 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                   3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7         +---+      \|- 7         +---+
--R    [x= ------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
--R                   3+-+                                3+-+
--R                  2\|3                                 \|3
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7       +---+
--R    [x= --------------------------,y= 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                  3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
--R    [x= ------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
--R                   3+-+                               3+-+
--R                  2\|3                                \|3
--R                                Type: List(List(Equation(Expression(Integer))))
--E 10

--S 11 of 14
radicalSolve([b,c])
--R
--R
--R   (11)
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7         +---+
--R   [[x= --------------------------,y= - 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                   3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7         +---+      \|- 7         +---+
--R    [x= ------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
--R                   3+-+                                3+-+
--R                  2\|3                                 \|3
--R          3+---+ +---+ +-+   3+---+
```

```
--R        - \|- 7 \|- 1 \|3  - \|- 7       +---+
--R    [x= ---------------------------,y= 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                 3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
--R    [x= -------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
--R                   3+-+                            3+-+
--R                  2\|3                             \|3
--R                              Type: List(List(Equation(Expression(Integer))))
--E 11

--S 12 of 14
radicalSolve([b=0,c=0],[x,y])
--R
--R
--R   (12)
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7         +---+
--R   [[x= ---------------------------,y= - 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                   3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7         +---+      \|- 7         +---+
--R    [x= -------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
--R                   3+-+                             3+-+
--R                  2\|3                              \|3
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7       +---+
--R    [x= ---------------------------,y= 2\|- 1 ],
--R                   3+-+
--R                  2\|3
--R        3+---+ +---+ +-+   3+---+                 3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
--R    [x= -------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
--R                   3+-+                            3+-+
--R                  2\|3                             \|3
--R                              Type: List(List(Equation(Expression(Integer))))
--E 12

--S 13 of 14
radicalSolve([b=0,c=0])
--R
--R
--R   (13)
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7         +---+
--R   [[x= ---------------------------,y= - 2\|- 1 ],
--R                   3+-+
--R                  2\|3
```

```
--R        3+---+ +---+ +-+   3+---+                    3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7         +---+       \|- 7         +---+
--R    [x= ------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
--R                  3+-+                                3+-+
--R                 2\|3                                 \|3
--R          3+---+ +---+ +-+   3+---+
--R        - \|- 7 \|- 1 \|3  - \|- 7       +---+
--R    [x= --------------------------,y= 2\|- 1 ],
--R                  3+-+
--R                 2\|3
--R        3+---+ +---+ +-+   3+---+                 3+---+
--R        \|- 7 \|- 1 \|3  - \|- 7       +---+     \|- 7       +---+
--R    [x= ------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
--R                  3+-+                              3+-+
--R                 2\|3                               \|3
--R                                Type: List(List(Equation(Expression(Integer))))
--E 13

--S 14 of 14
radicalRoots([b,c],[x,y])
--R
--R
--R   (14)
--R       3+---+ +---+ +-+   3+---+
--R     - \|- 7 \|- 1 \|3  - \|- 7      +---+
--R   [[--------------------------,- 2\|- 1 ],
--R                3+-+
--R               2\|3
--R     3+---+ +---+ +-+   3+---+             3+---+
--R     \|- 7 \|- 1 \|3  - \|- 7      +---+    \|- 7      +---+
--R    [------------------------,- 2\|- 1 ], [------,- 2\|- 1 ],
--R                3+-+                          3+-+
--R               2\|3                           \|3
--R       3+---+ +---+ +-+   3+---+            3+---+ +---+ +-+   3+---+
--R     - \|- 7 \|- 1 \|3  - \|- 7    +---+    \|- 7 \|- 1 \|3  - \|- 7    +---+
--R    [--------------------------,2\|- 1 ], [------------------------,2\|- 1 ],
--R                3+-+                                  3+-+
--R               2\|3                                  2\|3
--R     3+---+
--R     \|- 7    +---+
--R    [------,2\|- 1 ]]
--R      3+-+
--R      \|3
--R                                       Type: List(List(Expression(Integer)))
--E 14

)spool
```

— RadicalSolvePackage.help —

```
=====================================================================
RadicalSolvePackage examples
=====================================================================

b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
          3
         3x  + 7
        --------
          2
        5x  - 13


radicalSolve(b,x)
       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
   [x= ------,x= ------------------------,x= --------------------------]
        3+-+                3+-+                            3+-+
        \|3                2\|3                            2\|3


radicalSolve(b)
       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
   [x= ------,x= ------------------------,x= --------------------------]
        3+-+                3+-+                            3+-+
        \|3                2\|3                            2\|3


radicalSolve(b=0,x)
       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
   [x= ------,x= ------------------------,x= --------------------------]
        3+-+                3+-+                            3+-+
        \|3                2\|3                            2\|3


radicalSolve(b=0)
       3+---+    3+---+ +---+ +-+   3+---+      3+---+ +---+ +-+   3+---+
       \|- 7     \|- 7 \|- 1 \|3  - \|- 7     - \|- 7 \|- 1 \|3  - \|- 7
   [x= ------,x= ------------------------,x= --------------------------]
        3+-+                3+-+                            3+-+
        \|3                2\|3                            2\|3


radicalRoots(b,x)
        3+---+ 3+---+ +---+ +-+   3+---+   3+---+ +---+ +-+   3+---+
        \|- 7  \|- 7 \|- 1 \|3  - \|- 7  - \|- 7 \|- 1 \|3  - \|- 7
```

```
       [------,------------------------,--------------------------]
         3+-+             3+-+                      3+-+
         \|3              2\|3                      2\|3
```

```
contractSolve(b=0,x)
                  +---+ +-+     3+--+     +---+ +-+     3+--+
        3+--+ (\|- 1 \|3  - 1)\|%A  (- \|- 1 \|3  - 1)\|%A            7
       [\|%A ,---------------------,-----------------------] | [%A= - -]
                        2                      2                      3
```

```
contractSolve(b,x)
                  +---+ +-+     3+--+     +---+ +-+     3+--+
        3+--+ (\|- 1 \|3  - 1)\|%B  (- \|- 1 \|3  - 1)\|%B            7
       [\|%B ,---------------------,-----------------------] | [%B= - -]
                        2                      2                      3
```

```
c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
        2
       y  + 4
       ------
        y + 1
```

```
radicalSolve([b,c],[x,y])
         3+---+ +---+ +-+   3+---+
       - \|- 7 \|- 1 \|3  - \|- 7         +---+
   [[x= ---------------------------,y= - 2\|- 1 ],
                  3+-+
                 2\|3
       3+---+ +---+ +-+   3+---+                    3+---+
       \|- 7 \|- 1 \|3  - \|- 7         +---+       \|- 7         +---+
    [x= -------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
                  3+-+                                3+-+
                 2\|3                                 \|3
         3+---+ +---+ +-+   3+---+
       - \|- 7 \|- 1 \|3  - \|- 7       +---+
    [x= ---------------------------,y= 2\|- 1 ],
                  3+-+
                 2\|3
       3+---+ +---+ +-+   3+---+                  3+---+
       \|- 7 \|- 1 \|3  - \|- 7       +---+       \|- 7       +---+
    [x= -------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
                  3+-+                                3+-+
                 2\|3                                 \|3
```

```
radicalSolve([b,c])
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7         +---+
   [[x= ----------------------------,y= - 2\|- 1 ],
                   3+-+
                  2\|3
        3+---+ +---+ +-+   3+---+                    3+---+
        \|- 7 \|- 1 \|3  - \|- 7         +---+       \|- 7         +---+
    [x= -------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
                  3+-+                                3+-+
                 2\|3                                 \|3
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7       +---+
    [x= ----------------------------,y= 2\|- 1 ],
                   3+-+
                  2\|3
        3+---+ +---+ +-+   3+---+                 3+---+
        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
    [x= -------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
                  3+-+                               3+-+
                 2\|3                                \|3
```

```
radicalSolve([b=0,c=0],[x,y])
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7         +---+
   [[x= ----------------------------,y= - 2\|- 1 ],
                   3+-+
                  2\|3
        3+---+ +---+ +-+   3+---+                    3+---+
        \|- 7 \|- 1 \|3  - \|- 7         +---+       \|- 7         +---+
    [x= -------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
                  3+-+                                3+-+
                 2\|3                                 \|3
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7       +---+
    [x= ----------------------------,y= 2\|- 1 ],
                   3+-+
                  2\|3
        3+---+ +---+ +-+   3+---+                 3+---+
        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
    [x= -------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
                  3+-+                               3+-+
                 2\|3                                \|3
```

```
radicalSolve([b=0,c=0])
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7         +---+
   [[x= --------------------------,y= - 2\|- 1 ],
                  3+-+
                 2\|3
        3+---+ +---+ +-+   3+---+                   3+---+
        \|- 7 \|- 1 \|3  - \|- 7         +---+      \|- 7         +---+
    [x= ------------------------,y= - 2\|- 1 ], [x= ------,y= - 2\|- 1 ],
                  3+-+                                3+-+
                 2\|3                                 \|3
          3+---+ +---+ +-+   3+---+
        - \|- 7 \|- 1 \|3  - \|- 7       +---+
    [x= --------------------------,y= 2\|- 1 ],
                  3+-+
                 2\|3
        3+---+ +---+ +-+   3+---+                  3+---+
        \|- 7 \|- 1 \|3  - \|- 7       +---+      \|- 7       +---+
    [x= ------------------------,y= 2\|- 1 ], [x= ------,y= 2\|- 1 ]]
                  3+-+                                3+-+
                 2\|3                                 \|3

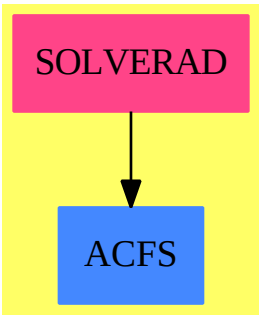


radicalRoots([b,c],[x,y])
       3+---+ +---+ +-+   3+---+
     - \|- 7 \|- 1 \|3  - \|- 7      +---+
   [[---------------------------,- 2\|- 1 ],
                3+-+
               2\|3
     3+---+ +---+ +-+   3+---+             3+---+
     \|- 7 \|- 1 \|3  - \|- 7      +---+    \|- 7      +---+
    [------------------------,- 2\|- 1 ], [------,- 2\|- 1 ],
               3+-+                           3+-+
              2\|3                            \|3
       3+---+ +---+ +-+   3+---+            3+---+ +---+ +-+   3+---+
     - \|- 7 \|- 1 \|3  - \|- 7    +---+    \|- 7 \|- 1 \|3  - \|- 7    +---+
    [--------------------------,2\|- 1 ], [------------------------,2\|- 1 ],
                3+-+                                  3+-+
               2\|3                                   2\|3
     3+---+
     \|- 7    +---+
    [------,2\|- 1 ]]
      3+-+
      \|3


See Also:
o )show RadicalSolvePackage
o )show RealSolvePackage
```

———-

19.2.1 RadicalSolvePackage (SOLVERAD)



Exports:
contractSolve radicalRoots radicalSolve

— package SOLVERAD RadicalSolvePackage —

```
)abbrev package SOLVERAD RadicalSolvePackage
++ Author: P.Gianni
++ Date Created: Summer 1990
++ Date Last Updated: October 1991
++ Basic Functions:
++ Related Constructors: SystemSolvePackage, FloatingRealPackage,
++ FloatingComplexPackage
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package tries to find solutions
++ expressed in terms of radicals for systems of equations
++ of rational functions with coefficients in an integral domain R.

RadicalSolvePackage(R): Cat == Capsule where
    R   :  Join(EuclideanDomain, OrderedSet, CharacteristicZero)
    PI ==> PositiveInteger
    NNI==> NonNegativeInteger
    Z  ==> Integer
    B  ==> Boolean
    ST ==> String
```

```
PR ==> Polynomial R
UP ==> SparseUnivariatePolynomial PR
LA ==> LocalAlgebra(PR, Z, Z)
RF ==> Fraction PR
RE ==> Expression R
EQ ==> Equation
SY ==> Symbol
SU ==> SuchThat(List RE, List Equation RE)
SUP==> SparseUnivariatePolynomial
L  ==> List
P  ==> Polynomial

SOLVEFOR ==> PolynomialSolveByFormulas(SUP RE, RE)
UPF2     ==> SparseUnivariatePolynomialFunctions2(PR,RE)

Cat ==> with

    radicalSolve :     (RF,SY)      -> L EQ RE
      ++ radicalSolve(rf,x) finds the solutions expressed in terms of
      ++ radicals of the equation rf = 0 with respect to the symbol x,
      ++ where rf is a rational function.
      ++
      ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
      ++X radicalSolve(b,x)
    radicalSolve :       RF         -> L EQ RE
      ++ radicalSolve(rf) finds the solutions expressed in terms of
      ++ radicals of the equation rf = 0, where rf is a
      ++ univariate rational function.
      ++
      ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
      ++X radicalSolve(b)
    radicalSolve :    (EQ RF,SY)    -> L EQ RE
      ++ radicalSolve(eq,x) finds the solutions expressed in terms of
      ++ radicals of the equation of rational functions eq
      ++ with respect to the symbol x.
      ++
      ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
      ++X radicalSolve(b=0,x)
    radicalSolve :      EQ RF       -> L EQ RE
      ++ radicalSolve(eq) finds the solutions expressed in terms of
      ++ radicals of the equation of rational functions eq
      ++ with respect to the unique symbol x appearing in eq.
      ++
      ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
      ++X radicalSolve(b=0)
    radicalSolve :    (L RF,L SY)   -> L L EQ RE
      ++ radicalSolve(lrf,lvar) finds the solutions expressed in terms of
      ++ radicals of the system of equations lrf = 0 with
      ++ respect to the list of symbols lvar,
      ++ where lrf is a list of rational functions.
```

```
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
  ++X radicalSolve([b,c],[x,y])
radicalSolve :       L RF       -> L L EQ RE
  ++ radicalSolve(lrf) finds the solutions expressed in terms of
  ++ radicals of the system of equations lrf = 0, where lrf is a
  ++ system of univariate rational functions.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
  ++X radicalSolve([b,c])
radicalSolve :   (L EQ RF,L SY) -> L L EQ RE
  ++ radicalSolve(leq,lvar) finds the solutions expressed in terms of
  ++ radicals of the system of equations of rational functions leq
  ++ with respect to the list of symbols lvar.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
  ++X radicalSolve([b=0,c=0],[x,y])
radicalSolve :     L EQ RF      -> L L EQ RE
  ++ radicalSolve(leq) finds the solutions expressed in terms of
  ++ radicals of the system of equations of rational functions leq
  ++ with respect to the unique symbol x appearing in leq.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
  ++X radicalSolve([b=0,c=0])
radicalRoots :      (RF,SY)     -> L RE
  ++ radicalRoots(rf,x) finds the roots expressed in terms of radicals
  ++ of the rational function rf with respect to the symbol x.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X radicalRoots(b,x)
radicalRoots :    (L RF,L SY)   -> L L RE
  ++ radicalRoots(lrf,lvar) finds the roots expressed in terms of
  ++ radicals of the list of rational functions lrf
  ++ with respect to the list of symbols lvar.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
  ++X c:Fraction(Polynomial(Integer)):=(y^2+4)/(y+1)
  ++X radicalRoots([b,c],[x,y])
contractSolve:    (EQ RF,SY)  -> SU
  ++ contractSolve(eq,x) finds the solutions expressed in terms of
  ++ radicals of the equation of rational functions eq
  ++ with respect to the symbol x.  The result contains new
  ++ symbols for common subexpressions in order to reduce the
  ++ size of the output.
  ++
  ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
```

```
      ++X contractSolve(b=0,x)
    contractSolve:    (RF,SY)     -> SU
      ++ contractSolve(rf,x) finds the solutions expressed in terms of
      ++ radicals of the equation rf = 0 with respect to the symbol x,
      ++ where rf is a rational function. The result contains  new
      ++ symbols for common subexpressions in order to reduce the
      ++ size of the output.
      ++
      ++X b:Fraction(Polynomial(Integer)):=(3*x^3+7)/(5*x^2-13)
      ++X contractSolve(b,x)
  Capsule ==> add
    import DegreeReductionPackage(PR, R)
    import SOLVEFOR

    SideEquations: List EQ RE := []
    ContractSoln:  B := false

    ---- Local Function Declarations ----
    solveInner:(PR, SY, B) -> SU
    linear:    UP -> List RE
    quadratic: UP -> List RE
    cubic:     UP -> List RE
    quartic:   UP -> List RE
    rad:       PI -> RE
    wrap:      RE -> RE
    New:       RE -> RE
    makeEq : (List RE,L SY) -> L EQ RE
    select :    L L RE      -> L L RE
    isGeneric? :  (L PR,L SY)  ->  Boolean
    findGenZeros :  (L PR,L SY) -> L L RE
    findZeros   :   (L PR,L SY) -> L L RE


    New s ==
        s = 0 => 0
        S := new()$Symbol ::PR::RF::RE
        SideEquations := append([S = s], SideEquations)
        S

    linear u    == [(-coefficient(u,0))::RE /(coefficient(u,1))::RE]
    quadratic u == quadratic(map(coerce,u)$UPF2)$SOLVEFOR
    cubic u     == cubic(map(coerce,u)$UPF2)$SOLVEFOR
    quartic u   == quartic(map(coerce,u)$UPF2)$SOLVEFOR
    rad n       == n::Z::RE
    wrap s      == (ContractSoln => New s; s)


    ---- Exported Functions ----
```

```
-- find the zeros of components in "generic" position --
 findGenZeros(rlp:L PR,rlv:L SY) : L L RE ==
  pp:=rlp.first
  v:=first rlv
  rlv:=rest rlv
  res:L L RE:=[]
  res:=append([reverse cons(r,[eval(
    (-coefficient(univariate(p,vv),0)::RE)/
     (leadingCoefficient univariate(p,vv))::RE,
       kernel(v)@Kernel(RE),r) for vv in rlv for p in rlp.rest])
         for r in radicalRoots(pp::RF,v)],res)
  res


 findZeros(rlp:L PR,rlv:L SY) : L L RE ==
  parRes:=[radicalRoots(p::RF,v) for p in rlp for v in rlv]
  parRes:=select parRes
  res:L L RE :=[]
  res1:L RE
  for par in parRes repeat
    res1:=[par.first]
    lv1:L Kernel(RE):=[kernel rlv.first]
    rlv1:=rlv.rest
    p1:=par.rest
    while p1^=[] repeat
      res1:=cons(eval(p1.first,lv1,res1),res1)
      p1:=p1.rest
      lv1:=cons(kernel rlv1.first,lv1)
      rlv1:=rlv1.rest
    res:=cons(res1,res)
  res

 radicalSolve(pol:RF,v:SY) ==
   [equation(v::RE,r) for r in radicalRoots(pol,v)]

 radicalSolve(p:RF) ==
   zero? p =>
      error "equation is always satisfied"
   lv:=removeDuplicates
      concat(variables numer p, variables denom p)
   empty? lv => error "inconsistent equation"
   #lv>1 => error "too many variables"
   radicalSolve(p,lv.first)

 radicalSolve(eq: EQ RF) ==
   radicalSolve(lhs eq -rhs eq)

 radicalSolve(eq: EQ RF,v:SY) ==
    radicalSolve(lhs eq - rhs eq,v)
```

```
radicalRoots(lp: L RF,lv: L SY) ==
  parRes:=triangularSystems(lp,lv)$SystemSolvePackage(R)
  parRes= list [] => []
   -- select the components in "generic" form
  rlv:=reverse lv
  rpRes:=[reverse res for res in parRes]
  listGen:= [res for res in rpRes|isGeneric?(res,rlv)]
  result:L L RE:=[]
  if listGen^=[] then
    result:="append"/[findGenZeros(res,rlv) for res in listGen]
    for res in listGen repeat
        rpRes:=delete(rpRes,position(res,rpRes))
   --  non-generic components
  rpRes = [] => result
  append("append"/[findZeros(res,rlv) for res in rpRes],
                result)

radicalSolve(lp:L RF,lv:L SY) ==
  [makeEq(lres,lv) for lres in radicalRoots(lp,lv)]

radicalSolve(lp: L RF) ==
  lv:="setUnion"/[setUnion(variables numer p,variables denom p)
                  for p in lp]
  [makeEq(lres,lv) for lres in radicalRoots(lp,lv)]

radicalSolve(le:L EQ RF,lv:L SY) ==
  lp:=[rhs p -lhs p for p in le]
  [makeEq(lres,lv) for lres in radicalRoots(lp,lv)]

radicalSolve(le: L EQ RF) ==
  lp:=[rhs p -lhs p for p in le]
  lv:="setUnion"/[setUnion(variables numer p,variables denom p)
                  for p in lp]
  [makeEq(lres,lv) for lres in radicalRoots(lp,lv)]

contractSolve(eq:EQ RF, v:SY)==
   solveInner(numer(lhs eq - rhs eq), v, true)

contractSolve(pq:RF, v:SY) == solveInner(numer pq, v, true)

radicalRoots(pq:RF, v:SY) == lhs solveInner(numer pq, v, false)


-- test if the ideal is radical in generic position --
isGeneric?(rlp:L PR,rlv:L SY) : Boolean ==
  "and"/[degree(f,x)=1 for f in rest rlp  for x in rest rlv]

---- select  the univariate factors
select(lp:L L RE) : L L RE ==
  lp=[] => list []
```

```
      [:[cons(f,lsel) for lsel in select lp.rest] for f in lp.first]

    ---- Local Functions ----
   -- construct the equation
    makeEq(nres:L RE,lv:L SY) : L EQ RE ==
      [equation(x :: RE,r) for x in lv for r in nres]

    solveInner(pq:PR,v:SY,contractFlag:B) ==
        SideEquations := []
        ContractSoln  := contractFlag

        factors:= factors
           (factor pq)$MultivariateFactorize(SY,IndexedExponents SY,R,PR)

        constants:  List PR     := []
        unsolved:   List PR     := []
        solutions:  List RE     := []

        for f in factors repeat
            ff:=f.factor
            ^ member?(v, variables (ff)) =>
                constants := cons(ff, constants)
            u := univariate(ff, v)
            t := reduce u
            u := t.pol
            n := degree u
            l: List RE :=
                n = 1 => linear u
                n = 2 => quadratic u
                n = 3 => cubic u
                n = 4 => quartic u
                unsolved := cons(ff, unsolved)
                []
            for s in l repeat
                if t.deg > 1 then s := wrap s
                T0 := expand(s, t.deg)
                for i in 1..f.exponent repeat
                    solutions := append(T0, solutions)
                re := SideEquations
        [solutions, SideEquations]$SU

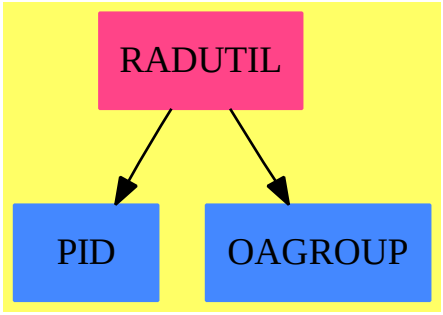

        ———-
```

— SOLVERAD.dotabb —

```
"SOLVERAD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVERAD"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"SOLVERAD" -> "ACFS"
```

19.3 package RADUTIL RadixUtilities

19.3.1 RadixUtilities (RADUTIL)



Exports:
radix

— package RADUTIL RadixUtilities —

```
)abbrev package RADUTIL RadixUtilities
++ Author: Stephen M. Watt
++ Date Created: October 1986
++ Date Last Updated: May 15, 1991
++ Basic Operations:
++ Related Domains: RadixExpansion
++ Also See:
++ AMS Classifications:
++ Keywords: radix, base, repeading decimal
++ Examples:
++ References:
++ Description:
++ This package provides tools for creating radix expansions.

RadixUtilities: Exports == Implementation where
  Exports ==> with
    radix: (Fraction Integer,Integer) -> Any
      ++ radix(x,b) converts x to a radix expansion in base b.
  Implementation ==> add
    radix(q, b) ==
      coerce(q :: RadixExpansion(b))$AnyFunctions1(RadixExpansion b)
```

———-

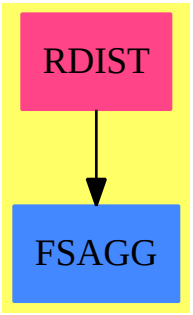
— RADUTIL.dotabb —

```
"RADUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RADUTIL"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"RADUTIL" -> "PID"
"RADUTIL" -> "OAGROUP"
```

———-

19.4 package RDIST RandomDistributions

19.4.1 RandomDistributions (RDIST)



Exports:
uniform rdHack1 weighted

— package RDIST RandomDistributions —

```
)abbrev package RDIST RandomDistributions
++ Description:
++ This package exports random distributions

RandomDistributions(S: SetCategory): with
        uniform:  Set S -> (() -> S)
          ++ uniform(s) \undocumented
        weighted: List Record(value: S, weight: Integer) -> (()->S)
          ++ weighted(l) \undocumented
        rdHack1:  (Vector S,Vector Integer,Integer)->(()->S)
          ++ rdHack1(v,u,n) \undocumented
    == add
```

```
        import RandomNumberSource()

        weighted lvw ==
            -- Collapse duplicates, adding weights.
            t: Table(S, Integer) := table()
            for r in lvw repeat
                u := search(r.value,t)
                w := (u case "failed" => 0; u::Integer)
                t r.value := w + r.weight

            -- Construct vectors of values and cumulative weights.
            kl := keys t
            n  := (#kl)::NonNegativeInteger
            n = 0 => error "Cannot select from empty set"
            kv: Vector(S)       := new(n, kl.0)
            wv: Vector(Integer) := new(n, 0)

            totwt: Integer := 0
            for k in kl for i in 1..n repeat
                kv.i := k
                totwt:= totwt + t k
                wv.i := totwt

            -- Function to generate an integer and lookup.
            rdHack1(kv, wv, totwt)

        rdHack1(kv, wv, totwt) ==
            w := randnum totwt
            -- do binary search in wv
            kv.1

        uniform fset ==
            l := members fset
            n := #l
            l.(randnum(n)+1)
```

———-

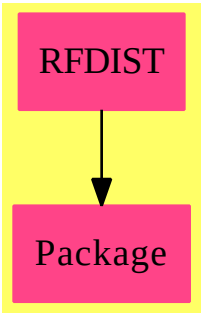
— RDIST.dotabb —

```
"RDIST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDIST"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"RDIST" -> "FSAGG"
```

———-

19.5 package RFDIST RandomFloatDistributions

19.5.1 RandomFloatDistributions (RFDIST)



Exports:

Beta	chiSquare	chiSquare1	exponential	exponential1
F	normal	t	normal01	uniform01
uniform				

— package RFDIST RandomFloatDistributions —

```
)abbrev package RFDIST RandomFloatDistributions
++ Description:
++ This package exports random floating-point distributions

RationalNumber==> Fraction Integer
RandomFloatDistributions(): Cat == Body where
    NNI ==> NonNegativeInteger

    Cat ==> with
        uniform01:   ()  -> Float
          ++ uniform01() \undocumented
        normal01:    ()  -> Float
          ++ normal01() \undocumented
        exponential1:()  -> Float
          ++ exponential1() \undocumented
        chiSquare1:  NNI -> Float
          ++ chiSquare1(n) \undocumented

        uniform:     (Float, Float) -> (() -> Float)
          ++ uniform(f,g) \undocumented
        normal:      (Float, Float) -> (() -> Float)
          ++ normal(f,g) \undocumented
        exponential: (Float)        -> (() -> Float)
          ++ exponential(f) \undocumented
        chiSquare:   (NNI)          -> (() -> Float)
          ++ chiSquare(n) \undocumented
```

```
        Beta:        (NNI, NNI)     -> (() -> Float)
          ++ Beta(n,m) \undocumented
        F:           (NNI, NNI)     -> (() -> Float)
          ++ F(n,m) \undocumented
        t:           (NNI)          -> (() -> Float)
          ++ t(n) \undocumented


    Body ==> add
        import RandomNumberSource()
--      FloatPackage0()

        -- random()  generates numbers in 0..rnmax
        rnmax := (size()$RandomNumberSource() - 1)::Float

        uniform01() ==
            randnum()::Float/rnmax
        uniform(a,b) ==
            a + uniform01()*(b-a)

        exponential1() ==
            u: Float := 0
            -- This test should really be  u < m where m is
            -- the minumum acceptible argument to log.
            while u = 0 repeat u := uniform01()
            - log u
        exponential(mean) ==
            mean*exponential1()

        -- This method is correct but slow.
        normal01() ==
            s := 2::Float
            while s >= 1 repeat
                v1 := 2 * uniform01() - 1
                v2 := 2 * uniform01() - 1
                s  := v1**2 + v2**2
            v1 * sqrt(-2 * log s/s)
        normal(mean, stdev) ==
            mean + stdev*normal01()

        chiSquare1 dgfree ==
            x: Float := 0
            for i in 1..dgfree quo 2 repeat
                x := x + 2*exponential1()
            if odd? dgfree then
                x := x + normal01()**2
            x
        chiSquare dgfree ==
            chiSquare1 dgfree
```

```
Beta(dgfree1, dgfree2) ==
    y1 := chiSquare1 dgfree1
    y2 := chiSquare1 dgfree2
    y1/(y1 + y2)

F(dgfree1, dgfree2) ==
    y1 := chiSquare1 dgfree1
    y2 := chiSquare1 dgfree2
    (dgfree2 * y1)/(dgfree1 * y2)

t dgfree ==
    n := normal01()
    d := chiSquare1(dgfree) / (dgfree::Float)
    n / sqrt d
```

———-

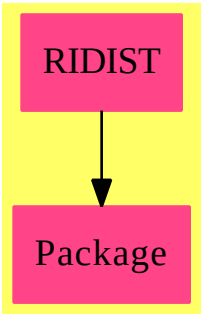
— **RFDIST.dotabb** —

```
"RFDIST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFDIST"]
"Package" [color="#FF4488"]
"RFDIST" -> "Package"
```

———-

19.6 package RIDIST RandomIntegerDistributions

19.6.1 RandomIntegerDistributions (RIDIST)



Exports:
binomial geometric poisson ridHack1 uniform

— **package RIDIST RandomIntegerDistributions** —

```
)abbrev package RIDIST RandomIntegerDistributions
++ Description:
++ This package exports integer distributions

RandomIntegerDistributions(): with
   uniform:   Segment Integer           -> (() -> Integer)
     ++ uniform(s) as
     ++     l + u0 + w*u1 + w**2*u2 +...+ w**(n-1)*u-1 + w**n*m
     ++ where
     ++     s = a..b
     ++     l = min(a,b)
     ++     m = abs(b-a) + 1
     ++     w**n < m < w**(n+1)
     ++     u0,...,un-1  are uniform on  0..w-1
     ++     m            is  uniform on  0..(m quo w**n)-1
   binomial:  (Integer, RationalNumber) -> (() -> Integer)
     ++ binomial(n,f) \undocumented
   poisson:   RationalNumber          -> (() -> Integer)
     ++ poisson(f) \undocumented
   geometric: RationalNumber          -> (() -> Integer)
     ++ geometric(f) \undocumented
   ridHack1:  (Integer,Integer,Integer,Integer) -> Integer
     ++ ridHack1(i,j,k,l) \undocumented
 == add
   import RandomNumberSource()
   import IntegerBits()

   uniform aTob ==
       a := lo aTob;  b := hi aTob
       l := min(a,b); m := abs(a-b) + 1

       w := 2**(bitLength size() quo 2)::NonNegativeInteger

       n  := 0
       mq := m  -- m quo w**n
       while (mqnext := mq quo w) > 0 repeat
           n  := n + 1
           mq := mqnext
       ridHack1(mq, n, w, l)

   ridHack1(mq, n, w, l) ==
       r := randnum mq
       for i in 1..n repeat r := r*w + randnum w
       r + l
```

———-

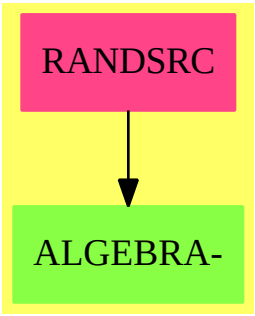
— RIDIST.dotabb —

```
"RIDIST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RIDIST"]
"Package" [color="#FF4488"]
"RIDIST" -> "Package"
```

———-

19.7 package RANDSRC RandomNumberSource

19.7.1 RandomNumberSource (RANDSRC)



Exports:
randnum reseed seed size

— package RANDSRC RandomNumberSource —

```
)abbrev package RANDSRC RandomNumberSource
++ Author:S.M.Watt
++ Date Created: April 87
++ Date Last Updated:Jan 92, May 1995 (MCD)
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ Random number generators.
++ All random numbers used in the system should originate from
++ the same generator.  This package is intended to be the source.
--
--  Possible improvements:
--  1) Start where the user left off
--  2) Be able to switch between methods in the random number source.
```

```
RandomNumberSource(): with
    -- If r := randnum() then  0 <= r < size().
        randnum:  () -> Integer
           ++ randnum() is a random number between 0 and size().
    --   If r := randnum() then  0 <= r < size().
        size:     () -> Integer
          ++ size() is the base of the random number generator

        -- If r := randnum n and n <= size()  then  0 <= r < n.
        randnum:  Integer -> Integer
           ++ randnum(n) is a random number between 0 and n.
        reseed:   Integer -> Void
           ++ reseed(n) restarts the random number generator at n.
        seed : () -> Integer
           ++ seed() returns the current seed value.

    == add
        -- This random number generator passes the spectral test
        -- with flying colours. [Knuth vol2, 2nd ed, p105]
        ranbase: Integer := 2**31-1
        x0:   Integer := 1231231231
        x1:   Integer := 3243232987

        randnum() ==
            t := (271828183 * x1 - 314159269 * x0) rem ranbase
            if t < 0 then t := t + ranbase
            x0:= x1
            x1:= t

        size() == ranbase
        reseed n ==
            x0 := n rem ranbase
            -- x1 := (n quo ranbase) rem ranbase
            x1 := n quo ranbase

        seed() == x1*ranbase + x0

        -- Compute an integer in 0..n-1.
        randnum n ==
            (n * randnum()) quo ranbase
```

———-

— RANDSRC.dotabb —

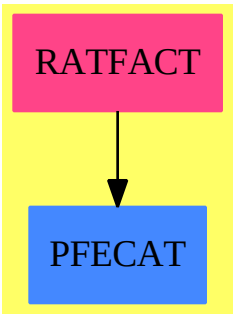
```
"RANDSRC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RANDSRC"]
"ALGEBRA-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALGEBRA"]
```

```
"RANDSRC" -> "ALGEBRA-"
```

———-

19.8 package RATFACT RationalFactorize

19.8.1 RationalFactorize (RATFACT)



Exports:
factor factorSquareFree

— package RATFACT RationalFactorize —

```
)abbrev package RATFACT RationalFactorize
++ Author: P. Gianni
++ Date created: ??
++ Date last updated: December 1993
++ Keywords: factorization, hensel, rational number
++ Description:
++ Factorization of extended polynomials with rational coefficients.
++ This package implements factorization of extended polynomials
++ whose coefficients are rational numbers. It does this by taking the
++ lcm of the coefficients of the polynomial and creating a polynomial
++ with integer coefficients. The algorithm in
++ \spadtype{GaloisGroupFactorizer} is then
++ used to factor the integer polynomial. The result is normalized
++ with respect to the original lcm of the denominators.


RationalFactorize(RP) : public == private where
  I  ==> Integer
  RN ==> Fraction Integer
  BP ==> SparseUnivariatePolynomial(I)
  RP : UnivariatePolynomialCategory RN
```

```
  public   ==> with

     factor           : RP ->  Factored RP
      ++ factor(p) factors an extended polynomial p over the rational numbers.
     factorSquareFree : RP -> Factored RP
      ++ factorSquareFree(p) factors an extended squareFree
      ++ polynomial p over the rational numbers.

  private  ==> add
     import GaloisGroupFactorizer (BP)
     ParFact   ==> Record(irr:BP,pow:I)
     FinalFact ==> Record(contp:I,factors:List(ParFact))
     URNI      ==> UnivariatePolynomialCategoryFunctions2(RN,RP,I,BP)
     UIRN      ==> UnivariatePolynomialCategoryFunctions2(I,BP,RN,RP)
     fUnion    ==> Union("nil", "sqfr", "irred", "prime")
     FFE       ==> Record(flg:fUnion, fctr:RP, xpnt:I)

     factor(p:RP) : Factored(RP) ==
       p = 0 => 0
       pden: I := lcm([denom c for c in coefficients p])
       pol : RP := pden*p
       ipol: BP := map(numer,pol)$URNI
       ffact: FinalFact := henselFact(ipol,false)
       makeFR(((ffact.contp)/pden)::RP,
         [["prime",map(coerce,u.irr)$UIRN,u.pow]$FFE
                              for u in ffact.factors])

     factorSquareFree(p:RP) : Factored(RP) ==
       p = 0 => 0
       pden: I := lcm([denom c for c in coefficients p])
       pol : RP := pden*p
       ipol: BP := map(numer,pol)$URNI
       ffact: FinalFact := henselFact(ipol,true)
       makeFR(((ffact.contp)/pden)::RP,
         [["prime",map(coerce,u.irr)$UIRN,u.pow]$FFE
                              for u in ffact.factors])
```

———-

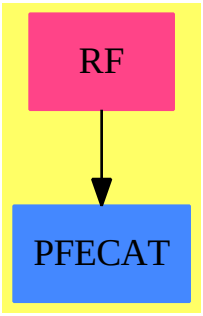
— RATFACT.dotabb —

```
"RATFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RATFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RATFACT" -> "PFECAT"
```

———-

19.9 package RF RationalFunction

19.9.1 RationalFunction (RF)



Exports:
coerce eval mainVariable multivariate univariate variables

— package RF RationalFunction —

```
)abbrev package RF RationalFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 18 April 1991
++ Keywords: polynomial, fraction
++ Description:
++ Utilities that provide the same top-level manipulations on
++ fractions than on polynomials.
-- Do not make into a domain!

RationalFunction(R:IntegralDomain): Exports == Implementation where
  V  ==> Symbol
  P  ==> Polynomial R
  Q  ==> Fraction P
  QF ==> PolynomialCategoryQuotientFunctions(IndexedExponents Symbol,
                                                    Symbol, R, P, Q)

  Exports ==> with
    variables   : Q -> List V
      ++ variables(f) returns the list of variables appearing
      ++ in the numerator or the denominator of f.
    mainVariable: Q -> Union(V, "failed")
      ++ mainVariable(f) returns the highest variable appearing
      ++ in the numerator or the denominator of f, "failed" if
      ++ f has no variables.
    univariate  : (Q, V) -> Fraction SparseUnivariatePolynomial Q
      ++ univariate(f, v) returns f viewed as a univariate
```

```
    ++ rational function in v.
  multivariate: (Fraction SparseUnivariatePolynomial Q, V) -> Q
    ++ multivariate(f, v) applies both the numerator and
    ++ denominator of f to v.
  eval        : (Q, V, Q) -> Q
    ++ eval(f, v, g) returns f with v replaced by g.
  eval        : (Q, List V, List Q) -> Q
    ++ eval(f, [v1,...,vn], [g1,...,gn]) returns f with
    ++ each vi replaced by gi in parallel, i.e. vi's appearing
    ++ inside the gi's are not replaced.
  eval        : (Q, Equation Q) -> Q
    ++ eval(f, v = g) returns f with v replaced by g.
    ++ Error: if v is not a symbol.
  eval        : (Q, List Equation Q) -> Q
    ++ eval(f, [v1 = g1,...,vn = gn]) returns f with
    ++ each vi replaced by gi in parallel, i.e. vi's appearing
    ++ inside the gi's are not replaced.
    ++ Error: if any vi is not a symbol.
  coerce      : R -> Q
    ++ coerce(r) returns r viewed as a rational function over R.

Implementation ==> add
  foo  : (List V, List Q, V) -> Q
  peval: (P, List V, List Q) -> Q

  coerce(r:R):Q            == r::P::Q
  variables f              == variables(f)$QF
  mainVariable f           == mainVariable(f)$QF
  univariate(f, x)         == univariate(f, x)$QF
  multivariate(f, x)       == multivariate(f, x)$QF
  eval(x:Q, s:V, y:Q)      == eval(x, [s], [y])
  eval(x:Q, eq:Equation Q) == eval(x, [eq])
  foo(ls, lv, x)           == match(ls, lv, x, x::Q)$ListToMap(V, Q)

  eval(x:Q, l:List Equation Q) ==
    eval(x, [retract(lhs eq)@V for eq in l]$List(V),
            [rhs eq for eq in l]$List(Q))

  eval(x:Q, ls:List V, lv:List Q) ==
    peval(numer x, ls, lv) / peval(denom x, ls, lv)

  peval(p, ls, lv) ==
    map(z1 +-> foo(ls, lv, z1), z2 +-> z2::Q,p)
     $PolynomialCategoryLifting(IndexedExponents V,V,R,P,Q)
```

———-

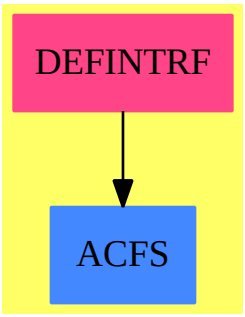
— RF.dotabb —

```
"RF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RF" -> "PFECAT"
```

———-

19.10 package DEFINTRF RationalFunctionDefiniteIntegration

19.10.1 RationalFunctionDefiniteIntegration (DEFINTRF)



Exports:
integrate

— package DEFINTRF RationalFunctionDefiniteIntegration —

```
)abbrev package DEFINTRF RationalFunctionDefiniteIntegration
++ Author: Manuel Bronstein
++ Date Created: 2 October 1989
++ Date Last Updated: 2 February 1993
++ Description:
++ Definite integration of rational functions.
++ \spadtype{RationalFunctionDefiniteIntegration} provides functions to
++ compute definite integrals of rational functions.

RationalFunctionDefiniteIntegration(R): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)

  SE  ==> Symbol
  RF  ==> Fraction Polynomial R
  FE  ==> Expression R
  ORF ==> OrderedCompletion RF
```

```
  OFE ==> OrderedCompletion FE
  U   ==> Union(f1:OFE, f2:List OFE, fail:"failed", pole:"potentialPole")

  Exports ==> with
    integrate: (RF, SegmentBinding OFE) -> U
      ++ integrate(f, x = a..b) returns the integral of
      ++ \spad{f(x)dx} from a to b.
      ++ Error: if f has a pole for x between a and b.
    integrate: (RF, SegmentBinding OFE, String) -> U
      ++ integrate(f, x = a..b, "noPole") returns the
      ++ integral of \spad{f(x)dx} from a to b.
      ++ If it is not possible to check whether f has a pole for x
      ++ between a and b (because of parameters), then this function
      ++ will assume that f has no such pole.
      ++ Error: if f has a pole for x between a and b or
      ++ if the last argument is not "noPole".
-- the following two are contained in the above, but they are for the
-- interpreter... DO NOT COMMENT OUT UNTIL THE INTERPRETER IS BETTER!
    integrate: (RF, SegmentBinding ORF) -> U
      ++ integrate(f, x = a..b) returns the integral of
      ++ \spad{f(x)dx} from a to b.
      ++ Error: if f has a pole for x between a and b.
    integrate: (RF, SegmentBinding ORF, String) -> U
      ++ integrate(f, x = a..b, "noPole") returns the
      ++ integral of \spad{f(x)dx} from a to b.
      ++ If it is not possible to check whether f has a pole for x
      ++ between a and b (because of parameters), then this function
      ++ will assume that f has no such pole.
      ++ Error: if f has a pole for x between a and b or
      ++ if the last argument is not "noPole".

  Implementation ==> add
    import DefiniteIntegrationTools(R, FE)
    import IntegrationResultRFToFunction(R)
    import OrderedCompletionFunctions2(RF, FE)

    int   : (RF, SE, OFE, OFE, Boolean) -> U
    nopole: (RF, SE, OFE, OFE) -> U

    integrate(f:RF, s:SegmentBinding OFE) ==
      int(f, variable s, lo segment s, hi segment s, false)

    nopole(f, x, a, b) ==
      k := kernel(x)@Kernel(FE)
      (u := integrate(f, x)) case FE =>
        (v := computeInt(k, u::FE, a, b, true)) case "failed" => ["failed"]
        [v::OFE]
      ans := empty()$List(OFE)
      for g in u::List(FE) repeat
        (v := computeInt(k, g, a, b, true)) case "failed" => return ["failed"]
```

```
        ans := concat_!(ans, [v::OFE])
      [ans]

    integrate(f:RF, s:SegmentBinding ORF) ==
      int(f, variable s, map(x +-> x::FE, lo segment s),
                         map(x +-> x::FE, hi segment s), false)

    integrate(f:RF, s:SegmentBinding ORF, str:String) ==
      int(f, variable s, map(x +-> x::FE, lo segment s),
                         map(x +-> x::FE, hi segment s), ignore? str)

    integrate(f:RF, s:SegmentBinding OFE, str:String) ==
      int(f, variable s, lo segment s, hi segment s, ignore? str)

    int(f, x, a, b, ignor?) ==
      a = b => [0::OFE]
      (z := checkForZero(denom f, x, a, b, true)) case "failed" =>
        ignor? => nopole(f, x, a, b)
        ["potentialPole"]
      z::Boolean => error "integrate: pole in path of integration"
      nopole(f, x, a, b)
```

———-

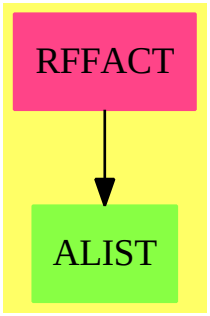
— **DEFINTRF.dotabb** —

```
"DEFINTRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEFINTRF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"DEFINTRF" -> "ACFS"
```

———-

19.11 package RFFACT RationalFunctionFactor

19.11.1 RationalFunctionFactor (RFFACT)



Exports:
factor

— package RFFACT RationalFunctionFactor —

```
)abbrev package RFFACT RationalFunctionFactor
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: ???
++ Description:
++ Factorization of univariate polynomials with coefficients which
++ are rational functions with integer coefficients.

RationalFunctionFactor(UP): Exports == Implementation where
  UP: UnivariatePolynomialCategory Fraction Polynomial Integer

  SE ==> Symbol
  P  ==> Polynomial Integer
  RF ==> Fraction P
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2

  Exports ==> with
    factor: UP -> Factored UP
      ++ factor(p) returns a prime factorisation of p.

  Implementation ==> add
    likuniv: (P, SE, P) -> UP

    dummy := new()$SE

    likuniv(p, x, d) ==
      map(y +-> y/d, univariate(p, x))$UPCF2(P,SparseUnivariatePolynomial P,
```

```
                                        RF, UP)

    factor p ==
      d  := denom(q := elt(p,dummy::P :: RF))
      map(x +-> likuniv(x,dummy,d),
          factor(numer q)$MultivariateFactorize(SE,
               IndexedExponents SE,Integer,P))$FactoredFunctions2(P, UP)
```

———-

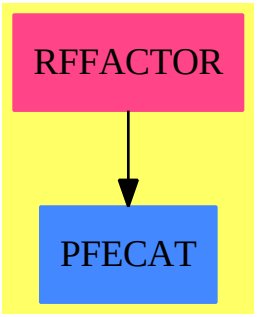
— **RFFACT.dotabb** —

```
"RFFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFFACT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"RFFACT" -> "ALIST"
```

———-

19.12 package RFFACTOR RationalFunctionFactorizer

19.12.1 RationalFunctionFactorizer (RFFACTOR)



Exports:
factor

— **package RFFACTOR RationalFunctionFactorizer** —

```
)abbrev package RFFACTOR RationalFunctionFactorizer
++ Author: P. Gianni
++ Date Created:
++ Date Last Updated: March 1995
++ Basic Functions:
```

```
++ Related Constructors: Fraction, Polynomial
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \spadtype{RationalFunctionFactorizer} contains the factor function
++ (called factorFraction) which factors fractions of polynomials by factoring
++ the numerator and denominator. Since any non zero fraction is a unit
++ the usual factor operation will just return the original fraction.

RationalFunctionFactorizer(R) : C == T
 where
  R  :    EuclideanDomain  -- R with factor for R[X]
  P  ==>  Polynomial R
  FP ==>  Fraction P
  SE ==>  Symbol

  C  == with
     factorFraction    : FP  ->   Fraction Factored(P)
       ++ factorFraction(r) factors the numerator and the denominator of
       ++ the polynomial fraction r.
  T  == add

     factorFraction(p:FP) : Fraction Factored(P) ==
       R is Fraction Integer =>
         MR:=MRationalFactorize(IndexedExponents SE,SE,
                                Integer,P)
         (factor(numer p)$MR)/ (factor(denom p)$MR)

       R has FiniteFieldCategory =>
         FF:=MultFiniteFactorize(SE,IndexedExponents SE,R,P)
         (factor(numer p))$FF/(factor(denom p))$FF

       R has CharacteristicZero =>
          MFF:=MultivariateFactorize(SE,IndexedExponents SE,R,P)
          (factor(numer p))$MFF/(factor(denom p))$MFF
       error "case not handled"
```

———-

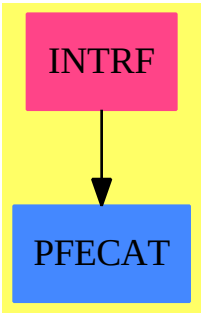
— RFFACTOR.dotabb —

```
"RFFACTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFFACTOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RFFACTOR" -> "PFECAT"
```

———-

19.13 package INTRF RationalFunctionIntegration

19.13.1 RationalFunctionIntegration (INTRF)



Exports:
extendedIntegrate infieldIntegrate internalIntegrate limitedIntegrate

— package INTRF RationalFunctionIntegration —

```
)abbrev package INTRF RationalFunctionIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 29 Mar 1990
++ Keywords: polynomial, fraction, integration.
++ Description:
++ This package provides functions for the integration of rational functions.

RationalFunctionIntegration(F): Exports == Implementation where
  F: Join(IntegralDomain, RetractableTo Integer, CharacteristicZero)

  SE  ==> Symbol
  P   ==> Polynomial F
  Q   ==> Fraction P
  UP  ==> SparseUnivariatePolynomial Q
  QF  ==> Fraction UP
  LGQ ==> List Record(coeff:Q, logand:Q)
  UQ  ==> Union(Record(ratpart:Q, coeff:Q), "failed")
  ULQ ==> Union(Record(mainpart:Q, limitedlogs:LGQ), "failed")

  Exports ==> with
    internalIntegrate: (Q, SE) -> IntegrationResult Q
       ++ internalIntegrate(f, x) returns g such that \spad{dg/dx = f}.
    infieldIntegrate : (Q, SE) -> Union(Q, "failed")
       ++ infieldIntegrate(f, x) returns a fraction
       ++ g such that \spad{dg/dx = f}
       ++ if g exists, "failed" otherwise.
```

```
    limitedIntegrate : (Q, SE, List Q) -> ULQ
       ++ \spad{limitedIntegrate(f, x, [g1,...,gn])} returns fractions
       ++ \spad{[h, [[ci,gi]]]} such that the gi's are among
       ++ \spad{[g1,...,gn]},
       ++ \spad{dci/dx = 0}, and  \spad{d(h + sum(ci log(gi)))/dx = f}
       ++ if possible, "failed" otherwise.
    extendedIntegrate: (Q, SE, Q) -> UQ
       ++ extendedIntegrate(f, x, g) returns fractions \spad{[h, c]} such that
       ++ \spad{dc/dx = 0} and \spad{dh/dx = f - cg}, if \spad{(h, c)} exist,
       ++ "failed" otherwise.

  Implementation ==> add
    import RationalIntegration(Q, UP)
    import IntegrationResultFunctions2(QF, Q)
    import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
                                                SE, F, P, Q)

    infieldIntegrate(f, x) ==
      map(x1 +-> multivariate(x1, x), infieldint univariate(f, x))

    internalIntegrate(f, x) ==
      map(x1 +-> multivariate(x1, x), integrate univariate(f, x))

    extendedIntegrate(f, x, g) ==
      map(x1 +-> multivariate(x1, x),
          extendedint(univariate(f, x), univariate(g, x)))

    limitedIntegrate(f, x, lu) ==
      map(x1 +-> multivariate(x1, x),
          limitedint(univariate(f, x), [univariate(u, x) for u in lu]))
```

———-

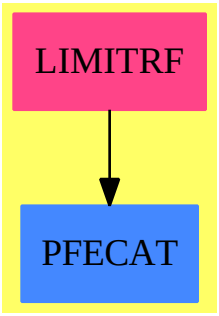
— INTRF.dotabb —

```
"INTRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTRF" -> "PFECAT"
```

———-

19.14 package LIMITRF RationalFunctionLimitPackage

19.14.1 RationalFunctionLimitPackage (LIMITRF)



Exports:
complexLimit limit

— package LIMITRF RationalFunctionLimitPackage —

```
)abbrev package LIMITRF RationalFunctionLimitPackage
++ Author: Manuel Bronstein
++ Date Created: 4 October 1989
++ Date Last Updated: 26 November 1991
++ Keywords: limit, rational function.
++ Description:
++ Computation of limits for rational functions.

RationalFunctionLimitPackage(R:GcdDomain):Exports==Implementation where
  Z       ==> Integer
  P       ==> Polynomial R
  RF      ==> Fraction P
  EQ      ==> Equation
  ORF     ==> OrderedCompletion RF
  OPF     ==> OnePointCompletion RF
  UP      ==> SparseUnivariatePolynomial RF
  SE      ==> Symbol
  QF      ==> Fraction SparseUnivariatePolynomial RF
  Result  ==> Union(ORF, "failed")
  TwoSide ==> Record(leftHandLimit:Result, rightHandLimit:Result)
  U       ==> Union(ORF, TwoSide, "failed")
  RFSGN   ==> RationalFunctionSign(R)

  Exports ==> with
-- The following are the one we really want, but the interpreter cannot
-- handle them...
--  limit: (RF,EQ ORF) -> U
```

```
--  ++ limit(f(x),x,a) computes the real two-sided limit lim(x -> a,f(x))

--  complexLimit: (RF,EQ OPF) -> OPF
--  ++ complexLimit(f(x),x,a) computes the complex limit lim(x -> a,f(x))

-- ... so we replace them by the following 4:
    limit: (RF,EQ OrderedCompletion P) -> U
      ++ limit(f(x),x = a) computes the real two-sided limit
      ++ of f as its argument x approaches \spad{a}.
    limit: (RF,EQ RF) -> U
      ++ limit(f(x),x = a) computes the real two-sided limit
      ++ of f as its argument x approaches \spad{a}.
    complexLimit: (RF,EQ OnePointCompletion P) -> OPF
      ++ \spad{complexLimit(f(x),x = a)} computes the complex limit
      ++ of \spad{f} as its argument x approaches \spad{a}.
    complexLimit: (RF,EQ RF) -> OPF
      ++ complexLimit(f(x),x = a) computes the complex limit
      ++ of f as its argument x approaches \spad{a}.
    limit: (RF,EQ RF,String) -> Result
      ++ limit(f(x),x,a,"left") computes the real limit
      ++ of f as its argument x approaches \spad{a} from the left;
      ++ limit(f(x),x,a,"right") computes the corresponding limit as x
      ++ approaches \spad{a} from the right.

  Implementation ==> add
    import ToolsForSign R
    import InnerPolySign(RF, UP)
    import RFSGN
    import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
                                                    SE, R, P, RF)

    finiteComplexLimit: (QF, RF) -> OPF
    finiteLimit       : (QF, RF) -> U
    fLimit            : (Z, UP, RF, Z) -> Result

-- These 2 should be exported, see comment above
    locallimit       : (RF, SE, ORF) -> U
    locallimitcomplex: (RF, SE, OPF) -> OPF

    limit(f:RF,eq:EQ RF) ==
      (xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
        error "limit: left hand side must be a variable"
      x := xx :: SE; a := rhs eq
      locallimit(f,x,a::ORF)

    complexLimit(f:RF,eq:EQ RF) ==
      (xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
        error "limit: left hand side must be a variable"
      x := xx :: SE; a := rhs eq
      locallimitcomplex(f,x,a::OPF)
```

```
limit(f:RF,eq:EQ OrderedCompletion P) ==
  (p := retractIfCan(lhs eq)@Union(P,"failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  (xx := retractIfCan(p)@Union(SE,"failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE
  a := map(y +-> y::RF,rhs eq)$OrderedCompletionFunctions2(P,RF)
  locallimit(f,x,a)

complexLimit(f:RF,eq:EQ OnePointCompletion P) ==
  (p := retractIfCan(lhs eq)@Union(P,"failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  (xx := retractIfCan(p)@Union(SE,"failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE
  a := map(y +-> y::RF,rhs eq)$OnePointCompletionFunctions2(P,RF)
  locallimitcomplex(f,x,a)

fLimit(n, d, a, dir) ==
  (s := signAround(d, a, dir, sign$RFSGN)) case "failed" => "failed"
  n * (s::Z) * plusInfinity()

finiteComplexLimit(f, a) ==
  zero?(n := (numer f) a) => 0
  zero?(d := (denom f) a) => infinity()
  (n / d)::OPF

finiteLimit(f, a) ==
  zero?(n := (numer f) a) => 0
  zero?(d := (denom f) a) =>
    (s := sign(n)$RFSGN) case "failed" => "failed"
    rhsl := fLimit(s::Z, denom f, a, 1)
    lhsl := fLimit(s::Z, denom f, a, -1)
    rhsl case "failed" =>
      lhsl case "failed" => "failed"
      [lhsl, rhsl]
    lhsl case "failed" => [lhsl, rhsl]
    rhsl::ORF = lhsl::ORF => lhsl::ORF
    [lhsl, rhsl]
  (n / d)::ORF

locallimit(f,x,a) ==
  g := univariate(f, x)
  zero?(n := whatInfinity a) => finiteLimit(g, retract a)
  (dn := degree numer g) > (dd := degree denom g) =>
    (sn := signAround(numer g, n, sign$RFSGN)) case "failed" => "failed"
    (sd := signAround(denom g, n, sign$RFSGN)) case "failed" => "failed"
    (sn::Z) * (sd::Z) * plusInfinity()
  dn < dd => 0
```

```
      ((leadingCoefficient numer g) / (leadingCoefficient denom g))::ORF

    limit(f,eq,st) ==
      (xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
        error "limit: left hand side must be a variable"
      x := xx :: SE; a := rhs eq
      zero?(n := (numer(g := univariate(f, x))) a) => 0
      zero?(d := (denom g) a) =>
        (s := sign(n)$RFSGN) case "failed" => "failed"
        fLimit(s::Z, denom g, a, direction st)
      (n / d)::ORF

    locallimitcomplex(f,x,a) ==
      g := univariate(f, x)
      (r := retractIfCan(a)@Union(RF, "failed")) case RF =>
        finiteComplexLimit(g, r::RF)
      (dn := degree numer g) > (dd := degree denom g) => infinity()
      dn < dd => 0
      ((leadingCoefficient numer g) / (leadingCoefficient denom g))::OPF
```

———-

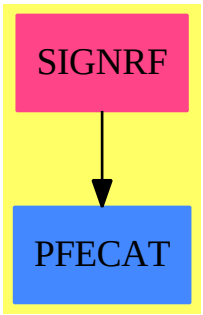
— **LIMITRF.dotabb** —

```
"LIMITRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIMITRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LIMITRF" -> "PFECAT"
```

———-

19.15 package SIGNRF RationalFunctionSign

19.15.1 RationalFunctionSign (SIGNRF)



Exports:
sign

— package SIGNRF RationalFunctionSign —

```
)abbrev package SIGNRF RationalFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 23 August 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Find the sign of a rational function around a point or infinity.

RationalFunctionSign(R:GcdDomain): Exports == Implementation where
  SE  ==> Symbol
  P   ==> Polynomial R
  RF  ==> Fraction P
  ORF ==> OrderedCompletion RF
  UP  ==> SparseUnivariatePolynomial RF
  U   ==> Union(Integer, "failed")
  SGN ==> ToolsForSign(R)

  Exports ==> with
    sign: RF -> U
      ++ sign f returns the sign of f if it is constant everywhere.
    sign: (RF, SE, ORF) -> U
      ++ sign(f, x, a) returns the sign of f as x approaches \spad{a},
      ++ from both sides if \spad{a} is finite.
    sign: (RF, SE, RF, String) -> U
      ++ sign(f, x, a, s) returns the sign of f as x nears \spad{a} from
      ++ the left (below) if s is the string \spad{"left"},
      ++ or from the right (above) if s is the string \spad{"right"}.

  Implementation ==> add
    import SGN
    import InnerPolySign(RF, UP)
    import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
                                                SE, R, P, RF)

    psign     : P -> U
    sqfrSign  : P -> U
    termSign  : P -> U
    listSign  : (List P, Integer) -> U
    finiteSign: (Fraction UP, RF) -> U

    sign f ==
      (un := psign numer f) case "failed" => "failed"
      (ud := psign denom f) case "failed" => "failed"
      (un::Integer) * (ud::Integer)
```

```
    finiteSign(g, a) ==
      (ud := signAround(denom g, a, sign$%)) case "failed" => "failed"
      (un := signAround(numer g, a, sign$%)) case "failed" => "failed"
      (un::Integer) * (ud::Integer)

    sign(f, x, a) ==
      g := univariate(f, x)
      zero?(n := whatInfinity a) => finiteSign(g, retract a)
      (ud := signAround(denom g, n, sign$%)) case "failed" => "failed"
      (un := signAround(numer g, n, sign$%)) case "failed" => "failed"
      (un::Integer) * (ud::Integer)

    sign(f, x, a, st) ==
      (ud := signAround(denom(g := univariate(f, x)), a,
                   d := direction st, sign$%)) case "failed" => "failed"
      (un := signAround(numer g, a, d, sign$%)) case "failed" => "failed"
      (un::Integer) * (ud::Integer)

    psign p ==
      (r := retractIfCan(p)@Union(R, "failed")) case R => sign(r::R)$SGN
      (u := sign(retract(unit(s := squareFree p))@R)$SGN) case "failed" =>
        "failed"
      ans := u::Integer
      for term in factors s | odd?(term.exponent) repeat
        (u := sqfrSign(term.factor)) case "failed" => return "failed"
        ans := ans * (u::Integer)
      ans

    sqfrSign p ==
      (u := termSign first(l := monomials p)) case "failed" => "failed"
      listSign(rest l, u::Integer)

    listSign(l, s) ==
      for term in l repeat
        (u := termSign term) case "failed" => return "failed"
        u::Integer ^= s => return "failed"
      s

    termSign term ==
      for var in variables term repeat
        odd? degree(term, var) => return "failed"
      sign(leadingCoefficient term)$SGN
```

———-

— SIGNRF.dotabb —

```
"SIGNRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIGNRF"]
```

```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SIGNRF" -> "PFECAT"
```

———-

19.16 package SUMRF RationalFunctionSum

— **RationalFunctionSum.input** —

```
)set break resume
)sys rm -f RationalFunctionSum.output
)spool RationalFunctionSum.output
)set message test on
)set message auto off
)clear all

--S 1 of 13
sum(i::Polynomial(Integer),variable(i=1..n))
--R
--R
--R          2
--R        6i  - 6i + 1
--R   (1)  ------------
--R             12
--R                                          Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 13
sum(i::Fraction(Polynomial(Integer)),i::Symbol)
--R
--R
--R          2
--R        6i  - 6i + 1
--R   (2)  ------------
--R             12
--R                                Type: Union(Fraction(Polynomial(Integer)),...)
--E 2

--S 3 of 13
sum(i,i=1..n)
--R
--R
--R         2
--R        n  + n
--R   (3)  ------
--R           2
```

```
--R                                          Type: Fraction(Polynomial(Integer))
--E 3

--S 4 of 13
sum(i::Fraction(Polynomial(Integer)),i=1..n)
--R
--R
--R         2
--R        n  + n
--R   (4)  ------
--R          2
--R                                Type: Union(Fraction(Polynomial(Integer)),...)
--E 4

--S 5 of 13
s:=i=1..n
--R
--R
--R   (5)  i= 1..n
--R                                     Type: SegmentBinding(Polynomial(Integer))
--E 5

--S 6 of 13
hiseg:=high(segment(s))
--R
--R
--R   (6)  n
--R                                                  Type: Polynomial(Integer)
--E 6

--S 7 of 13
loseg:=low(segment(s))
--R
--R
--R   (7)  1
--R                                                  Type: Polynomial(Integer)
--E 7

--S 8 of 13
v:=variable s
--R
--R
--R   (8)  i
--R                                                               Type: Symbol
--E 8

--S 9 of 13
p:=i::Polynomial(Integer)
--R
--R
```

```
--R   (9)  i
--R                                                  Type: Polynomial(Integer)
--E 9

--S 10 of 13
f:=sum(p,v)
--R
--R
--R           2
--R         6i  - 6i + 1
--R   (10)  ------------
--R              12
--R                                        Type: Fraction(Polynomial(Integer))
--E 10

--S 11 of 13
t1:=eval(f,v,(1+hiseg))
--R
--R
--R           2
--R         6n  + 6n + 1
--R   (11)  ------------
--R              12
--R                                        Type: Fraction(Polynomial(Integer))
--E 11

--S 12 of 13
t2:=eval(f,v,loseg)
--R
--R
--R          1
--R   (12)  --
--R         12
--R                                        Type: Fraction(Polynomial(Integer))
--E 12

--S 13 of 13
t1-t2
--R
--R
--R          2
--R         n  + n
--R   (13)  ------
--R            2
--R                                        Type: Fraction(Polynomial(Integer))
--E 13

)spool
```

———-

— RationalFunctionSum.help —

```
=====================================================================
RationalFunctionSum examples
=====================================================================

There are 4 different forms of the sum operator in this domain.
They are variations of the types of the input

   sum(i::Polynomial(Integer),variable(i=1..n))

          2
       6i  - 6i + 1
       ------------
            12


   sum(i::Fraction(Polynomial(Integer)),i::Symbol)

          2
       6i  - 6i + 1
       ------------
            12


   sum(i,i=1..n)

        2
       n  + n
       ------
          2


   sum(i::Fraction(Polynomial(Integer)),i=1..n)

        2
       n  + n
       ------
          2


We can compute the sum form on part at a time to see what happens.
First, we make a SegmentBinding(Polynomial(Integer)):

   s:=i=1..n

        i= 1..n
```

```
We pick out the upper and lower bounds of the segment:

   hiseg:=high(segment(s))

        n


   loseg:=low(segment(s))

        1


We pick out the variable from the segment:

   v:=variable s

        i


We create a polynomial we wish to sum:

   p:=i::Polynomial(Integer)

         i


And we create the sum for that polynomial:

   f:=sum(p,v)

           2
         6i  - 6i + 1
         ------------
              12


The new evaluate it at the upper endpoint:

   t1:=eval(f,v,(1+hiseg))

           2
         6n  + 6n + 1
         ------------
              12

And the lower endpoint:

   t2:=eval(f,v,loseg)
```

```
         1
        --
        12
```

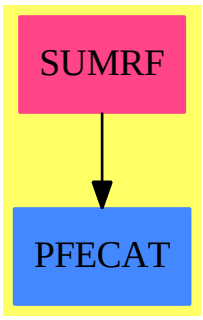
And we take the difference of the endpoints:

```
   t1-t2

         2
        n  + n
        ------
           2
```

See Also:
o)show RationalFunctionSum

19.16.1 RationalFunctionSum (SUMRF)



Exports:
sum

— package SUMRF RationalFunctionSum —

```
)abbrev package SUMRF RationalFunctionSum
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description:
```

```
++ Computes sums of rational functions;

RationalFunctionSum(R): Exports == Impl where
  R: Join(IntegralDomain, OrderedSet, RetractableTo Integer)

  P   ==> Polynomial R
  RF  ==> Fraction P
  FE  ==> Expression R
  SE  ==> Symbol

  Exports ==> with
    sum: (P, SE) -> RF
      ++ sum(a(n), n) returns \spad{A} which
      ++ is the indefinite sum of \spad{a} with respect to
      ++ upward difference on \spad{n}, i.e. \spad{A(n+1) - A(n) = a(n)}.
      ++
      ++X sum(i::Polynomial(Integer),variable(i=1..n))
    sum: (RF, SE) -> Union(RF, FE)
      ++ sum(a(n), n) returns \spad{A} which
      ++ is the indefinite sum of \spad{a} with respect to
      ++ upward difference on \spad{n}, i.e. \spad{A(n+1) - A(n) = a(n)}.
      ++
      ++X sum(i::Fraction(Polynomial(Integer)),i::Symbol)
    sum: (P, SegmentBinding P) -> RF
      ++ sum(f(n), n = a..b) returns \spad{f(a) + f(a+1) + ... f(b)}.
      ++
      ++X sum(i,i=1..n)
    sum: (RF, SegmentBinding RF) -> Union(RF, FE)
      ++ sum(f(n), n = a..b) returns \spad{f(a) + f(a+1) + ... f(b)}.
      ++
      ++X sum(i::Fraction(Polynomial(Integer)),i=1..n)

  Impl ==> add
    import RationalFunction R
    import GosperSummationMethod(IndexedExponents SE, SE, R, P, RF)

    innersum    : (RF, SE) -> Union(RF, "failed")
    innerpolysum: (P, SE) -> RF

    sum(f:RF, s:SegmentBinding RF) ==
      (indef := innersum(f, v := variable s)) case "failed" =>
        summation(f::FE,map((z:RF):FE +->z::FE,s)
          $SegmentBindingFunctions2(RF,FE))
      eval(indef::RF, v, 1 + hi segment s)
        - eval(indef::RF, v,lo segment s)

    sum(an:RF, n:SE) ==
      (u := innersum(an, n)) case "failed" => summation(an::FE, n)
      u::RF
```

```
    sum(p:P, s:SegmentBinding P) ==
      f := sum(p, v := variable s)
      eval(f, v, (1 + hi segment s)::RF) - eval(f,v,lo(segment s)::RF)

    innersum(an, n) ==
      (r := retractIfCan(an)@Union(P, "failed")) case "failed" =>
         an1 := eval(an, n, -1 + n::RF)
         (u := GospersMethod(an/an1, n, new$SE)) case "failed" =>
           "failed"
         an1 * eval(u::RF, n, -1 + n::RF)
      sum(r::P, n)

    sum(p:P, n:SE) ==
      rec := sum(p, n)$InnerPolySum(IndexedExponents SE, SE, R, P)
      rec.num / (rec.den :: P)
```

———-

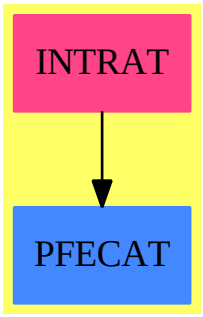
— **SUMRF.dotabb** —

```
"SUMRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUMRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUMRF" -> "PFECAT"
```

———-

19.17 package INTRAT RationalIntegration

19.17.1 RationalIntegration (INTRAT)



Exports:
extendedint infieldint integrate limitedint

— package INTRAT RationalIntegration —

```
)abbrev package INTRAT RationalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ Rational function integration
++ This package provides functions for the base case of the Risch algorithm.
-- Used internally bt the integration packages

RationalIntegration(F, UP): Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP: UnivariatePolynomialCategory F

  RF  ==> Fraction UP
  IR  ==> IntegrationResult RF
  LLG ==> List Record(coeff:RF, logand:RF)
  URF ==> Union(Record(ratpart:RF, coeff:RF), "failed")
  U   ==> Union(Record(mainpart:RF, limitedlogs:LLG), "failed")

  Exports ==> with
    integrate  : RF -> IR
      ++ integrate(f) returns g such that \spad{g' = f}.
    infieldint : RF -> Union(RF, "failed")
      ++ infieldint(f) returns g such that \spad{g' = f} or "failed"
      ++ if the integral of f is not a rational function.
    extendedint: (RF, RF) -> URF
       ++ extendedint(f, g) returns fractions \spad{[h, c]} such that
       ++ \spad{c' = 0} and \spad{h' = f - cg},
       ++ if \spad{(h, c)} exist, "failed" otherwise.
    limitedint : (RF, List RF) -> U
       ++ \spad{limitedint(f, [g1,...,gn])} returns
       ++ fractions \spad{[h,[[ci, gi]]]}
       ++ such that the gi's are among \spad{[g1,...,gn]}, \spad{ci' = 0}, and
       ++ \spad{(h+sum(ci log(gi)))' = f}, if possible, "failed" otherwise.

  Implementation ==> add
    import TranscendentalIntegration(F, UP)

    infieldint f ==
      rec := baseRDE(0, f)$TranscendentalRischDE(F, UP)
      rec.nosol => "failed"
      rec.ans

    integrate f ==
      rec := monomialIntegrate(f, differentiate)
      integrate(rec.polypart)::RF::IR + rec.ir
```

```
    limitedint(f, lu) ==
      quorem := divide(numer f, denom f)
      (u := primlimintfrac(quorem.remainder / (denom f), differentiate,
        lu)) case "failed" => "failed"
      [u.mainpart + integrate(quorem.quotient)::RF, u.limitedlogs]

    extendedint(f, g) ==
      fqr := divide(numer f, denom f)
      gqr := divide(numer g, denom g)
      (i1 := primextintfrac(fqr.remainder / (denom f), differentiate,
                  gqr.remainder / (denom g))) case "failed" => "failed"
      i2:=integrate(fqr.quotient-retract(i1.coeff)@UP *gqr.quotient)::RF
      [i2 + i1.ratpart, i1.coeff]
```

———-

— INTRAT.dotabb —

```
"INTRAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTRAT" -> "PFECAT"
```

———-

19.18 package RINTERP RationalInterpolation

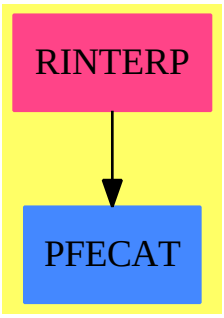
19.18.1 Introduction

This file contains a crude naïve implementation of rational interpolation, where the coefficients of the rational function are in any given field.

19.18.2 Questions and Outlook

- Maybe this file should be joined with pinterp.spad, where polynomial Lagrange interpolation is implemented. I have a second version that parallels the structure of pinterp.spad closely.
- There are probably better ways to implement rational interpolation. Maybe http://www.cs.ucsb.edu contains something useful, but I don't know.
- Comments welcome!

19.18.3 RationalInterpolation (RINTERP)



Exports:
interpolate

— **package RINTERP RationalInterpolation** —

```
)abbrev package RINTERP RationalInterpolation
++ Description:
++ This package exports rational interpolation algorithms

RationalInterpolation(xx,F): Exports == Implementation  where
    xx: Symbol
    F: Field
    Exports == with
        interpolate: (List F, List F, NonNegativeInteger,
                                NonNegativeInteger) -> Fraction Polynomial F
```

———-

The implementation sets up a system of linear equations and solves it.

— **package RINTERP RationalInterpolation** —

```
    Implementation == add
        interpolate(xlist, ylist, m, k) ==
```

———-

First we check whether we have the right number of points and values. Clearly the number of points and the number of values must be identical. Note that we want to determine the numerator and denominator polynomials only up to a factor. Thus, we want to determine $m + k + 1$ coefficients, where m is the degree of the polynomial in the numerator and k is the degree of the polynomial in the denominator.

In fact, we could also leave – for example – k unspecified and determine it as `k=#xlist-m-1`: I don't know whether this would be better.

— **package RINTERP RationalInterpolation** —

```
#xlist ^= #ylist =>
    error "Different number of points and values."
#xlist ^= m+k+1 =>
    error "wrong number of points"
```

———-

The next step is to set up the matrix. Suppose that our numerator polynomial is $p(x) = a_0 + a_1 x + \ldots + a_m x^m$ and that our denominator polynomial is $q(x) = b_0 + b_1 x + \ldots + b_m x^m$. Then we have the following equations, writing n for $m + k + 1$:

$$\begin{aligned}
p(x_1) - y_1 q(x_1) &= a_0 + a_1 x_1 + \ldots + a_m x_1^m - y_1(b_0 + b_1 x_1 + \ldots + b_k x_1^k) = 0 \\
p(x_2) - y_2 q(x_2) &= a_0 + a_1 x_2 + \ldots + a_m x_2^m - y_2(b_0 + b_1 x_2 + \ldots + b_k x_2^k) = 0 \\
&\vdots \\
p(x_n) - y_n q(x_n) &= a_0 + a_1 x_n + \ldots + a_m x_n^m - y_n(b_0 + b_1 x_n + \ldots + b_k x_n^k) = 0
\end{aligned}$$

This can be written as

$$\begin{bmatrix}
1 & x_1 & \ldots & x_1^m & -y_1 & -y_1 x_1 & \ldots & -y_1 x_1^k \\
1 & x_2 & \ldots & x_2^m & -y_2 & -y_2 x_2 & \ldots & -y_2 x_2^k \\
 & & & \vdots & & & & \\
1 & x_n & \ldots & x_n^m & -y_n & -y_n x_n & \ldots & -y_n x_2^k
\end{bmatrix}
\begin{bmatrix}
a_0 \\ a_1 \\ \vdots \\ a_m \\ b_0 \\ b_1 \\ \vdots \\ b_k
\end{bmatrix} = \mathbf{0}$$

We generate this matrix columnwise:

— **package RINTERP RationalInterpolation** —

```
tempvec: List F := [1 for i in 1..(m+k+1)]

collist: List List F := cons(tempvec,
                             [(tempvec := [tempvec.i * xlist.i _
                                           for i in 1..(m+k+1)]) _
                              for j in 1..max(m,k)])

collist := append([collist.j for j in 1..(m+1)], _
                  [[- collist.j.i * ylist.i for i in 1..(m+k+1)] _
                   for j in 1..(k+1)])
```

———-

Now we can solve the system:

— **package RINTERP RationalInterpolation** —

```
res: List Vector F := nullSpace((transpose matrix collist) _
                                ::Matrix F)
```

———-

Note that it may happen that the system has several solutions. In this case, some of the data points may not be interpolated correctly. However, the solution is often still useful, thus we do not signal an error.

— **package RINTERP RationalInterpolation** —

```
if #res~=1 then output("Warning: unattainable points!" _
                       ::OutputForm)$OutputPackage
```

———-

In this situation, all the solutions will be equivalent, thus we can always simply take the first one:

— **package RINTERP RationalInterpolation** —

```
reslist: List List Polynomial F := _
          [[(res.1).(i+1)*(xx::Polynomial F)**i for i in 0..m], _
          [(res.1).(i+m+2)*(xx::Polynomial F)**i for i in 0..k]]
```

———-

Finally, we generate the rational function:

— **package RINTERP RationalInterpolation** —

```
reduce((_+),reslist.1)/reduce((_+),reslist.2)
```

———-

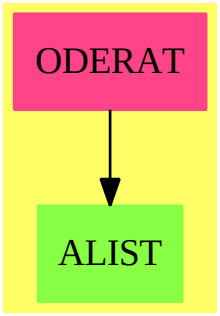
— **RINTERP.dotabb** —

```
"RINTERP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RINTERP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RINTERP" -> "PFECAT"
```

———-

19.19 package ODERAT RationalLODE

19.19.1 RationalLODE (ODERAT)



Exports:
indicialEquationAtInfinity ratDsolve

— package ODERAT RationalLODE —

```
)abbrev package ODERAT RationalLODE
++ Author: Manuel Bronstein
++ Date Created: 13 March 1991
++ Date Last Updated: 13 April 1994
++ Description:
++ \spad{RationalLODE} provides functions for in-field solutions of linear
++ ordinary differential equations, in the rational case.

RationalLODE(F, UP): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Integer,
                                       RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  Z   ==> Integer
  RF  ==> Fraction UP
  U   ==> Union(RF, "failed")
  V   ==> Vector F
  M   ==> Matrix F
  LODO ==> LinearOrdinaryDifferentialOperator1 RF
  LODO2==> LinearOrdinaryDifferentialOperator2(UP, RF)

  Exports ==> with
    ratDsolve: (LODO, RF) -> Record(particular: U, basis: List RF)
      ++ ratDsolve(op, g) returns \spad{["failed", []]} if the equation
      ++ \spad{op y = g} has no rational solution. Otherwise, it returns
      ++ \spad{[f, [y1,...,ym]]} where f is a particular rational solution
```

```
    ++ and the yi's form a basis for the rational solutions of the
    ++ homogeneous equation.
  ratDsolve: (LODO, List RF) -> Record(basis:List RF, mat:Matrix F)
    ++ ratDsolve(op, [g1,...,gm]) returns \spad{[[h1,...,hq], M]} such
    ++ that any rational solution of \spad{op y = c1 g1 + ... + cm gm}
    ++ is of the form \spad{d1 h1 + ... + dq hq} where
    ++ \spad{M [d1,...,dq,c1,...,cm] = 0}.
  ratDsolve: (LODO2, RF) -> Record(particular: U, basis: List RF)
    ++ ratDsolve(op, g) returns \spad{["failed", []]} if the equation
    ++ \spad{op y = g} has no rational solution. Otherwise, it returns
    ++ \spad{[f, [y1,...,ym]]} where f is a particular rational solution
    ++ and the yi's form a basis for the rational solutions of the
    ++ homogeneous equation.
  ratDsolve: (LODO2, List RF) -> Record(basis:List RF, mat:Matrix F)
    ++ ratDsolve(op, [g1,...,gm]) returns \spad{[[h1,...,hq], M]} such
    ++ that any rational solution of \spad{op y = c1 g1 + ... + cm gm}
    ++ is of the form \spad{d1 h1 + ... + dq hq} where
    ++ \spad{M [d1,...,dq,c1,...,cm] = 0}.
  indicialEquationAtInfinity: LODO -> UP
    ++ indicialEquationAtInfinity op returns the indicial equation of
    ++ \spad{op} at infinity.
  indicialEquationAtInfinity: LODO2 -> UP
    ++ indicialEquationAtInfinity op returns the indicial equation of
    ++ \spad{op} at infinity.

Implementation ==> add
  import BoundIntegerRoots(F, UP)
  import RationalIntegration(F, UP)
  import PrimitiveRatDE(F, UP, LODO2, LODO)
  import LinearSystemMatrixPackage(F, V, V, M)
  import InnerCommonDenominator(UP, RF, List UP, List RF)

  nzero?             : V -> Boolean
  evenodd            : N -> F
  UPfact             : N -> UP
  infOrder           : RF -> Z
  infTau             : (UP, N) -> F
  infBound           : (LODO2, List RF) -> N
  regularPoint       : (LODO2, List RF) -> Z
  infIndicialEquation: (List N, List UP) -> UP
  makeDot            : (Vector F, List RF) -> RF
  unitlist           : (N, N) -> List F
  infMuLambda: LODO2 -> Record(mu:Z, lambda:List N, func:List UP)
  ratDsolve0: (LODO2, RF) -> Record(particular: U, basis: List RF)
  ratDsolve1: (LODO2, List RF) -> Record(basis:List RF, mat:Matrix F)
  candidates: (LODO2,List RF,UP) -> Record(basis:List RF,particular:List RF)

  dummy := new()$Symbol

  infOrder f == (degree denom f) - (degree numer f)
```

```
evenodd n  == (even? n => 1; -1)

ratDsolve1(op, lg) ==
  d := denomLODE(op, lg)
  rec := candidates(op, lg, d)
  l := concat([op q for q in rec.basis],
              [op(rec.particular.i) - lg.i for i in 1..#(rec.particular)])
  sys1 := reducedSystem(matrix [l])@Matrix(UP)
  [rec.basis, reducedSystem sys1]

ratDsolve0(op, g) ==
  zero? degree op => [inv(leadingCoefficient(op)::RF) * g, empty()]
  minimumDegree op > 0 =>
    sol := ratDsolve0(monicRightDivide(op, monomial(1, 1)).quotient, g)
    b:List(RF) := [1]
    for f in sol.basis repeat
      if (uu := infieldint f) case RF then b := concat(uu::RF, b)
    sol.particular case "failed" => ["failed", b]
    [infieldint(sol.particular::RF), b]
  (u := denomLODE(op, g)) case "failed" => ["failed", empty()]
  rec := candidates(op, [g], u::UP)
  l := lb := lsol := empty()$List(RF)
  for q in rec.basis repeat
      if zero?(opq := op q) then lsol := concat(q, lsol)
      else (l := concat(opq, l); lb := concat(q, lb))
  h:RF := (zero? g => 0; first(rec.particular))
  empty? l =>
      zero? g => [0, lsol]
      [(g = op h => h; "failed"), lsol]
  m:M
  v:V
  if zero? g then
      m := reducedSystem(reducedSystem(matrix [l])@Matrix(UP))@M
      v := new(ncols m, 0)$V
  else
      sys1 := reducedSystem(matrix [l], vector [g - op h]
                            )@Record(mat: Matrix UP, vec: Vector UP)
      sys2 := reducedSystem(sys1.mat, sys1.vec)@Record(mat:M, vec:V)
      m := sys2.mat
      v := sys2.vec
  sol := solve(m, v)
  part:U :=
    zero? g => 0
    sol.particular case "failed" => "failed"
    makeDot(sol.particular::V, lb) + first(rec.particular)
  [part,
   concat_!(lsol, [makeDot(v, lb) for v in sol.basis | nzero? v])]

indicialEquationAtInfinity(op:LODO2) ==
  rec := infMuLambda op
```

```
      infIndicialEquation(rec.lambda, rec.func)

    indicialEquationAtInfinity(op:LODO) ==
      rec := splitDenominator(op, empty())
      indicialEquationAtInfinity(rec.eq)

    regularPoint(l, lg) ==
      a := leadingCoefficient(l) * commonDenominator lg
      coefficient(a, 0) ^= 0 => 0
      for i in 1.. repeat
        a(j := i::F) ^= 0 => return i
        a(-j) ^= 0 => return(-i)

    unitlist(i, q) ==
      v := new(q, 0)$Vector(F)
      v.i := 1
      parts v

    candidates(op, lg, d) ==
      n := degree d + infBound(op, lg)
      m := regularPoint(op, lg)
      uts := UnivariateTaylorSeries(F, dummy, m::F)
      tools := UTSodetools(F, UP, LODO2, uts)
      solver := UnivariateTaylorSeriesODESolver(F, uts)
      dd := UP2UTS(d)$tools
      f := LODO2FUN(op)$tools
      q := degree op
      e := unitlist(1, q)
      hom := [UTS2UP(dd * ode(f, unitlist(i, q))$solver, n)$tools /$RF d
                  for i in 1..q]$List(RF)
      a1 := inv(leadingCoefficient(op)::RF)
      part :=
       [UTS2UP(dd *
         ode((l1:List(uts)):uts +->
              RF2UTS(a1 * g)$tools + f l1, e)$solver, n)$tools
           /$RF d for g in lg | g ^= 0]$List(RF)
      [hom, part]

    nzero? v ==
      for i in minIndex v .. maxIndex v repeat
        not zero? qelt(v, i) => return true
      false

-- returns z(z+1)...(z+(n-1))
    UPfact n ==
      zero? n => 1
      z := monomial(1, 1)$UP
      */[z + i::F::UP for i in 0..(n-1)::N]

    infMuLambda l ==
```

```
    lamb:List(N) := [d := degree l]
    lf:List(UP) := [a := leadingCoefficient l]
    mup := degree(a)::Z - d
    while (l := reductum l) ^= 0 repeat
        a := leadingCoefficient l
        if (m := degree(a)::Z - (d := degree l)) > mup then
          mup := m
          lamb := [d]
          lf := [a]
        else if (m = mup) then
          lamb := concat(d, lamb)
          lf := concat(a, lf)
    [mup, lamb, lf]

  infIndicialEquation(lambda, lf) ==
    ans:UP := 0
    for i in lambda for f in lf repeat
      ans := ans + evenodd i * leadingCoefficient f * UPfact i
    ans

  infBound(l, lg) ==
    rec := infMuLambda l
    n := min(- degree(l)::Z - 1,
             integerBound infIndicialEquation(rec.lambda, rec.func))
    while not(empty? lg) and zero? first lg repeat lg := rest lg
    empty? lg => (-n)::N
    m := infOrder first lg
    for g in rest lg repeat
      if not(zero? g) and (mm := infOrder g) < m then m := mm
    (-min(n, rec.mu - degree(leadingCoefficient l)::Z + m))::N

  makeDot(v, bas) ==
    ans:RF := 0
    for i in 1.. for b in bas repeat ans := ans + v.i::UP * b
    ans

  ratDsolve(op:LODO, g:RF) ==
    rec := splitDenominator(op, [g])
    ratDsolve0(rec.eq, first(rec.rh))

  ratDsolve(op:LODO, lg:List RF) ==
    rec := splitDenominator(op, lg)
    ratDsolve1(rec.eq, rec.rh)

  ratDsolve(op:LODO2, g:RF) ==
    unit?(c := content op) => ratDsolve0(op, g)
    ratDsolve0((op exquo c)::LODO2, inv(c::RF) * g)

  ratDsolve(op:LODO2, lg:List RF) ==
    unit?(c := content op) => ratDsolve1(op, lg)
```

```
ratDsolve1((op exquo c)::LODO2, [inv(c::RF) * g for g in lg])
```

———-

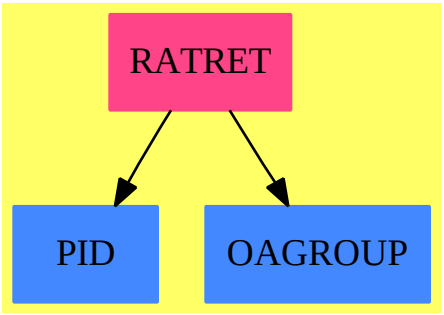
— **ODERAT.dotabb** —

```
"ODERAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODERAT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"ODERAT" -> "ALIST"
```

———-

19.20 package RATRET RationalRetractions

19.20.1 RationalRetractions (RATRET)



Exports:
rational rational? rationalIfCan

— **package RATRET RationalRetractions** —

```
)abbrev package RATRET RationalRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Rational number testing and retraction functions.

RationalRetractions(S:RetractableTo(Fraction Integer)): with
    rational     : S -> Fraction Integer
      ++ rational(x) returns x as a rational number;
      ++ error if x is not a rational number;
```

```
    rational?    : S -> Boolean
      ++ rational?(x) returns true if x is a rational number,
      ++ false otherwise;
    rationalIfCan: S -> Union(Fraction Integer, "failed")
      ++ rationalIfCan(x) returns x as a rational number,
      ++ "failed" if x is not a rational number;
  == add
    rational s      == retract s
    rational? s     == retractIfCan(s) case Fraction(Integer)
    rationalIfCan s == retractIfCan s
```

———-

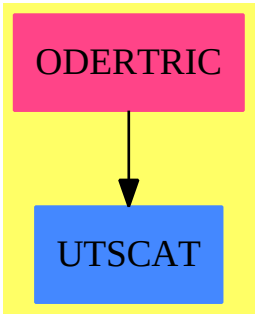
— **RATRET.dotabb** —

```
"RATRET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RATRET"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"RATRET" -> "PID"
"RATRET" -> "OAGROUP"
```

———-

19.21 package ODERTRIC RationalRicDE

19.21.1 RationalRicDE (ODERTRIC)



Exports:
polyRicDE ricDsolve singRicDE

— **package ODERTRIC RationalRicDE** —

```
)abbrev package ODERTRIC RationalRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ In-field solution of Riccati equations, rational case.

RationalRicDE(F, UP): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Integer,
                                       RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  Z   ==> Integer
  SY  ==> Symbol
  P   ==> Polynomial F
  RF  ==> Fraction P
  EQ  ==> Equation RF
  QF  ==> Fraction UP
  UP2 ==> SparseUnivariatePolynomial UP
  SUP ==> SparseUnivariatePolynomial P
  REC ==> Record(poly:SUP, vars:List SY)
  SOL ==> Record(var:List SY, val:List F)
  POL ==> Record(poly:UP, eq:L)
  FRC ==> Record(frac:QF, eq:L)
  CNT ==> Record(constant:F, eq:L)
  UTS ==> UnivariateTaylorSeries(F, dummy, 0)
  UPS ==> SparseUnivariatePolynomial UTS
  L   ==> LinearOrdinaryDifferentialOperator2(UP, QF)
  LQ  ==> LinearOrdinaryDifferentialOperator1 QF

  Exports ==> with
    ricDsolve: (LQ, UP -> List F) -> List QF
      ++ ricDsolve(op, zeros) returns the rational solutions of the associated
      ++ Riccati equation of \spad{op y = 0}.
      ++ \spad{zeros} is a zero finder in \spad{UP}.
    ricDsolve: (LQ, UP -> List F, UP -> Factored UP) -> List QF
      ++ ricDsolve(op, zeros, ezfactor) returns the rational
      ++ solutions of the associated Riccati equation of \spad{op y = 0}.
      ++ \spad{zeros} is a zero finder in \spad{UP}.
      ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
      ++ not necessarily into irreducibles.
    ricDsolve: (L, UP -> List F) -> List QF
      ++ ricDsolve(op, zeros) returns the rational solutions of the associated
      ++ Riccati equation of \spad{op y = 0}.
      ++ \spad{zeros} is a zero finder in \spad{UP}.
    ricDsolve: (L, UP -> List F, UP -> Factored UP) -> List QF
      ++ ricDsolve(op, zeros, ezfactor) returns the rational
      ++ solutions of the associated Riccati equation of \spad{op y = 0}.
      ++ \spad{zeros} is a zero finder in \spad{UP}.
```

```
    ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
    ++ not necessarily into irreducibles.
  singRicDE: (L, UP -> Factored UP) -> List FRC
    ++ singRicDE(op, ezfactor) returns \spad{[[f1,L1], [f2,L2],..., [fk,Lk]]}
    ++ such that the singular ++ part of any rational solution of the
    ++ associated Riccati equation of \spad{op y = 0} must be one of the fi's
    ++ (up to the constant coefficient), in which case the equation for
    ++ \spad{z = y e^{-int ai}} is \spad{Li z = 0}.
    ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
    ++ not necessarily into irreducibles.
  polyRicDE: (L, UP -> List F) -> List POL
    ++ polyRicDE(op, zeros) returns \spad{[[p1,L1], [p2,L2], ... , [pk,Lk]]}
    ++ such that the polynomial part of any rational solution of the
    ++ associated Riccati equation of \spad{op y = 0} must be one of the pi's
    ++ (up to the constant coefficient), in which case the equation for
    ++ \spad{z = y e^{-int p}} is \spad{Li z = 0}.
    ++ \spad{zeros} is a zero finder in \spad{UP}.
  if F has AlgebraicallyClosedField then
    ricDsolve: LQ -> List QF
      ++ ricDsolve(op) returns the rational solutions of the associated
      ++ Riccati equation of \spad{op y = 0}.
    ricDsolve: (LQ, UP -> Factored UP) -> List QF
      ++ ricDsolve(op, ezfactor) returns the rational solutions of the
      ++ associated Riccati equation of \spad{op y = 0}.
      ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
      ++ not necessarily into irreducibles.
    ricDsolve: L -> List QF
      ++ ricDsolve(op) returns the rational solutions of the associated
      ++ Riccati equation of \spad{op y = 0}.
    ricDsolve: (L, UP -> Factored UP) -> List QF
      ++ ricDsolve(op, ezfactor) returns the rational solutions of the
      ++ associated Riccati equation of \spad{op y = 0}.
      ++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
      ++ not necessarily into irreducibles.

Implementation ==> add
  import RatODETools(P, SUP)
  import RationalLODE(F, UP)
  import NonLinearSolvePackage F
  import PrimitiveRatDE(F, UP, L, LQ)
  import PrimitiveRatRicDE(F, UP, L, LQ)

  FifCan           : RF -> Union(F, "failed")
  UP2SUP           : UP -> SUP
  innersol         : (List UP, Boolean) -> List QF
  mapeval          : (SUP, List SY, List F) -> UP
  ratsol           : List List EQ -> List SOL
  ratsln           : List EQ -> Union(SOL, "failed")
  solveModulo      : (UP, UP2) -> List UP
  logDerOnly       : L -> List QF
```

```
    nonSingSolve     : (N, L, UP -> List F) -> List QF
    constantRic      : (UP, UP -> List F) -> List F
    nopoly           : (N, UP, L, UP -> List F) -> List QF
    reverseUP        : UP -> UTS
    reverseUTS       : (UTS, N) -> UP
    newtonSolution   : (L, F, N, UP -> List F) -> UP
    newtonSolve      : (UPS, F, N) -> Union(UTS, "failed")
    genericPolynomial: (SY, Z) -> Record(poly:SUP, vars:List SY)
      -- genericPolynomial(s, n) returns
      -- \spad{[[s0 + s1 X +...+ sn X^n],[s0,...,sn]]}.

    dummy := new()$SY

    UP2SUP p == map(z +-> z::P,p)
                          $UnivariatePolynomialCategoryFunctions2(F,UP,P,SUP)
    logDerOnly l == [differentiate(s) / s for s in ratDsolve(l, 0).basis]
    ricDsolve(l:LQ, zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)
    ricDsolve(l:L,  zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)
    singRicDE(l, ezfactor)              == singRicDE(l, solveModulo, ezfactor)

    ricDsolve(l:LQ, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
      ricDsolve(splitDenominator(l, empty()).eq, zeros, ezfactor)

    mapeval(p, ls, lv) ==
      map(z +-> ground eval(z, ls, lv),p)
                       $UnivariatePolynomialCategoryFunctions2(P, SUP, F, UP)

    FifCan f ==
      ((n := retractIfCan(numer f))@Union(F, "failed") case F) and
        ((d := retractIfCan(denom f))@Union(F, "failed") case F) =>
           (n::F) / (d::F)
      "failed"

-- returns [0, []] if n < 0
    genericPolynomial(s, n) ==
      ans:SUP := 0
      l:List(SY) := empty()
      for i in 0..n repeat
        ans := ans + monomial((sy := new s)::P, i::N)
        l := concat(sy, l)
      [ans, reverse_! l]

    ratsln l ==
      ls:List(SY) := empty()
      lv:List(F) := empty()
      for eq in l repeat
        ((u := FifCan rhs eq) case "failed") or
          ((v := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed")
             => return "failed"
        lv := concat(u::F, lv)
```

```
        ls := concat(v::SY, ls)
      [ls, lv]

    ratsol l ==
      ans:List(SOL) := empty()
      for sol in l repeat
        if ((u := ratsln sol) case SOL) then ans := concat(u::SOL, ans)
      ans

-- returns [] if the solutions of l have no polynomial component
    polyRicDE(l, zeros) ==
      ans:List(POL) := [[0, 1]]
      empty?(lc := leadingCoefficientRicDE l) => ans
      rec := first lc                          -- one with highest degree
      for a in zeros(rec.eq) | a ^= 0 repeat
        if (p := newtonSolution(l, a, rec.deg, zeros)) ^= 0 then
            ans := concat([p, changeVar(l, p)], ans)
      ans

-- reverseUP(a_0 + a_1 x + ... + an x^n) = a_n + ... + a_0 x^n
    reverseUP p ==
      ans:UTS := 0
      n := degree(p)::Z
      while p ^= 0 repeat
        ans := ans + monomial(leadingCoefficient p, (n - degree p)::N)
        p   := reductum p
      ans

-- reverseUTS(a_0 + a_1 x + ..., n) = a_n + ... + a_0 x^n
    reverseUTS(s, n) ==
      +/[monomial(coefficient(s, i), (n - i)::N)$UP for i in 0..n]

-- returns a potential polynomial solution p with leading coefficient a*?**n
    newtonSolution(l, a, n, zeros) ==
      i:N
      m:Z := 0
      aeq:UPS := 0
      op := l
      while op ^= 0 repeat
        mu := degree(op) * n + degree leadingCoefficient op
        op := reductum op
        if mu > m then m := mu
      while l ^= 0 repeat
        c := leadingCoefficient l
        d := degree l
        s:UTS := monomial(1, (m - d * n - degree c)::N)$UTS * reverseUP c
        aeq := aeq + monomial(s, d)
        l := reductum l
      (u := newtonSolve(aeq, a, n)) case UTS => reverseUTS(u::UTS, n)
      -- newton lifting failed, so revert to traditional method
```

```
      atn := monomial(a, n)$UP
      neq := changeVar(l, atn)
      sols := [sol.poly for sol in polyRicDE(neq, zeros) | degree(sol.poly) < n]
      empty? sols => atn
      atn + first sols

-- solves the algebraic equation eq for y, returns a solution of degree n with
-- initial term a
-- uses naive newton approximation for now
-- an example where this fails is   y^2 + 2 x y + 1 + x^2 = 0
-- which arises from the differential operator D^2 + 2 x D + 1 + x^2
    newtonSolve(eq, a, n) ==
      deq := differentiate eq
      sol := a::UTS
      for i in 1..n repeat
        (xquo := eq(sol) exquo deq(sol)) case "failed" => return "failed"
        sol := truncate(sol - xquo::UTS, i)
      sol

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
    ricDsolve(l:L, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
      n := degree l
      ans:List(QF) := empty()
      for rec in singRicDE(l, ezfactor) repeat
        ans := removeDuplicates_! concat_!(ans,
                        [rec.frac + f for f in nonSingSolve(n, rec.eq, zeros)])
        #ans = n => return ans
      ans

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
    nonSingSolve(n, l, zeros) ==
      ans:List(QF) := empty()
      for rec in polyRicDE(l, zeros) repeat
        ans := removeDuplicates_! concat_!(ans, nopoly(n,rec.poly,rec.eq,zeros))
        #ans = n => return ans
      ans

    constantRic(p, zeros) ==
      zero? degree p => empty()
      zeros squareFreePart p

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
    nopoly(n, p, l, zeros) ==
      ans:List(QF) := empty()
      for rec in constantCoefficientRicDE(l,z+->constantRic(z, zeros)) repeat
        ans := removeDuplicates_! concat_!(ans,
                  [(rec.constant::UP + p)::QF + f for f in logDerOnly(rec.eq)])
```

```
        #ans = n => return ans
      ans

-- returns [p1,...,pn] s.t. h(x,pi(x)) = 0 mod c(x)
    solveModulo(c, h) ==
      rec := genericPolynomial(dummy, degree(c)::Z - 1)
      unk:SUP := 0
      while not zero? h repeat
        unk := unk + UP2SUP(leadingCoefficient h) * (rec.poly ** degree h)
        h   := reductum h
      sol := ratsol solve(coefficients(monicDivide(unk,UP2SUP c).remainder),
                          rec.vars)
      [mapeval(rec.poly, s.var, s.val) for s in sol]

    if F has AlgebraicallyClosedField then
      zro1: UP -> List F
      zro : (UP, UP -> Factored UP) -> List F

      ricDsolve(l:L)  == ricDsolve(l, squareFree)
      ricDsolve(l:LQ) == ricDsolve(l, squareFree)

      ricDsolve(l:L, ezfactor:UP -> Factored UP) ==
        ricDsolve(l, z +-> zro(z, ezfactor), ezfactor)

      ricDsolve(l:LQ, ezfactor:UP -> Factored UP) ==
        ricDsolve(l, z +-> zro(z, ezfactor), ezfactor)

      zro(p, ezfactor) ==
        concat [zro1(r.factor) for r in factors ezfactor p]

      zro1 p ==
        [zeroOf(map((z:F):F +-> z, p)
          $UnivariatePolynomialCategoryFunctions2(F, UP, F,
              SparseUnivariatePolynomial F))]
```

———-

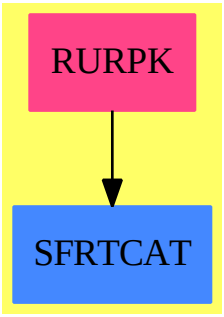
— ODERTRIC.dotabb —

```
"ODERTRIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODERTRIC"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"ODERTRIC" -> "UTSCAT"
```

———-

19.22 package RURPK RationalUnivariateRepresentationPackage

19.22.1 RationalUnivariateRepresentationPackage (RURPK)



Exports:
rur

— package RURPK RationalUnivariateRepresentationPackage —

```
)abbrev package RURPK RationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Description:
++ A package for computing the rational univariate representation
++ of a zero-dimensional algebraic variety given by a regular
++ triangular set. This package is essentially an interface for the
++ \spadtype{InternalRationalUnivariateRepresentationPackage} constructor.
++ It is used in the \spadtype{ZeroDimensionalSolvePackage}
++ for solving polynomial systems with finitely many solutions.

RationalUnivariateRepresentationPackage(R,ls): Exports == Implementation where
  R : Join(EuclideanDomain,CharacteristicZero)
  ls: List Symbol
  N ==> NonNegativeInteger
  Z ==> Integer
  P ==> Polynomial R
  LP ==> List P
  U ==> SparseUnivariatePolynomial(R)
  RUR ==> Record(complexRoots: U, coordinates: LP)
```

```
Exports ==  with

   rur: (LP,Boolean) -> List RUR
     ++ \spad{rur(lp,univ?)} returns a rational univariate representation
     ++ of \spad{lp}. This assumes that \spad{lp} defines a regular
     ++ triangular \spad{ts} whose associated variety is zero-dimensional
     ++ over \spad{R}. \spad{rur(lp,univ?)} returns a list of items
     ++ \spad{[u,lc]} where \spad{u} is an irreducible univariate polynomial
     ++ and each \spad{c} in \spad{lc} involves two variables: one from \spad{ls},
     ++ called the coordinate of \spad{c}, and an extra variable which
     ++ represents any root of \spad{u}. Every root of \spad{u} leads to
     ++ a tuple of values for the coordinates of \spad{lc}. Moreover,
     ++ a point \spad{x} belongs to the variety associated with \spad{lp} iff
     ++ there exists an item \spad{[u,lc]} in \spad{rur(lp,univ?)} and
     ++ a root \spad{r} of \spad{u} such that \spad{x} is given by the
     ++ tuple of values for the coordinates of \spad{lc} evaluated at \spad{r}.
     ++ If \spad{univ?} is \spad{true} then each polynomial \spad{c}
     ++ will have a constant leading coefficient w.r.t. its coordinate.
     ++ See the example which illustrates the \spadtype{ZeroDimensionalSolvePackage}
     ++ package constructor.
   rur: (LP) -> List RUR
     ++ \spad{rur(lp)} returns the same as \spad{rur(lp,true)}
   rur: (LP,Boolean,Boolean) -> List RUR
     ++ \spad{rur(lp,univ?,check?)} returns the same as \spad{rur(lp,true)}.
     ++ Moreover, if \spad{check?} is \spad{true} then the result is checked.

Implementation == add
   news: Symbol := new()$Symbol
   lv: List Symbol := concat(ls,news)
   V ==> OrderedVariableList(lv)
   Q ==> NewSparseMultivariatePolynomial(R,V)
   E ==> IndexedExponents V
   TS ==> SquareFreeRegularTriangularSet(R,E,V,Q)
   QWT ==> Record(val: Q, tower: TS)
   LQWT ==> Record(val: List Q, tower: TS)
   polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,Q)
   normpack ==> NormalizationPackage(R,E,V,Q,TS)
   rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E,V,Q,TS)
   newv: V := variable(news)::V
   newq : Q := newv :: Q

   rur(lp: List P, univ?: Boolean, check?: Boolean): List RUR ==
     lp := remove(zero?,lp)
     empty? lp =>
       error "rur$RURPACK: #1 is empty"
     any?(ground?,lp) =>
       error "rur$RURPACK: #1 is not a triangular set"
     ts: TS := [[newq]$(List Q)]
     lq: List Q := []
```

```
    for p in lp repeat
      rif: Union(Q,"failed") := retractIfCan(p)$Q
      rif case "failed" =>
        error "rur$RURPACK: #1 is not a subset of R[ls]"
      q: Q := rif::Q
      lq := cons(q,lq)
    lq := sort(infRittWu?,lq)
    toSee: List LQWT := [[lq,ts]$LQWT]
    toSave: List TS := []
    while not empty? toSee repeat
      lqwt := first toSee; toSee := rest toSee
      lq := lqwt.val; ts := lqwt.tower
      empty? lq =>
        -- output(ts::OutputForm)$OutputPackage
        toSave := cons(ts,toSave)
      q := first lq; lq := rest lq
      not (mvar(q) > mvar(ts)) =>
        error "rur$RURPACK: #1 is not a triangular set"
      empty? (rest(ts)::TS) =>
        lfq := irreducibleFactors([q])$polsetpack
        for fq in lfq repeat
          newts := internalAugment(fq,ts)
          newlq := [remainder(q,newts).polnum for q in lq]
          toSee := cons([newlq,newts]$LQWT,toSee)
      lsfqwt: List QWT := squareFreePart(q,ts)
      for qwt in lsfqwt repeat
        q := qwt.val; ts := qwt.tower
        if not ground? init(q)
          then
            q := normalizedAssociate(q,ts)$normpack
        newts := internalAugment(q,ts)
        newlq := [remainder(q,newts).polnum for q in lq]
        toSee := cons([newlq,newts]$LQWT,toSee)
    toReturn: List RUR := []
    for ts in toSave repeat
      lus := rur(ts,univ?)$rurpack
      check? and (not checkRur(ts,lus)$rurpack) =>
        output("RUR for: ")$OutputPackage
        output(ts::OutputForm)$OutputPackage
        output("Is: ")$OutputPackage
        for us in lus repeat output(us::OutputForm)$OutputPackage
        error "rur$RURPACK: bad result with function rur$IRURPK"
      for us in lus repeat
         g: U  := univariate(select(us,newv)::Q)$Q
         lc: LP := [convert(q)@P for q in parts(collectUpper(us,newv))]
         toReturn := cons([g,lc]$RUR, toReturn)
    toReturn

  rur(lp: List P, univ?: Boolean): List RUR ==
    rur(lp,univ?,false)
```

```
rur(lp: List P): List RUR == rur(lp,true)
```

———-

— **RURPK.dotabb** —

```
"RURPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RURPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"RURPK" -> "SFRTCAT"
```

———-

19.23 package POLUTIL RealPolynomialUtilitiesPackage

This file describes the Real Closure 1.0 package which consists of different packages, categoris and domains :

The package RealPolynomialUtilitiesPackage whichs receives a field and a univariate polynomial domain with coefficients in the field. It computes some simple functions such as Strum and Sylvester sequences.

The category RealRootCharacterizationCategory provides abstarct functionalities to work with "real roots" of univariate polynomials. These resemble variables with some functionalities needed to compute important operations.

RealClosedField is a category with provides common operations available over real closed fiels. These include finding all the roots of univariate polynomial, taking square roots, ...

CAVEATS

Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every cretaion function raises a new "real root". This has the effect that when you type something like sqrt(2) + sqrt(2) you have two new variables which happen to be equal. To avoid this name the expression such as in s2 := sqrt(2) ; s2 + s2

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which most natural to you.

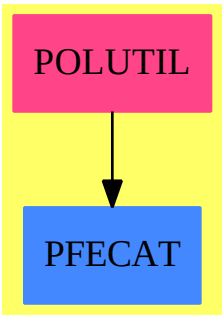
LIMITATIONS

The file reclos.input show some basic use of the package. This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants

methods usually work best. Betas versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done excatly. They can thus be quite time consuming when depending on several "real roots".

19.23.1 RealPolynomialUtilitiesPackage (POLUTIL)



Exports:
boundOfCauchy lazyVariations sturmSequence sturmVariationsOf sylvesterSequence

— package POLUTIL RealPolynomialUtilitiesPackage —

```
)abbrev package POLUTIL RealPolynomialUtilitiesPackage
++ Author: Renaud Rioboo
++ Date Created: summer 1992
++ Basic Functions: provides polynomial utilities
++ Related Constructors: RealClosure,
++ Date Last Updated: July 2004
++ Also See:
++ AMS Classifications:
++ Keywords: Sturm sequences
++ References:
++ Description:
++ \axiomType{RealPolynomialUtilitiesPackage} provides common functions used
++ by interval coding.

RealPolynomialUtilitiesPackage(TheField,ThePols) : PUB == PRIV where

    TheField : Field
    ThePols : UnivariatePolynomialCategory(TheField)

    Z ==> Integer
```

```
    N ==> NonNegativeInteger
    P ==> ThePols

    PUB == with

       sylvesterSequence : (ThePols,ThePols) -> List ThePols
         ++ \axiom{sylvesterSequence(p,q)} is the negated remainder sequence
         ++ of p and q divided by the last computed term
       sturmSequence : ThePols -> List ThePols
         ++ \axiom{sturmSequence(p) = sylvesterSequence(p,p')}
       if TheField has OrderedRing then
         boundOfCauchy : ThePols -> TheField
           ++ \axiom{boundOfCauchy(p)} bounds the roots of p
         sturmVariationsOf : List TheField -> N
           ++ \axiom{sturmVariationsOf(l)} is the number of sign variations
           ++ in the list of numbers l,
           ++ note that the first term counts as a sign
         lazyVariations : (List(TheField), Z, Z) -> N
           ++ \axiom{lazyVariations(l,s1,sn)} is the number of sign variations
           ++ in the list of non null numbers [s1::l]@sn,


    PRIV == add

     sturmSequence(p) ==
       sylvesterSequence(p,differentiate(p))

     sylvesterSequence(p1,p2) ==
       res : List(ThePols) := [p1]
       while (p2 ^= 0) repeat
         res := cons(p2 , res)
         (p1 , p2) := (p2 , -(p1 rem p2))
       if degree(p1) > 0
       then
         p1 := unitCanonical(p1)
         res := [ term quo p1 for term in res ]
       reverse! res

     if TheField has OrderedRing
     then

       boundOfCauchy(p) ==
         c :TheField := inv(leadingCoefficient(p))
         l := [ c*term for term in rest(coefficients(p))]
         null(l) => 1
         1 + ("max" / [ abs(t) for t in l ])

--       sturmVariationsOf(l) ==
--         res : N := 0
--         lsg := sign(first(l))
```

```
--         for term in l repeat
--           if ^( (sg := sign(term) ) = 0 ) then
--             if (sg ^= lsg) then res := res + 1
--             lsg := sg
--         res

       sturmVariationsOf(l) ==
         null(l) => error "POLUTIL: sturmVariationsOf: empty list !"
         l1 := first(l)
         -- first 0 counts as a sign
         ll : List(TheField) := []
         for term in rest(l) repeat
           -- zeros don't count
           if not(zero?(term)) then ll := cons(term,ll)
         -- if l1 is not zero then ll = reverse(l)
         null(ll) => error "POLUTIL: sturmVariationsOf: Bad sequence"
         ln := first(ll)
         ll := reverse(rest(ll))
         -- if l1 is not zero then first(l) = first(ll)
         -- if l1 is zero then first zero should count as a sign
         zero?(l1) => 1 + lazyVariations(rest(ll),sign(first(ll)),sign(ln))
         lazyVariations(ll, sign(l1), sign(ln))

       lazyVariations(l,sl,sh) ==
         zero?(sl) or zero?(sh) => error "POLUTIL: lazyVariations: zero sign!"
         null(l) =>
           if sl = sh then 0 else 1
         null(rest(l)) =>
           if zero?(first(l))
           then error "POLUTIL: lazyVariations: zero sign!"
           else
             if sl = sh
             then
               if (sl = sign(first(l)))
               then 0
               else 2
             -- in this case we save one test
             else 1
         s := sign(l.2)
         lazyVariations([first(l)],sl,s) +
           lazyVariations(rest(rest(l)),s,sh)
```

———-

— POLUTIL.dotabb —

```
"POLUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLUTIL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
```

```
"POLUTIL" -> "PFECAT"
```

19.24 package REALSOLV RealSolvePackage

— RealSolvePackage.input —

```
)set break resume
)sys rm -f RealSolvePackage.output
)spool RealSolvePackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 13
p := 4*x^3 - 3*x^2 + 2*x - 4
--R
--R
--R          3     2
--R   (1)  4x  - 3x  + 2x - 4
--R                                                  Type: Polynomial(Integer)
--E 1

--S 2 of 13
ans1 := solve(p,0.01)$REALSOLV
--R
--R
--R   (2)  [1.11328125]
--R                                                        Type: List(Float)
--E 2

--S 3 of 13
ans2 := solve(p::POLY(FRAC(INT)),0.01)$REALSOLV
--R
--R
--R   (3)  [1.11328125]
--R                                                        Type: List(Float)
--E 3

--S 4 of 13
R := Integer
--R
--R
--R   (4)  Integer
--R                                                             Type: Domain
--E 4
```

```
--S 5 of 13
ls : List Symbol := [x,y,z,t]
--R
--R
--R   (5)  [x,y,z,t]
--R                                                    Type: List(Symbol)
--E 5

--S 6 of 13
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
--R
--R
--R   (6)  [x,y,z,t,%A]
--R                                                    Type: List(Symbol)
--E 6

--S 7 of 13
pack := ZDSOLVE(R,ls,ls2)
--R
--R
--R   (7)  ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])
--R                                                          Type: Domain
--E 7

--S 8 of 13
p1 := x**2*y*z + y*z
--R
--R
--R          2
--R   (8)  (x  + 1)y z
--R                                              Type: Polynomial(Integer)
--E 8

--S 9 of 13
p2 := x**2*y**2*z + x + z
--R
--R
--R          2 2
--R   (9)  (x y  + 1)z + x
--R                                              Type: Polynomial(Integer)
--E 9

--S 10 of 13
p3 := x**2*y**2*z**2 +  z + 1
--R
--R
--R          2 2 2
--R   (10)  x y z  + z + 1
--R                                              Type: Polynomial(Integer)
```

```
--E 10

--S 11 of 13
lp := [p1, p2, p3]
--R
--R
--R            2           2 2            2 2 2
--R   (11)  [(x  + 1)y z,(x y  + 1)z + x,x y z  + z + 1]
--R                                              Type: List(Polynomial(Integer))
--E 11

--S 12 of 13
lsv:List(Symbol):=[x,y,z]
--R
--R
--R   (12)  [x,y,z]
--R                                                    Type: List(Symbol)
--E 12

--S 13 of 13
ans3 := realSolve(lp,lsv,0.01)$REALSOLV
--R
--R
--R   (13)  [[1.0,0.0,- 1.0]]
--R                                                  Type: List(List(Float))
--E 13
)spool
```

———-

— RealSolvePackage.help —

```
=====================================================================
RealSolvePackage examples
=====================================================================

  p := 4*x^3 - 3*x^2 + 2*x - 4

  ans1 := solve(p,0.01)$REALSOLV

  ans2 := solve(p::POLY(FRAC(INT)),0.01)$REALSOLV

  R := Integer

  ls : List Symbol := [x,y,z,t]

  ls2 : List Symbol := [x,y,z,t,new()$Symbol]
```

```
  pack := ZDSOLVE(R,ls,ls2)

  p1 := x**2*y*z + y*z

  p2 := x**2*y**2*z + x + z

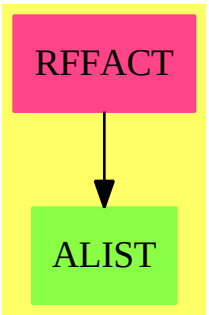
  p3 := x**2*y**2*z**2 +  z + 1

  lp := [p1, p2, p3]

  ans3 := realSolve(lp,[x,y,z],0.01)

See Also:
o )show RealSolvePackage
```

19.24.1 RealSolvePackage (REALSOLV)



Exports:
realSolve solve

— package REALSOLV RealSolvePackage —

```
)abbrev package REALSOLV RealSolvePackage
++ Description:
++ This package provides numerical solutions of systems of
++ polynomial equations for use in ACPLOT

RealSolvePackage(): Exports == Implementation where
  I    ==> Integer
  IE   ==> IndexedExponents Symbol
  L    ==> List
```

```
NF   ==> Float
P    ==> Polynomial
RN   ==> Fraction Integer
SE   ==> Symbol
RFI  ==> Fraction Polynomial Integer
LIFT ==> PolynomialCategoryLifting(IE,SE,RN,P RN, RFI)
SOLV ==> FloatingRealPackage Float

Exports ==> with
  solve: (P RN,NF) -> L NF
    ++ solve(p,eps) finds the real zeroes of a
    ++ univariate rational polynomial p with precision eps.
    ++
    ++X p := 4*x^3 - 3*x^2 + 2*x - 4
    ++X solve(p::POLY(FRAC(INT)),0.01)$REALSOLV

  solve: (P I,NF) -> L NF
    ++ solve(p,eps) finds the real zeroes of a univariate
    ++ integer polynomial p with precision eps.
    ++
    ++X p := 4*x^3 - 3*x^2 + 2*x - 4
    ++X solve(p,0.01)$REALSOLV

  realSolve: (L P I,L SE,NF) -> L L NF
    ++ realSolve(lp,lv,eps) = compute the list of the real
    ++ solutions of the list lp of polynomials with integer
    ++ coefficients with respect to the variables in lv,
    ++ with precision eps.
    ++
    ++X p1 := x**2*y*z + y*z
    ++X p2 := x**2*y**2*z + x + z
    ++X p3 := x**2*y**2*z**2 +  z + 1
    ++X lp := [p1, p2, p3]
    ++X realSolve(lp,[x,y,z],0.01)

Implementation ==>  add

  prn2rfi: P RN -> RFI
  prn2rfi p ==
    map(x+->x::RFI, x+->(numer(x)::RFI)/(denom(x)::RFI), p)$LIFT

  pi2rfi: P I -> RFI
  pi2rfi p == p :: RFI

  solve(p:P RN,eps:NF) == realRoots(prn2rfi p, eps)$SOLV

  solve(p:P I,eps:NF)  == realRoots(p::RFI, eps)$SOLV

  realSolve(lp,lv,eps) ==
    realRoots(map(pi2rfi, lp)$ListFunctions2(P I,RFI),lv,eps)$SOLV
```

———-

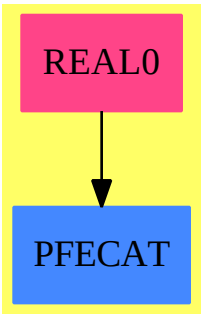
— **REALSOLV.dotabb** —

```
"REALSOLV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REALSOLV"]
"Package" [color="#FF4488"]
"REALSOLV" -> "Package"
```

———-

19.25 package REAL0 RealZeroPackage

19.25.1 RealZeroPackage (REAL0)



Exports:
midpoint midpoints realZeros refine

— **package REAL0 RealZeroPackage** —

```
)abbrev package REAL0 RealZeroPackage
++ Author: Andy Neff
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors: UnivariatePolynomial, RealZeroPackageQ
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides functions for finding the real zeros
```

```
++ of univariate polynomials over the integers to arbitrary user-specified
++ precision. The results are returned as a list of
++ isolating intervals which are expressed as records with
++ "left" and "right" rational number components.

RealZeroPackage(Pol): T == C where
   Pol: UnivariatePolynomialCategory Integer
   RN ==> Fraction Integer
   Interval ==> Record(left : RN, right : RN)
   isoList ==> List(Interval)
   T == with
    -- next two functions find isolating intervals
      realZeros: (Pol) -> isoList
        ++ realZeros(pol) returns a list of isolating intervals for
        ++ all the real zeros of the univariate polynomial pol.
      realZeros: (Pol, Interval) -> isoList
        ++ realZeros(pol, range) returns a list of isolating intervals
        ++ for all the real zeros of the univariate polynomial pol which
        ++ lie in the interval expressed by the record range.
    -- next two functions return intervals smaller then tolerence
      realZeros: (Pol, RN) -> isoList
        ++ realZeros(pol, eps) returns a list of intervals of length less
        ++ than the rational number eps for all the real roots of the
        ++ polynomial pol.
      realZeros: (Pol, Interval, RN) -> isoList
        ++ realZeros(pol, int, eps) returns a list of intervals of length
        ++ less than the rational number eps for all the real roots of the
        ++ polynomial pol which lie in the interval expressed by the
        ++ record int.
      refine: (Pol, Interval, RN) -> Interval
        ++ refine(pol, int, eps) refines the interval int containing
        ++ exactly one root of the univariate polynomial pol to size less
        ++ than the rational number eps.
      refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
        ++ refine(pol, int, range) takes a univariate polynomial pol and
        ++ and isolating interval int containing exactly one real
        ++ root of pol; the operation returns an isolating interval which
        ++ is contained within range, or "failed" if no such isolating interval exists.
      midpoint: Interval -> RN
        ++ midpoint(int) returns the midpoint of the interval int.
      midpoints: isoList -> List RN
        ++ midpoints(isolist) returns the list of midpoints for the list
        ++ of intervals isolist.
   C == add
      --Local Functions
      makeSqfr: Pol -> Pol
      ReZeroSqfr: (Pol) -> isoList
      PosZero: (Pol) -> isoList
      Zero1: (Pol) -> isoList
      transMult: (Integer, Pol) -> Pol
```

```
transMultInv: (Integer, Pol) -> Pol
transAdd1: (Pol) -> Pol
invert: (Pol) -> Pol
minus: (Pol) -> Pol
negate: Interval -> Interval
rootBound: (Pol) -> Integer
var: (Pol) -> Integer

negate(int : Interval):Interval == [-int.right,-int.left]

midpoint(i : Interval):RN ==  (1/2)*(i.left + i.right)

midpoints(li : isoList) : List RN ==
  [midpoint x for x in li]

makeSqfr(F : Pol):Pol ==
   sqfr := squareFree F
   F := */[s.factor for s in factors(sqfr)]

realZeros(F : Pol) ==
   ReZeroSqfr makeSqfr F

realZeros(F : Pol, rn : RN) ==
   F := makeSqfr F
   [refine(F,int,rn) for int in ReZeroSqfr(F)]

realZeros(F : Pol, bounds : Interval) ==
   F := makeSqfr F
   [rint::Interval for int in ReZeroSqfr(F) |
       (rint:=refine(F,int,bounds)) case Interval]

realZeros(F : Pol, bounds : Interval, rn : RN) ==
   F := makeSqfr F
   [refine(F,int,rn) for int in realZeros(F,bounds)]

ReZeroSqfr(F : Pol) ==
   F = 0 => error "ReZeroSqfr: zero polynomial"
   L : isoList := []
   degree(F) = 0 => L
   if (r := minimumDegree(F)) > 0 then
        L := [[0,0]$Interval]
        tempF := F exquo monomial(1, r)
        if not (tempF case "failed") then
             F := tempF
   J:isoList := [negate int for int in reverse(PosZero(minus(F)))]
   K : isoList := PosZero(F)
   append(append(J, L), K)

PosZero(F : Pol) ==   --F is square free, primitive
                    --and F(0) ^= 0; returns isoList for positive
```

```
                    --roots of F

   b : Integer := rootBound(F)
   F := transMult(b,F)
   L : isoList := Zero1(F)
   int : Interval
   L := [[b*int.left, b*int.right]$Interval for int in L]

Zero1(F : Pol) ==   --returns isoList for roots of F in (0,1)
   J : isoList
   K : isoList
   L : isoList
   L := []
   (v := var(transAdd1(invert(F)))) = 0 => []
   v = 1 => L := [[0,1]$Interval]
   G : Pol := transMultInv(2, F)
   H : Pol := transAdd1(G)
   if minimumDegree H > 0 then
           -- H has a root at 0 => F has one at 1/2, and G at 1
        L := [[1/2,1/2]$Interval]
        Q : Pol := monomial(1, 1)
        tempH : Union(Pol, "failed") := H exquo Q
        if not (tempH case "failed") then H := tempH
        Q := Q + monomial(-1, 0)
        tempG : Union(Pol, "failed") := G exquo Q
        if not (tempG case "failed") then G := tempG
   int : Interval
   J := [[(int.left+1)* (1/2),(int.right+1) * (1/2)]$Interval
                                            for int in Zero1(H)]
   K := [[int.left * (1/2), int.right * (1/2)]$Interval
                                            for int in Zero1(G)]
   append(append(J, L), K)

rootBound(F : Pol) ==  --returns power of 2 that is a bound
                       --for the positive roots of F
   if leadingCoefficient(F) < 0 then F := -F
   lcoef := leadingCoefficient(F)
   F := reductum(F)
   i : Integer := 0
   while not (F = 0) repeat
        if (an := leadingCoefficient(F)) < 0 then i := i - an
        F := reductum(F)
   b : Integer := 1
   while (b * lcoef) <= i repeat
        b := 2 * b
   b

transMult(c : Integer, F : Pol) ==
           --computes Pol G such that G(x) = F(c*x)
   G : Pol := 0
```

```
        while not (F = 0) repeat
             n := degree(F)
             G := G + monomial((c**n) * leadingCoefficient(F), n)
             F := reductum(F)
        G

      transMultInv(c : Integer, F : Pol) ==
                 --computes Pol G such that G(x) = (c**n) * F(x/c)
        d := degree(F)
        cc : Integer := 1
        G : Pol := monomial(leadingCoefficient F,d)
        while (F:=reductum(F)) ^= 0 repeat
             n := degree(F)
             cc := cc*(c**(d-n):NonNegativeInteger)
             G := G + monomial(cc * leadingCoefficient(F), n)
             d := n
        G

--    otransAdd1(F : Pol) ==
--                --computes Pol G such that G(x) = F(x+1)
--       G : Pol := F
--       n : Integer := 1
--       while (F := differentiate(F)) ^= 0 repeat
--            if not ((tempF := F exquo n) case "failed") then F := tempF
--            G := G + F
--            n := n + 1
--       G

      transAdd1(F : Pol) ==
                 --computes Pol G such that G(x) = F(x+1)
        n := degree F
        v := vectorise(F, n+1)
        for i in 0..(n-1) repeat
           for j in (n-i)..n repeat
              qsetelt_!(v,j, qelt(v,j) + qelt(v,(j+1)))
        ans : Pol := 0
        for i in 0..n repeat
           ans := ans + monomial(qelt(v,(i+1)),i)
        ans


      minus(F : Pol) ==
                 --computes Pol G such that G(x) = F(-x)
        G : Pol := 0
        while not (F = 0) repeat
             n := degree(F)
             coef := leadingCoefficient(F)
             odd? n =>
                  G := G + monomial(-coef, n)
                  F := reductum(F)
```

```
        G := G + monomial(coef, n)
        F := reductum(F)
   G

invert(F : Pol) ==
           --computes Pol G such that G(x) = (x**n) * F(1/x)
   G : Pol := 0
   n := degree(F)
   while not (F = 0) repeat
        G := G + monomial(leadingCoefficient(F),
                          (n-degree(F))::NonNegativeInteger)
        F := reductum(F)
   G

var(F : Pol) ==    --number of sign variations in coefs of F
   i : Integer := 0
   LastCoef : Boolean
   next : Boolean
   LastCoef := leadingCoefficient(F) < 0
   while not ((F := reductum(F)) = 0) repeat
        next := leadingCoefficient(F) < 0
        if ((not LastCoef) and next) or
            ((not next) and LastCoef) then i := i+1
        LastCoef := next
   i

refine(F : Pol, int : Interval, bounds : Interval) ==
   lseg := min(int.right,bounds.right) - max(int.left,bounds.left)
   lseg < 0 => "failed"
   lseg = 0 =>
      pt :=
         int.left = bounds.right => int.left
         int.right
      elt(transMultInv(denom(pt),F),numer pt) = 0 => [pt,pt]
      "failed"
   lseg = int.right - int.left => int
   refine(F, refine(F, int, lseg), bounds)

refine(F : Pol, int : Interval, eps : RN) ==
   a := int.left
   b := int.right
   a=b => [a,b]$Interval
   an : Integer := numer(a)
   ad : Integer := denom(a)
   bn : Integer := numer(b)
   bd : Integer := denom(b)
   xfl : Boolean := false
   if (u:=elt(transMultInv(ad, F), an)) = 0 then
       F := (F exquo (monomial(ad,1)-monomial(an,0)))::Pol
       u:=elt(transMultInv(ad, F), an)
```

```
        if (v:=elt(transMultInv(bd, F), bn)) = 0 then
            F := (F exquo (monomial(bd,1)-monomial(bn,0)))::Pol
            v:=elt(transMultInv(bd, F), bn)
            u:=elt(transMultInv(ad, F), an)
        if u > 0 then (F:=-F;v:=-v)
        if v < 0 then
           error [int, "is not a valid isolation interval for", F]
        if eps <= 0 then error "precision must be positive"
        while (b - a) >= eps repeat
             mid : RN := (b + a) * (1/2)
             midn : Integer := numer(mid)
             midd : Integer := denom(mid)
             (v := elt(transMultInv(midd, F), midn)) < 0 =>
                  a := mid
                  an := midn
                  ad := midd
             v > 0 =>
                  b := mid
                  bn := midn
                  bd := midd
             v = 0 =>
                  a := mid
                  b := mid
                  an := midn
                  ad := midd
                  xfl := true
        [a, b]$Interval
```

———-

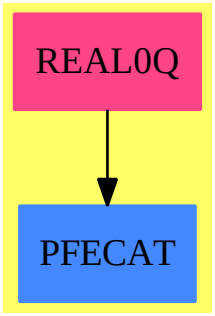
— **REAL0.dotabb** —

```
"REAL0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REAL0"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"REAL0" -> "PFECAT"
```

———-

19.26 package REAL0Q RealZeroPackageQ

19.26.1 RealZeroPackageQ (REAL0Q)



Exports:
realZeros refine

— package REAL0Q RealZeroPackageQ —

```
)abbrev package REAL0Q RealZeroPackageQ
++ Author: Andy Neff, Barry Trager
++ Date Created:
++ Date Last Updated: 7 April 1991
++ Basic Functions:
++ Related Constructors: UnivariatePolynomial, RealZeroPackageQ
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides functions for finding the real zeros of univariate
++ polynomials over the rational numbers to arbitrary user-specified
++ precision. The results are returned as a list of isolating intervals,
++ expressed as records with "left" and "right" rational number components.

RealZeroPackageQ(Pol): T == C where
   RN  ==> Fraction Integer
   I   ==> Integer
   SUP ==> SparseUnivariatePolynomial
   Pol: UnivariatePolynomialCategory RN
   Interval ==> Record(left : RN, right : RN)
   isoList ==> List(Interval)
   ApproxInfo ==> Record(approx : RN, exFlag : Boolean)
   T == with
    -- next two functions find isolating intervals
      realZeros: (Pol) -> isoList
```

```
      ++ realZeros(pol) returns a list of isolating intervals for
      ++ all the real zeros of the univariate polynomial pol.
    realZeros: (Pol, Interval) -> isoList
      ++ realZeros(pol, range) returns a list of isolating intervals
      ++ for all the real zeros of the univariate polynomial pol which
      ++ lie in the interval expressed by the record range.
  -- next two functions return intervals smaller then tolerence
    realZeros: (Pol, RN) -> isoList
      ++ realZeros(pol, eps) returns a list of intervals of length less
      ++ than the rational number eps for all the real roots of the
      ++ polynomial pol.
    realZeros: (Pol, Interval, RN) -> isoList
      ++ realZeros(pol, int, eps) returns a list of intervals of length
      ++ less than the rational number eps for all the real roots of the
      ++ polynomial pol which lie in the interval expressed by the
      ++ record int.
    refine: (Pol, Interval, RN) -> Interval
      ++ refine(pol, int, eps) refines the interval int containing
      ++ exactly one root of the univariate polynomial pol to size less
      ++ than the rational number eps.
    refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
      ++ refine(pol, int, range) takes a univariate polynomial pol and
      ++ and isolating interval int which must contain exactly one real
      ++ root of pol, and returns an isolating interval which
      ++ is contained within range, or "failed" if no such isolating interval exists.
  C == add
    import RealZeroPackage SparseUnivariatePolynomial Integer

    convert2PolInt: Pol -> SparseUnivariatePolynomial Integer

    convert2PolInt(f : Pol) ==
       pden:I :=lcm([denom c for c in coefficients f])
       map(numer,pden * f)$UnivariatePolynomialCategoryFunctions2(RN,Pol,I,SUP I)

    realZeros(f : Pol) == realZeros(convert2PolInt f)
    realZeros(f : Pol, rn : RN) == realZeros(convert2PolInt f, rn)
    realZeros(f : Pol, bounds : Interval) ==
             realZeros(convert2PolInt f, bounds)
    realZeros(f : Pol, bounds : Interval, rn : RN) ==
             realZeros(convert2PolInt f, bounds, rn)
    refine(f : Pol, int : Interval, eps : RN) ==
             refine(convert2PolInt f, int, eps)
    refine(f : Pol, int : Interval, bounds : Interval) ==
             refine(convert2PolInt f, int, bounds)
```

———-

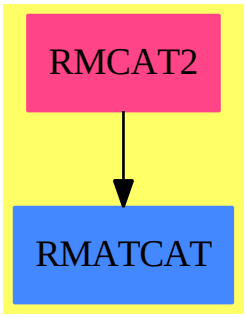
— REAL0Q.dotabb —

```
"REAL0Q" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REAL0Q"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"REAL0Q" -> "PFECAT"
```

———-

19.27 package RMCAT2 RectangularMatrixCategoryFunctions2

19.27.1 RectangularMatrixCategoryFunctions2 (RMCAT2)



Exports:
map reduce

— package RMCAT2 RectangularMatrixCategoryFunctions2 —

```
)abbrev package RMCAT2 RectangularMatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 12 June 1991
++ Basic Operations:
++ Related Domains: IndexedMatrix(R,minRow,minCol), Matrix(R),
++    RectangularMatrix(n,m,R), SquareMatrix(n,R)
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Keywords: matrix, map, reduce
++ Examples:
++ References:
++ Description:
++ \spadtype{RectangularMatrixCategoryFunctions2} provides functions between
++ two matrix domains.  The functions provided are \spadfun{map} and
++ \spadfun{reduce}.
```

```
RectangularMatrixCategoryFunctions2(m,n,R1,Row1,Col1,M1,R2,Row2,Col2,M2):_
        Exports == Implementation where
  m,n  : NonNegativeInteger
  R1   : Ring
  Row1 : DirectProductCategory(n, R1)
  Col1 : DirectProductCategory(m, R1)
  M1   : RectangularMatrixCategory(m,n,R1,Row1,Col1)
  R2   : Ring
  Row2 : DirectProductCategory(n, R2)
  Col2 : DirectProductCategory(m, R2)
  M2   : RectangularMatrixCategory(m,n,R2,Row2,Col2)

  Exports ==> with
    map: (R1 -> R2,M1) -> M2
      ++ \spad{map(f,m)} applies the function \spad{f} to the elements of the
      ++ matrix \spad{m}.
    reduce: ((R1,R2) -> R2,M1,R2) -> R2
      ++ \spad{reduce(f,m,r)} returns a matrix \spad{n} where
      ++ \spad{n[i,j] = f(m[i,j],r)} for all indices spad{i} and \spad{j}.

  Implementation ==> add
    minr ==> minRowIndex
    maxr ==> maxRowIndex
    minc ==> minColIndex
    maxc ==> maxColIndex

    map(f,mat) ==
      ans : M2 := new(m,n,0)$Matrix(R2) pretend M2
      for i in minr(mat)..maxr(mat) for k in minr(ans)..maxr(ans) repeat
        for j in minc(mat)..maxc(mat) for l in minc(ans)..maxc(ans) repeat
          qsetelt_!(ans pretend Matrix R2,k,l,f qelt(mat,i,j))
      ans

    reduce(f,mat,ident) ==
      s := ident
      for i in minr(mat)..maxr(mat) repeat
       for j in minc(mat)..maxc(mat) repeat
         s := f(qelt(mat,i,j),s)
      s
```

———-

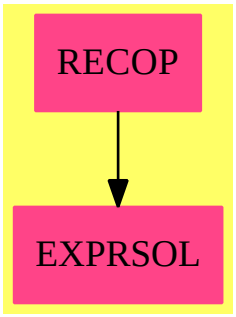
— RMCAT2.dotabb —

```
"RMCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RMCAT2"]
"RMATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RMATCAT"]
"RMCAT2" -> "RMATCAT"
```

19.28 package RECOP RecurrenceOperator

The package defined in this file provide an operator for the n^{th} term of a recurrence and an operator for the coefficient of x^n in a function specified by a functional equation.

19.28.1 RecurrenceOperator (RECOP)



Exports:

getEq	getOp	evalADE	evalRec
getShiftRec	numberOfValuesNeeded	shiftInfoRec	

— package RECOP RecurrenceOperator —

```
)abbrev package RECOP RecurrenceOperator
++ Author: Martin Rubey
++ Description:
++ This package provides an operator for the n-th term of a recurrence and an
++ operator for the coefficient of x^n in a function specified by a functional
++ equation.

RecurrenceOperator(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
  F: Join(FunctionSpace R, AbelianMonoid, RetractableTo Integer, _
          RetractableTo Symbol, PartialDifferentialRing Symbol, _
          CombinatorialOpsCategory)
--RecurrenceOperator(F): Exports == Implementation where
--  F: Join(ExpressionSpace, AbelianMonoid, RetractableTo Integer,
--          RetractableTo Symbol, PartialDifferentialRing Symbol)

  Exports == with
```

```
    evalRec: (BasicOperator, Symbol, F, F, F, List F) -> F
      ++ \spad{evalRec(u, dummy, n, n0, eq, values)} creates an expression that
      ++ stands for u(n0), where u(n) is given by the equation eq. However, for
      ++ technical reasons the variable n has to be replaced by a dummy
      ++ variable dummy in eq. The argument values specifies the initial values
      ++ of the recurrence u(0), u(1),...
      ++ For the moment we don't allow recursions that contain u inside of
      ++ another operator.

    evalADE: (BasicOperator, Symbol, F, F, F, List F) -> F
      ++ \spad{evalADE(f, dummy, x, n, eq, values)} creates an expression that
      ++ stands for the coefficient of x^n in the Taylor expansion of f(x),
      ++ where f(x) is given by the functional equation eq. However, for
      ++ technical reasons the variable x has to be replaced by a dummy
      ++ variable dummy in eq. The argument values specifies the first few
      ++ Taylor coefficients.

    getEq: F -> F
      ++ \spad{getEq f} returns the defining equation, if f represents the
      ++ coefficient of an ADE or a recurrence.

    getOp: F -> BasicOperator
      ++ \spad{getOp f}, if f represents the coefficient of a recurrence or
      ++ ADE, returns the operator representing the solution


-- should be local
    numberOfValuesNeeded: (Integer, BasicOperator, Symbol, F) -> Integer

-- should be local
    if R has Ring then
      getShiftRec: (BasicOperator, Kernel F, Symbol) -> Union(Integer, "failed")

      shiftInfoRec: (BasicOperator, Symbol, F) ->
                     Record(max: Union(Integer, "failed"),
                            ord: Union(Integer, "failed"),
                            ker: Kernel F)

  Implementation == add
\getchunk{implementation: RecurrenceOperator}
```

———-

19.28.2 Defining new operators

We define two new operators, one for recurrences, the other for functional equations. The operators for recurrences represents the n^{th} term of the corresponding sequence, the other the coefficient of x^n in the Taylor series expansion.

— **implementation: RecurrenceOperator** —

```
oprecur := operator("rootOfRec"::Symbol)$BasicOperator

opADE := operator("rootOfADE"::Symbol)$BasicOperator

setProperty(oprecur, "%dummyVar", 2 pretend None)
setProperty(opADE, "%dummyVar", 2 pretend None)
```

———-

Setting these properties implies that the second and third arguments of oprecur are dummy variables and affects `tower$ES`: the second argument will not appear in `tower$ES`, if it does not appear in any argument but the first and second. The third argument will not appear in `tower$ES`, if it does not appear in any other argument. (`%defsum` is a good example)

The arguments of the two operators are as follows:

1. eq, i.e. the vanishing expression

   — **implementation: RecurrenceOperator** —

   ```
   eqAsF: List F -> F
   eqAsF l == l.1
   ```

   ———-

2. dummy, a dummy variable to make substitutions possible

   — **implementation: RecurrenceOperator** —

   ```
   dummy: List F -> Symbol
   dummy l == retract(l.2)@Symbol

   dummyAsF: List F -> F
   dummyAsF l == l.2
   ```

   ———-

3. the variable for display

   — **implementation: RecurrenceOperator** —

   ```
   displayVariable: List F -> F
   displayVariable l == l.3
   ```

   ———-

4. operatorName(argument)

— **implementation: RecurrenceOperator** —

```
operatorName: List F -> BasicOperator
operatorName l == operator(kernels(l.4).1)

operatorNameAsF: List F -> F
operatorNameAsF l == l.4

operatorArgument: List F -> F
operatorArgument l == argument(kernels(l.4).1).1
```

———-

Concerning rootOfADE, note that although we have arg as argument of the operator, it is intended to indicate the coefficient, not the argument of the power series.

5. values in reversed order.

   - rootOfRec: maybe values should be preceded by the index of the first given value. Currently, the last value is interpreted as $f(0)$.
   - rootOfADE: values are the first few coefficients of the power series expansion in order.

— **implementation: RecurrenceOperator** —

```
initialValues: List F -> List F
initialValues l == rest(l, 4)
```

———-

19.28.3 Recurrences

Extracting some information from the recurrence

We need to find out wether we can determine the next term of the sequence, and how many initial values are necessary.

— **implementation: RecurrenceOperator** —

```
if R has Ring then
  getShiftRec(op: BasicOperator, f: Kernel F, n: Symbol)
              : Union(Integer, "failed") ==
      a := argument f
      if every?(z +-> freeOf?(z, n::F), a) then return 0
```

```
          if #a ~= 1 then error "RECOP: operator should have only one argument"

          p := univariate(a.1, retract(n::F)@Kernel(F))
          if denominator p ~= 1 then return "failed"

          num := numer p

          if degree num = 1 and coefficient(num, 1) = 1
             and every?(z +-> freeOf?(z, n::F), coefficients num)
          then return retractIfCan(coefficient(num, 0))
          else return "failed"

-- if the recurrence is of the form
-- $p(n, f(n+m-o), f(n+m-o+1), \dots, f(n+m)) = 0$
-- in which case shiftInfoRec returns [m, o, f(n+m)].

      shiftInfoRec(op: BasicOperator, argsym: Symbol, eq: F):
                  Record(max: Union(Integer, "failed"),
                         ord: Union(Integer, "failed"),
                         ker: Kernel F) ==

-- ord and ker are valid only if all shifts are Integers
-- ker is the kernel of the maximal shift.
        maxShift: Integer
        minShift: Integer
        nextKernel: Kernel F

-- We consider only those kernels that have op as operator. If there is none,
-- we raise an error. For the moment we don't allow recursions that contain op
-- inside of another operator.

        error? := true

        for f in kernels eq repeat
          if is?(f, op) then
            shift := getShiftRec(op, f, argsym)
            if error? then
              error? := false
              nextKernel := f
              if shift case Integer then
                maxShift := shift
                minShift := shift
              else return ["failed", "failed", nextKernel]
            else
              if shift case Integer then
                if maxShift < shift then
                  maxShift := shift
                  nextKernel := f
                if minShift > shift then
```

```
                minShift := shift
              else return ["failed", "failed", nextKernel]

        if error? then error "evalRec: equation does not contain operator"

        [maxShift, maxShift - minShift, nextKernel]
```

———-

Evaluating a recurrence

— **implementation: RecurrenceOperator** —

```
      evalRec(op, argsym, argdisp, arg, eq, values) ==
        if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
           or (n < 0)
        then
          shiftInfo := shiftInfoRec(op, argsym, eq)

          if (shiftInfo.ord case "failed") or ((shiftInfo.ord)::Integer > 0)
          then
            kernel(oprecur,
                   append([eq, argsym::F, argdisp, op(arg)], values))
          else
            p := univariate(eq, shiftInfo.ker)
            num := numer p

-- If the degree is 1, we can return the function explicitly.

            if degree num = 1 then
              eval(-coefficient(num, 0)/coefficient(num, 1), argsym::F,
                   arg::F-(shiftInfo.max)::Integer::F)
            else
              kernel(oprecur,
                     append([eq, argsym::F, argdisp, op(arg)], values))
        else
          len: Integer := #values
          if n < len
          then values.(len-n)
          else
            shiftInfo := shiftInfoRec(op, argsym, eq)

            if shiftInfo.max case Integer then
              p := univariate(eq, shiftInfo.ker)

              num := numer p

              if degree num = 1 then
```

```
            next := -coefficient(num, 0)/coefficient(num, 1)
            nextval := eval(next, argsym::F,
                            (len-(shiftInfo.max)::Integer)::F)
            newval := eval(nextval, op,
                       z+->evalRec(op, argsym, argdisp, z, eq, values))
            evalRec(op, argsym, argdisp, arg, eq, cons(newval, values))
          else
            kernel(oprecur,
                   append([eq, argsym::F, argdisp, op(arg)], values))

        else
          kernel(oprecur,
                 append([eq, argsym::F, argdisp, op(arg)], values))

  numberOfValuesNeeded(numberOfValues: Integer,
                       op: BasicOperator,
                        argsym: Symbol, eq: F): Integer ==
    order := shiftInfoRec(op, argsym, eq).ord
    if order case Integer
    then min(numberOfValues, retract(order)@Integer)
    else numberOfValues


else
  evalRec(op, argsym, argdisp, arg, eq, values) ==
    kernel(oprecur,
           append([eq, argsym::F, argdisp, op(arg)], values))

  numberOfValuesNeeded(numberOfValues: Integer,
                       op: BasicOperator, argsym: Symbol, eq: F): Integer ==
    numberOfValues
```

———-

Setting the evaluation property of oprecur

irecur is just a wrapper that allows us to write a recurrence relation as an operator.

— **implementation: RecurrenceOperator** —

```
irecur: List F -> F
irecur l ==
  evalRec(operatorName l,
          dummy l, displayVariable l,
          operatorArgument l, eqAsF l, initialValues l)

evaluate(oprecur, irecur)$BasicOperatorFunctions1(F)
```

———-

Displaying a recurrence relation

— **implementation: RecurrenceOperator** —

```
ddrec: List F -> OutputForm
ddrec l ==
  op := operatorName l
  values := reverse l
  eq := eqAsF l

  numberOfValues := numberOfValuesNeeded(#values-4, op, dummy l, eq)

  vals: List OutputForm
       := cons(eval(eq, dummyAsF l, displayVariable l)::OutputForm = _
               0::OutputForm,
               [elt(op::OutputForm, [(i-1)::OutputForm]) = _
                (values.i)::OutputForm
                for i in 1..numberOfValues])

  bracket(hconcat([(operatorNameAsF l)::OutputForm,
                   ": ",
                   commaSeparate vals]))

setProperty(oprecur, "%specialDisp",
            ddrec@(List F -> OutputForm) pretend None)
```

———-

19.28.4 Functional Equations

Determining the number of initial values for ADE's

We use Joris van der Hoeven's instructions for ADE's. Given $Q = p(f, f', \ldots, f^{(r)})$ we first need to differentiate Q with respect to $f^{(i)}$ for $i \in \{0, 1, \ldots, r\}$, plug in the given truncated power series solution and determine the valuation.

— **NOTYET implementation: RecurrenceOperator** —

```
getValuation(op, argsym, eq, maxorder, values): Integer ==
    max: Integer := -1;
    ker: Kernel F
    for i in 0..maxorder repeat
        ker := D(op(argsym), argsym, i)::Kernel F
```

```
            pol := univariate(eq, ker)
            dif := D pol
            ground numer(dif.D(op(argsym), argsym, i))
```

———-

Extracting some information from the functional equation

getOrder returns the maximum derivative of op occurring in f.

— implementation: RecurrenceOperator —

```
    getOrder(op: BasicOperator, f: Kernel F): NonNegativeInteger ==
      res: NonNegativeInteger := 0
      g := f
      while is?(g, %diff) repeat
        g := kernels(argument(g).1).1
        res := res+1

      if is?(g, op) then res else 0
```

———-

Extracting a coefficient given a functional equation

— implementation: RecurrenceOperator —

```
    evalADE(op, argsym, argdisp, arg, eq, values) ==
      if not freeOf?(eq, retract(argdisp)@Symbol)
      then error "RECOP: The argument should not be used in the equation of the_
 ADE"

      if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
      then
-- try to determine the necessary number of initial values
         keq := kernels eq
         order := getOrder(op, keq.1)
         for k in rest keq repeat order := max(order, getOrder(op, k))

         p: Fraction SparseUnivariatePolynomial F
           := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F

         if one? degree numer p
-- the equation is holonomic
         then kernel(opADE,
```

```
                    append([eq, argsym::F, argdisp, op(arg)],
                           reverse first(reverse values, order)))
         else kernel(opADE,
                    append([eq, argsym::F, argdisp, op(arg)], values))
      else
        if n < 0
        then 0
        else
          keq := kernels eq
          order := getOrder(op, keq.1)
--          output(hconcat("The order is ", order::OutputForm))$OutputPackage
          for k in rest keq repeat order := max(order, getOrder(op, k))

          p: Fraction SparseUnivariatePolynomial F
            := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F

--          output(hconcat("p: ", p::OutputForm))$OutputPackage

          if degree numer p > 1
          then
--            kernel(opADE,
--                   append([eq, argsym::F, argdisp, op(arg)], values))

            s := seriesSolve(eq, op, argsym, reverse values)
                          $ExpressionSolve(R, F,
                                           UnivariateFormalPowerSeries F,
                                           UnivariateFormalPowerSeries
                                             SparseUnivariatePolynomialExpressions F)

            elt(s, n::Integer::NonNegativeInteger)

          else
            s := seriesSolve(eq, op, argsym, first(reverse values, order))
                          $ExpressionSolve(R, F,
                                           UnivariateFormalPowerSeries F,
                                           UnivariateFormalPowerSeries
                                             SparseUnivariatePolynomialExpressions F)

            elt(s, n::Integer::NonNegativeInteger)


    iADE: List F -> F
-- This is just a wrapper that allows us to write a recurrence relation as an
-- operator.
    iADE l ==
      evalADE(operatorName l,
              dummy l, displayVariable l,
              operatorArgument l, eqAsF l, initialValues l)

    evaluate(opADE, iADE)$BasicOperatorFunctions1(F)
```

```
getEq(f: F): F ==
  ker := kernels f
  if one?(#ker) and _
     (is?(operator(ker.1), "rootOfADE"::Symbol) or _
      is?(operator(ker.1), "rootOfRec"::Symbol)) then
    l := argument(ker.1)
    eval(eqAsF l, dummyAsF l, displayVariable l)
  else
    error "getEq: argument should be a single rootOfADE or rootOfRec object"

getOp(f: F): BasicOperator ==
  ker := kernels f
  if one?(#ker) and _
     (is?(operator(ker.1), "rootOfADE"::Symbol) or _
      is?(operator(ker.1), "rootOfRec"::Symbol)) then
    operatorName argument(ker.1)
  else
    error "getOp: argument should be a single rootOfADE or rootOfRec object"
```

———-

Displaying a functional equation

— **implementation: RecurrenceOperator** —

```
ddADE: List F -> OutputForm
ddADE l ==
  op := operatorName l
  values := reverse l

  vals: List OutputForm
       := cons(eval(eqAsF l, dummyAsF l, displayVariable l)::OutputForm = _
               0::OutputForm,
               [eval(D(op(dummyAsF l), dummy l, i), _
                     dummyAsF l=0)::OutputForm = _
                (values.(i+1))::OutputForm * _
                factorial(box(i::R::F)$F)::OutputForm _
                for i in 0..min(4,#values-5)])

  bracket(hconcat([bracket((displayVariable l)::OutputForm ** _
                           (operatorArgument l)::OutputForm),
                   (op(displayVariable l))::OutputForm, ": ",
                   commaSeparate vals]))

setProperty(opADE, "%specialDisp",
```

```
            ddADE@(List F -> OutputForm) pretend None)
```

———-

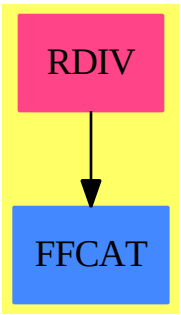
— **RECOP.dotabb** —

```
"RECOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RECOP"]
"EXPRSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPRSOL"]
"RECOP" -> "EXPRSOL"
```

———-

19.29 package RDIV ReducedDivisor

19.29.1 ReducedDivisor (RDIV)



Exports:
order

— **package RDIV ReducedDivisor** —

```
)abbrev package RDIV ReducedDivisor
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 8 November 1994
++ Description:
++ Finds the order of a divisor over a finite field

ReducedDivisor(F1, UP, UPUP, R, F2): Exports == Implementation where
  F1    : Field
  UP    : UnivariatePolynomialCategory F1
  UPUP  : UnivariatePolynomialCategory Fraction UP
  R     : FunctionFieldCategory(F1, UP, UPUP)
```

```
  F2    : Join(Finite, Field)

  N     ==> NonNegativeInteger
  FD    ==> FiniteDivisor(F1, UP, UPUP, R)
  UP2   ==> SparseUnivariatePolynomial F2
  UPUP2 ==> SparseUnivariatePolynomial Fraction UP2

  Exports ==> with
    order: (FD, UPUP, F1 -> F2) -> N
      ++ order(f,u,g) \undocumented

  Implementation ==> add
    algOrder : (FD, UPUP, F1 -> F2)  -> N
    rootOrder: (FD, UP, N, F1 -> F2) -> N

-- pp is not necessarily monic
    order(d, pp, f) ==
      (r := retractIfCan(reductum pp)@Union(Fraction UP, "failed"))
        case "failed" => algOrder(d, pp, f)
      rootOrder(d, - retract(r::Fraction(UP) / leadingCoefficient pp)@UP,
                degree pp, f)

    algOrder(d, modulus, reduce) ==
      redmod := map(reduce, modulus)$MultipleMap(F1,UP,UPUP,F2,UP2,UPUP2)
      curve  := AlgebraicFunctionField(F2, UP2, UPUP2, redmod)
      order(map(reduce,
              d)$FiniteDivisorFunctions2(F1,UP,UPUP,R,F2,UP2,UPUP2,curve)
                                )$FindOrderFinite(F2, UP2, UPUP2, curve)

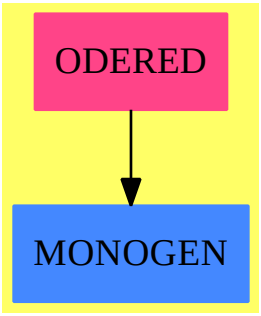
    rootOrder(d, radicand, n, reduce) ==
      redrad := map(reduce,
           radicand)$UnivariatePolynomialCategoryFunctions2(F1,UP,F2,UP2)
      curve  := RadicalFunctionField(F2, UP2, UPUP2, redrad::Fraction UP2, n)
      order(map(reduce,
              d)$FiniteDivisorFunctions2(F1,UP,UPUP,R,F2,UP2,UPUP2,curve)
                                )$FindOrderFinite(F2, UP2, UPUP2, curve)
```

— RDIV.dotabb —

```
"RDIV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDIV"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"RDIV" -> "FFCAT"
```

19.30 package ODERED ReduceLODE

19.30.1 ReduceLODE (ODERED)



Exports:
reduceLODE

— package ODERED ReduceLODE —

```
)abbrev package ODERED ReduceLODE
++ Author: Manuel Bronstein
++ Date Created: 19 August 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ Elimination of an algebraic from the coefficentss
++ of a linear ordinary differential equation.

ReduceLODE(F, L, UP, A, LO): Exports == Implementation where
  F : Field
  L : LinearOrdinaryDifferentialOperatorCategory F
  UP: UnivariatePolynomialCategory F
  A : MonogenicAlgebra(F, UP)
  LO: LinearOrdinaryDifferentialOperatorCategory A

  V ==> Vector F
  M ==> Matrix L

  Exports ==> with
    reduceLODE: (LO, A) -> Record(mat:M, vec:V)
      ++ reduceLODE(op, g) returns \spad{[m, v]} such that
      ++ any solution in \spad{A} of \spad{op z = g}
      ++ is of the form \spad{z = (z_1,...,z_m) . (b_1,...,b_m)} where
      ++ the \spad{b_i's} are the basis of \spad{A} over \spad{F} returned
      ++ by \spadfun{basis}() from \spad{A}, and the \spad{z_i's} satisfy the
      ++ differential system \spad{M.z = v}.
```

```
  Implementation ==> add
    matF2L: Matrix F -> M

    diff := D()$L

-- coerces a matrix of elements of F into a matrix of (order 0) L.0.D.0's
    matF2L m ==
      map((f1:F):L+->f1::L, m)$MatrixCategoryFunctions2(F, V, V, Matrix F,
                                                 L, Vector L, Vector L, M)

-- This follows the algorithm and notation of
--  "The Risch Differential Equation on an Algebraic Curve", M. Bronstein,
-- in 'Proceedings of ISSAC '91', Bonn, BRD, ACM Press, pp.241-246, July 1991.
    reduceLODE(l, g) ==
      n := rank()$A
-- md is the basic differential matrix (D x I + Dy)
      md := matF2L transpose derivationCoordinates(basis(), (f1:F):F+->diff f1)
      for i in minRowIndex md .. maxRowIndex md
        for j in minColIndex md .. maxColIndex md repeat
          md(i, j) := diff + md(i, j)
-- mdi will go through the successive powers of md
      mdi := copy md
      sys := matF2L(transpose regularRepresentation coefficient(l, 0))
      for i in 1..degree l repeat
        sys := sys +
                matF2L(transpose regularRepresentation coefficient(l, i)) * mdi
        mdi := md * mdi
      [sys, coordinates g]
```

———-

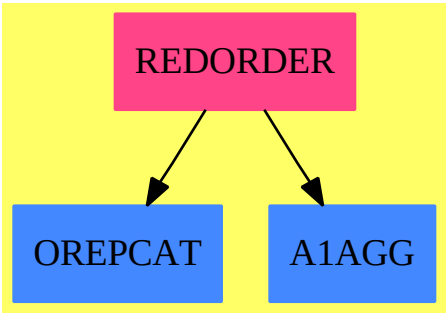
— ODERED.dotabb —

```
"ODERED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODERED"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"ODERED" -> "MONOGEN"
```

———-

19.31 package REDORDER ReductionOfOrder

19.31.1 ReductionOfOrder (REDORDER)



Exports:
ReduceOrder

— package REDORDER ReductionOfOrder —

```
)abbrev package REDORDER ReductionOfOrder
++ Author: Manuel Bronstein
++ Date Created: 4 November 1991
++ Date Last Updated: 3 February 1994
++ Keywords: differential equation, ODE
++ Description:
++ \spadtype{ReductionOfOrder} provides
++ functions for reducing the order of linear ordinary differential equations
++ once some solutions are known.

ReductionOfOrder(F, L): Exports == Impl where
  F: Field
  L: LinearOrdinaryDifferentialOperatorCategory F

  Z ==> Integer
  A ==> PrimitiveArray F

  Exports ==> with
    ReduceOrder: (L, F) -> L
      ++ ReduceOrder(op, s) returns \spad{op1} such that for any solution
      ++ \spad{z} of \spad{op1 z = 0}, \spad{y = s \int z} is a solution of
      ++ \spad{op y = 0}. \spad{s} must satisfy \spad{op s = 0}.
    ReduceOrder: (L, List F) -> Record(eq:L, op:List F)
      ++ ReduceOrder(op, [f1,...,fk]) returns \spad{[op1,[g1,...,gk]]} such that
      ++ for any solution \spad{z} of \spad{op1 z = 0},
      ++ \spad{y = gk \int(g_{k-1} \int(... \int(g1 \int z)...)} is a solution
      ++ of \spad{op y = 0}. Each \spad{fi} must satisfy \spad{op fi = 0}.
```

```
  Impl ==> add
    ithcoef   : (L, Z, A) -> F
    locals    : (A, Z, Z) -> F
    localbinom: (Z, Z) -> Z

    diff := D()$L

    localbinom(j, i) == (j > i => binomial(j, i+1); 0)
    locals(s, j, i)  == (j > i => qelt(s, j - i - 1); 0)

    ReduceOrder(l:L, sols:List F) ==
      empty? sols => [l, empty()]
      neweq := ReduceOrder(l, sol := first sols)
      rec := ReduceOrder(neweq, [diff(s / sol) for s in rest sols])
      [rec.eq, concat_!(rec.op, sol)]

    ithcoef(eq, i, s) ==
      ans:F := 0
      while eq ^= 0 repeat
          j   := degree eq
          ans := ans + localbinom(j, i) * locals(s,j,i) * leadingCoefficient eq
          eq  := reductum eq
      ans

    ReduceOrder(eq:L, sol:F) ==
      s:A := new(n := degree eq, 0)         -- will contain derivatives of sol
      si := sol                             -- will run through the derivatives
      qsetelt_!(s, 0, si)
      for i in 1..(n-1)::NonNegativeInteger repeat
          qsetelt_!(s, i, si := diff si)
      ans:L := 0
      for i in 0..(n-1)::NonNegativeInteger repeat
          ans := ans + monomial(ithcoef(eq, i, s), i)
      ans
```

———-

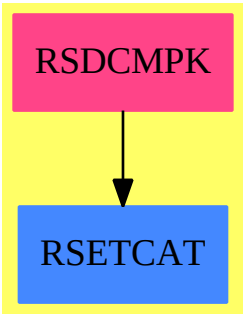
— REDORDER.dotabb —

```
"REDORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REDORDER"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"REDORDER" -> "OREPCAT"
"REDORDER" -> "A1AGG"
```

———-

19.32 package RSDCMPK RegularSetDecompositionPackage

19.32.1 RegularSetDecompositionPackage (RSDCMPK)



Exports:

KrullNumber	algebraicDecompose	convert	decompose
internalDecompose	numberOfVariables	printInfo	transcendentalDecompose
upDateBranches			

— package RSDCMPK RegularSetDecompositionPackage —

```
)abbrev package RSDCMPK RegularSetDecompositionPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/16/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A package providing a new algorithm for solving polynomial systems
++ by means of regular chains. Two ways of solving are proposed:
++ in the sense of Zariski closure (like in Kalkbrener's algorithm)
++ or in the sense of the regular zeros (like in Wu, Wang or Lazard
++ methods). This algorithm is valid for nay type
++ of regular set. It does not care about the way a polynomial is
++ added in an regular set, or how two quasi-components are compared
++ (by an inclusion-test), or how the invertibility test is made in
++ the tower of simple extensions associated with a regular set.
++ These operations are realized respectively by the domain \spad{TS}
++ and the packages
```

```
++ \axiomType{QCMPACK}(R,E,V,P,TS) and \axiomType{RSETGCD}(R,E,V,P,TS).
++ The same way it does not care about the way univariate polynomial
++ gcd (with coefficients in the tower of simple extensions associated
++ with a regular set) are computed. The only requirement is that these
++ gcd need to have invertible initials (normalized or not).
++ WARNING. There is no need for a user to call diectly any operation
++ of this package since they can be accessed by the domain \axiom{TS}.
++ Thus, the operations of this package are not documented.

RegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : RegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  LP ==> List P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : TS)
  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Wip ==> Record(done: Split, todo: List LpWT)
  Branch ==> Record(eq: List P, tower: TS, ineq: List P)
  UBF ==> Union(Branch,"failed")
  Split ==> List TS
  iprintpack ==> InternalPrintPackage()
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)
  regsetgcdpack ==> RegularTriangularSetGcdPackage(R,E,V,P,TS)

  Exports ==  with

     KrullNumber: (LP, Split) -> N
     numberOfVariables: (LP, Split) -> N
     algebraicDecompose: (P,TS,B) -> Record(done: Split, todo: List LpWT)
     transcendentalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
     transcendentalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
     internalDecompose: (P,TS,N,B) -> Record(done: Split, todo: List LpWT)
     internalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
     internalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
     decompose: (LP, Split, B, B) -> Split
     decompose: (LP, Split, B, B, B, B, B) -> Split
     upDateBranches: (LP,Split,List LpWT,Wip,N) -> List LpWT
     convert: Record(val: List P,tower: TS) -> String
     printInfo: (List Record(val: List P,tower: TS), N) -> Void

  Implementation == add
```

```
KrullNumber(lp: LP, lts: Split): N ==
  ln: List N := [#(ts) for ts in lts]
  n := #lp + reduce(max,ln)

numberOfVariables(lp: LP, lts: Split): N ==
  lv: List V := variables([lp]$PS)
  for ts in lts repeat lv := concat(variables(ts), lv)
  # removeDuplicates(lv)

algebraicDecompose(p: P, ts: TS, clos?: B): Record(done: Split, todo: List LpWT) ==
  ground? p =>
    error " in algebraicDecompose$REGSET: should never happen !"
  v := mvar(p); n := #ts
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  ts_v := select(ts,v)::P
  if mdeg(p) < mdeg(ts_v)
    then
      lgwt := internalLastSubResultant(ts_v,p,ts_v_-,true,false)$regsetgcdpack
    else
      lgwt := internalLastSubResultant(p,ts_v,ts_v_-,true,false)$regsetgcdpack
  lts: Split := []
  llpwt: List LpWT := []
  for gwt in lgwt repeat
    g := gwt.val; us := gwt.tower
    zero? g =>
      error " in algebraicDecompose$REGSET: should never happen !!"
    ground? g => "leave"
    if mvar(g) = v then lts := concat(augment(members(ts_v_+),augment(g,us)),lts)
    h := leadingCoefficient(g,v)
    b: Boolean := purelyAlgebraic?(us)
    lsfp := squareFreeFactors(h)$polsetpack
    lus := augment(members(ts_v_+),augment(ts_v,us)@Split)
    for f in lsfp repeat
      ground? f => "leave"
      b and purelyAlgebraic?(f,us) => "leave"
      for vs in lus repeat
         llpwt := cons([[f,p],vs]$LpWT, llpwt)
  [lts,llpwt]

transcendentalDecompose(p: P, ts: TS,bound: N): Record(done: Split, todo: List LpWT) ==
  lts: Split
  if #ts < bound
    then
      lts := augment(p,ts)
    else
      lts := []
  llpwt: List LpWT := []
  [lts,llpwt]
```

```
transcendentalDecompose(p: P, ts: TS): Record(done: Split, todo: List LpWT) ==
  lts: Split:= augment(p,ts)
  llpwt: List LpWT := []
  [lts,llpwt]

internalDecompose(p: P, ts: TS,bound: N,clos?:B): Record(done: Split, todo: List LpWT) ==
  clos? => internalDecompose(p,ts,bound)
  internalDecompose(p,ts)

internalDecompose(p: P, ts: TS,bound: N): Record(done: Split, todo: List LpWT) ==
  -- ASSUME p not constant
  llpwt: List LpWT := []
  lts: Split := []
  -- EITHER mvar(p) is null
  if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
    then
      llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
      p := (p exquo lmp)::P
  ip := squareFreePart init(p); tp := tail p
  p := mainPrimitivePart p
  -- OR init(p) is null or not
  lbwt := invertible?(ip,ts)@(List BWT)
  for bwt in lbwt repeat
    bwt.val =>
      if algebraic?(mvar(p),bwt.tower)
        then
          rsl := algebraicDecompose(p,bwt.tower,true)
        else
          rsl := transcendentalDecompose(p,bwt.tower,bound)
      lts := concat(rsl.done,lts)
      llpwt := concat(rsl.todo,llpwt)
      -- purelyAlgebraicLeadingMonomial?(ip,bwt.tower) => "leave"  -- UNPROVED CRITERIA
      purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave" -- SAFE
      (not ground? ip) =>
        zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
        (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
    riv := removeZero(ip,bwt.tower)
    (zero? riv) =>
      zero? tp => lts := cons(bwt.tower,lts)
      (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
    llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
  [lts,llpwt]

internalDecompose(p: P, ts: TS): Record(done: Split, todo: List LpWT) ==
  -- ASSUME p not constant
  llpwt: List LpWT := []
  lts: Split := []
  -- EITHER mvar(p) is null
  if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
```

```
    then
      llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
      p := (p exquo lmp)::P
  ip := squareFreePart init(p); tp := tail p
  p := mainPrimitivePart p
  -- OR init(p) is null or not
  lbwt := invertible?(ip,ts)@(List BWT)
  for bwt in lbwt repeat
    bwt.val =>
      if algebraic?(mvar(p),bwt.tower)
        then
          rsl := algebraicDecompose(p,bwt.tower,false)
        else
          rsl := transcendentalDecompose(p,bwt.tower)
      lts := concat(rsl.done,lts)
      llpwt :=  concat(rsl.todo,llpwt)
      purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave"
      (not ground? ip) =>
        zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
        (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
    riv := removeZero(ip,bwt.tower)
    (zero? riv) =>
      zero? tp => lts := cons(bwt.tower,lts)
      (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
    llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
  [lts,llpwt]

decompose(lp: LP, lts: Split, clos?: B, info?: B): Split ==
  decompose(lp,lts,false,false,clos?,true,info?)

convert(lpwt: LpWT): String ==
  ls: List String := ["<", string((#(lpwt.val))::Z), ",", string((#(lpwt.tower))::Z), ">" ]
  concat ls

printInfo(toSee: List LpWT, n: N): Void ==
  lpwt := first toSee
  s: String := concat ["[", string((#toSee)::Z), " ", convert(lpwt)@String]
  m: N := #(lpwt.val)
  toSee := rest toSee
  for lpwt in toSee repeat
    m := m + #(lpwt.val)
    s := concat [s, ",", convert(lpwt)@String]
  s := concat [s, " -> |", string(m::Z), "|; {", string(n::Z),"}]"]
  iprint(s)$iprintpack
  void()

decompose(lp: LP, lts: Split, cleanW?: B, sqfr?: B, clos?: B, rem?: B, info?: B): Split ==
  -- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
  -- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
  -- if clos? then SOLVE in the closure sense
```

```
  -- if rem? then REDUCE the current p by using remainder
  -- if info? then PRINT info
  empty? lp => lts
  branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$quasicomppack
  empty? branches => []
  toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
  toSave: Split := []
  if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
  while (not empty? toSee) repeat
    if info? then printInfo(toSee,#toSave)
    lpwt := first toSee; toSee := rest toSee
    lp := lpwt.val; ts := lpwt.tower
    empty? lp =>
      toSave := cons(ts, toSave)
    p := first lp;  lp := rest lp
    if rem? and (not ground? p) and (not empty? ts)
       then
         p := remainder(p,ts).polnum
    p := removeZero(p,ts)
    zero? p => toSee := cons([lp,ts]$LpWT, toSee)
    ground? p => "leave"
    rsl := internalDecompose(p,ts,bound,clos?)
    toSee := upDateBranches(lp,toSave,toSee,rsl,bound)
  removeSuperfluousQuasiComponents(toSave)$quasicomppack

upDateBranches(leq:LP,lts:Split,current:List LpWT,wip: Wip,n:N): List LpWT ==
  newBranches: List LpWT := wip.todo
  newComponents: Split := wip.done
  branches1, branches2:  List LpWT
  branches1 := []; branches2  := []
  for branch in newBranches repeat
    us := branch.tower
    #us > n => "leave"
    newleq := sort(infRittWu?,concat(leq,branch.val))
    --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
    --any?(ground?,foo)  => "leave"
    branches1 := cons([newleq,us]$LpWT, branches1)
  for us in newComponents repeat
    #us > n => "leave"
    subQuasiComponent?(us,lts)$quasicomppack => "leave"
    --newleq := leq
    --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
    --any?(ground?,foo)  => "leave"
    branches2 := cons([leq,us]$LpWT, branches2)
  empty? branches1 =>
    empty? branches2 => current
    concat(branches2, current)
  branches := concat [branches2, branches1, current]
  -- branches := concat(branches,current)
  removeSuperfluousCases(branches)$quasicomppack
```

———-

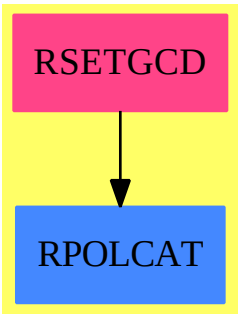
— **RSDCMPK.dotabb** —

```
"RSDCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RSDCMPK"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
"RSDCMPK" -> "RSETCAT"
```

———-

19.33 package RSETGCD RegularTriangularSetGcdPackage

19.33.1 RegularTriangularSetGcdPackage (RSETGCD)



Exports:

stopTableGcd!	stopTableInvSet!	integralLastSubResultant
prepareSubResAlgo	startTableGcd!	startTableInvSet!
toseInvertible?	toseInvertibleSet	toseLastSubResultant
toseSquareFreePart		

— **package RSETGCD RegularTriangularSetGcdPackage** —

```
)abbrev package RSETGCD RegularTriangularSetGcdPackage
++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 08/30/1998
++ Date Last Updated: 12/15/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
```

```
++ Keywords:
++ References :
++  [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++      algebraic towers of simple extensions" In proceedings of AAECC11
++      Paris, 1995.
++  [2] M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++      d'extensions simples et resolution des systemes d'equations
++      algebriques" These, Universite P.etM. Curie, Paris, 1997.
++  [3] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ An internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.

RegularTriangularSetGcdPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : RegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  LP ==> List P
  PtoP ==> P -> P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : TS)
  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Branch ==> Record(eq: List P, tower: TS, ineq: List P)
  UBF ==> Union(Branch,"failed")
  Split ==> List TS
  KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
  EntryGcd ==> List PWT
  HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
  KeyInvSet ==> Record(arg1: P, arg3: TS)
  EntryInvSet ==> List TS
  HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)

  Exports ==  with
     startTableGcd!: (S,S,S) -> Void
         ++ \axiom{startTableGcd!(s1,s2,s3)}
         ++ is an internal subroutine, exported only for developement.
     stopTableGcd!: () -> Void
         ++ \axiom{stopTableGcd!()}
         ++ is an internal subroutine, exported only for developement.
```

```
    startTableInvSet!: (S,S,S) -> Void
        ++ \axiom{startTableInvSet!(s1,s2,s3)}
        ++ is an internal subroutine, exported only for developement.
    stopTableInvSet!: () -> Void
        ++ \axiom{stopTableInvSet!()} is an internal subroutine,
        ++ exported only for developement.
    prepareSubResAlgo: (P,P,TS) -> List LpWT
        ++ \axiom{prepareSubResAlgo(p1,p2,ts)}
        ++ is an internal subroutine, exported only for developement.
    internalLastSubResultant: (P,P,TS,B,B) -> List PWT
        ++ \axiom{internalLastSubResultant(p1,p2,ts,inv?,break?)}
        ++ is an internal subroutine, exported only for developement.
    internalLastSubResultant: (List LpWT,V,B) -> List PWT
        ++ \axiom{internalLastSubResultant(lpwt,v,flag)} is an internal
        ++ subroutine, exported only for developement.
    integralLastSubResultant: (P,P,TS) -> List PWT
        ++ \axiom{integralLastSubResultant(p1,p2,ts)}
        ++ is an internal subroutine, exported only for developement.
    toseLastSubResultant: (P,P,TS) -> List PWT
        ++ \axiom{toseLastSubResultant(p1,p2,ts)} has the same specifications
        ++ as lastSubResultant from RegularTriangularSetCategory.
    toseInvertible?: (P,TS) -> B
        ++ \axiom{toseInvertible?(p1,p2,ts)} has the same specifications as
        ++ invertible? from RegularTriangularSetCategory.
    toseInvertible?: (P,TS) -> List BWT
        ++ \axiom{toseInvertible?(p1,p2,ts)} has the same specifications as
        ++ invertible? from RegularTriangularSetCategory.
    toseInvertibleSet: (P,TS) -> Split
        ++ \axiom{toseInvertibleSet(p1,p2,ts)} has the same specifications as
        ++ invertibleSet from RegularTriangularSetCategory.
    toseSquareFreePart: (P,TS) -> List PWT
        ++ \axiom{toseSquareFreePart(p,ts)} has the same specifications as
        ++ squareFreePart from RegularTriangularSetCategory.

  Implementation == add

    startTableGcd!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$HGcd
      printInfo!(ok,ko)$HGcd
      startStats!(domainName)$HGcd
      void()

    stopTableGcd!(): Void ==
      if makingStats?()$HGcd then printStats!()$HGcd
      clearTable!()$HGcd

    startTableInvSet!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$HInvSet
      printInfo!(ok,ko)$HInvSet
      startStats!(domainName)$HInvSet
```

```
    void()

  stopTableInvSet!(): Void ==
    if makingStats?()$HInvSet then printStats!()$HInvSet
    clearTable!()$HInvSet

  toseInvertible?(p:P,ts:TS): Boolean ==
    q := primitivePart initiallyReduce(p,ts)
    zero? q => false
    normalized?(q,ts) => true
    v := mvar(q)
    not algebraic?(v,ts) =>
      toCheck: List BWT := toseInvertible?(p,ts)@(List BWT)
      for bwt in toCheck repeat
        bwt.val = false => return false
      return true
    ts_v := select(ts,v)::P
    ts_v_- := collectUnder(ts,v)
    lgwt := internalLastSubResultant(ts_v,q,ts_v_-,false,true)
    for gwt in lgwt repeat
      g := gwt.val;
      (not ground? g) and (mvar(g) = v) =>
        return false
    true

  toseInvertible?(p:P,ts:TS): List BWT ==
    q := primitivePart initiallyReduce(p,ts)
    zero? q => [[false,ts]$BWT]
    normalized?(q,ts) => [[true,ts]$BWT]
    v := mvar(q)
    not algebraic?(v,ts) =>
      lbwt: List BWT := []
      toCheck: List BWT := toseInvertible?(init(q),ts)@(List BWT)
      for bwt in toCheck repeat
        bwt.val => lbwt := cons(bwt,lbwt)
        newq := removeZero(q,bwt.tower)
        zero? newq => lbwt := cons(bwt,lbwt)
        lbwt := concat(toseInvertible?(newq,bwt.tower)@(List BWT), lbwt)
      return lbwt
    ts_v := select(ts,v)::P
    ts_v_- := collectUnder(ts,v)
    ts_v_+ := collectUpper(ts,v)
    lgwt := internalLastSubResultant(ts_v,q,ts_v_-,false,false)
    lbwt: List BWT := []
    for gwt in lgwt repeat
      g := gwt.val; ts := gwt.tower
      (ground? g) or (mvar(g) < v) =>
        ts := internalAugment(ts_v,ts)
        ts := internalAugment(members(ts_v_+),ts)
        lbwt := cons([true, ts]$BWT,lbwt)
```

```
      g := mainPrimitivePart g
      ts_g := internalAugment(g,ts)
      ts_g := internalAugment(members(ts_v_+),ts_g)
      -- USE internalAugment with parameters ??
      lbwt := cons([false, ts_g]$BWT,lbwt)
      h := lazyPquo(ts_v,g)
      (ground? h) or (mvar(h) < v) => "leave"
      h := mainPrimitivePart h
      ts_h := internalAugment(h,ts)
      ts_h := internalAugment(members(ts_v_+),ts_h)
      -- USE internalAugment with parameters ??
      -- CAN BE OPTIMIZED if the input tower is separable
      inv := toseInvertible?(q,ts_h)@(List BWT)
      lbwt := concat([bwt for bwt in inv | bwt.val],lbwt)
    sort((x,y) +-> x.val < y.val,lbwt)

  toseInvertibleSet(p:P,ts:TS): Split ==
    k: KeyInvSet := [p,ts]
    e := extractIfCan(k)$HInvSet
    e case EntryInvSet => e::EntryInvSet
    q := primitivePart initiallyReduce(p,ts)
    zero? q => []
    normalized?(q,ts) => [ts]
    v := mvar(q)
    toSave: Split := []
    not algebraic?(v,ts) =>
      toCheck: List BWT := toseInvertible?(init(q),ts)@(List BWT)
      for bwt in toCheck repeat
        bwt.val => toSave := cons(bwt.tower,toSave)
        newq := removeZero(q,bwt.tower)
        zero? newq => "leave"
        toSave := concat(toseInvertibleSet(newq,bwt.tower), toSave)
      toSave := removeDuplicates toSave
      return algebraicSort(toSave)$quasicomppack
    ts_v := select(ts,v)::P
    ts_v_- := collectUnder(ts,v)
    ts_v_+ := collectUpper(ts,v)
    lgwt := internalLastSubResultant(ts_v,q,ts_v_-,false,false)
    for gwt in lgwt repeat
      g := gwt.val; ts := gwt.tower
      (ground? g) or (mvar(g) < v) =>
        ts := internalAugment(ts_v,ts)
        ts := internalAugment(members(ts_v_+),ts)
        toSave := cons(ts,toSave)
      g := mainPrimitivePart g
      h := lazyPquo(ts_v,g)
      h := mainPrimitivePart h
      (ground? h) or (mvar(h) < v) => "leave"
      ts_h := internalAugment(h,ts)
      ts_h := internalAugment(members(ts_v_+),ts_h)
```

```
          inv := toseInvertibleSet(q,ts_h)
          toSave := removeDuplicates concat(inv,toSave)
        toSave := algebraicSort(toSave)$quasicomppack
        insert!(k,toSave)$HInvSet
        toSave

      toseSquareFreePart_wip(p:P, ts: TS): List PWT ==
      -- ASSUME p is not constant and mvar(p) > mvar(ts)
      -- ASSUME init(p) is invertible w.r.t. ts
      -- ASSUME p is mainly primitive
--        one? mdeg(p) => [[p,ts]$PWT]
        mdeg(p) = 1 => [[p,ts]$PWT]
        v := mvar(p)$P
        q: P := mainPrimitivePart D(p,v)
        lgwt: List PWT := internalLastSubResultant(p,q,ts,true,false)
        lpwt : List PWT := []
        sfp : P
        for gwt in lgwt repeat
          g := gwt.val; us := gwt.tower
          (ground? g) or (mvar(g) < v) =>
            lpwt := cons([p,us],lpwt)
          g := mainPrimitivePart g
          sfp := lazyPquo(p,g)
          sfp := mainPrimitivePart stronglyReduce(sfp,us)
          lpwt := cons([sfp,us],lpwt)
        lpwt

      toseSquareFreePart_base(p:P, ts: TS): List PWT == [[p,ts]$PWT]

      toseSquareFreePart(p:P, ts: TS): List PWT == toseSquareFreePart_wip(p,ts)

      prepareSubResAlgo(p1:P,p2:P,ts:TS): List LpWT ==
        -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
        -- ASSUME init(p1) invertible modulo ts !!!
        toSee: List LpWT := [[[p1,p2],ts]$LpWT]
        toSave: List LpWT := []
        v := mvar(p1)
        while (not empty? toSee) repeat
          lpwt := first toSee; toSee := rest toSee
          p1 := lpwt.val.1; p2 := lpwt.val.2
          ts := lpwt.tower
          lbwt := toseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
          for bwt in lbwt repeat
            (bwt.val = true) and (degree(p2,v) > 0) =>
              p3 := prem(p1, -p2)
              s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
              toSave := cons([[p2,p3,s],bwt.tower]$LpWT,toSave)
            -- p2 := initiallyReduce(p2,bwt.tower)
            newp2 := primitivePart initiallyReduce(p2,bwt.tower)
            (bwt.val = true) =>
```

```
        -- toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
        toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
      -- zero? p2 =>
      zero? newp2 =>
        toSave := cons([[p1,0,1],bwt.tower]$LpWT,toSave)
      -- toSee := cons([[p1,p2],ts]$LpWT,toSee)
      toSee := cons([[p1,newp2],bwt.tower]$LpWT,toSee)
  toSave

integralLastSubResultant(p1:P,p2:P,ts:TS): List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME p1 and p2 have no algebraic coefficients
  lsr := lastSubResultant(p1, p2)
  ground?(lsr) => [[lsr,ts]$PWT]
  mvar(lsr) < mvar(p1) => [[lsr,ts]$PWT]
  gi1i2 := gcd(init(p1),init(p2))
  ex: Union(P,"failed") := (gi1i2 * lsr) exquo$P init(lsr)
  ex case "failed" => [[lsr,ts]$PWT]
  [[ex::P,ts]$PWT]

internalLastSubResultant(p1:P,p2:P,ts:TS,b1:B,b2:B): List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- if b1 ASSUME init(p2) invertible w.r.t. ts
  -- if b2 BREAK with the first non-trivial gcd
  k: KeyGcd := [p1,p2,ts,b2]
  e := extractIfCan(k)$HGcd
  e case EntryGcd => e::EntryGcd
  toSave: List PWT
  empty? ts =>
    toSave := integralLastSubResultant(p1,p2,ts)
    insert!(k,toSave)$HGcd
    return toSave
  toSee: List LpWT
  if b1
    then
      p3 := prem(p1, -p2)
      s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
      toSee := [[[p2,p3,s],ts]$LpWT]
    else
      toSee := prepareSubResAlgo(p1,p2,ts)
  toSave := internalLastSubResultant(toSee,mvar(p1),b2)
  insert!(k,toSave)$HGcd
  toSave

internalLastSubResultant(llpwt: List LpWT,v:V,b2:B): List PWT ==
  toReturn: List PWT := []; toSee: List LpWT;
  while (not empty? llpwt) repeat
    toSee := llpwt; llpwt := []
    -- CONSIDER FIRST the vanishing current last subresultant
    for lpwt in toSee repeat
```

```
      p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3; ts := lpwt.tower
      lbwt := toseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
      for bwt in lbwt repeat
        bwt.val = false =>
          toReturn := cons([p1,bwt.tower]$PWT, toReturn)
          b2 and positive?(degree(p1,v)) => return toReturn
        llpwt := cons([[p1,p2,s],bwt.tower]$LpWT, llpwt)
    empty? llpwt => "leave"
    -- CONSIDER NOW the branches where the computations continue
    toSee := llpwt; llpwt := []
    lpwt := first toSee; toSee := rest toSee
    p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
    delta: N := (mdeg(p1) - degree(p2,v))::N
    p3: P := LazardQuotient2(p2, leadingCoefficient(p2,v), s, delta)
    zero?(degree(p3,v)) =>
      toReturn := cons([p3,lpwt.tower]$PWT, toReturn)
      for lpwt in toSee repeat
        toReturn := cons([p3,lpwt.tower]$PWT, toReturn)
    (p1, p2) := (p3, next_subResultant2(p1, p2, p3, s))
    s := leadingCoefficient(p1,v)
    llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
    for lpwt in toSee repeat
      llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
  toReturn

toseLastSubResultant(p1:P,p2:P,ts:TS): List PWT ==
  ground? p1 =>
    error"in toseLastSubResultantElseSplit$TOSEGCD  : bad #1"
  ground? p2 =>
    error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
  not (mvar(p2) = mvar(p1)) =>
    error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
  algebraic?(mvar(p1),ts) =>
    error"in toseLastSubResultantElseSplit$TOSEGCD : bad #1"
  not initiallyReduced?(p1,ts) =>
    error"in toseLastSubResultantElseSplit$TOSEGCD : bad #1"
  not initiallyReduced?(p2,ts) =>
    error"in toseLastSubResultantElseSplit$TOSEGCD : bad #2"
  purelyTranscendental?(p1,ts) and purelyTranscendental?(p2,ts) =>
    integralLastSubResultant(p1,p2,ts)
  if mdeg(p1) < mdeg(p2) then
     (p1, p2) := (p2, p1)
     if odd?(mdeg(p1)) and odd?(mdeg(p2)) then p2 := - p2
  internalLastSubResultant(p1,p2,ts,false,false)
```

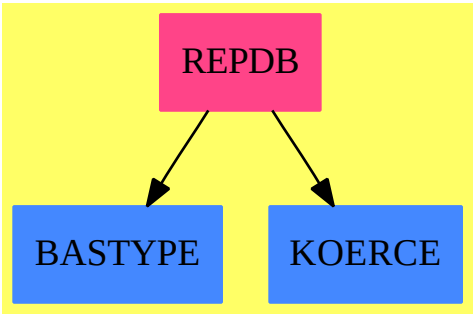
———-

— RSETGCD.dotabb —

```
"RSETGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RSETGCD"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"RSETGCD" -> "RPOLCAT"
```

19.34 package REPDB RepeatedDoubling

19.34.1 RepeatedDoubling (REPDB)



Exports:
double

— package REPDB RepeatedDoubling —

```
)abbrev package REPDB RepeatedDoubling
++ Description:
++ Implements multiplication by repeated addition

-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedDoubling, $mutableDomains)

RepeatedDoubling(S):Exports ==Implementation where
   S: SetCategory with
     "+":(%,%)->%
       ++ x+y returns the sum of x and y
   Exports == with
     double: (PositiveInteger,S) -> S
       ++ double(i, r) multiplies r by i using repeated doubling.
   Implementation == add
     x: S
```

```
     n: PositiveInteger
     double(n,x) ==
--         one? n => x
        (n = 1) => x
        odd?(n)$Integer =>
           x + double(shift(n,-1) pretend PositiveInteger,(x+x))
        double(shift(n,-1) pretend PositiveInteger,(x+x))
```

———-

— **REPDB.dotabb** —

```
"REPDB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REPDB"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"REPDB" -> "BASTYPE"
"REPDB" -> "KOERCE"
```

———-

19.35 package REPSQ RepeatedSquaring

This code computes x^n by repeated squaring. It takes advantage of the law of exponents so that if $n = a * b$ then

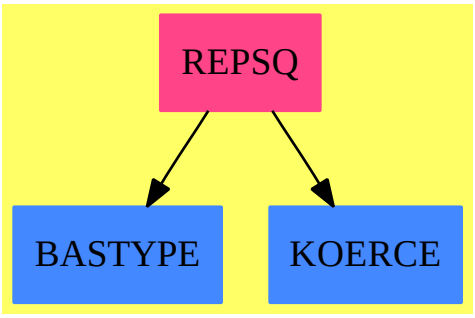
$$x^n = x^a x^b$$

It also takes advantage of the fact that n has a binary representation so that a one-bit right-shift of a number divides it by 2. Thus we get three cases:

$$\begin{aligned} &n = 1 \rightarrow x^1 \\ &n \text{ odd} \rightarrow \text{repeatedsquare}(x * x, n/2) * x \\ &n \text{ even} \rightarrow \text{repeatedsquare}(x * x, n/2) \end{aligned}$$

Since each recursive call divides n by 2 the algorithm is O(log(n)).

19.35.1 RepeatedSquaring (REPSQ)



Exports:
expt

— package REPSQ RepeatedSquaring —

```
)abbrev package REPSQ RepeatedSquaring
++ Description:
++ Implements exponentiation by repeated squaring

-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedSquaring, $mutableDomains)

RepeatedSquaring(S): Exports == Implementation where
   S: SetCategory with
     "*":(%,%)->%
       ++ x*y returns the product of x and y
   Exports == with
     expt: (S,PositiveInteger) -> S
       ++ expt(r, i) computes r**i  by repeated squaring
   Implementation == add
     x: S
     n: PositiveInteger
     expt(x, n) ==
--         one? n => x
         (n = 1) => x
         odd?(n)$Integer=> x * expt(x*x,shift(n,-1) pretend PositiveInteger)
         expt(x*x,shift(n,-1) pretend PositiveInteger)
```

———-

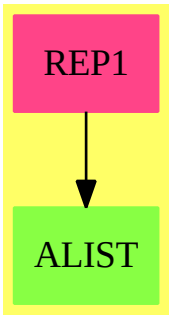
— **REPSQ.dotabb** —

```
"REPSQ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REPSQ"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"REPSQ" -> "BASTYPE"
"REPSQ" -> "KOERCE"
```

———-

19.36 package REP1 RepresentationPackage1

19.36.1 RepresentationPackage1 (REP1)



Exports:

antisymmetricTensors	createGenericMatrix	permutationRepresentation
symmetricTensors	tensorProduct	

— **package REP1 RepresentationPackage1** —

```
)abbrev package REP1 RepresentationPackage1
++ Authors: Holger Gollan, Johannes Grabmeier, Thorsten Werther
++ Date Created: 12 September 1987
++ Date Last Updated: 24 May 1991
++ Basic Operations: antisymmetricTensors,symmetricTensors,
++   tensorProduct, permutationRepresentation
++ Related Constructors: RepresentationPackage1, Permutation
++ Also See: IrrRepSymNatPackage
++ AMS Classifications:
++ Keywords: representation, symmetrization, tensor product
++ References:
++   G. James, A. Kerber: The Representation Theory of the Symmetric
++    Group. Encycl. of Math. and its Appl. Vol 16., Cambr. Univ Press 1981;
++   J. Grabmeier, A. Kerber: The Evaluation of Irreducible
```

```
++    Polynomial Representations of the General Linear Groups
++    and of the Unitary Groups over Fields of Characteristic 0,
++    Acta Appl. Math. 8 (1987), 271-291;
++   H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++    their Realization in the Computer Algebra System Scratchpad,
++    Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
++ Description:
++ RepresentationPackage1 provides functions for representation theory
++ for finite groups and algebras.
++ The package creates permutation representations and uses tensor products
++ and its symmetric and antisymmetric components to create new
++ representations of larger degree from given ones.
++ Note that instead of having parameters from \spadtype{Permutation}
++ this package allows list notation of permutations as well:
++ e.g. \spad{[1,4,3,2]} denotes permutes 2 and 4 and fixes 1 and 3.

RepresentationPackage1(R): public == private where

   R   : Ring
   OF    ==> OutputForm
   NNI   ==> NonNegativeInteger
   PI    ==> PositiveInteger
   I     ==> Integer
   L     ==> List
   M     ==> Matrix
   P     ==> Polynomial
   SM    ==> SquareMatrix
   V     ==> Vector
   ICF   ==> IntegerCombinatoricFunctions Integer
   SGCF  ==> SymmetricGroupCombinatoricFunctions
   PERM  ==> Permutation

   public ==> with

     if R has commutative("*") then
       antisymmetricTensors : (M R,PI) ->  M R
         ++ antisymmetricTensors(a,n) applies to the square matrix
         ++ a the irreducible, polynomial representation of the
         ++ general linear group GLm, where m is the number of
         ++ rows of a, which corresponds to the partition
         ++ (1,1,...,1,0,0,...,0) of n.
         ++ Error: if n is greater than m.
         ++ Note that this corresponds to the symmetrization of the
         ++ representation with the sign representation of the symmetric group
         ++ Sn. The carrier spaces of the representation are the antisymmetric
         ++ tensors of the n-fold tensor product.
     if R has commutative("*") then
       antisymmetricTensors : (L M R, PI) -> L M R
         ++ antisymmetricTensors(la,n) applies to each
         ++ m-by-m square matrix in
```

```
    ++ the list la the irreducible, polynomial representation
    ++ of the general linear group GLm
    ++ which corresponds
    ++ to the partition (1,1,...,1,0,0,...,0) of n.
    ++ Error: if n is greater than m.
    ++ Note that this corresponds to the symmetrization of the
    ++ representation with the sign representation of the symmetric group
    ++ Sn. The carrier spaces of the representation are the antisymmetric
    ++ tensors of the n-fold tensor product.
createGenericMatrix : NNI -> M P R
  ++ createGenericMatrix(m) creates a square matrix of dimension k
  ++ whose entry at the i-th row and j-th column is the
  ++ indeterminate x[i,j] (double subscripted).
symmetricTensors : (M R, PI) -> M R
  ++ symmetricTensors(a,n) applies to the m-by-m
  ++ square matrix a the
  ++ irreducible, polynomial representation of the general linear
  ++ group GLm
  ++ which corresponds to the partition (n,0,...,0) of n.
  ++ Error: if a is not a square matrix.
  ++ Note that this corresponds to the symmetrization of the
  ++ representation with the trivial representation of the symmetric
  ++ group Sn.
  ++ The carrier spaces of the representation are the symmetric
  ++ tensors of the n-fold tensor product.
symmetricTensors : (L M R, PI)  ->  L M R
  ++ symmetricTensors(la,n) applies to each m-by-m square matrix in the
  ++ list la the irreducible, polynomial representation
  ++ of the general linear group GLm
  ++ which corresponds
  ++ to the partition (n,0,...,0) of n.
  ++ Error: if the matrices in la are not square matrices.
  ++ Note that this corresponds to the symmetrization of the
  ++ representation with the trivial representation of the symmetric
  ++ group Sn.
  ++ The carrier spaces of the representation are the symmetric
  ++ tensors of the n-fold tensor product.
tensorProduct : (M R, M R) -> M R
  ++ tensorProduct(a,b) calculates the Kronecker product
  ++ of the matrices a and b.
  ++ Note that if each matrix corresponds to a group representation
  ++ (repr. of  generators) of one group, then these matrices
  ++ correspond to the tensor product of the two representations.
tensorProduct : (L M R, L M R) -> L M R
  ++ tensorProduct([a1,...,ak],[b1,...,bk]) calculates the list of
  ++ Kronecker products of the matrices ai and bi
  ++ for {1 <= i <= k}.
  ++ Note that if each list of matrices corresponds to a group
  ++ representation (repr. of generators) of one group, then these
  ++ matrices correspond to the tensor product of the two representations.
```

```
    tensorProduct : M R -> M R
      ++ tensorProduct(a) calculates the Kronecker product
      ++ of the matrix a with itself.
    tensorProduct : L M R -> L M R
      ++ tensorProduct([a1,...ak]) calculates the list of
      ++ Kronecker products of each matrix ai with itself
      ++ for {1 <= i <= k}.
      ++ Note that if the list of matrices corresponds to a group
      ++ representation (repr. of generators) of one group, then these
      ++ matrices correspond to the tensor product of the representation
      ++ with itself.
    permutationRepresentation : (PERM I, I) -> M I
      ++ permutationRepresentation(pi,n) returns the matrix
      ++ (deltai,pi(i)) (Kronecker delta) for a permutation
      ++ pi of {1,2,...,n}.
    permutationRepresentation : L I -> M I
      ++ permutationRepresentation(pi,n) returns the matrix
      ++ (deltai,pi(i)) (Kronecker delta) if the permutation
      ++ pi is in list notation and permutes {1,2,...,n}.
    permutationRepresentation : (L PERM I, I) -> L M I
      ++ permutationRepresentation([pi1,...,pik],n) returns the list
      ++ of matrices [(deltai,pi1(i)),...,(deltai,pik(i))]
      ++ (Kronecker delta) for the permutations pi1,...,pik
      ++ of {1,2,...,n}.
    permutationRepresentation : L L I -> L M I
      ++ permutationRepresentation([pi1,...,pik],n) returns the list
      ++ of matrices [(deltai,pi1(i)),...,(deltai,pik(i))]
      ++ if the permutations pi1,...,pik are in
      ++ list notation and are permuting {1,2,...,n}.

  private ==> add

    -- import of domains and packages

    import OutputForm

    -- declaration of local functions:


    calcCoef : (L I, M I) -> I
      -- calcCoef(beta,C) calculates the term
      -- |S(beta) gamma S(alpha)| / |S(beta)|


    invContent : L I -> V I
      -- invContent(alpha) calculates the weak monoton function f with
      -- f : m -> n with invContent alpha. f is stored in the returned
      -- vector
```

```
-- definition of local functions


calcCoef(beta,C) ==
  prod : I := 1
  for i in 1..maxIndex beta repeat
    prod := prod * multinomial(beta(i), entries row(C,i))$ICF
  prod


invContent(alpha) ==
  n : NNI := (+/alpha)::NNI
  f : V I  := new(n,0)
  i : NNI := 1
  j : I   := - 1
  for og in alpha repeat
    j := j + 1
    for k in 1..og repeat
      f(i) := j
      i := i + 1
  f


-- exported functions:



if R has commutative("*") then
  antisymmetricTensors ( a : M R , k : PI ) ==

    n      : NNI   := nrows a
    k = 1 => a
    k > n =>
      error("second parameter for antisymmetricTensors is too large")
    m      :   I   := binomial(n,k)$ICF
    il     : L L I   := [subSet(n,k,i)$SGCF for i in 0..m-1]
    b      :  M R   := zero(m::NNI, m::NNI)
    for i in 1..m repeat
      for j in 1..m repeat
        c : M R := zero(k,k)
        lr: L I := il.i
        lt: L I := il.j
        for  r in 1..k repeat
          for t in 1..k repeat
            rr : I := lr.r
            tt : I := lt.t
            --c.r.t := a.(1+rr).(1+tt)
            setelt(c,r,t,elt(a, 1+rr, 1+tt))
        setelt(b, i, j, determinant c)
    b
```

```
if R has commutative("*") then
  antisymmetricTensors(la: L M R, k: PI) ==
    [antisymmetricTensors(ma,k) for ma in la]



symmetricTensors (a : M R, n : PI) ==

  m : NNI := nrows a
  m ^= ncols a =>
    error("Input to symmetricTensors is no square matrix")
  n = 1 => a

  dim : NNI := (binomial(m+n-1,n)$ICF)::NNI
  c : M R := new(dim,dim,0)
  f : V I := new(n,0)
  g : V I := new(n,0)
  nullMatrix : M I := new(1,1,0)
  colemanMatrix : M I

  for i in 1..dim repeat
    -- unrankImproperPartitions1 starts counting from 0
    alpha := unrankImproperPartitions1(n,m,i-1)$SGCF
    f := invContent(alpha)
    for j in 1..dim repeat
      -- unrankImproperPartitions1 starts counting from 0
      beta := unrankImproperPartitions1(n,m,j-1)$SGCF
      g := invContent(beta)
      colemanMatrix := nextColeman(alpha,beta,nullMatrix)$SGCF
      while colemanMatrix ^= nullMatrix repeat
        gamma := inverseColeman(alpha,beta,colemanMatrix)$SGCF
        help : R := calcCoef(beta,colemanMatrix)::R
        for k in 1..n repeat
          help := help * a( (1+f k)::NNI, (1+g(gamma k))::NNI )
        c(i,j) := c(i,j) + help
        colemanMatrix := nextColeman(alpha,beta,colemanMatrix)$SGCF
      -- end of while
    -- end of j-loop
  -- end of i-loop

  c


symmetricTensors(la : L M R, k : PI) ==
  [symmetricTensors (ma, k) for ma in la]


tensorProduct(a: M R, b: M R) ==
```

```
  n      : NNI := nrows a
  m      : NNI := nrows b
  nc     : NNI := ncols a
  mc     : NNI := ncols b
  c      : M R   := zero(n * m, nc * mc)
  indexr : NNI := 1                              --   row index
  for i in 1..n repeat
     for k in 1..m repeat
        indexc : NNI := 1                        --   column index
        for j in 1..nc repeat
           for l in 1..mc repeat
              c(indexr,indexc) := a(i,j) * b(k,l)
              indexc          := indexc + 1
        indexr := indexr + 1
  c


tensorProduct (la: L M R, lb: L M R) ==
  [tensorProduct(la.i, lb.i) for i in 1..maxIndex la]


tensorProduct(a : M R) == tensorProduct(a, a)

tensorProduct(la : L M R) ==
  tensorProduct(la :: L M R, la :: L M R)

permutationRepresentation (p : PERM I, n : I) ==
  -- permutations are assumed to permute {1,2,...,n}
  a : M I := zero(n :: NNI, n :: NNI)
  for i in 1..n repeat
     a(eval(p,i)$(PERM I),i) := 1
  a


permutationRepresentation (p : L I) ==
  -- permutations are assumed to permute {1,2,...,n}
  n : I := #p
  a : M I := zero(n::NNI, n::NNI)
  for i in 1..n repeat
     a(p.i,i) := 1
  a


permutationRepresentation(listperm : L PERM I, n : I) ==
  -- permutations are assumed to permute {1,2,...,n}
  [permutationRepresentation(perm, n) for perm in listperm]

permutationRepresentation (listperm : L L I) ==
  -- permutations are assumed to permute {1,2,...,n}
  [permutationRepresentation perm for perm in listperm]
```

```
createGenericMatrix(m) ==
  res : M P R := new(m,m,0$(P R))
  for i in 1..m repeat
    for j in 1..m repeat
       iof : OF := coerce(i)$Integer
       jof : OF := coerce(j)$Integer
       le : L OF := cons(iof,list jof)
       sy : Symbol := subscript(x::Symbol, le)$Symbol
       res(i,j) := (sy :: P R)
  res
```

———-

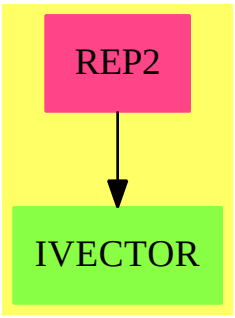
— **REP1.dotabb** —

```
"REP1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP1"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"REP1" -> "ALIST"
```

———-

19.37 package REP2 RepresentationPackage2

19.37.1 RepresentationPackage2 (REP2)



Exports:

areEquivalent?	completeEchelonBasis
createRandomElement	cyclicSubmodule
isAbsolutelyIrreducible?	meatAxe
scanOneDimSubspaces	split
standardBasisOfCyclicSubmodule	

— package REP2 RepresentationPackage2 —

```
)abbrev package REP2 RepresentationPackage2
++ Authors: Holger Gollan, Johannes Grabmeier
++ Date Created: 10 September 1987
++ Date Last Updated: 20 August 1990
++ Basic Operations: areEquivalent?, isAbsolutelyIrreducible?,
++   split, meatAxe
++ Related Constructors: RepresentationTheoryPackage1
++ Also See: IrrRepSymNatPackage
++ AMS Classifications:
++ Keywords: meat-axe, modular representation
++ Reference:
++   R. A. Parker: The Computer Calculation of Modular Characters
++   (The Meat-Axe), in M. D. Atkinson (Ed.), Computational Group Theory
++   Academic Press, Inc., London 1984
++   H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++    their Realization in the Computer Algebra System Scratchpad,
++    Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23.
++ Description:
++ RepresentationPackage2 provides functions for working with
++ modular representations of finite groups and algebra.
++ The routines in this package are created, using ideas of R. Parker,
++ (the meat-Axe) to get smaller representations from bigger ones,
++ i.e. finding sub- and factormodules, or to show, that such the
++ representations are irreducible.
++ Note that most functions are randomized functions of Las Vegas type
++ i.e. every answer is correct, but with small probability
++ the algorithm fails to get an answer.

RepresentationPackage2(R): public == private where

  R    : Ring
  OF    ==> OutputForm
  I     ==> Integer
  L     ==> List
  SM    ==> SquareMatrix
  M     ==> Matrix
  NNI   ==> NonNegativeInteger
  V     ==> Vector
  PI    ==> PositiveInteger
  B     ==> Boolean
  RADIX ==> RadixExpansion

  public ==> with

    completeEchelonBasis : V V R  ->  M R
      ++ completeEchelonBasis(lv) completes the basis lv assumed
      ++ to be in echelon form of a subspace of R**n (n the length
```

```
    ++ of all the vectors in lv with unit vectors to a basis of
    ++ R**n. It is assumed that the argument is not an empty
    ++ vector and that it is not the basis of the 0-subspace.
    ++ Note that the rows of the result correspond to the vectors
    ++ of the basis.
  createRandomElement : (L M R, M R) -> M R
    ++ createRandomElement(aG,x) creates a random element of the group
    ++ algebra generated by aG.
  -- randomWord : (L L I, L M)  ->  M R
    --++ You can create your own 'random' matrix with "randomWord(lli, lm)".
    --++ Each li in lli determines a product of matrices, the entries in li
    --++ determine which matrix from lm is chosen. Finally we sum over all
    --++ products. The result "sm" can be used to call split with (e.g.)
    --++ second parameter "first nullSpace sm"
  if R has EuclideanDomain then   -- using rowEchelon
    cyclicSubmodule  : (L M R, V R) ->   V V R
      ++ cyclicSubmodule(lm,v) generates a basis as follows.
      ++ It is assumed that the size n of the vector equals the number
      ++ of rows and columns of the matrices. Then the matrices generate
      ++ a subalgebra, say \spad{A}, of the algebra of all square matrices of
      ++ dimension n. V R is an \spad{A}-module in the natural way.
      ++ cyclicSubmodule(lm,v) generates the R-Basis of Av as
      ++ described in section 6 of R. A. Parker's "The Meat-Axe".
      ++ Note that in contrast to the description in "The Meat-Axe" and to
      ++ standardBasisOfCyclicSubmodule the result is in echelon form.
    standardBasisOfCyclicSubmodule  : (L M R, V R) ->   M R
      ++ standardBasisOfCyclicSubmodule(lm,v) returns a matrix as follows.
      ++ It is assumed that the size n of the vector equals the number
      ++ of rows and columns of the matrices.  Then the matrices generate
      ++ a subalgebra, say \spad{A},
      ++ of the algebra of all square matrices of
      ++ dimension n. V R is an \spad{A}-module in the natural way.
      ++ standardBasisOfCyclicSubmodule(lm,v) calculates a matrix whose
      ++ non-zero column vectors are the R-Basis of Av achieved
      ++ in the way as described in section 6
      ++ of R. A. Parker's "The Meat-Axe".
      ++ Note that in contrast to cyclicSubmodule, the result is not
      ++ in echelon form.
  if R has Field then  -- only because of inverse in SM
    areEquivalent? : (L M R, L M R, B, I) -> M R
      ++ areEquivalent?(aG0,aG1,randomelements,numberOfTries) tests
      ++ whether the two lists of matrices, all assumed of same
      ++ square shape, can be simultaneously conjugated by a non-singular
      ++ matrix. If these matrices represent the same group generators,
      ++ the representations are equivalent.
      ++ The algorithm tries
      ++ numberOfTries times to create elements in the
      ++ generated algebras in the same fashion. If their ranks differ,
      ++ they are not equivalent. If an
      ++ isomorphism is assumed, then
```

```
  ++ the kernel of an element of the first algebra
  ++ is mapped to the kernel of the corresponding element in the
  ++ second algebra. Now consider the one-dimensional ones.
  ++ If they generate the whole space (e.g. irreducibility !)
  ++ we use standardBasisOfCyclicSubmodule to create the
  ++ only possible transition matrix. The method checks whether the
  ++ matrix conjugates all corresponding matrices from aGi.
  ++ The way to choose the singular matrices is as in meatAxe.
  ++ If the two representations are equivalent, this routine
  ++ returns the transformation matrix TM with
  ++ aG0.i * TM = TM * aG1.i for all i. If the representations
  ++ are not equivalent, a small 0-matrix is returned.
  ++ Note that the case
  ++ with different sets of group generators cannot be handled.
areEquivalent? : (L M R, L M R) -> M R
  ++ areEquivalent?(aG0,aG1) calls areEquivalent?(aG0,aG1,true,25).
  ++ Note that the choice of 25 was rather arbitrary.
areEquivalent? : (L M R, L M R, I) -> M R
  ++ areEquivalent?(aG0,aG1,numberOfTries) calls
  ++ areEquivalent?(aG0,aG1,true,25).
  ++ Note that the choice of 25 was rather arbitrary.
isAbsolutelyIrreducible? : (L M R, I) -> B
  ++ isAbsolutelyIrreducible?(aG, numberOfTries) uses
  ++ Norton's irreducibility test to check for absolute
  ++ irreduciblity, assuming if a one-dimensional kernel is found.
  ++ As no field extension changes create "new" elements
  ++ in a one-dimensional space, the criterium stays true
  ++ for every extension. The method looks for one-dimensionals only
  ++ by creating random elements (no fingerprints) since
  ++ a run of meatAxe would have proved absolute irreducibility
  ++ anyway.
isAbsolutelyIrreducible? : L M R -> B
  ++ isAbsolutelyIrreducible?(aG) calls
  ++ isAbsolutelyIrreducible?(aG,25).
  ++ Note that the choice of 25 was rather arbitrary.
split : (L M R, V R)  ->  L L M R
  ++ split(aG, vector) returns a subalgebra \spad{A} of all
  ++ square matrix of dimension n as a list of list of matrices,
  ++ generated by the list of matrices aG, where n denotes both
  ++ the size of vector as well as the dimension of each of the
  ++ square matrices.
  ++ V R is an A-module in the natural way.
  ++ split(aG, vector) then checks whether the cyclic submodule
  ++ generated by vector is a proper submodule of V R.
  ++ If successful, it returns a two-element list, which contains
  ++ first the list of the representations of the submodule,
  ++ then the list of the representations of the factor module.
  ++ If the vector generates the whole module, a one-element list
  ++ of the old representation is given.
  ++ Note that a later version this should call the other split.
```

```
    split:  (L M R, V V R) -> L L M R
      ++ split(aG,submodule) uses a proper submodule of R**n
      ++ to create the representations of the submodule and of
      ++ the factor module.
  if (R has Finite) and (R has Field) then
    meatAxe  : (L M R, B, I, I)  ->  L L M R
      ++ meatAxe(aG,randomElements,numberOfTries, maxTests) returns
      ++ a 2-list of representations as follows.
      ++ All matrices of argument aG are assumed to be square
      ++ and of equal size.
      ++ Then \spad{aG} generates a subalgebra, say \spad{A}, of the algebra
      ++ of all square matrices of dimension n. V R is an A-module
      ++ in the usual way.
      ++ meatAxe(aG,numberOfTries, maxTests) creates at most
      ++ numberOfTries random elements of the algebra, tests
      ++ them for singularity. If singular, it tries at most maxTests
      ++ elements of its kernel to generate a proper submodule.
      ++ If successful, a 2-list is returned: first, a list
      ++ containing first the list of the
      ++ representations of the submodule, then a list of the
      ++ representations of the factor module.
      ++ Otherwise, if we know that all the kernel is already
      ++ scanned, Norton's irreducibility test can be used either
      ++ to prove irreducibility or to find the splitting.
      ++ If randomElements is false, the first 6 tries
      ++ use Parker's fingerprints.
    meatAxe : L M R -> L L M R
      ++ meatAxe(aG) calls meatAxe(aG,false,25,7) returns
      ++ a 2-list of representations as follows.
      ++ All matrices of argument \spad{aG} are assumed to be square
      ++ and of
      ++ equal size. Then \spad{aG} generates a subalgebra,
      ++ say \spad{A}, of the algebra
      ++ of all square matrices of dimension n. V R is an A-module
      ++ in the usual way.
      ++ meatAxe(aG) creates at most 25 random elements
      ++ of the algebra, tests
      ++ them for singularity. If singular, it tries at most 7
      ++ elements of its kernel to generate a proper submodule.
      ++ If successful a list which contains first the list of the
      ++ representations of the submodule, then a list of the
      ++ representations of the factor module is returned.
      ++ Otherwise, if we know that all the kernel is already
      ++ scanned, Norton's irreducibility test can be used either
      ++ to prove irreducibility or to find the splitting.
      ++ Notes: the first 6 tries use Parker's fingerprints.
      ++ Also, 7 covers the case of three-dimensional kernels over
      ++ the field with 2 elements.
    meatAxe: (L M R, B) ->  L L M R
      ++ meatAxe(aG, randomElements) calls meatAxe(aG,false,6,7),
```

```
      ++ only using Parker's fingerprints, if randomElemnts is false.
      ++ If it is true, it calls meatAxe(aG,true,25,7),
      ++ only using random elements.
      ++ Note that the choice of 25 was rather arbitrary.
      ++ Also, 7 covers the case of three-dimensional kernels over the field
      ++ with 2 elements.
    meatAxe : (L M R, PI)  ->  L L M R
      ++ meatAxe(aG, numberOfTries) calls
      ++ meatAxe(aG,true,numberOfTries,7).
      ++ Notes: 7 covers the case of three-dimensional
      ++ kernels over the field with 2 elements.
    scanOneDimSubspaces: (L V R, I)  ->  V R
      ++ scanOneDimSubspaces(basis,n) gives a canonical representative
      ++ of the n-th one-dimensional subspace of the vector space
      ++ generated by the elements of basis, all from R**n.
      ++ The coefficients of the representative are of shape
      ++ (0,...,0,1,*,...,*), * in R. If the size of R
      ++ is q, then there are (q**n-1)/(q-1) of them.
      ++ We first reduce n modulo this number, then find the
      ++ largest i such that +/[q**i for i in 0..i-1] <= n.
      ++ Subtracting this sum of powers from n results in an
      ++ i-digit number to basis q. This fills the positions of the
      ++ stars.
      -- would prefer to have (V V R,....   but nullSpace  results
      -- in L V R

private ==> add

  -- import of domain and packages
  import OutputForm

  -- declarations and definitions of local variables and
  -- local function

  blockMultiply: (M R, M R, L I, I) -> M R
    -- blockMultiply(a,b,li,n) assumes that a has n columns
    -- and b has n rows, li is a sublist of the rows of a and
    -- a sublist of the columns of b. The result is the
    -- multiplication of the (li x n) part of a with the
    -- (n x li) part of b. We need this, because just matrix
    -- multiplying the parts would require extra storage.
  blockMultiply(a, b, li, n) ==
    matrix([[ +/[a(i,s) * b(s,j) for s in 1..n ] _
      for j in li  ]  for i in li])

  fingerPrint: (NNI, M R, M R, M R) -> M R
    -- is local, because one should know all the results for smaller i
  fingerPrint (i : NNI, a : M R, b : M R, x :M R) ==
    -- i > 2 only gives the correct result if the value of x from
    -- the parameter list equals the result of fingerprint(i-1,...)
```

```
    (i::PI) = 1 => x := a + b + a*b
    (i::PI) = 2 => x := (x + a*b)*b
    (i::PI) = 3 => x := a + b*x
    (i::PI) = 4 => x := x + b
    (i::PI) = 5 => x := x + a*b
    (i::PI) = 6 => x := x - a + b*a
    error "Sorry, but there are only 6 fingerprints!"
    x


  -- definition of exported functions


  --randomWord(lli,lm)  ==
  --  -- we assume that all matrices are square of same size
  --  numberOfMatrices := #lm
  --  +/[*/[lm.(1+i rem numberOfMatrices) for i in li ] for li in lli]

  completeEchelonBasis(basis) ==

    dimensionOfSubmodule : NNI := #basis
    n : NNI := # basis.1
    indexOfVectorToBeScanned : NNI := 1
    row : NNI := dimensionOfSubmodule

    completedBasis : M R := zero(n, n)
    for i in 1..dimensionOfSubmodule repeat
      completedBasis := setRow_!(completedBasis, i, basis.i)
    if #basis <= n then
      newStart : NNI := 1
      for j in 1..n
        while indexOfVectorToBeScanned <= dimensionOfSubmodule repeat
          if basis.indexOfVectorToBeScanned.j = 0 then
            completedBasis(1+row,j) := 1  --put unit vector into basis
            row := row + 1
          else
            indexOfVectorToBeScanned := indexOfVectorToBeScanned + 1
          newStart : NNI := j + 1
      for j in newStart..n repeat
        completedBasis(j,j) := 1  --put unit vector into basis
    completedBasis


  createRandomElement(aG,algElt) ==
    numberOfGenerators : NNI := #aG
    -- randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    algElt := algElt * aG.randomIndex
    -- randomIndxElement := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
```

```
    algElt + aG.randomIndex


  if R has EuclideanDomain then
    cyclicSubmodule (lm : L M R, v : V R)  ==
      basis : M R := rowEchelon matrix list entries v
      -- normalizing the vector
      -- all these elements lie in the submodule generated by v
      furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
      --furtherElts has elements of the generated submodule. It will
      --will be checked whether they are in the span of the vectors
      --computed so far. Of course we stop if we have got the whole
      --space.
      while (^null furtherElts) and (nrows basis < #v)  repeat
        w : V R := first furtherElts
        nextVector : M R := matrix list entries w -- normalizing the vector
        -- will the rank change if we add this nextVector
        -- to the basis so far computed?
        addedToBasis : M R := vertConcat(basis, nextVector)
        if rank addedToBasis ^= nrows basis then
           basis := rowEchelon addedToBasis  -- add vector w to basis
           updateFurtherElts : L V R := _
             [(lm.i*w)::V R for i in 1..maxIndex lm]
           furtherElts := append (rest furtherElts, updateFurtherElts)
        else
           -- the vector w lies in the span of matrix, no updating
           -- of the basis
           furtherElts := rest furtherElts
      vector [row(basis, i) for i in 1..maxRowIndex basis]


    standardBasisOfCyclicSubmodule (lm : L M R, v : V R)  ==
      dim   : NNI := #v
      standardBasis : L L R := list(entries v)
      basis : M R := rowEchelon matrix list entries v
      -- normalizing the vector
      -- all these elements lie in the submodule generated by v
      furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
      --furtherElts has elements of the generated submodule. It will
      --will be checked whether they are in the span of the vectors
      --computed so far. Of course we stop if we  have got the whole
      --space.
      while (^null furtherElts) and (nrows basis < #v)  repeat
        w : V R := first furtherElts
        nextVector : M R := matrix list entries w  -- normalizing the vector
        -- will the rank change if we add this nextVector
        -- to the basis so far computed?
        addedToBasis : M R := vertConcat(basis, nextVector)
        if rank addedToBasis ^= nrows basis then
           standardBasis := cons(entries w, standardBasis)
```

```
          basis := rowEchelon addedToBasis  -- add vector w to basis
          updateFurtherElts : L V R := _
            [lm.i*w for i in 1..maxIndex lm]
          furtherElts := append (rest furtherElts, updateFurtherElts)
        else
          -- the vector w lies in the span of matrix, therefore
          -- no updating of matrix
          furtherElts := rest furtherElts
      transpose matrix standardBasis


  if R has Field then  -- only because of inverse in Matrix

    -- as conditional local functions, *internal have to be here

    splitInternal: (L M R, V R, B) -> L L M R
    splitInternal(algebraGenerators : L M R, vector: V R,doSplitting? : B) ==

      n : I := # vector    -- R-rank of representation module =
                           -- degree of representation
      submodule : V V R := cyclicSubmodule (algebraGenerators,vector)
      rankOfSubmodule : I := # submodule  -- R-Rank of submodule
      submoduleRepresentation    : L M R := nil()
      factormoduleRepresentation : L M R := nil()
      if n ^= rankOfSubmodule then
        messagePrint "  A proper cyclic submodule is found."
        if doSplitting? then   -- no else !!
          submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
          factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..n]
          transitionMatrix : M R := _
            transpose completeEchelonBasis submodule
          messagePrint "  Transition matrix computed"
          inverseTransitionMatrix : M R :=  _
           autoCoerce(inverse transitionMatrix)$Union(M R,"failed")
          messagePrint "  The inverse of the transition matrix computed"
          messagePrint "  Now transform the matrices"
          for i in 1..maxIndex algebraGenerators repeat
            helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
            -- in order to not create extra space and regarding the fact
            -- that we only want the two blocks in the main diagonal we
            -- multiply with the aid of the local function blockMultiply
            submoduleRepresentation := cons( blockMultiply( _
              helpMatrix,transitionMatrix,submoduleIndices,n), _
              submoduleRepresentation)
            factormoduleRepresentation := cons( blockMultiply( _
              helpMatrix,transitionMatrix,factormoduleIndices,n), _
              factormoduleRepresentation)
        [reverse submoduleRepresentation, reverse _
          factormoduleRepresentation]
      else -- represesentation is irreducible
```

```
    messagePrint "  The generated cyclic submodule was not proper"
    [algebraGenerators]



irreducibilityTestInternal: (L M R, M R, B) -> L L M R
irreducibilityTestInternal(algebraGenerators,_
    singularMatrix,split?) ==
  algebraGeneratorsTranspose : L M R := [transpose _
     algebraGenerators.j for j in 1..maxIndex algebraGenerators]
  xt : M R := transpose singularMatrix
  messagePrint "  We know that all the cyclic submodules generated by all"
  messagePrint "    non-trivial element of the singular matrix under view are"
  messagePrint "    not proper, hence Norton's irreducibility test can be done:"
  -- actually we only would need one (!) non-trivial element from
  -- the kernel of xt, such an element must exist as the transpose
  -- of a singular matrix is of course singular. Question: Can
  -- we get it more easily from the kernel of x = singularMatrix?
  kernel : L V R := nullSpace xt
  result : L L M R :=  _
    splitInternal(algebraGeneratorsTranspose,first kernel,split?)
  if null rest result then  -- this means first kernel generates
    -- the whole module
    if 1 = #kernel then
       messagePrint "  Representation is absolutely irreducible"
    else
       messagePrint "  Representation is irreducible, but we don't know "
       messagePrint "    whether it is absolutely irreducible"
  else
    if split? then
      messagePrint "  Representation is not irreducible and it will be split:"
      -- these are the dual representations, so calculate the
      -- dual to get the desired result, i.e. "transpose inverse"
      -- improvements??
      for i in 1..maxIndex result repeat
        for j in 1..maxIndex (result.i) repeat
          mat : M R := result.i.j
          result.i.j := _
            transpose autoCoerce(inverse mat)$Union(M R,"failed")
    else
      messagePrint "  Representation is not irreducible, use meatAxe to split"
  -- if "split?" then dual representation interchange factor
  -- and submodules, hence reverse
  reverse result



-- exported functions for FiniteField-s.
```

```
areEquivalent? (aG0, aG1) ==
  areEquivalent? (aG0, aG1, true, 25)


areEquivalent? (aG0, aG1, numberOfTries) ==
  areEquivalent? (aG0, aG1, true, numberOfTries)


areEquivalent? (aG0, aG1, randomelements, numberOfTries) ==
    result : B := false
    transitionM : M R := zero(1, 1)
    numberOfGenerators  : NNI := #aG0
    -- need a start value for creating random matrices:
    -- if we switch to randomelements later, we take the last
    -- fingerprint.
    if randomelements then   -- random should not be from I
       --randomIndex  : I   := randnum numberOfGenerators
       randomIndex := 1+(random()$Integer rem numberOfGenerators)
       x0 : M R := aG0.randomIndex
       x1 : M R := aG1.randomIndex
    n : NNI := #row(x0,1)   -- degree  of representation
    foundResult : B := false
    for i in 1..numberOfTries until foundResult repeat
      -- try to create a non-singular element of the algebra
      -- generated by "aG". If only two generators,
      -- i < 7 and not "randomelements" use Parker's  fingerprints
      -- i >= 7 create random elements recursively:
      -- x_i+1 :=x_i * mr1 + mr2, where mr1 and mr2 are randomly
      -- chosen elements form "aG".
      if i = 7 then randomelements := true
      if randomelements then
         --randomIndex := randnum numberOfGenerators
         randomIndex := 1+(random()$Integer rem numberOfGenerators)
         x0 := x0 * aG0.randomIndex
         x1 := x1 * aG1.randomIndex
         --randomIndex := randnum numberOfGenerators
         randomIndex := 1+(random()$Integer rem numberOfGenerators)
         x0 := x0 + aG0.randomIndex
         x1 := x1 + aG1.randomIndex
      else
         x0 := fingerPrint (i, aG0.0, aG0.1 ,x0)
         x1 := fingerPrint (i, aG1.0, aG1.1 ,x1)
      -- test singularity of x0 and x1
      rk0 : NNI := rank x0
      rk1 : NNI := rank x1
      rk0 ^= rk1 =>
        messagePrint  "Dimensions of kernels differ"
        foundResult := true
        result := false
      -- can assume dimensions are equal
```

```
  rk0 ^= n - 1 =>
    -- not of any use here if kernel not one-dimensional
    if randomelements then
      messagePrint  "Random element in generated algebra does"
      messagePrint  "  not have a one-dimensional kernel"
    else
      messagePrint  "Fingerprint element in generated algebra does"
      messagePrint  "  not have a one-dimensional kernel"
  -- can assume dimensions are equal and equal to n-1
  if randomelements then
    messagePrint  "Random element in generated algebra has"
    messagePrint  "  one-dimensional kernel"
  else
    messagePrint  "Fingerprint element in generated algebra has"
    messagePrint  "  one-dimensional kernel"
  kernel0 : L V R := nullSpace x0
  kernel1 : L V R := nullSpace x1
  baseChange0 : M R := standardBasisOfCyclicSubmodule(_
    aG0,kernel0.1)
  baseChange1 : M R := standardBasisOfCyclicSubmodule(_
    aG1,kernel1.1)
  (ncols baseChange0) ^= (ncols baseChange1) =>
    messagePrint  "  Dimensions of generated cyclic submodules differ"
    foundResult := true
    result := false
  -- can assume that dimensions of cyclic submodules are equal
  (ncols baseChange0) = n =>   -- full dimension
    transitionM := baseChange0 * _
      autoCoerce(inverse baseChange1)$Union(M R,"failed")
    foundResult := true
    result := true
    for j in 1..numberOfGenerators while result repeat
      if (aG0.j*transitionM) ^= (transitionM*aG1.j) then
        result := false
        transitionM := zero(1 ,1)
        messagePrint "  There is no isomorphism, as the only possible one"
        messagePrint "    fails to do the necessary base change"
  -- can assume that dimensions of cyclic submodules are not "n"
  messagePrint  "  Generated cyclic submodules have equal, but not full"
  messagePrint  "    dimension, hence we can not draw any conclusion"
-- here ends the for-loop
if not foundResult then
  messagePrint  " "
  messagePrint  "Can neither prove equivalence nor inequivalence."
  messagePrint  "  Try again."
else
  if result then
    messagePrint  " "
    messagePrint  "Representations are equivalent."
  else
```

```
        messagePrint  " "
        messagePrint  "Representations are not equivalent."
    transitionM


isAbsolutelyIrreducible?(aG) == isAbsolutelyIrreducible?(aG,25)


isAbsolutelyIrreducible?(aG, numberOfTries) ==
  result : B := false
  numberOfGenerators  : NNI := #aG
  -- need a start value for creating random matrices:
  -- randomIndex  : I   := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  x : M R := aG.randomIndex
  n : NNI := #row(x,1)   -- degree  of representation
  foundResult : B := false
  for i in 1..numberOfTries until foundResult repeat
    -- try to create a non-singular element of the algebra
    -- generated by "aG", dimension of its kernel being 1.
    -- create random elements recursively:
    -- x_i+1 :=x_i * mr1 + mr2, where mr1 and mr2 are randomly
    -- chosen elements form "aG".
    -- randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    x := x * aG.randomIndex
    --randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    x := x + aG.randomIndex
    -- test whether rank of x is n-1
    rk : NNI := rank x
    if rk = n - 1 then
      foundResult := true
      messagePrint "Random element in generated algebra has"
      messagePrint "  one-dimensional kernel"
      kernel : L V R := nullSpace x
      if n=#cyclicSubmodule(aG, first kernel) then
        result := (irreducibilityTestInternal(aG,x,false)).1 ^= nil()$(L M R)
        -- result := not null? first irreducibilityTestInternal(aG,x,false) -- this down't compile !!
      else -- we found a proper submodule
        result := false
        --split(aG,kernel.1) -- to get the splitting
    else -- not of any use here if kernel not one-dimensional
      messagePrint "Random element in generated algebra does"
      messagePrint "  not have a one-dimensional kernel"
  -- here ends the for-loop
  if not foundResult then
    messagePrint "We have not found a one-dimensional kernel so far,"
    messagePrint "  as we do a random search you could try again"
  --else
```

```
    --  if not result then
    --    messagePrint "Representation is not irreducible."
    --  else
    --    messagePrint "Representation is irreducible."
    result



  split(algebraGenerators: L M R, vector: V R) ==
    splitInternal(algebraGenerators, vector, true)


  split(algebraGenerators : L M R, submodule: V V R) == --not zero submodule
    n : NNI := #submodule.1 -- R-rank of representation module =
                            -- degree of representation
    rankOfSubmodule : I := (#submodule) :: I --R-Rank of submodule
    submoduleRepresentation    : L M R := nil()
    factormoduleRepresentation : L M R := nil()
    submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
    factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..(n::I)]
    transitionMatrix : M R := _
      transpose completeEchelonBasis submodule
    messagePrint "  Transition matrix computed"
    inverseTransitionMatrix : M R :=
      autoCoerce(inverse transitionMatrix)$Union(M R,"failed")
    messagePrint "  The inverse of the transition matrix computed"
    messagePrint "  Now transform the matrices"
    for i in 1..maxIndex algebraGenerators repeat
      helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
      -- in order to not create extra space and regarding the fact
      -- that we only want the two blocks in the main diagonal we
      -- multiply with the aid of the local function blockMultiply
      submoduleRepresentation := cons( blockMultiply( _
        helpMatrix,transitionMatrix,submoduleIndices,n), _
        submoduleRepresentation)
      factormoduleRepresentation := cons( blockMultiply( _
        helpMatrix,transitionMatrix,factormoduleIndices,n), _
        factormoduleRepresentation)
    cons(reverse submoduleRepresentation, list( reverse _
      factormoduleRepresentation)::(L L M R))


-- the following is "under"  "if R has Field", as there are compiler
-- problems with conditinally defined local functions, i.e. it
-- doesn't know, that "FiniteField" has "Field".


  -- we are scanning through the vectorspaces
  if (R has Finite) and (R has Field) then
```

```
meatAxe(algebraGenerators, randomelements, numberOfTries, _
   maxTests) ==
  numberOfGenerators  : NNI := #algebraGenerators
  result : L L M R := nil()$(L L M R)
  q   : PI  := size()$R:PI
  -- need a start value for creating random matrices:
  -- if we switch to randomelements later, we take the last
  -- fingerprint.
  if randomelements then   -- random should not be from I
     --randomIndex  : I   := randnum numberOfGenerators
     randomIndex := 1+(random()$Integer rem numberOfGenerators)
     x : M R := algebraGenerators.randomIndex
  foundResult : B := false
  for i in 1..numberOfTries until foundResult repeat
    -- try to create a non-singular element of the algebra
    -- generated by "algebraGenerators". If only two generators,
    -- i < 7 and not "randomelements" use Parker's  fingerprints
    -- i >= 7 create random elements recursively:
    -- x_i+1 :=x_i * mr1 + mr2, where mr1 and mr2 are randomly
    -- chosen elements form "algebraGenerators".
    if i = 7 then randomelements := true
    if randomelements then
       --randomIndex := randnum numberOfGenerators
       randomIndex := 1+(random()$Integer rem numberOfGenerators)
       x := x * algebraGenerators.randomIndex
       --randomIndex := randnum numberOfGenerators
       randomIndex := 1+(random()$Integer rem numberOfGenerators)
       x := x + algebraGenerators.randomIndex
    else
       x := fingerPrint (i, algebraGenerators.1,_
         algebraGenerators.2 , x)
    -- test singularity of x
    n : NNI := #row(x, 1)  -- degree  of representation
    if (rank x) ^= n then  -- x singular
      if randomelements then
         messagePrint "Random element in generated algebra is singular"
      else
         messagePrint "Fingerprint element in generated algebra is singular"
      kernel : L V R := nullSpace x
      -- the first number is the maximal number of one dimensional
      -- subspaces of the kernel, the second is a user given
      -- constant
      numberOfOneDimSubspacesInKernel : I := (q**(#kernel)-1)quo(q-1)
      numberOfTests : I :=  _
        min(numberOfOneDimSubspacesInKernel, maxTests)
      for j in 1..numberOfTests repeat
         --we create an element in the kernel, there is a good
         --probability for it to generate a proper submodule, the
         --called "split" does the further work:
         result := _
```

```
            split(algebraGenerators,scanOneDimSubspaces(kernel,j))
          -- we had "not null rest result" directly in the following
          -- if .. then, but the statment there foundResult := true
          -- didn't work properly
          foundResult :=  not null rest result
          if foundResult then
            leave -- inner for-loop
            -- finish here with result
          else  -- no proper submodule
            -- we were not successfull, i.e gen. submodule was
            -- not proper, if the whole kernel is already scanned,
            -- Norton's irreducibility test is used now.
            if (j+1)>numberOfOneDimSubspacesInKernel then
              -- we know that all the cyclic submodules generated
              -- by all non-trivial elements of the kernel are proper.
              foundResult := true
              result : L L M R := irreducibilityTestInternal (_
                algebraGenerators,x,true)
              leave  -- inner for-loop
        -- here ends the inner for-loop
      else  -- x non-singular
        if randomelements then
          messagePrint "Random element in generated algebra is non-singular"
        else
          messagePrint "Fingerprint element in generated algebra is non-singular"
    -- here ends the outer for-loop
    if not foundResult then
       result : L L M R := [nil()$(L M R), nil()$(L M R)]
       messagePrint " "
       messagePrint "Sorry, no result, try meatAxe(...,true)"
       messagePrint "  or consider using an extension field."
    result


meatAxe (algebraGenerators) ==
  meatAxe(algebraGenerators, false, 25, 7)


meatAxe (algebraGenerators, randomElements?) ==
  randomElements? => meatAxe (algebraGenerators, true, 25, 7)
  meatAxe(algebraGenerators, false, 6, 7)


meatAxe (algebraGenerators:L M R, numberOfTries:PI) ==
  meatAxe (algebraGenerators, true, numberOfTries, 7)



scanOneDimSubspaces(basis,n) ==
  -- "dimension" of subspace generated by "basis"
```

```
dim : NNI := #basis
-- "dimension of the whole space:
nn : NNI := #(basis.1)
q : NNI := size()$R
-- number of all one-dimensional subspaces:
nred : I := n rem ((q**dim -1) quo (q-1))
pos : I := nred
i : I := 0
for i in 0..dim-1 while nred >= 0 repeat
  pos := nred
  nred := nred - (q**i)
i  := if i = 0 then 0 else i-1
coefficients : V R := new(dim,0$R)
coefficients.(dim-i) := 1$R
iR : L I := wholeRagits(pos::RADIX q)
for j in 1..(maxIndex iR) repeat
  coefficients.(dim-((#iR)::I) +j) := index((iR.j+(q::I))::PI)$R
result : V R := new(nn,0)
for i in 1..maxIndex coefficients repeat
  newAdd : V R := coefficients.i * basis.i
  for j in 1..nn repeat
    result.j := result.j + newAdd.j
result
```

———-

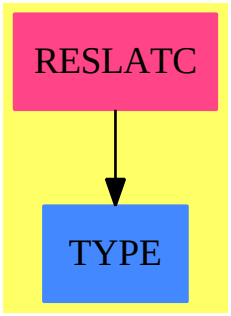
— **REP2.dotabb** —

```
"REP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP2"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"REP2" -> "IVECTOR"
```

———-

19.38 package RESLATC ResolveLatticeCompletion

19.38.1 ResolveLatticeCompletion (RESLATC)



Exports:
coerce

— package RESLATC ResolveLatticeCompletion —

```
)abbrev package RESLATC ResolveLatticeCompletion
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Basic Operations:
++ Related Domains: ErrorFunctions, Exit, Void
++ Also See:
++ AMS Classifications:
++ Keywords: mode, resolve, type lattice
++ Examples:
++ References:
++ Description:
++ This package provides coercions for the special types \spadtype{Exit}
++ and \spadtype{Void}.

ResolveLatticeCompletion(S: Type): with
        coerce: S -> Void
             ++ coerce(s) throws all information about s away.
             ++ This coercion allows values of any type to appear
             ++ in contexts where they will not be used.
             ++ For example, it allows the resolution of different types in
             ++ the \spad{then} and \spad{else} branches when an \spad{if}
             ++ is in a context where the resulting value is not used.
        coerce: Exit -> S
             ++ coerce(e) is never really evaluated.  This coercion is
             ++ used for formal type correctness when a function will not
             ++ return directly to its caller.
```

```
    == add
        coerce(s: S): Void == void()
        coerce(e: Exit): S ==
            error "Bug: Should not be able to obtain value of type Exit"
```

———-

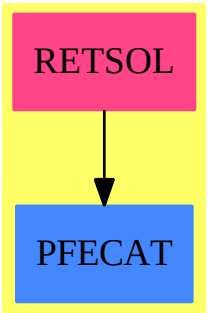
— **RESLATC.dotabb** —

```
"RESLATC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RESLATC"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"RESLATC" -> "TYPE"
```

———-

19.39 package RETSOL RetractSolvePackage

19.39.1 RetractSolvePackage (RETSOL)



Exports:
solveRetract

— **package RETSOL RetractSolvePackage** —

```
)abbrev package RETSOL RetractSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 31 October 1991
++ Description:
++ RetractSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving.
```

```
RetractSolvePackage(Q, R): Exports == Implementation where
  Q: IntegralDomain
  R: Join(IntegralDomain, RetractableTo Q)

  PQ  ==> Polynomial Q
  FQ  ==> Fraction PQ
  SY  ==> Symbol
  P   ==> Polynomial R
  F   ==> Fraction P
  EQ  ==> Equation
  SSP ==> SystemSolvePackage

  Exports ==> with
    solveRetract: (List P, List SY) -> List List EQ F
      ++ solveRetract(lp,lv) finds the solutions of the list lp of
      ++ rational functions with respect to the list of symbols lv.
      ++ The function tries to retract all the coefficients of the equations
      ++ to Q before solving if possible.

  Implementation ==> add
    LEQQ2F : List EQ FQ -> List EQ F
    FQ2F   : FQ -> F
    PQ2P   : PQ -> P
    QIfCan : List P -> Union(List FQ, "failed")
    PQIfCan: P -> Union(FQ, "failed")

    PQ2P p   == map((q1:Q):R +-> q1::R, p)$PolynomialFunctions2(Q, R)
    FQ2F f   == PQ2P numer f / PQ2P denom f
    LEQQ2F l == [equation(FQ2F lhs eq, FQ2F rhs eq) for eq in l]

    solveRetract(lp, lv) ==
      (u := QIfCan lp) case "failed" =>
        solve([p::F for p in lp]$List(F), lv)$SSP(R)
      [LEQQ2F l for l in solve(u::List(FQ), lv)$SSP(Q)]

    QIfCan l ==
      ans:List(FQ) := empty()
      for p in l repeat
        (u := PQIfCan p) case "failed" => return "failed"
        ans := concat(u::FQ, ans)
      ans

    PQIfCan p ==
      (u := mainVariable p) case "failed" =>
        (r := retractIfCan(ground p)@Union(Q,"failed")) case Q => r::Q::PQ::FQ
        "failed"
      up := univariate(p, s := u::SY)
      ans:FQ := 0
      while up ^= 0 repeat
```

```
        (v := PQIfCan leadingCoefficient up) case "failed" => return "failed"
        ans := ans + monomial(1, s, degree up)$PQ * (v::FQ)
        up  := reductum up
      ans
```

———-

— **RETSOL.dotabb** —

```
"RETSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RETSOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RETSOL" -> "PFECAT"
```

———-

19.40 package RFP RootsFindingPackage

— **RootsFindingPackage.input** —

```
)set break resume
)sys rm -f RootsFindingPackage.output
)spool RootsFindingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RootsFindingPackage
--R
--R RootsFindingPackage(K: Field)  is a package constructor
--R Abbreviation for RootsFindingPackage is RFP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFP
--R
--R------------------------------ Operations --------------------------------
--R foundZeroes : () -> List(K)
--R distinguishedCommonRootsOf : (List(SparseUnivariatePolynomial(K)),K) -> Record(zeros: List(K),extDegree
--R distinguishedRootsOf : (SparseUnivariatePolynomial(K),K) -> Record(zeros: List(K),extDegree: Integer)
--R setFoundZeroes : List(K) -> List(K)
--R
--E 1

)spool
)lisp (bye)
```

———-

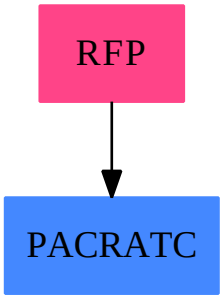
— RootsFindingPackage.help —

```
====================================================================
RootsFindingPackage examples
====================================================================

See Also:
o )show RootsFindingPackage
```

———-

19.40.1 RootsFindingPackage (RFP)



Exports:
distinguishedCommonRootsOf distinguishedRootsOf foundZeroes setFoundZeroes

— package RFP RootsFindingPackage —

```
)abbrev package RFP RootsFindingPackage
++ Authors: G. Hache
++ Date Created: 6 Oct 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This pacackage finds all the roots of a polynomial. If the constant field is
++ not large enough then it returns the list of found zeros and the degree
++ of the extension need to find the other roots missing. If the return
++ degree is 1 then all the roots have been found. If 0 is return
++ for the extension degree then there are an infinite number of zeros,
++ that is you ask for the zeroes of 0. In the case of infinite field
++ a list of all found zeros is kept and for each other call of a function
++ that finds zeroes, a check is made on that list; this is to keep
```

```
++ a kind of "canonical" representation of the elements.
RootsFindingPackage(K):P==T where
  K:Field

  LIST     ==> List
  INT      ==> Integer
  NNI      ==> NonNegativeInteger
  MFINFACT ==> MultFiniteFactorize
  FFFACTSE ==> FiniteFieldFactorizationWithSizeParseBySideEffect
  SUP      ==> SparseUnivariatePolynomial
  REC      ==> Record(zeros:List(K),extDegree:INT)

  P== with

    distinguishedRootsOf: (SUP(K),K) -> REC
      ++ distinguishedRootsOf returns a record consisting of a list of zeros
      ++ of the input polynomial followed by the smallest extension degree
      ++ needed to find all the zeros. If K has
      ++ \spad{PseudoAlgebraicClosureOfFiniteFieldCategory} or
      ++ \spad{PseudoAlgebraicClosureOfRationalNumberCategory} then
      ++ a root is created for each irreducible factor, and only these
      ++ roots are returns and not their conjugate.

    distinguishedCommonRootsOf: (List SUP(K),K) -> REC
      ++ distinguishedCommonRootsOf returns the common zeros of a list of
      ++ polynomial. It returns a record as in distinguishedRootsOf. If 0
      ++ is returned as extension degree then there are an infinite number
      ++ of common zeros (in this case, the polynomial 0 was given in the
      ++ list of input polynomials).

    foundZeroes: () -> List K
      ++ foundZeroes returns the list of already
      ++ found zeros by the functions
      ++ distinguishedRootsOf  and
      ++ distinguishedCommonRootsOf.

    setFoundZeroes: List K -> List K
      ++ setFoundZeroes sets the list of foundZeroes to the given one.

  T== add
    -- signature of local function
    zeroOfLinearPoly: SUP(K) -> K
    -- local variable
    listOfAllZeros:List(K):=empty()

    foundZeroes==listOfAllZeros

    if K has PseudoAlgebraicClosureOfPerfectFieldCategory then
      distinguishedRootsOf(polyZero, theExtension) ==
        --PRECONDITION: setExtension! is called in K to set the extension to
```

```
      --the extension of factorization
      zero?(polyZero) =>
        [empty(),0]
      listOfZeros:List(K):=distinguishedRootsOf(polyZero,theExtension)$K
      [listOfZeros,1]

  if K has FiniteFieldCategory  and _
   ^(K has PseudoAlgebraicClosureOfFiniteFieldCategory) then
    distinguishedRootsOf(polyZero,dummy)==
      zero?(polyZero) => [empty(),0]
      factorpolyZero:=factor(polyZero)$FFFACTSE(K,SUP(K))
      listOfFactor:=factorList(factorpolyZero)
      listFact:= [pol.fctr for pol in listOfFactor]
      degExt:INT:=
        lcm(([degree(poly) for poly in listFact]) pretend LIST(INT))
      listOfZeros:List(K):=removeDuplicates_
        [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
      [listOfZeros,degExt]

  if K has QuotientFieldCategory( Integer ) and _
   ^(K has PseudoAlgebraicClosureOfRationalNumberCategory) then
    distinguishedRootsOf(polyZero,dummy)==
      zero?(polyZero) => [empty(),0]
      factorpolyZero:=factor(polyZero)$RationalFactorize( SUP(K) )
      listOfFactor:=factorList(factorpolyZero)
      listFact:= [pol.fctr for pol in listOfFactor]
      degExt:INT:=
        lcm(([degree(poly) for poly in listFact]) pretend LIST(INT))
      listOfZeros:List(K):=removeDuplicates_
        [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
      [listOfZeros,degExt]

  distinguishedCommonRootsOf(listOfPoly1,theExtension)==
    listOfPoly:List(SUP(K)):=[pol for pol in listOfPoly1 | ^zero?(pol)]
    empty?(listOfPoly) ==> [empty(),0]
    reco:= distinguishedRootsOf(gcd(listOfPoly),theExtension)
    listOfZeros:= reco.zeros
    degExt:INT:= reco.extDegree
    [listOfZeros,degExt]

  zeroOfLinearPoly(pol)==
    ^one?(degree(pol)) => error "the polynomial is not linear"
    listCoef:List(K):=coefficients(pol)
    one?(#listCoef) => 0
    - last(listCoef) / first(listCoef)

  setFoundZeroes(setlist)==
    oldListOfAllZeroes:= copy listOfAllZeros
    listOfAllZeros:=setlist
    oldListOfAllZeroes
```

———-

— **RFP.dotabb** —

```
"RFP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFP"]
"PACRATC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PACRATC"]
"RFP" -> "PACRATC"
```

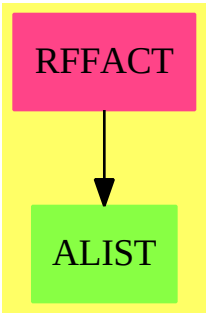
———-

Chapter 20

Chapter S

20.1 package SAERFFC SAERationalFunctionAlgFactor

20.1.1 SAERationalFunctionAlgFactor (SAERFFC)



Exports:
factor

— package SAERFFC SAERationalFunctionAlgFactor —

```
)abbrev package SAERFFC SAERationalFunctionAlgFactor
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: ???
++ Keywords: factorization, algebraic extension, univariate polynomial
++ Description:
++ Factorization of univariate polynomials with coefficients in an
++ algebraic extension of \spadtype{Fraction Polynomial Integer}.
```

```
SAERationalFunctionAlgFactor(UP, SAE, UPA): Exports == Implementation where
  UP : UnivariatePolynomialCategory Fraction Polynomial Integer
  SAE : Join(Field, CharacteristicZero,
                     MonogenicAlgebra(Fraction Polynomial Integer, UP))
  UPA: UnivariatePolynomialCategory SAE

  Exports ==> with
    factor: UPA -> Factored UPA
      ++ factor(p) returns a prime factorisation of p.

  Implementation ==> add
    factor q ==
      factor(q, factor$RationalFunctionFactor(UP)
              )$InnerAlgFactor(Fraction Polynomial Integer, UP, SAE, UPA)
```

———-

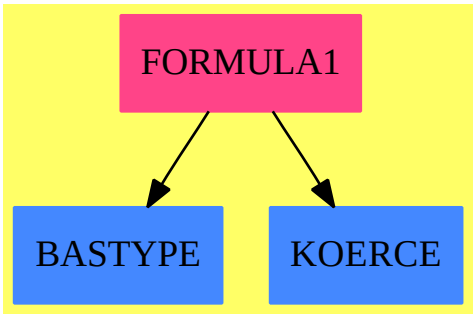
— **SAERFFC.dotabb** —

```
"SAERFFC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SAERFFC"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"SAERFFC" -> "MONOGEN"
```

———-

20.2 package FORMULA1 ScriptFormulaFormat1

20.2.1 ScriptFormulaFormat1 (FORMULA1)



Exports:
coerce

— package FORMULA1 ScriptFormulaFormat1 —

```
)abbrev package FORMULA1 ScriptFormulaFormat1
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ Change History:
++ Basic Operations: coerce
++ Related Constructors: ScriptFormulaFormat
++ Also See: TexFormat, TexFormat1
++ AMS Classifications:
++ Keywords: output, format, SCRIPT, BookMaster, formula
++ References:
++   SCRIPT Mathematical Formula Formatter User's Guide, SH20-6453,
++   IBM Corporation, Publishing Systems Information Development,
++   Dept. G68, P.O. Box 1900, Boulder, Colorado, USA 80301-9191.
++ Description:
++ \spadtype{ScriptFormulaFormat1} provides a utility coercion for
++ changing to SCRIPT formula format anything that has a coercion to
++ the standard output format.

ScriptFormulaFormat1(S : SetCategory): public == private where
  public  ==  with
    coerce: S -> ScriptFormulaFormat()
      ++ coerce(s) provides a direct coercion from an expression s of domain S
      ++ to SCRIPT formula format. This allows the user to skip the step of
      ++ first manually coercing the object to standard output format
      ++ before it is coerced to SCRIPT formula format.

  private == add
    import ScriptFormulaFormat()

    coerce(s : S): ScriptFormulaFormat ==
      coerce(s :: OutputForm)$ScriptFormulaFormat
```

———-

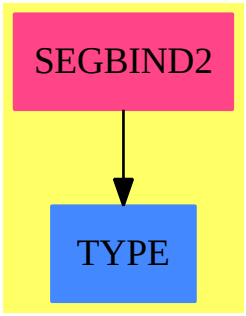
— FORMULA1.dotabb —

```
"FORMULA1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORMULA1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"FORMULA1" -> "BASTYPE"
"FORMULA1" -> "KOERCE"
```

———-

20.3 package SEGBIND2 SegmentBindingFunctions2

20.3.1 SegmentBindingFunctions2 (SEGBIND2)



Exports:
map

— **package SEGBIND2 SegmentBindingFunctions2** —

```
)abbrev package SEGBIND2 SegmentBindingFunctions2
++ Author:
++ Date Created:
++ Date Last Updated: June 4, 1991
++ Basic Operations:
++ Related Domains: SegmentBinding, Segment, Equation
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ This package provides operations for mapping functions onto
++ \spadtype{SegmentBinding}s.

SegmentBindingFunctions2(R:Type, S:Type): with
  map: (R -> S, SegmentBinding R) -> SegmentBinding S
      ++ map(f,v=a..b) returns the value given by \spad{v=f(a)..f(b)}.
 == add
  map(f, b) ==
    equation(variable b, map(f, segment b)$SegmentFunctions2(R, S))
```

————-

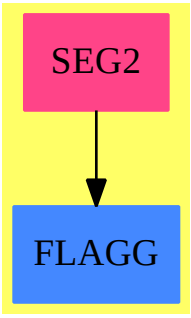
— **SEGBIND2.dotabb** —

```
"SEGBIND2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SEGBIND2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"SEGBIND2" -> "TYPE"
```

———-

20.4 package SEG2 SegmentFunctions2

20.4.1 SegmentFunctions2 (SEG2)



Exports:
map

— package SEG2 SegmentFunctions2 —

```
)abbrev package SEG2 SegmentFunctions2
++ Author:
++ Date Created:
++ Date Last Updated: June 4, 1991
++ Basic Operations:
++ Related Domains: Segment, UniversalSegment
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ This package provides operations for mapping functions onto segments.

SegmentFunctions2(R:Type, S:Type): public == private where
  public ==> with
    map: (R -> S, Segment R) -> Segment S
        ++ map(f,l..h) returns a new segment \spad{f(l)..f(h)}.
```

```
    if R has OrderedRing then
      map: (R -> S, Segment R) -> List S
        ++ map(f,s) expands the segment s, applying \spad{f} to each
        ++ value.  For example, if \spad{s = l..h by k}, then the list
        ++ \spad{[f(l), f(l+k),..., f(lN)]} is computed, where
        ++ \spad{lN <= h < lN+k}.


  private ==> add
    map(f : R->S, r : Segment R): Segment S ==
      SEGMENT(f lo r,f hi r)$Segment(S)

    if R has OrderedRing then
     map(f : R->S, r : Segment R): List S ==
       lr := nil()$List(S)
       l := lo r
       h := hi r
       inc := (incr r)::R
       if inc > 0 then
         while l <= h repeat
           lr := concat(f(l), lr)
           l := l + inc
       else
         while l >= h repeat
           lr := concat(f(l), lr)
           l := l + inc
       reverse_! lr
```

———-

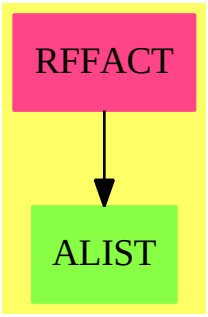
— SEG2.dotabb —

```
"SEG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SEG2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"SEG2" -> "FLAGG"
```

———-

20.5 package SAEFACT SimpleAlgebraicExtensionAlgFactor

20.5.1 SimpleAlgebraicExtensionAlgFactor (SAEFACT)



Exports:
factor

— package SAEFACT SimpleAlgebraicExtensionAlgFactor —

```
)abbrev package SAEFACT SimpleAlgebraicExtensionAlgFactor
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: ???
++ Keywords: factorization, algebraic extension, univariate polynomial
++ Description:
++ Factorization of univariate polynomials with coefficients in an
++ algebraic extension of the rational numbers (\spadtype{Fraction Integer}).

SimpleAlgebraicExtensionAlgFactor(UP,SAE,UPA):Exports==Implementation where
  UP : UnivariatePolynomialCategory Fraction Integer
  SAE : Join(Field, CharacteristicZero,
                        MonogenicAlgebra(Fraction Integer, UP))
  UPA: UnivariatePolynomialCategory SAE

  Exports ==> with
    factor: UPA -> Factored UPA
      ++ factor(p) returns a prime factorisation of p.

  Implementation ==> add
    factor q ==
      factor(q, factor$RationalFactorize(UP)
                     )$InnerAlgFactor(Fraction Integer, UP, SAE, UPA)
```

———-

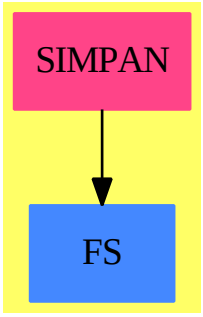
— SAEFACT.dotabb —

```
"SAEFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SAEFACT"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"SAEFACT" -> "MONOGEN"
```

———-

20.6 package SIMPAN SimplifyAlgebraicNumberConvertPackage

20.6.1 SimplifyAlgebraicNumberConvertPackage (SIMPAN)



Exports:
simplify

— package SIMPAN SimplifyAlgebraicNumberConvertPackage —

```
)abbrev package SIMPAN SimplifyAlgebraicNumberConvertPackage
++ Description:
++ Package to allow simplify to be called on AlgebraicNumbers
++ by converting to EXPR(INT)

SimplifyAlgebraicNumberConvertPackage(): with
  simplify: AlgebraicNumber -> Expression(Integer)
    ++ simplify(an) applies simplifications to an
 == add
  simplify(a:AlgebraicNumber) ==
    simplify(a::Expression(Integer))$TranscendentalManipulations(Integer, Expression Integer)
```

———-

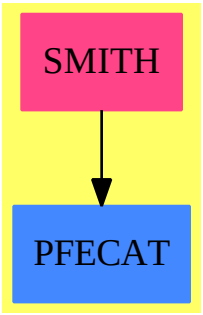
— **SIMPAN.dotabb** —

```
"SIMPAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIMPAN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SIMPAN" -> "FS"
```

———-

20.7 package SMITH SmithNormalForm

20.7.1 SmithNormalForm (SMITH)



Exports:
completeHermite completeSmith diophantineSystem hermite smith

— **package SMITH SmithNormalForm** —

```
)abbrev package SMITH SmithNormalForm
++ Author: Patrizia Gianni
++ Date Created: October 1992
++ Date Last Updated:
++ Basic Operations:
++ Related Domains: Matrix(R)
++ Also See:
++ AMS Classifications:
++ Keywords: matrix, canonical forms, linear algebra
++ Examples:
++ References:
++ Description:
++ \spadtype{SmithNormalForm} is a package
++ which provides some standard canonical forms for matrices.
```

```
SmithNormalForm(R,Row,Col,M) : Exports == Implementation where

  R   : EuclideanDomain
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M   : MatrixCategory(R,Row,Col)

  I            ==> Integer
  NNI          ==> NonNegativeInteger
  HermiteForm  ==>   Record(Hermite:M,eqMat:M)
  SmithForm    ==>   Record(Smith : M, leftEqMat : M, rightEqMat : M)
  PartialV     ==> Union(Col, "failed")
  Both         ==> Record(particular: PartialV, basis: List Col)

  Exports ==  with
    hermite : M -> M
      ++ \spad{hermite(m)} returns the Hermite normal form of the
      ++ matrix m.
    completeHermite : M -> HermiteForm
      ++ \spad{completeHermite} returns a record  that contains
      ++ the Hermite normal form H of the matrix and the equivalence matrix
      ++ U such that U*m = H
    smith : M -> M
      ++ \spad{smith(m)} returns the Smith Normal form of the matrix m.
    completeSmith : M ->  SmithForm
      ++ \spad{completeSmith} returns a record  that contains
      ++ the Smith normal form H of the matrix and the left and right
      ++ equivalence matrices U and V such that U*m*v = H
    diophantineSystem : (M,Col) -> Both
      ++ \spad{diophantineSystem(A,B)} returns a particular integer solution and
      ++ an integer  basis  of the equation \spad{AX = B}.

  Implementation == add
    MATCAT1 ==> MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Col2,M2)
    MATCAT2 ==> MatrixCategoryFunctions2(QF,Row2,Col2,M2,R,Row,Col,M)
    QF      ==> Fraction R
    Row2    ==> Vector QF
    Col2    ==> Vector QF
    M2      ==> Matrix QF

                ------  Local Functions -----
    elRow1       :   (M,I,I)         ->  M
    elRow2       :  (M,R,I,I)        ->  M
    elColumn2    :  (M,R,I,I)        ->  M
    isDiagonal?  :      M            ->  Boolean
    ijDivide     : (SmithForm ,I,I)  ->  SmithForm
    lastStep     :   SmithForm       ->  SmithForm
    test1        :  (M,Col,NNI)      ->  Union(NNI, "failed")
    test2        : (M, Col,NNI,NNI)  ->  Union( Col, "failed")
```

```
 -- inconsistent system : case  0 = c --
test1(sm:M,b:Col,m1 : NNI) : Union(NNI , "failed") ==
  km:=m1
  while zero? sm(km,km) repeat
    if not zero?(b(km)) then return "failed"
    km:= (km - 1) :: NNI
  km

if Col has shallowlyMutable then

  test2(sm : M ,b : Col, n1:NNI,dk:NNI) : Union( Col, "failed") ==
    -- test divisibility --
    sol:Col := new(n1,0)
    for k in 1..dk repeat
      if (c:=(b(k) exquo sm(k,k))) case "failed" then return "failed"
      sol(k):= c::R
    sol

 -- test if the matrix is diagonal or pseudo-diagonal --
isDiagonal?(m : M) : Boolean ==
  m1:= nrows m
  n1:= ncols m
  for i in 1..m1 repeat
    for j in 1..n1 | (j ^= i) repeat
      if  not zero?(m(i,j)) then return false
  true

   -- elementary operation of first kind: exchange two rows --
elRow1(m:M,i:I,j:I) : M ==
  vec:=row(m,i)
  setRow!(m,i,row(m,j))
  setRow!(m,j,vec)
  m

        -- elementary operation of second kind: add to row i--
                    -- a*row j  (i^=j) --
elRow2(m : M,a:R,i:I,j:I) : M ==
  vec:= map(x +-> a*x,row(m,j))
  vec:=map("+",row(m,i),vec)
  setRow!(m,i,vec)
  m
        -- elementary operation of second kind: add to column i --
                      -- a*column j (i^=j) --
elColumn2(m : M,a:R,i:I,j:I) : M ==
  vec:= map(x +-> a*x,column(m,j))
  vec:=map("+",column(m,i),vec)
  setColumn!(m,i,vec)
  m

   -- modify SmithForm in such a way that the term m(i,i) --
```

```
          -- divides the term m(j,j). m is diagonal --
    ijDivide(sf : SmithForm , i : I,j : I) : SmithForm ==
      m:=sf.Smith
      mii:=m(i,i)
      mjj:=m(j,j)
      extGcd:=extendedEuclidean(mii,mjj)
      d := extGcd.generator
      mii:=(mii exquo d)::R
      mjj := (mjj exquo d) :: R
      -- add to row j extGcd.coef1*row i --
      lMat:=elRow2(sf.leftEqMat,extGcd.coef1,j,i)
      -- switch rows i and j --
      lMat:=elRow1(lMat,i,j)
      -- add to row j -mii*row i --
      lMat := elRow2(lMat,-mii,j,i)
--      lMat := ijModify(mii,mjj,extGcd.coef1,extGcd.coef2,sf.leftEqMat,i,j)
      m(j,j):= m(i,i) * mjj
      m(i,i):= d
      -- add to column i extGcd.coef2 * column j --
      rMat := elColumn2(sf.rightEqMat,extGcd.coef2,i,j)
      -- add to column j -mjj*column i --
      rMat:=elColumn2(rMat,-mjj,j,i)
      -- multiply by -1 column j --
      setColumn!(rMat,j,map(x +-> -1*x,column(rMat,j)))
      [m,lMat,rMat]


     -- given a diagonal matrix compute its Smith form --
    lastStep(sf : SmithForm) : SmithForm ==
      m:=sf.Smith
      m1:=min(nrows m,ncols m)
      for i in 1..m1 while (mii:=m(i,i)) ^=0 repeat
        for j in i+1..m1 repeat
          if (m(j,j) exquo mii) case "failed" then return
             lastStep(ijDivide(sf,i,j))
      sf

    -- given m and t row-equivalent matrices, with t in upper triangular --
          -- form  compute the matrix u such that u*m=t --
    findEqMat(m :  M,t : M) : Record(Hermite : M, eqMat : M) ==
      m1:=nrows m
      n1:=ncols m
      "and"/[zero? t(m1,j) for j in 1..n1] => -- there are 0 rows
         if "and"/[zero? t(1,j) for j in 1..n1]
         then return [m,scalarMatrix(m1,1)]  -- m is the zero matrix
         mm:=horizConcat(m,scalarMatrix(m1,1))
         mmh:=rowEchelon mm
         [subMatrix(mmh,1,m1,1,n1), subMatrix(mmh,1,m1,n1+1,n1+m1)]
      u:M:=zero(m1,m1)
      j:=1
```

```
  while t(1,j)=0 repeat j:=j+1  -- there are 0 columns
  t1:=copy t
  mm:=copy m
  if j>1 then
    t1:=subMatrix(t,1,m1,j,n1)
    mm:=subMatrix(m,1,m1,j,n1)
  t11:=t1(1,1)
  for i in 1..m1 repeat
    u(i,1) := (mm(i,1) exquo t11) :: R
    for j in 2..m1 repeat
      j0:=j
      while zero?(tjj:=t1(j,j0)) repeat j0:=j0+1
      u(i,j) :=
       ((mm(i,j0)-("+"/[u(i,k)*t1(k,j0) for k in 1..(j-1)])) exquo tjj)::R
  u1:M2:= map(x +-> x::QF,u)$MATCAT1
  [t,map(retract$QF,(inverse u1)::M2)$MATCAT2]

            --- Hermite normal form of m ---
hermite(m:M) : M == rowEchelon m

 -- Hermite normal form and equivalence matrix --
completeHermite(m : M) : Record(Hermite : M, eqMat : M) ==
  findEqMat(m,rowEchelon m)

smith(m : M) : M == completeSmith(m).Smith

completeSmith(m : M) : Record(Smith : M, leftEqMat : M, rightEqMat : M) ==
  cm1:=completeHermite m
  leftm:=cm1.eqMat
  m1:=cm1.Hermite
  isDiagonal? m1 => lastStep([m1,leftm,scalarMatrix(ncols m,1)])
  nr:=nrows m
  cm1:=completeHermite transpose m1
  rightm:= transpose cm1.eqMat
  m1:=cm1.Hermite
  isDiagonal? m1 =>
    cm2:=lastStep([m1,leftm,rightm])
    nrows(m:=cm2.Smith) = nr => cm2
    [transpose m,cm2.leftEqMat, cm2.rightEqMat]
  cm2:=completeSmith m1
  cm2:=lastStep([cm2.Smith,transpose(cm2.rightEqMat)*leftm,
           rightm*transpose(cm2.leftEqMat)])
  nrows(m:=cm2.Smith) = nr => cm2
  [transpose m, cm2.leftEqMat, cm2.rightEqMat]

-- Find the solution in R of the linear system mX = b --
diophantineSystem(m : M, b : Col) : Both  ==
  sf:=completeSmith m
  sm:=sf.Smith
  m1:=nrows sm
```

```
lm:=sf.leftEqMat
b1:Col:= lm* b
(t1:=test1(sm,b1,m1)) case "failed" => ["failed",empty()]
dk:=t1 :: NNI
n1:=ncols sm
(t2:=test2(sm,b1,n1,dk)) case "failed" => ["failed",empty()]
rm := sf.rightEqMat
sol:=rm*(t2 :: Col)  -- particular solution
dk = n1  => [sol,list new(n1,0)]
lsol:List Col := [column(rm,i) for i in (dk+1)..n1]
[sol,lsol]
```

———-

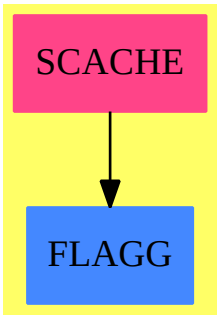
— **SMITH.dotabb** —

```
"SMITH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SMITH"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SMITH" -> "PFECAT"
```

———-

20.8 package SCACHE SortedCache

20.8.1 SortedCache (SCACHE)



Exports:
cache clearCache enterInCache

— **package SCACHE SortedCache** —

```
)abbrev package SCACHE SortedCache
```

```
++ Author: Manuel Bronstein
++ Date Created: 31 Oct 1988
++ Date Last Updated: 14 May 1991
++ Description:
++ A sorted cache of a cachable set S is a dynamic structure that
++ keeps the elements of S sorted and assigns an integer to each
++ element of S once it is in the cache. This way, equality and ordering
++ on S are tested directly on the integers associated with the elements
++ of S, once they have been entered in the cache.

SortedCache(S:CachableSet): Exports == Implementation where
  N    ==> NonNegativeInteger
  DIFF ==> 1024

  Exports ==> with
    clearCache  : () -> Void
      ++ clearCache() empties the cache.
    cache       : () -> List S
      ++ cache() returns the current cache as a list.
    enterInCache: (S, S -> Boolean) -> S
      ++ enterInCache(x, f) enters x in the cache, calling \spad{f(y)} to
      ++ determine whether x is equal to y. It returns x with an integer
      ++ associated with it.
    enterInCache: (S, (S, S) -> Integer) -> S
      ++ enterInCache(x, f) enters x in the cache, calling \spad{f(x, y)} to
      ++ determine whether \spad{x < y (f(x,y) < 0), x = y (f(x,y) = 0)}, or
      ++ \spad{x > y (f(x,y) > 0)}.
      ++ It returns x with an integer associated with it.

  Implementation ==> add
    shiftCache   : (List S, N) -> Void
    insertInCache: (List S, List S, S, N) -> S

    cach := [nil()]$Record(cche:List S)

    cache() == cach.cche

    shiftCache(l, n) ==
      for x in l repeat setPosition(x, n + position x)
      void

    clearCache() ==
      for x in cache repeat setPosition(x, 0)
      cach.cche := nil()
      void

    enterInCache(x:S, equal?:S -> Boolean) ==
      scan := cache()
      while not null scan repeat
        equal?(y := first scan) =>
```

```
          setPosition(x, position y)
          return y
        scan := rest scan
      setPosition(x, 1 + #cache())
      cach.cche := concat(cache(), x)
      x

    enterInCache(x:S, triage:(S, S) -> Integer) ==
      scan := cache()
      pos:N:= 0
      for i in 1..#scan repeat
        zero?(n := triage(x, y := first scan)) =>
          setPosition(x, position y)
          return y
        n<0 => return insertInCache(first(cache(),(i-1)::N),scan,x,pos)
        scan := rest scan
        pos  := position y
      setPosition(x, pos + DIFF)
      cach.cche := concat(cache(), x)
      x

    insertInCache(before, after, x, pos) ==
      if ((pos+1) = position first after) then shiftCache(after, DIFF)
      setPosition(x, pos + (((position first after) - pos)::N quo 2))
      cach.cche := concat(before, concat(x, after))
      x
```

———-

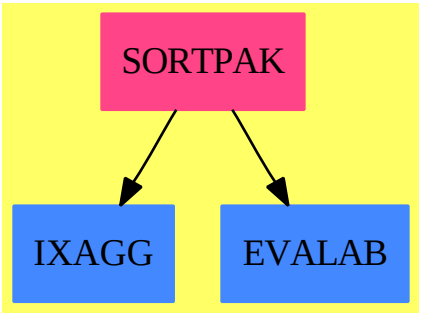
— SCACHE.dotabb —

```
"SCACHE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SCACHE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"SCACHE" -> "FLAGG"
```

———-

20.9 package SORTPAK SortPackage

20.9.1 SortPackage (SORTPAK)



Exports:
bubbleSort! insertionSort!

— **package SORTPAK SortPackage** —

```
)abbrev package SORTPAK SortPackage
++ Description:
++ This package exports sorting algorithnms

SortPackage(S,A) : Exports == Implementation where
  S: Type
  A: IndexedAggregate(Integer,S)
    with (finiteAggregate; shallowlyMutable)

  Exports == with
    bubbleSort_!: (A,(S,S) -> Boolean) -> A
      ++ bubbleSort!(a,f) \undocumented
    insertionSort_!: (A, (S,S) -> Boolean) -> A
      ++ insertionSort!(a,f) \undocumented
    if S has OrderedSet then
      bubbleSort_!: A -> A
        ++ bubbleSort!(a) \undocumented
      insertionSort_!: A -> A
        ++ insertionSort! \undocumented

  Implementation == add
    bubbleSort_!(m,f) ==
      n := #m
      for i in 1..(n-1) repeat
        for j in n..(i+1) by -1 repeat
          if f(m.j,m.(j-1)) then swap_!(m,j,j-1)
      m
```

```
    insertionSort_!(m,f) ==
      for i in 2..#m repeat
        j := i
        while j > 1 and f(m.j,m.(j-1)) repeat
          swap_!(m,j,j-1)
          j := (j - 1) pretend PositiveInteger
      m
    if S has OrderedSet then
      bubbleSort_!(m) == bubbleSort_!(m,_<$S)
      insertionSort_!(m) == insertionSort_!(m,_<$S)
    if A has UnaryRecursiveAggregate(S) then
      bubbleSort_!(m,fn) ==
        empty? m => m
        l := m
        while not empty? (r := l.rest) repeat
           r := bubbleSort_!(r,fn)
           x := l.first
           if fn(r.first,x) then
             l.first := r.first
             r.first := x
           l.rest := r
           l := l.rest
        m
```

———-

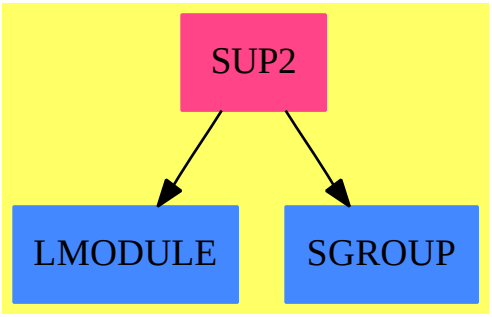
— SORTPAK.dotabb —

```
"SORTPAK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SORTPAK"]
"IXAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IXAGG"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"SORTPAK" -> "IXAGG"
"SORTPAK" -> "EVALAB"
```

———-

20.10 package SUP2 SparseUnivariatePolynomialFunctions2

20.10.1 SparseUnivariatePolynomialFunctions2 (SUP2)



Exports:
map

— package SUP2 SparseUnivariatePolynomialFunctions2 —

```
)abbrev package SUP2 SparseUnivariatePolynomialFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package lifts a mapping from coefficient rings R to S to
++ a mapping from sparse univariate polynomial over R to
++ a sparse univariate polynomial over S.
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

SparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
  map:(R->S,SparseUnivariatePolynomial R) -> SparseUnivariatePolynomial S
    ++ map(func, poly) creates a new polynomial by applying func to
    ++ every non-zero coefficient of the polynomial poly.
 == add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
           SparseUnivariatePolynomial R, S, SparseUnivariatePolynomial S)
```

———-

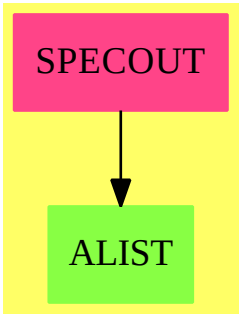
— **SUP2.dotabb** —

```
"SUP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"SUP2" -> "LMODULE"
"SUP2" -> "SGROUP"
```

———-

20.11 package SPECOUT SpecialOutputPackage

20.11.1 SpecialOutputPackage (SPECOUT)



Exports:
outputAsTex outputAsFortran outputAsScript

— **package SPECOUT SpecialOutputPackage** —

```
)abbrev package SPECOUT SpecialOutputPackage
++ Author: Stephen M. Watt
++ Date Created: September 1986
++ Date Last Updated: May 23, 1991
++ Basic Operations: outputAsFortran, outputAsScript, outputAsTex
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
```

```
++ Description:
++ SpecialOutputPackage allows FORTRAN, Tex and
++ Script Formula Formatter output from programs.

SpecialOutputPackage: public == private where
  public == with
    outputAsFortran: (String,OutputForm) -> Void
      ++ outputAsFortran(v,o) sends output v = o in FORTRAN format
      ++ to the destination defined by \spadsyscom{set output fortran}.
    outputAsFortran: OutputForm          -> Void
      ++ outputAsFortran(o) sends output o in FORTRAN format.
    outputAsScript:  OutputForm          -> Void
      ++ outputAsScript(o) sends output o in Script Formula Formatter format
      ++ to the destination defined by \spadsyscom{set output formula}.
    outputAsTex:     OutputForm          -> Void
      ++ outputAsTex(o) sends output o in Tex format to the destination
      ++ defined by \spadsyscom{set output tex}.
    outputAsFortran: List OutputForm     -> Void
      ++ outputAsFortran(l) sends (for each expression in the list l)
      ++ output in FORTRAN format to the destination defined by
      ++ \spadsyscom{set output fortran}.
    outputAsScript:  List OutputForm     -> Void
      ++ outputAsScript(l) sends (for each expression in the list l)
      ++ output in Script Formula Formatter format to the destination defined.
      ++ by \spadsyscom{set output forumula}.
    outputAsTex:     List OutputForm     -> Void
      ++ outputAsTex(l) sends (for each expression in the list l)
      ++ output in Tex format to the destination as defined by
      ++ \spadsyscom{set output tex}.

  private == add
    e : OutputForm
    l : List OutputForm
    var : String
    --ExpressionPackage()

    juxtaposeTerms: List OutputForm -> OutputForm
    juxtaposeTerms l == blankSeparate l

    outputAsFortran e ==
      dispfortexp$Lisp e
      void()$Void

    outputAsFortran(var,e) ==
      e := var::Symbol::OutputForm  = e
      dispfortexp(e)$Lisp
      void()$Void

    outputAsFortran l ==
      dispfortexp$Lisp juxtaposeTerms l
```

```
      void()$Void

    outputAsScript e ==
      formulaFormat$Lisp e
      void()$Void

    outputAsScript l ==
      formulaFormat$Lisp juxtaposeTerms l
      void()$Void

    outputAsTex e ==
      texFormat$Lisp e
      void()$Void

    outputAsTex l ==
      texFormat$Lisp juxtaposeTerms l
      void()$Void
```

———-

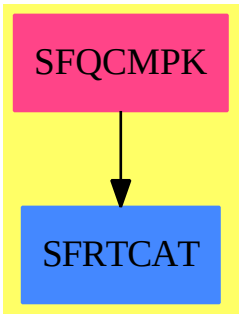
— **SPECOUT.dotabb** —

```
"SPECOUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SPECOUT"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SPECOUT" -> "ALIST"
```

———-

20.12 package SFQCMPK SquareFreeQuasiComponentPackage

20.12.1 SquareFreeQuasiComponentPackage (SFQCMPK)



Exports:

algebraicSort	branchIfCan
infRittWu?	internalInfRittWu?
internalSubPolSet?	internalSubQuasiComponent?
moreAlgebraic?	prepareDecompose
removeSuperfluousCases	removeSuperfluousQuasiComponents
startTable!	stopTable!
subCase?	subPolSet?
subQuasiComponent?	subTriSet?
supDimElseRittWu?	

— **SFQCMPK.dotabb** —

```
"SFQCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFQCMPK"]
```

———-

— **package SFQCMPK SquareFreeQuasiComponentPackage** —

```
)abbrev package SFQCMPK SquareFreeQuasiComponentPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] D. LAZARD "A new method for solving algebraic systems of
++      positive dimension" Discr. App. Math. 33:147-160,1991
++      Tech. Report (PoSSo project)
++  [2] M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++      d'extensions simples et resolution des systemes d'equations
++      algebriques" These, Universite P.etM. Curie, Paris, 1997.
++  [3] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A internal package for removing redundant quasi-components and redundant
++ branches when decomposing a variety by means of quasi-components
++ of regular triangular sets.

SquareFreeQuasiComponentPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
```

```
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
Key ==>  Record(left:TS, right:TS)
Entry ==> Boolean
H ==> TabulatedComputationPackage(Key, Entry)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)

Exports ==  with
   startTable!: (S,S,S) -> Void
    ++ \axiom{startTableGcd!(s1,s2,s3)}
    ++ is an internal subroutine, exported only for developement.
   stopTable!: () -> Void
    ++ \axiom{stopTableGcd!()}
    ++ is an internal subroutine, exported only for developement.
   supDimElseRittWu?: (TS,TS) -> Boolean
    ++ \axiom{supDimElseRittWu(ts,us)} returns true iff \axiom{ts}
    ++ has less elements than \axiom{us} otherwise if \axiom{ts}
    ++ has higher rank than \axiom{us} w.r.t. Riit and Wu ordering.
   algebraicSort: Split -> Split
    ++ \axiom{algebraicSort(lts)} sorts \axiom{lts} w.r.t
    ++ supDimElseRittWu from QuasiComponentPackage.
   moreAlgebraic?: (TS,TS) -> Boolean
    ++ \axiom{moreAlgebraic?(ts,us)} returns false iff \axiom{ts}
    ++ and \axiom{us} are both empty, or \axiom{ts}
    ++ has less elements than \axiom{us}, or some variable is
    ++ algebraic w.r.t. \axiom{us} and is not w.r.t. \axiom{ts}.
   subTriSet?: (TS,TS) -> Boolean
    ++ \axiom{subTriSet?(ts,us)} returns true iff \axiom{ts} is
    ++ a sub-set of \axiom{us}.
   subPolSet?: (LP, LP) -> Boolean
    ++ \axiom{subPolSet?(lp1,lp2)} returns true iff \axiom{lp1} is
    ++ a sub-set of \axiom{lp2}.
   internalSubPolSet?: (LP, LP) -> Boolean
    ++ \axiom{internalSubPolSet?(lp1,lp2)} returns true iff \axiom{lp1} is
    ++ a sub-set of \axiom{lp2} assuming that these lists are sorted
    ++ increasingly w.r.t.
    ++ infRittWu? from RecursivePolynomialCategory.
```

```
    internalInfRittWu?: (LP, LP) -> Boolean
     ++ \axiom{internalInfRittWu?(lp1,lp2)}
     ++ is an internal subroutine, exported only for developement.
    infRittWu?: (LP, LP) -> Boolean
     ++ \axiom{infRittWu?(lp1,lp2)}
     ++ is an internal subroutine, exported only for developement.
    internalSubQuasiComponent?: (TS,TS) -> Union(Boolean,"failed")
     ++ \axiom{internalSubQuasiComponent?(ts,us)} returns a boolean
     ++ \spad{b} value if the fact the regular zero set of \axiom{us}
     ++ contains that of \axiom{ts} can be decided (and in that case
     ++ \axiom{b} gives this inclusion) otherwise returns \axiom{"failed"}.
    subQuasiComponent?: (TS,TS) -> Boolean
     ++ \axiom{subQuasiComponent?(ts,us)} returns true iff
     ++ internalSubQuasiComponent?(ts,us) from QuasiComponentPackage
     ++ returns true.
    subQuasiComponent?: (TS,Split) -> Boolean
     ++ \axiom{subQuasiComponent?(ts,lus)} returns true iff
     ++ \axiom{subQuasiComponent?(ts,us)} holds for one
     ++ \spad{us} in \spad{lus}.
    removeSuperfluousQuasiComponents: Split -> Split
     ++ \axiom{removeSuperfluousQuasiComponents(lts)} removes from
     ++ \axiom{lts} any \spad{ts} such that
     ++ \axiom{subQuasiComponent?(ts,us)} holds for
     ++ another \spad{us} in \axiom{lts}.
    subCase?: (LpWT,LpWT) -> Boolean
     ++ \axiom{subCase?(lpwt1,lpwt2)}
     ++ is an internal subroutine, exported only for developement.
    removeSuperfluousCases: List LpWT -> List LpWT
     ++ \axiom{removeSuperfluousCases(llpwt)}
     ++ is an internal subroutine, exported only for developement.
    prepareDecompose: (LP, List(TS),B,B) -> List Branch
     ++ \axiom{prepareDecompose(lp,lts,b1,b2)}
     ++ is an internal subroutine, exported only for developement.
    branchIfCan: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
     ++ \axiom{branchIfCan(leq,ts,lineq,b1,b2,b3,b4,b5)}
     ++ is an internal subroutine, exported only for developement.

  Implementation == add

    squareFreeFactors(lp: LP): LP ==
      lsflp: LP := []
      for p in lp repeat
        lsfp := squareFreeFactors(p)$polsetpack
        lsflp := concat(lsfp,lsflp)
      sort(infRittWu?,removeDuplicates lsflp)

    startTable!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$H
      if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
      if (not empty? domainName) then startStats!(domainName)$H
```

```
  void()

stopTable!(): Void ==
  if makingStats?()$H then printStats!()$H
  clearTable!()$H

supDimElseRittWu? (ts:TS,us:TS): Boolean ==
  #ts < #us => true
  #ts > #us => false
  lp1 :LP := members(ts)
  lp2 :LP := members(us)
  while (not empty? lp1) and (not infRittWu?(first(lp2),first(lp1)))
   repeat
    lp1 := rest lp1
    lp2 := rest lp2
  not empty? lp1

algebraicSort (lts:Split): Split ==
  lts := removeDuplicates lts
  sort(supDimElseRittWu?,lts)

moreAlgebraic?(ts:TS,us:TS): Boolean  ==
  empty? ts => empty? us
  empty? us => true
  #ts < #us => false
  for p in (members us) repeat
     not algebraic?(mvar(p),ts) => return false
  true

subTriSet?(ts:TS,us:TS): Boolean  ==
  empty? ts => true
  empty? us => false
  mvar(ts) > mvar(us) => false
  mvar(ts) < mvar(us) => subTriSet?(ts,rest(us)::TS)
  first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS,rest(us)::TS)
  false

internalSubPolSet?(lp1: LP, lp2: LP): Boolean  ==
  empty? lp1 => true
  empty? lp2 => false
  associates?(first lp1, first lp2) =>
    internalSubPolSet?(rest lp1, rest lp2)
  infRittWu?(first lp1, first lp2) => false
  internalSubPolSet?(lp1, rest lp2)

subPolSet?(lp1: LP, lp2: LP): Boolean  ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1,lp2)
```

```
infRittWu?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1,lp2)

internalInfRittWu?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?(first lp1, first lp2)$P => true
  infRittWu?(first lp2, first lp1)$P => false
  infRittWu?(rest lp1, rest lp2)$$

subCase? (lpwt1:LpWT,lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false
  subQuasiComponent?(lpwt1.tower,lpwt2.tower)

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P)
then

  internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
    subTriSet?(us,ts) => true
    not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
    for p in (members us) repeat
      mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
        return("failed"::Union(Boolean,"failed"))
    for p in (members us) repeat
      not zero? initiallyReduce(p,ts) =>
        return("failed"::Union(Boolean,"failed"))
    lsfp := squareFreeFactors(initials us)
    for p in lsfp repeat
      b: B := invertible?(p,ts)$TS
      not b =>
        return(false::Union(Boolean,"failed"))
    true::Union(Boolean,"failed")

else

  internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
    subTriSet?(us,ts) => true
    not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
    for p in (members us) repeat
      mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
        return("failed"::Union(Boolean,"failed"))
    for p in (members us) repeat
      not zero? reduceByQuasiMonic(p,ts) =>
        return("failed"::Union(Boolean,"failed"))
    true::Union(Boolean,"failed")

subQuasiComponent?(ts:TS,us:TS): Boolean ==
```

```
    k: Key := [ts, us]
    e := extractIfCan(k)$H
    e case Entry => e::Entry
    ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts,us)
    b: Boolean := (ubf case Boolean) and (ubf::Boolean)
    insert!(k,b)$H
    b

  subQuasiComponent?(ts:TS,lus:Split): Boolean ==
    for us in lus repeat
       subQuasiComponent?(ts,us)@B => return true
    false

  removeSuperfluousCases (cases:List LpWT) ==
    #cases < 2 => cases
    toSee :=
      sort((x:LpWT,y:LpWT):Boolean +-> supDimElseRittWu?(x.tower,y.tower),
           cases)
    lpwt1,lpwt2 : LpWT
    toSave,headmaxcases,maxcases,copymaxcases : List LpWT
    while not empty? toSee repeat
      lpwt1 := first toSee
      toSee := rest toSee
      toSave := []
      for lpwt2 in toSee repeat
         if subCase?(lpwt1,lpwt2)
           then
             lpwt1 := lpwt2
           else
             if not subCase?(lpwt2,lpwt1)
               then
                 toSave := cons(lpwt2,toSave)
      if empty? maxcases
        then
          headmaxcases := [lpwt1]
          maxcases := headmaxcases
        else
          copymaxcases := maxcases
          while (not empty? copymaxcases) and _
            (not subCase?(lpwt1,first(copymaxcases))) repeat
              copymaxcases := rest copymaxcases
          if empty? copymaxcases
            then
              setrest!(headmaxcases,[lpwt1])
              headmaxcases := rest headmaxcases
      toSee := reverse toSave
    maxcases

  removeSuperfluousQuasiComponents(lts: Split): Split ==
    lts := removeDuplicates lts
```

```
    #lts < 2 => lts
    toSee := algebraicSort lts
    toSave,headmaxlts,maxlts,copymaxlts : Split
    while not empty? toSee repeat
      ts := first toSee
      toSee := rest toSee
      toSave := []
      for us in toSee repeat
         if subQuasiComponent?(ts,us)@B
           then
             ts := us
           else
             if not subQuasiComponent?(us,ts)@B
               then
                 toSave := cons(us,toSave)
      if empty? maxlts
        then
          headmaxlts := [ts]
          maxlts := headmaxlts
        else
          copymaxlts := maxlts
          while (not empty? copymaxlts) and _
            (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
              copymaxlts := rest copymaxlts
          if empty? copymaxlts
            then
              setrest!(headmaxlts,[ts])
              headmaxlts := rest headmaxlts
      toSee := reverse toSave
    algebraicSort maxlts

  removeAssociates (lp:LP):LP ==
    removeDuplicates [primitivePart(p) for p in lp]

  branchIfCan(leq: LP,ts: TS,lineq: LP, b1:B,b2:B,b3:B,b4:B,b5:B):UBF ==
     -- ASSUME pols in leq are squarefree and mainly primitive
     -- if b1 then CLEAN UP leq
     -- if b2 then CLEAN UP lineq
     -- if b3 then SEARCH for ZERO in lineq with leq
     -- if b4 then SEARCH for ZERO in lineq with ts
     -- if b5 then SEARCH for ONE in leq with lineq
     if b1
       then
         leq := removeAssociates(leq)
         leq := remove(zero?,leq)
         any?(ground?,leq) =>
           return("failed"::Union(Branch,"failed"))
     if b2
       then
         any?(zero?,lineq) =>
```

```
            return("failed"::Union(Branch,"failed"))
          lineq := removeRedundantFactors(lineq)$polsetpack
      if b3
        then
          ps: PS := construct(leq)$PS
          for q in lineq repeat
            zero? remainder(q,ps).polnum =>
              return("failed"::Union(Branch,"failed"))
      (empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch)::UBF
      if b4
        then
          for q in lineq repeat
            zero? initiallyReduce(q,ts) =>
              return("failed"::Union(Branch,"failed"))
      if b5
        then
          newleq: LP := []
          for p in leq repeat
            for q in lineq repeat
              if mvar(p) = mvar(q)
                then
                  g := gcd(p,q)
                  newp := (p exquo g)::P
                  ground? newp =>
                    return("failed"::Union(Branch,"failed"))
                  newleq := cons(newp,newleq)
                else
                  newleq := cons(p,newleq)
          leq := newleq
      leq := sort(infRittWu?, removeDuplicates leq)
      ([leq, ts, lineq]$Branch)::UBF

    prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
      -- if b1 then REMOVE REDUNDANT COMPONENTS in lts
      -- if b2 then SPLIT the input system with squareFree
      lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))
      any?(ground?,lp) => []
      empty? lts => []
      if b1 then lts := removeSuperfluousQuasiComponents lts
      not b2 =>
        [[lp,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
      toSee: List Branch
      lq: LP := []
      toSee := [[lq,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
      empty? lp => toSee
      for p in lp repeat
        lsfp := squareFreeFactors(p)$polsetpack
        branches: List Branch := []
        lq := []
        for f in lsfp repeat
```

```
        for branch in toSee repeat
          leq : LP := branch.eq
          ts := branch.tower
          lineq : LP := branch.ineq
          ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true)@UBF
          ubf1 case "failed" => "leave"
          ubf2: UBF :=
            branchIfCan([f],ts,lineq,false,false,true,true,true)@UBF
          ubf2 case "failed" => "leave"
          leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))
          lineq :=
            sort(infRittWu?,removeDuplicates concat(ubf1.ineq,ubf2.ineq))
          newBranch :=
            branchIfCan(leq,ts,lineq,false,false,false,false,false)
          branches:= cons(newBranch::Branch,branches)
        lq := cons(f,lq)
      toSee := branches
    sort((x,y) +-> supDimElseRittWu?(x.tower,y.tower),toSee)
```

———-

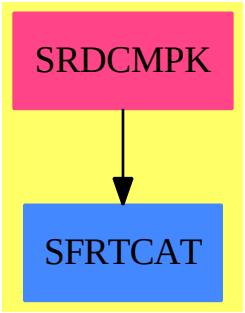
— **SFQCMPK.dotabb** —

```
"SFQCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFQCMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SFQCMPK" -> "SFRTCAT"
```

———-

20.13 package SRDCMPK SquareFreeRegularSetDecompositionPackage

20.13.1 SquareFreeRegularSetDecompositionPackage (SRDCMPK)



Exports:

algebraicDecompose	convert	decompose
internalDecompose	KrullNumber	numberOfVariables
printInfo	transcendentalDecompose	upDateBranches

— **package SRDCMPK SquareFreeRegularSetDecompositionPackage** —

```
)abbrev package SRDCMPK SquareFreeRegularSetDecompositionPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] M. MORENO MAZA "A new algorithm for computing triangular
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A package providing a new algorithm for solving polynomial systems
++ by means of regular chains. Two ways of solving are provided:
++ in the sense of Zariski closure (like in Kalkbrener's algorithm)
++ or in the sense of the regular zeros (like in Wu, Wang or Lazard-
++ Moreno methods). This algorithm is valid for nay type
++ of regular set. It does not care about the way a polynomial is
++ added in an regular set, or how two quasi-components are compared
++ (by an inclusion-test), or how the invertibility test is made in
++ the tower of simple extensions associated with a regular set.
++ These operations are realized respectively by the domain \spad{TS}
++ and the packages \spad{QCMPPK(R,E,V,P,TS)} and \spad{RSETGCD(R,E,V,P,TS)}.
++ The same way it does not care about the way univariate polynomial
++ gcds (with coefficients in the tower of simple extensions associated
++ with a regular set) are computed. The only requirement is that these
++ gcds need to have invertible initials (normalized or not).
++ WARNING. There is no need for a user to call diectly any operation
++ of this package since they can be accessed by the domain \axiomType{TS}.
++ Thus, the operations of this package are not documented.

SquareFreeRegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
```

```
Z ==> Integer
B ==> Boolean
LP ==> List P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Wip ==> Record(done: Split, todo: List LpWT)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports ==  with

   KrullNumber: (LP, Split) -> N
   numberOfVariables: (LP, Split) -> N
   algebraicDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
   transcendentalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
   transcendentalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
   internalDecompose: (P,TS,N,B) -> Record(done: Split, todo: List LpWT)
   internalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
   internalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
   decompose: (LP, Split, B, B) -> Split
   decompose: (LP, Split, B, B, B, B, B) -> Split
   upDateBranches: (LP,Split,List LpWT,Wip,N) -> List LpWT
   convert: Record(val: List P,tower: TS) -> String
   printInfo: (List Record(val: List P,tower: TS), N) -> Void

Implementation == add

   KrullNumber(lp: LP, lts: Split): N ==
     ln: List N := [#(ts) for ts in lts]
     n := #lp + reduce(max,ln)

   numberOfVariables(lp: LP, lts: Split): N ==
     lv: List V := variables([lp]$PS)
     for ts in lts repeat lv := concat(variables(ts), lv)
     # removeDuplicates(lv)

   algebraicDecompose(p: P, ts: TS): Record(done: Split, todo: List LpWT) ==
     ground? p =>
       error " in algebraicDecompose$REGSET: should never happen !"
     v := mvar(p); n := #ts
     ts_v_- := collectUnder(ts,v)
     ts_v_+ := collectUpper(ts,v)
     ts_v := select(ts,v)::P
```

```
    lgwt: List PWT
    if mdeg(p) < mdeg(ts_v)
      then
        lgwt := stoseInternalLastSubResultant(ts_v,p,ts_v_-,true,false)$regsetgcdpack
      else
        lgwt := stoseInternalLastSubResultant(p,ts_v,ts_v_-,true,false)$regsetgcdpack
    lts: Split := []
    llpwt: List LpWT := []
    for gwt in lgwt repeat
      g := gwt.val; us := gwt.tower
      zero? g =>
        error " in algebraicDecompose$REGSET: should never happen !!"
      ground? g => "leave"
      h := leadingCoefficient(g,v)
      lus := augment(members(ts_v_+),augment(ts_v,us)$TS)$TS
      lsfp := squareFreeFactors(h)$polsetpack
      for f in lsfp repeat
        ground? f => "leave"
        for vs in lus repeat
          llpwt := cons([[f,p],vs]$LpWT, llpwt)
      n < #us =>
        error " in algebraicDecompose$REGSET: should never happen !!!"
      mvar(g) = v =>
        lts := concat(augment(members(ts_v_+),augment(g,us)$TS)$TS,lts)
    [lts,llpwt]

  transcendentalDecompose(p: P, ts: TS,bound: N): Record(done: Split, todo: List LpWT) ==
    lts: Split
    if #ts < bound
      then
        lts := augment(p,ts)$TS
      else
        lts := []
    llpwt: List LpWT := []
    [lts,llpwt]

  transcendentalDecompose(p: P, ts: TS): Record(done: Split, todo: List LpWT) ==
    lts: Split:= augment(p,ts)$TS
    llpwt: List LpWT := []
    [lts,llpwt]

  internalDecompose(p: P, ts: TS,bound: N,clos?:B): Record(done: Split, todo: List LpWT) ==
    clos? => internalDecompose(p,ts,bound)
    internalDecompose(p,ts)

  internalDecompose(p: P, ts: TS,bound: N): Record(done: Split, todo: List LpWT) ==
    -- ASSUME p not constant
    llpwt: List LpWT := []
    lts: Split := []
    -- EITHER mvar(p) is null
```

```
    if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
      then
        llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
        p := (p exquo lmp)::P
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
    -- OR init(p) is null or not
    lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
    for bwt in lbwt repeat
      bwt.val =>
        if algebraic?(mvar(p),bwt.tower)
          then
            rsl := algebraicDecompose(p,bwt.tower)
          else
            rsl := transcendentalDecompose(p,bwt.tower,bound)
        lts := concat(rsl.done,lts)
        llpwt :=  concat(rsl.todo,llpwt)
        (not ground? ip) =>
          zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
          (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
      riv := removeZero(ip,bwt.tower)
      (zero? riv) =>
        zero? tp => lts := cons(bwt.tower,lts)
        (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
      llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
    [lts,llpwt]

  internalDecompose(p: P, ts: TS): Record(done: Split, todo: List LpWT) ==
    -- ASSUME p not constant
    llpwt: List LpWT := []
    lts: Split := []
    -- EITHER mvar(p) is null
    if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
      then
        llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
        p := (p exquo lmp)::P
    ip := squareFreePart init(p); tp := tail p
    p := mainPrimitivePart p
    -- OR init(p) is null or not
    lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
    for bwt in lbwt repeat
      bwt.val =>
        if algebraic?(mvar(p),bwt.tower)
          then
            rsl := algebraicDecompose(p,bwt.tower)
          else
            rsl := transcendentalDecompose(p,bwt.tower)
        lts := concat(rsl.done,lts)
        llpwt :=  concat(rsl.todo,llpwt)
        (not ground? ip) =>
```

```
          zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
          (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
      riv := removeZero(ip,bwt.tower)
      (zero? riv) =>
        zero? tp => lts := cons(bwt.tower,lts)
        (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
      llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
    [lts,llpwt]

  decompose(lp: LP, lts: Split, clos?: B, info?: B): Split ==
    decompose(lp,lts,false,false,clos?,true,info?)

  convert(lpwt: LpWT): String ==
    ls: List String := ["<", string((#(lpwt.val))::Z), ",", string((#(lpwt.tower))::Z), ">" ]
    concat ls

  printInfo(toSee: List LpWT, n: N): Void ==
    lpwt := first toSee
    s: String := concat ["[", string((#toSee)::Z), " ", convert(lpwt)@String]
    m: N := #(lpwt.val)
    toSee := rest toSee
    for lpwt in toSee repeat
      m := m + #(lpwt.val)
      s := concat [s, ",", convert(lpwt)@String]
    s := concat [s, " -> |", string(m::Z), "|; {", string(n::Z),"}]"]
    iprint(s)$iprintpack
    void()

  decompose(lp: LP, lts: Split, cleanW?: B, sqfr?: B, clos?: B, rem?: B, info?: B): Split ==
    -- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
    -- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
    -- if clos? then SOLVE in the closure sense
    -- if rem? then REDUCE the current p by using remainder
    -- if info? then PRINT info
    empty? lp => lts
    branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$quasicomppack
    empty? branches => []
    toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
    toSave: Split := []
    if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
    while (not empty? toSee) repeat
      if info? then printInfo(toSee,#toSave)
      lpwt := first toSee; toSee := rest toSee
      lp := lpwt.val; ts := lpwt.tower
      empty? lp =>
        toSave := cons(ts, toSave)
      p := first lp;  lp := rest lp
      if rem? and (not ground? p) and (not empty? ts)
         then
           p := remainder(p,ts).polnum
```

```
      p := removeZero(p,ts)
      zero? p => toSee := cons([lp,ts]$LpWT, toSee)
      ground? p => "leave"
      rsl := internalDecompose(p,ts,bound,clos?)
      toSee := upDateBranches(lp,toSave,toSee,rsl,bound)
    removeSuperfluousQuasiComponents(toSave)$quasicomppack

  upDateBranches(leq:LP,lts:Split,current:List LpWT,wip: Wip,n:N): List LpWT ==
    newBranches: List LpWT := wip.todo
    newComponents: Split := wip.done
    branches1, branches2:  List LpWT
    branches1 := []; branches2  := []
    for branch in newBranches repeat
      us := branch.tower
      #us > n => "leave"
      newleq := sort(infRittWu?,concat(leq,branch.val))
      --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
      --any?(ground?,foo)  => "leave"
      branches1 := cons([newleq,us]$LpWT, branches1)
    for us in newComponents repeat
      #us > n => "leave"
      subQuasiComponent?(us,lts)$quasicomppack => "leave"
      --newleq := leq
      --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
      --any?(ground?,foo)  => "leave"
      branches2 := cons([leq,us]$LpWT, branches2)
    empty? branches1 =>
      empty? branches2 => current
      concat(branches2, current)
    branches := concat [branches2, branches1, current]
    -- branches := concat(branches,current)
    removeSuperfluousCases(branches)$quasicomppack
```

———-

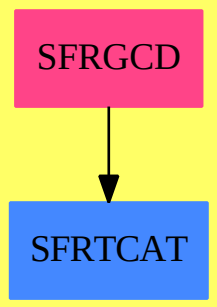
— **SRDCMPK.dotabb** —

```
"SRDCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SRDCMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SRDCMPK" -> "SFRTCAT"
```

———-

20.14 package SFRGCD SquareFreeRegularTriangularSetGcdPackage

20.14.1 SquareFreeRegularTriangularSetGcdPackage (SFRGCD)



Exports:

stopTableGcd!	stopTableInvSet!
startTableGcd!	startTableInvSet!
stoseIntegralLastSubResultant	stoseInternalLastSubResultant
stoseInvertible?	stoseInvertible?reg
stoseInvertible?sqfreg	stoseInvertibleSet
stoseInvertibleSetreg	stoseInvertibleSetsqfreg
stoseLastSubResultant	stosePrepareSubResAlgo
stoseSquareFreePart	

— **package SFRGCD SquareFreeRegularTriangularSetGcdPackage** —

```
)abbrev package SFRGCD SquareFreeRegularTriangularSetGcdPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 10/01/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References :
++  [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++      algebraic towers of simple extensions" In proceedings of AAECC11
++      Paris, 1995.
++  [2] M. MORENO MAZA "Calculs de pgcd au-dessus des tours
++      d'extensions simples et resolution des systemes d'equations
++      algebriques" These, Universite P.etM. Curie, Paris, 1997.
++  [3] M. MORENO MAZA "A new algorithm for computing triangular
```

```
++      decomposition of algebraic varieties" NAG Tech. Rep. 4/98.
++ Description:
++ A internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.
++ There is no need to use directly this package since its main operations are
++ available from \spad{TS}.

SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
  TS : RegularTriangularSetCategory(R,E,V,P)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  S ==> String
  LP ==> List P
  PtoP ==> P -> P
  PS ==> GeneralPolynomialSet(R,E,V,P)
  PWT ==> Record(val : P, tower : TS)
  BWT ==> Record(val : Boolean, tower : TS)
  LpWT ==> Record(val : (List P), tower : TS)
  Branch ==> Record(eq: List P, tower: TS, ineq: List P)
  UBF ==> Union(Branch,"failed")
  Split ==> List TS
  KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
  EntryGcd ==> List PWT
  HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
  KeyInvSet ==> Record(arg1: P, arg3: TS)
  EntryInvSet ==> List TS
  HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
  iprintpack ==> InternalPrintPackage()
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
  quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)

  SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)

  Exports ==  with
     startTableGcd!: (S,S,S) -> Void
     stopTableGcd!: () -> Void
     startTableInvSet!: (S,S,S) -> Void
     stopTableInvSet!: () -> Void
     stosePrepareSubResAlgo: (P,P,TS) -> List LpWT
     stoseInternalLastSubResultant: (P,P,TS,B,B) -> List PWT
     stoseInternalLastSubResultant: (List LpWT,V,B) -> List PWT
     stoseIntegralLastSubResultant: (P,P,TS) -> List PWT
     stoseLastSubResultant: (P,P,TS) -> List PWT
     stoseInvertible?: (P,TS) -> B
```

```
    stoseInvertible?_sqfreg: (P,TS) -> List BWT
    stoseInvertibleSet_sqfreg: (P,TS) -> Split
    stoseInvertible?_reg: (P,TS) -> List BWT
    stoseInvertibleSet_reg: (P,TS) -> Split
    stoseInvertible?: (P,TS) -> List BWT
    stoseInvertibleSet: (P,TS) -> Split
    stoseSquareFreePart: (P,TS) -> List PWT

  Implementation == add

    startTableGcd!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$HGcd
      printInfo!(ok,ko)$HGcd
      startStats!(domainName)$HGcd
      void()

    stopTableGcd!(): Void ==
      if makingStats?()$HGcd then printStats!()$HGcd
      clearTable!()$HGcd

    startTableInvSet!(ok: S, ko: S, domainName: S): Void ==
      initTable!()$HInvSet
      printInfo!(ok,ko)$HInvSet
      startStats!(domainName)$HInvSet
      void()

    stopTableInvSet!(): Void ==
      if makingStats?()$HInvSet then printStats!()$HInvSet
      clearTable!()$HInvSet

    stoseInvertible?(p:P,ts:TS): Boolean ==
      q := primitivePart initiallyReduce(p,ts)
      zero? q => false
      normalized?(q,ts) => true
      v := mvar(q)
      not algebraic?(v,ts) =>
        toCheck: List BWT := stoseInvertible?(p,ts)@(List BWT)
        for bwt in toCheck repeat
          bwt.val = false => return false
        return true
      ts_v := select(ts,v)::P
      ts_v_- := collectUnder(ts,v)
      lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,true)
      for gwt in lgwt repeat
        g := gwt.val;
        (not ground? g) and (mvar(g) = v) =>
          return false
      true

    stosePrepareSubResAlgo(p1:P,p2:P,ts:TS): List LpWT ==
```

```
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME init(p1) invertible modulo ts !!!
  toSee: List LpWT := [[[p1,p2],ts]$LpWT]
  toSave: List LpWT := []
  v := mvar(p1)
  while (not empty? toSee) repeat
    lpwt := first toSee; toSee := rest toSee
    p1 := lpwt.val.1; p2 := lpwt.val.2
    ts := lpwt.tower
    lbwt := stoseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
    for bwt in lbwt repeat
      (bwt.val = true) and (degree(p2,v) > 0) =>
        p3 := prem(p1, -p2)
        s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
        toSave := cons([[p2,p3,s],bwt.tower]$LpWT,toSave)
      -- p2 := initiallyReduce(p2,bwt.tower)
      newp2 := primitivePart initiallyReduce(p2,bwt.tower)
      (bwt.val = true) =>
        -- toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
        toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
      -- zero? p2 =>
      zero? newp2 =>
        toSave := cons([[p1,0,1],bwt.tower]$LpWT,toSave)
      -- toSee := cons([[p1,p2],bwt.tower]$LpWT,toSee)
      toSee := cons([[p1,newp2],bwt.tower]$LpWT,toSee)
  toSave

stoseIntegralLastSubResultant(p1:P,p2:P,ts:TS): List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME p1 and p2 have no algebraic coefficients
  lsr := lastSubResultant(p1, p2)
  ground?(lsr) => [[lsr,ts]$PWT]
  mvar(lsr) < mvar(p1) => [[lsr,ts]$PWT]
  gi1i2 := gcd(init(p1),init(p2))
  ex: Union(P,"failed") := (gi1i2 * lsr) exquo$P init(lsr)
  ex case "failed" => [[lsr,ts]$PWT]
  [[ex::P,ts]$PWT]

stoseInternalLastSubResultant(p1:P,p2:P,ts:TS,b1:B,b2:B): List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- if b1 ASSUME init(p2) invertible w.r.t. ts
  -- if b2 BREAK with the first non-trivial gcd
  k: KeyGcd := [p1,p2,ts,b2]
  e := extractIfCan(k)$HGcd
  e case EntryGcd => e::EntryGcd
  toSave: List PWT
  empty? ts =>
    toSave := stoseIntegralLastSubResultant(p1,p2,ts)
    insert!(k,toSave)$HGcd
    return toSave
```

```
    toSee: List LpWT
    if b1
      then
        p3 := prem(p1, -p2)
        s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
        toSee := [[[p2,p3,s],ts]$LpWT]
      else
        toSee := stosePrepareSubResAlgo(p1,p2,ts)
    toSave := stoseInternalLastSubResultant(toSee,mvar(p1),b2)
    insert!(k,toSave)$HGcd
    toSave

  stoseInternalLastSubResultant(llpwt: List LpWT,v:V,b2:B): List PWT ==
    toReturn: List PWT := []; toSee: List LpWT;
    while (not empty? llpwt) repeat
      toSee := llpwt; llpwt := []
      -- CONSIDER FIRST the vanishing current last subresultant
      for lpwt in toSee repeat
        p1 := lpwt.val.1;
        p2 := lpwt.val.2;
        s := lpwt.val.3;
        ts := lpwt.tower
        lbwt := stoseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
        for bwt in lbwt repeat
          bwt.val = false =>
            toReturn := cons([p1,bwt.tower]$PWT, toReturn)
            b2 and positive?(degree(p1,v)) => return toReturn
          llpwt := cons([[p1,p2,s],bwt.tower]$LpWT, llpwt)
      empty? llpwt => "leave"
      -- CONSIDER NOW the branches where the computations continue
      toSee := llpwt; llpwt := []
      lpwt := first toSee; toSee := rest toSee
      p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
      delta: N := (mdeg(p1) - degree(p2,v))::N
      p3: P := LazardQuotient2(p2, leadingCoefficient(p2,v), s, delta)
      zero?(degree(p3,v)) =>
        toReturn := cons([p3,lpwt.tower]$PWT, toReturn)
        for lpwt in toSee repeat
          toReturn := cons([p3,lpwt.tower]$PWT, toReturn)
      (p1, p2) := (p3, next_subResultant2(p1, p2, p3, s))
      s := leadingCoefficient(p1,v)
      llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
      for lpwt in toSee repeat
        llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)
    toReturn

  stoseLastSubResultant(p1:P,p2:P,ts:TS): List PWT ==
    ground? p1 =>
      error"in stoseLastSubResultantElseSplit$SFRGCD  : bad #1"
    ground? p2 =>
```

```
        error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"
      not (mvar(p2) = mvar(p1)) =>
        error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"
      algebraic?(mvar(p1),ts) =>
        error"in stoseLastSubResultantElseSplit$SFRGCD : bad #1"
      not initiallyReduced?(p1,ts) =>
        error"in stoseLastSubResultantElseSplit$SFRGCD : bad #1"
      not initiallyReduced?(p2,ts) =>
        error"in stoseLastSubResultantElseSplit$SFRGCD : bad #2"
      purelyTranscendental?(p1,ts) and purelyTranscendental?(p2,ts) =>
        stoseIntegralLastSubResultant(p1,p2,ts)
      if mdeg(p1) < mdeg(p2) then
         (p1, p2) := (p2, p1)
         if odd?(mdeg(p1)) and odd?(mdeg(p2)) then p2 := - p2
      stoseInternalLastSubResultant(p1,p2,ts,false,false)

    stoseSquareFreePart_wip(p:P, ts: TS): List PWT ==
    -- ASSUME p is not constant and mvar(p) > mvar(ts)
    -- ASSUME init(p) is invertible w.r.t. ts
    -- ASSUME p is mainly primitive
--      one? mdeg(p) => [[p,ts]$PWT]
      mdeg(p) = 1 => [[p,ts]$PWT]
      v := mvar(p)$P
      q: P := mainPrimitivePart D(p,v)
      lgwt: List PWT := stoseInternalLastSubResultant(p,q,ts,true,false)
      lpwt : List PWT := []
      sfp : P
      for gwt in lgwt repeat
        g := gwt.val; us := gwt.tower
        (ground? g) or (mvar(g) < v) =>
          lpwt := cons([p,us],lpwt)
        g := mainPrimitivePart g
        sfp := lazyPquo(p,g)
        sfp := mainPrimitivePart stronglyReduce(sfp,us)
        lpwt := cons([sfp,us],lpwt)
      lpwt

    stoseSquareFreePart_base(p:P, ts: TS): List PWT == [[p,ts]$PWT]

    stoseSquareFreePart(p:P, ts:TS): List PWT == stoseSquareFreePart_wip(p,ts)

    stoseInvertible?_sqfreg(p:P,ts:TS): List BWT ==
      --iprint("+")$iprintpack
      q := primitivePart initiallyReduce(p,ts)
      zero? q => [[false,ts]$BWT]
      normalized?(q,ts) => [[true,ts]$BWT]
      v := mvar(q)
      not algebraic?(v,ts) =>
        lbwt: List BWT := []
        toCheck: List BWT := stoseInvertible?_sqfreg(init(q),ts)@(List BWT)
```

```
      for bwt in toCheck repeat
        bwt.val => lbwt := cons(bwt,lbwt)
        newq := removeZero(q,bwt.tower)
        zero? newq => lbwt := cons(bwt,lbwt)
        lbwt :=
         concat(stoseInvertible?_sqfreg(newq,bwt.tower)@(List BWT), lbwt)
      return lbwt
    ts_v := select(ts,v)::P
    ts_v_- := collectUnder(ts,v)
    ts_v_+ := collectUpper(ts,v)
    lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
    lbwt: List BWT := []
    lts, lts_g, lts_h: Split
    for gwt in lgwt repeat
      g := gwt.val; ts := gwt.tower
      (ground? g) or (mvar(g) < v) =>
        lts := augment(ts_v,ts)$TS
        lts := augment(members(ts_v_+),lts)$TS
        for ts in lts repeat
          lbwt := cons([true, ts]$BWT,lbwt)
      g := mainPrimitivePart g
      lts_g := augment(g,ts)$TS
      lts_g := augment(members(ts_v_+),lts_g)$TS
      -- USE stoseInternalAugment with parameters ??
      for ts_g in lts_g repeat
        lbwt := cons([false, ts_g]$BWT,lbwt)
      h := lazyPquo(ts_v,g)
      (ground? h) or (mvar(h) < v) => "leave"
      h := mainPrimitivePart h
      lts_h := augment(h,ts)$TS
      lts_h := augment(members(ts_v_+),lts_h)$TS
      -- USE stoseInternalAugment with parameters ??
      for ts_h in lts_h repeat
        lbwt := cons([true, ts_h]$BWT,lbwt)
    sort((x,y) +-> x.val < y.val,lbwt)

  stoseInvertibleSet_sqfreg(p:P,ts:TS): Split ==
    --iprint("*")$iprintpack
    k: KeyInvSet := [p,ts]
    e := extractIfCan(k)$HInvSet
    e case EntryInvSet => e::EntryInvSet
    q := primitivePart initiallyReduce(p,ts)
    zero? q => []
    normalized?(q,ts) => [ts]
    v := mvar(q)
    toSave: Split := []
    not algebraic?(v,ts) =>
      toCheck: List BWT := stoseInvertible?_sqfreg(init(q),ts)@(List BWT)
      for bwt in toCheck repeat
        bwt.val => toSave := cons(bwt.tower,toSave)
```

```
      newq := removeZero(q,bwt.tower)
      zero? newq => "leave"
      toSave := concat(stoseInvertibleSet_sqfreg(newq,bwt.tower), toSave)
    toSave := removeDuplicates toSave
    return algebraicSort(toSave)$quasicomppack
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
  lts, lts_h: Split
  for gwt in lgwt repeat
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
      lts := augment(ts_v,ts)$TS
      lts := augment(members(ts_v_+),lts)$TS
      toSave := concat(lts,toSave)
    g := mainPrimitivePart g
    h := lazyPquo(ts_v,g)
    h := mainPrimitivePart h
    (ground? h) or (mvar(h) < v) => "leave"
    lts_h := augment(h,ts)$TS
    lts_h := augment(members(ts_v_+),lts_h)$TS
    toSave := concat(lts_h,toSave)
  toSave := algebraicSort(toSave)$quasicomppack
  insert!(k,toSave)$HInvSet
  toSave

stoseInvertible?_reg(p:P,ts:TS): List BWT ==
  --iprint("-")$iprintpack
  q := primitivePart initiallyReduce(p,ts)
  zero? q => [[false,ts]$BWT]
  normalized?(q,ts) => [[true,ts]$BWT]
  v := mvar(q)
  not algebraic?(v,ts) =>
    lbwt: List BWT := []
    toCheck: List BWT := stoseInvertible?_reg(init(q),ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val => lbwt := cons(bwt,lbwt)
      newq := removeZero(q,bwt.tower)
      zero? newq => lbwt := cons(bwt,lbwt)
      lbwt :=
       concat(stoseInvertible?_reg(newq,bwt.tower)@(List BWT), lbwt)
    return lbwt
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
  lbwt: List BWT := []
  lts, lts_g, lts_h: Split
  for gwt in lgwt repeat
```

```
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
      lts := augment(ts_v,ts)$TS
      lts := augment(members(ts_v_+),lts)$TS
      for ts in lts repeat
        lbwt := cons([true, ts]$BWT,lbwt)
    g := mainPrimitivePart g
    lts_g := augment(g,ts)$TS
    lts_g := augment(members(ts_v_+),lts_g)$TS
    -- USE internalAugment with parameters ??
    for ts_g in lts_g repeat
      lbwt := cons([false, ts_g]$BWT,lbwt)
    h := lazyPquo(ts_v,g)
    (ground? h) or (mvar(h) < v) => "leave"
    h := mainPrimitivePart h
    lts_h := augment(h,ts)$TS
    lts_h := augment(members(ts_v_+),lts_h)$TS
    -- USE internalAugment with parameters ??
    for ts_h in lts_h repeat
      inv := stoseInvertible?_reg(q,ts_h)@(List BWT)
      lbwt := concat([bwt for bwt in inv | bwt.val],lbwt)
  sort((x,y) +-> x.val < y.val,lbwt)

stoseInvertibleSet_reg(p:P,ts:TS): Split ==
  --iprint("/")$iprintpack
  k: KeyInvSet := [p,ts]
  e := extractIfCan(k)$HInvSet
  e case EntryInvSet => e::EntryInvSet
  q := primitivePart initiallyReduce(p,ts)
  zero? q => []
  normalized?(q,ts) => [ts]
  v := mvar(q)
  toSave: Split := []
  not algebraic?(v,ts) =>
    toCheck: List BWT := stoseInvertible?_reg(init(q),ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val => toSave := cons(bwt.tower,toSave)
      newq := removeZero(q,bwt.tower)
      zero? newq => "leave"
      toSave := concat(stoseInvertibleSet_reg(newq,bwt.tower), toSave)
    toSave := removeDuplicates toSave
    return algebraicSort(toSave)$quasicomppack
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
  lts, lts_h: Split
  for gwt in lgwt repeat
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
```

```
          lts := augment(ts_v,ts)$TS
          lts := augment(members(ts_v_+),lts)$TS
          toSave := concat(lts,toSave)
        g := mainPrimitivePart g
        h := lazyPquo(ts_v,g)
        h := mainPrimitivePart h
        (ground? h) or (mvar(h) < v) => "leave"
        lts_h := augment(h,ts)$TS
        lts_h := augment(members(ts_v_+),lts_h)$TS
        for ts_h in lts_h repeat
          inv := stoseInvertibleSet_reg(q,ts_h)
          toSave := removeDuplicates concat(inv,toSave)
      toSave := algebraicSort(toSave)$quasicomppack
      insert!(k,toSave)$HInvSet
      toSave

    if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P)
    then

      stoseInvertible?(p:P,ts:TS): List BWT == stoseInvertible?_sqfreg(p,ts)

      stoseInvertibleSet(p:P,ts:TS): Split == stoseInvertibleSet_sqfreg(p,ts)

    else

      stoseInvertible?(p:P,ts:TS): List BWT == stoseInvertible?_reg(p,ts)

      stoseInvertibleSet(p:P,ts:TS): Split == stoseInvertibleSet_reg(p,ts)
```

———-

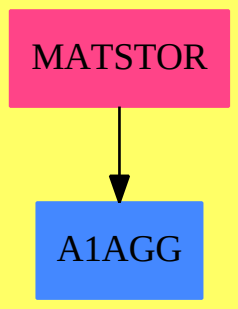
— SFRGCD.dotabb —

```
"SFRGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFRGCD"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SFRGCD" -> "SFRTCAT"
```

———-

20.15 package MATSTOR StorageEfficientMatrixOperations

20.15.1 StorageEfficientMatrixOperations (MATSTOR)



Exports:

copy!	leftScalarTimes!	minus!	plus!	power!
rightScalarTimes!	times!	?**?		

— package MATSTOR StorageEfficientMatrixOperations —

```
)abbrev package MATSTOR StorageEfficientMatrixOperations
++ Author: Clifton J. Williamson
++ Date Created: 18 July 1990
++ Date Last Updated: 18 July 1990
++ Basic Operations:
++ Related Domains: Matrix(R)
++ Also See:
++ AMS Classifications:
++ Keywords: matrix, linear algebra
++ Examples:
++ References:
++ Description:
++ This package provides standard arithmetic operations on matrices.
++ The functions in this package store the results of computations
++ in existing matrices, rather than creating new matrices.  This
++ package works only for matrices of type Matrix and uses the
++ internal representation of this type.

StorageEfficientMatrixOperations(R): Exports == Implementation where
  R     : Ring
  M   ==> Matrix R
  NNI ==> NonNegativeInteger
  ARR ==> PrimitiveArray R
  REP ==> PrimitiveArray PrimitiveArray R
```

```
Exports ==> with
  copy_! : (M,M) -> M
   ++ \spad{copy!(c,a)} copies the matrix \spad{a} into the matrix c.
   ++ Error: if \spad{a} and c do not have the same
   ++ dimensions.
  plus_! : (M,M,M) -> M
   ++ \spad{plus!(c,a,b)} computes the matrix sum \spad{a + b} and stores the
   ++ result in the matrix c.
   ++ Error: if \spad{a}, b, and c do not have the same dimensions.
  minus_! : (M,M) -> M
   ++ \spad{minus!(c,a)} computes \spad{-a} and stores the result in the
   ++ matrix c.
   ++ Error: if a and c do not have the same dimensions.
  minus_! : (M,M,M) -> M
   ++ \spad{!minus!(c,a,b)} computes the matrix difference \spad{a - b}
   ++ and stores the result in the matrix c.
   ++ Error: if \spad{a}, b, and c do not have the same dimensions.
  leftScalarTimes_! : (M,R,M) -> M
   ++ \spad{leftScalarTimes!(c,r,a)} computes the scalar product
   ++ \spad{r * a} and stores the result in the matrix c.
   ++ Error: if \spad{a} and c do not have the same dimensions.
  rightScalarTimes_! : (M,M,R) -> M
   ++ \spad{rightScalarTimes!(c,a,r)} computes the scalar product
   ++ \spad{a * r} and stores the result in the matrix c.
   ++ Error: if \spad{a} and c do not have the same dimensions.
  times_! : (M,M,M) -> M
   ++ \spad{times!(c,a,b)} computes the matrix product \spad{a * b}
   ++ and stores the result in the matrix c.
   ++ Error: if \spad{a}, b, and c do not have
   ++ compatible dimensions.
  power_! : (M,M,M,M,NNI) -> M
   ++ \spad{power!(a,b,c,m,n)} computes m ** n and stores the result in
   ++ \spad{a}. The matrices b and c are used to store intermediate results.
   ++ Error: if \spad{a}, b, c, and m are not square
   ++ and of the same dimensions.
  "**" : (M,NNI) -> M
   ++ \spad{x ** n} computes the n-th power
   ++ of a square matrix. The power n is assumed greater than 1.

Implementation ==> add

  rep : M -> REP
  rep m == m pretend REP

  copy_!(c,a) ==
    m := nrows a; n := ncols a
    not((nrows c) = m and (ncols c) = n) =>
      error "copy!: matrices of incompatible dimensions"
    aa := rep a; cc := rep c
    for i in 0..(m-1) repeat
```

```
      aRow := qelt(aa,i); cRow := qelt(cc,i)
      for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,qelt(aRow,j))
    c

  plus_!(c,a,b) ==
    m := nrows a; n := ncols a
    not((nrows b) = m and (ncols b) = n) =>
      error "plus!: matrices of incompatible dimensions"
    not((nrows c) = m and (ncols c) = n) =>
      error "plus!: matrices of incompatible dimensions"
    aa := rep a; bb := rep b; cc := rep c
    for i in 0..(m-1) repeat
      aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)
      for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,qelt(aRow,j) + qelt(bRow,j))
    c

  minus_!(c,a) ==
    m := nrows a; n := ncols a
    not((nrows c) = m and (ncols c) = n) =>
      error "minus!: matrices of incompatible dimensions"
    aa := rep a; cc := rep c
    for i in 0..(m-1) repeat
      aRow := qelt(aa,i); cRow := qelt(cc,i)
      for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,-qelt(aRow,j))
    c

  minus_!(c,a,b) ==
    m := nrows a; n := ncols a
    not((nrows b) = m and (ncols b) = n) =>
      error "minus!: matrices of incompatible dimensions"
    not((nrows c) = m and (ncols c) = n) =>
      error "minus!: matrices of incompatible dimensions"
    aa := rep a; bb := rep b; cc := rep c
    for i in 0..(m-1) repeat
      aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)
      for j in 0..(n-1) repeat
        qsetelt_!(cRow,j,qelt(aRow,j) - qelt(bRow,j))
    c

  leftScalarTimes_!(c,r,a) ==
    m := nrows a; n := ncols a
    not((nrows c) = m and (ncols c) = n) =>
      error "leftScalarTimes!: matrices of incompatible dimensions"
    aa := rep a; cc := rep c
    for i in 0..(m-1) repeat
      aRow := qelt(aa,i); cRow := qelt(cc,i)
      for j in 0..(n-1) repeat
```

```
      qsetelt_!(cRow,j,r * qelt(aRow,j))
  c

rightScalarTimes_!(c,a,r) ==
  m := nrows a; n := ncols a
  not((nrows c) = m and (ncols c) = n) =>
    error "rightScalarTimes!: matrices of incompatible dimensions"
  aa := rep a; cc := rep c
  for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
      qsetelt_!(cRow,j,qelt(aRow,j) * r)
  c

copyCol_!: (ARR,REP,Integer,Integer) -> ARR
copyCol_!(bCol,bb,j,n1) ==
  for i in 0..n1 repeat qsetelt_!(bCol,i,qelt(qelt(bb,i),j))

times_!(c,a,b) ==
  m := nrows a; n := ncols a; p := ncols b
  not((nrows b) = n and (nrows c) = m and (ncols c) = p) =>
    error "times!: matrices of incompatible dimensions"
  aa := rep a; bb := rep b; cc := rep c
  bCol : ARR := new(n,0)
  m1 := (m :: Integer) - 1; n1 := (n :: Integer) - 1
  for j in 0..(p-1) repeat
    copyCol_!(bCol,bb,j,n1)
    for i in 0..m1 repeat
      aRow := qelt(aa,i); cRow := qelt(cc,i)
      sum : R := 0
      for k in 0..n1 repeat
        sum := sum + qelt(aRow,k) * qelt(bCol,k)
      qsetelt_!(cRow,j,sum)
  c

power_!(a,b,c,m,p) ==
  mm := nrows a; nn := ncols a
  not(mm = nn) =>
    error "power!: matrix must be square"
  not((nrows b) = mm and (ncols b) = nn) =>
    error "power!: matrices of incompatible dimensions"
  not((nrows c) = mm and (ncols c) = nn) =>
    error "power!: matrices of incompatible dimensions"
  not((nrows m) = mm and (ncols m) = nn) =>
    error "power!: matrices of incompatible dimensions"
  flag := false
  copy_!(b,m)
  repeat
    if odd? p then
      flag =>
```

```
            times_!(c,b,a)
            copy_!(a,c)
          flag := true
          copy_!(a,b)
--        one? p => return a
        (p = 1) => return a
        p := p quo 2
        times_!(c,b,b)
        copy_!(b,c)

    m ** n ==
      not square? m => error "**: matrix must be square"
      a := copy m; b := copy m; c := copy m
      power_!(a,b,c,m,n)
```

———-

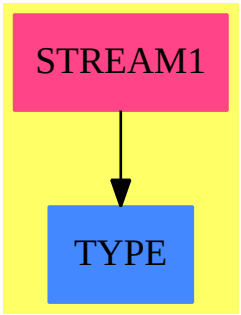
— **MATSTOR.dotabb** —

```
"MATSTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MATSTOR"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"MATSTOR" -> "A1AGG"
```

———-

20.16 package STREAM1 StreamFunctions1

20.16.1 StreamFunctions1 (STREAM1)



Exports:
concat

— **package STREAM1 StreamFunctions1** —

```
)abbrev package STREAM1 StreamFunctions1
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Keywords: stream, infinite list, infinite sequence
++ Description:
++ Functions defined on streams with entries in one set.

StreamFunctions1(S:Type): Exports == Implementation where
  ST  ==> Stream

  Exports ==> with
    concat: ST ST S -> ST S
      ++ concat(u) returns the left-to-right concatentation of the
      ++ streams in u. Note that \spad{concat(u) = reduce(concat,u)}.
      ++
      ++X m:=[i for i in 10..]
      ++X n:=[j for j in 1.. | prime? j]
      ++X p:=[m,n]::Stream(Stream(PositiveInteger))
      ++X concat(p)

  Implementation ==> add

    concat z == delay
      empty? z => empty()
      empty?(x := frst z) => concat rst z
      concat(frst x,concat(rst x,concat rst z))
```

———-

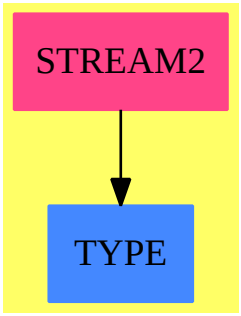
— STREAM1.dotabb —

```
"STREAM1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM1" -> "TYPE"
```

———-

20.17 package STREAM2 StreamFunctions2

20.17.1 StreamFunctions2 (STREAM2)



Exports:
map reduce scan

— package STREAM2 StreamFunctions2 —

```
)abbrev package STREAM2 StreamFunctions2
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Keywords: stream, infinite list, infinite sequence
++ Description:
++ Functions defined on streams with entries in two sets.

StreamFunctions2(A:Type,B:Type): Exports == Implementation where
  ST   ==> Stream

  Exports ==> with
    map: ((A -> B),ST A) -> ST B
      ++ map(f,s) returns a stream whose elements are the function f applied
      ++ to the corresponding elements of s.
      ++ Note that \spad{map(f,[x0,x1,x2,...]) = [f(x0),f(x1),f(x2),..]}.
      ++
      ++X m:=[i for i in 1..]
      ++X f(i:PositiveInteger):PositiveInteger==i**2
      ++X map(f,m)

    scan: (B,((A,B) -> B),ST A) -> ST B
      ++ scan(b,h,[x0,x1,x2,...]) returns \spad{[y0,y1,y2,...]}, where
      ++ \spad{y0 = h(x0,b)},
      ++ \spad{y1 = h(x1,y0)},\spad{...}
      ++ \spad{yn = h(xn,y(n-1))}.
      ++
```

```
      ++X m:=[i for i in 1..]::Stream(Integer)
      ++X f(i:Integer,j:Integer):Integer==i+j
      ++X scan(1,f,m)

    reduce:  (B,(A,B) -> B,ST A) -> B
      ++ reduce(b,f,u), where u is a finite stream \spad{[x0,x1,...,xn]},
      ++ returns the value \spad{r(n)} computed as follows:
      ++ \spad{r0 = f(x0,b),
      ++ r1 = f(x1,r0),...,
      ++ r(n) = f(xn,r(n-1))}.
      ++
      ++X m:=[i for i in 1..300]::Stream(Integer)
      ++X f(i:Integer,j:Integer):Integer==i+j
      ++X reduce(1,f,m)

--  rreduce: (B,(A,B) -> B,ST A) -> B
--    ++ reduce(b,h,[x0,x1,..,xn]) = h(x1,h(x2(..,h(x(n-1),h(xn,b))..)
--  reshape: (ST B,ST A) -> ST B
--    ++ reshape(y,x) = y

  Implementation ==> add

    mapp: (A -> B,ST A) -> ST B
    mapp(f,x)== delay
      empty? x => empty()
      concat(f frst x, map(f,rst x))

    map(f,x) ==
      explicitlyEmpty? x => empty()
      eq?(x,rst x) => repeating([f frst x])
      mapp(f, x)

--  reshape(y,x) == y

    scan(b,h,x) == delay
      empty? x => empty()
      c := h(frst x,b)
      concat(c,scan(c,h,rst x))

    reduce(b,h,x) ==
      empty? x => b
      reduce(h(frst x,b),h,rst x)
--  rreduce(b,h,x) ==
--    empty? x => b
--    h(frst x,rreduce(b,h,rst x))
```

———————

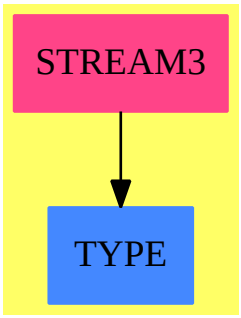
— **STREAM2.dotabb** —

```
"STREAM2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM2" -> "TYPE"
```

———-

20.18 package STREAM3 StreamFunctions3

20.18.1 StreamFunctions3 (STREAM3)



Exports:
map

— **package STREAM3 StreamFunctions3** —

```
)abbrev package STREAM3 StreamFunctions3
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Keywords: stream, infinite list, infinite sequence
++ Description:
++ Functions defined on streams with entries in three sets.

StreamFunctions3(A,B,C): Exports == Implementation where
  A  : Type
  B  : Type
  C  : Type
  ST ==> Stream

  Exports ==> with
    map: ((A,B) -> C,ST A,ST B) -> ST C
      ++ map(f,st1,st2) returns the stream whose elements are the
```

```
      ++ function f applied to the corresponding elements of st1 and st2.
      ++ \spad{map(f,[x0,x1,x2,..],[y0,y1,y2,..]) = [f(x0,y0),f(x1,y1),..]}.
      ++
      ++S
      ++X m:=[i for i in 1..]::Stream(Integer)
      ++X n:=[i for i in 1..]::Stream(Integer)
      ++X f(i:Integer,j:Integer):Integer == i+j
      ++X map(f,m,n)

  Implementation ==> add

    mapp:((A,B) -> C,ST A,ST B) -> ST C
    mapp(g,x,y) == delay
      empty? x or empty? y => empty()
      concat(g(frst x,frst y), map(g,rst x,rst y))

    map(g,x,y) ==
      explicitlyEmpty? x => empty()
      eq?(x,rst x) => map(z +-> g(frst x,z),y)$StreamFunctions2(B,C)
      explicitlyEmpty? y => empty()
      eq?(y,rst y) => map(z +-> g(z,frst y),x)$StreamFunctions2(A,C)
      mapp(g,x,y)
```

———-

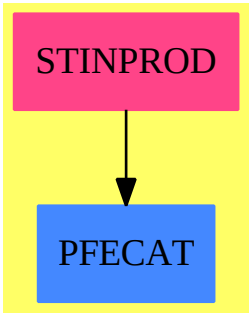
— **STREAM3.dotabb** —

```
"STREAM3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM3" -> "TYPE"
```

———-

20.19 package STINPROD StreamInfiniteProduct

20.19.1 StreamInfiniteProduct (STINPROD)



Exports:

evenInfiniteProduct	generalInfiniteProduct
infiniteProduct	oddInfiniteProduct

— package STINPROD StreamInfiniteProduct —

```
)abbrev package STINPROD StreamInfiniteProduct
++ Author: Clifton J. Williamson
++ Date Created: 23 February 1990
++ Date Last Updated: 23 February 1990
++ Basic Operations: infiniteProduct, evenInfiniteProduct, oddInfiniteProduct,
++   generalInfiniteProduct
++ Related Domains: UnivariateTaylorSeriesCategory
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, infinite product
++ Examples:
++ References:
++ Description:
++ This package computes infinite products of Taylor series over an
++ integral domain of characteristic 0.  Here Taylor series are
++ represented by streams of Taylor coefficients.

StreamInfiniteProduct(Coef): Exports == Implementation where
  Coef: Join(IntegralDomain,CharacteristicZero)
  I  ==> Integer
  QF ==> Fraction
  ST ==> Stream

  Exports ==> with

    infiniteProduct: ST Coef -> ST Coef
      ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
```

```
      ++ The series \spad{f(x)} should have constant coefficient 1.
    evenInfiniteProduct: ST Coef -> ST Coef
      ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    oddInfiniteProduct: ST Coef -> ST Coef
      ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.
    generalInfiniteProduct: (ST Coef,I,I) -> ST Coef
      ++ generalInfiniteProduct(f(x),a,d) computes
      ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
      ++ The series \spad{f(x)} should have constant coefficient 1.

  Implementation ==> add

    if Coef has Field then

      import StreamTaylorSeriesOperations(Coef)
      import StreamTranscendentalFunctions(Coef)

      infiniteProduct st             == exp lambert log st
      evenInfiniteProduct st         == exp evenlambert log st
      oddInfiniteProduct st          == exp oddlambert log st
      generalInfiniteProduct(st,a,d) == exp generalLambert(log st,a,d)

    else

      import StreamTaylorSeriesOperations(QF Coef)
      import StreamTranscendentalFunctions(QF Coef)

      applyOverQF:(ST QF Coef -> ST QF Coef,ST Coef) -> ST Coef
      applyOverQF(f,st) ==
        stQF := map(z1 +-> z1::QF(Coef),st)$StreamFunctions2(Coef,QF Coef)
        map(z1 +-> retract(z1)@Coef,f stQF)$StreamFunctions2(QF Coef,Coef)

      infiniteProduct st     == applyOverQF(z1 +-> exp lambert log z1,st)
      evenInfiniteProduct st == applyOverQF(z1 +-> exp evenlambert log z1,st)
      oddInfiniteProduct st  == applyOverQF(z1 +-> exp oddlambert log z1,st)
      generalInfiniteProduct(st,a,d) ==
        applyOverQF(z1 +-> exp generalLambert(log z1,a,d),st)
```

———-

— STINPROD.dotabb —

```
"STINPROD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STINPROD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"STINPROD" -> "PFECAT"
```

———-

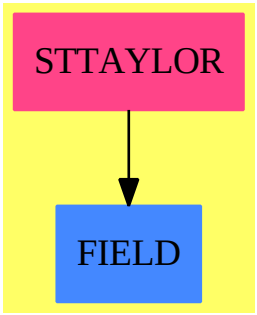
20.20 package STTAYLOR StreamTaylorSeriesOperations

Problems raising a UTS to a negative integer power.

The code in powern(rn,x) which raises an unnecessary error where no distinction between rational and integer powers are made.

The fix is easy. Since the problem does not exist in SUPS we can just take the definition there.

20.20.1 StreamTaylorSeriesOperations (STTAYLOR)



Exports:

addiag	coerce	compose	deriv	eval
evenlambert	exquo	gderiv	generalLambert	int
integers	integrate	invmultisect	lagrange	lambert
lazyGintegrate	lazyIntegrate	mapdiv	mapmult	monom
multisect	nlde	oddintegers	oddlambert	power
powern	recip	revert	?*?	?+?
?-?	?/?	?*?	-?	

— **package STTAYLOR StreamTaylorSeriesOperations** —

```
)abbrev package STTAYLOR StreamTaylorSeriesOperations
++ Author: William Burge, Stephen Watt, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 26 May 1994
++ Basic Operations:
++ Related Domains: Stream(A), ParadoxicalCombinatorsForStreams(A),
++   StreamTranscendentalFunctions(A),
++   StreamTranscendentalFunctionsNonCommutative(A)
++ Also See:
++ AMS Classifications:
```

```
++ Keywords: stream, Taylor series
++ Examples:
++ References:
++ Description:
++ StreamTaylorSeriesOperations implements Taylor series arithmetic,
++ where a Taylor series is represented by a stream of its coefficients.

StreamTaylorSeriesOperations(A): Exports == Implementation where
  A :        Ring
  RN     ==> Fraction Integer
  I      ==> Integer
  NNI    ==> NonNegativeInteger
  ST     ==> Stream
  SP2    ==> StreamFunctions2
  SP3    ==> StreamFunctions3
  L      ==> List
  LA     ==> List A
  YS     ==> Y$ParadoxicalCombinatorsForStreams(A)
  UN     ==> Union(ST A,"failed")
  Exports ==> with
    "+"          : (ST A,ST A) -> ST A
      ++ a + b returns the power series sum of \spad{a} and \spad{b}:
      ++ \spad{[a0,a1,..] + [b0,b1,..] = [a0 + b0,a1 + b1,..]}
    "-"          : (ST A,ST A) -> ST A
      ++ a - b returns the power series difference of \spad{a} and
      ++ \spad{b}: \spad{[a0,a1,..] - [b0,b1,..] = [a0 - b0,a1 - b1,..]}
    "-"          : ST A -> ST A
      ++ - a returns the power series negative of \spad{a}:
      ++ \spad{- [a0,a1,...] = [- a0,- a1,...]}
    "*"          : (ST A,ST A) -> ST A
      ++ a * b returns the power series (Cauchy) product of \spad{a} and b:
      ++ \spad{[a0,a1,...] * [b0,b1,...] = [c0,c1,...]} where
      ++ \spad{ck = sum(i + j = k,ai * bk)}.
    "*"          : (A,ST A) -> ST A
      ++ r * a returns the power series scalar multiplication of r by \spad{a}:
      ++ \spad{r * [a0,a1,...] = [r * a0,r * a1,...]}
    "*"          : (ST A,A) -> ST A
      ++ a * r returns the power series scalar multiplication of \spad{a} by r:
      ++ \spad{[a0,a1,...] * r = [a0 * r,a1 * r,...]}
    "exquo"      : (ST A,ST A) -> Union(ST A,"failed")
      ++ exquo(a,b) returns the power series quotient of \spad{a} by b,
      ++ if the quotient exists, and "failed" otherwise
    "/"          : (ST A,ST A) -> ST A
      ++ a / b returns the power series quotient of \spad{a} by b.
      ++ An error message is returned if \spad{b} is not invertible.
      ++ This function is used in fixed point computations.
    recip        : ST A -> UN
      ++ recip(a) returns the power series reciprocal of \spad{a}, or
      ++ "failed" if not possible.
    monom        : (A,I) -> ST A
```

```
  ++ monom(deg,coef) is a monomial of degree deg with coefficient
  ++ coef.
integers     : I -> ST I
  ++ integers(n) returns \spad{[n,n+1,n+2,...]}.
oddintegers  : I -> ST I
  ++ oddintegers(n) returns \spad{[n,n+2,n+4,...]}.
int          : A -> ST A
  ++ int(r) returns [r,r+1,r+2,...], where r is a ring element.
mapmult      : (ST A,ST A) -> ST A
  ++ mapmult([a0,a1,..],[b0,b1,..])
  ++ returns \spad{[a0*b0,a1*b1,..]}.
deriv        : ST A -> ST A
  ++ deriv(a) returns the derivative of the power series with
  ++ respect to the power series variable. Thus
  ++ \spad{deriv([a0,a1,a2,...])} returns \spad{[a1,2 a2,3 a3,...]}.
gderiv       : (I -> A,ST A)  -> ST A
  ++ gderiv(f,[a0,a1,a2,..]) returns
  ++ \spad{[f(0)*a0,f(1)*a1,f(2)*a2,..]}.
coerce       : A -> ST A
  ++ coerce(r) converts a ring element r to a stream with one element.
eval         : (ST A,A) -> ST A
  ++ eval(a,r) returns a stream of partial sums of the power series
  ++ \spad{a} evaluated at the power series variable equal to r.
compose      : (ST A,ST A) -> ST A
  ++ compose(a,b) composes the power series \spad{a} with
  ++ the power series b.
lagrange     : ST A -> ST A
  ++ lagrange(g) produces the power series for f where f is
  ++ implicitly defined as \spad{f(z) = z*g(f(z))}.
revert       : ST A -> ST A
  ++ revert(a) computes the inverse of a power series \spad{a}
  ++ with respect to composition.
  ++ the series should have constant coefficient 0 and first
  ++ order coefficient 1.
addiag       : ST ST A -> ST A
  ++ addiag(x) performs diagonal addition of a stream of streams. if x =
  ++ \spad{[[a<0,0>,a<0,1>,..],[a<1,0>,a<1,1>,..],[a<2,0>,a<2,1>,..],..]}
  ++ and \spad{addiag(x) = [b<0,b<1>,...], then b<k> = sum(i+j=k,a<i,j>)}.
lambert      : ST A -> ST A
  ++ lambert(st) computes \spad{f(x) + f(x**2) + f(x**3) + ...}
  ++ if st is a stream representing \spad{f(x)}.
  ++ This function is used for computing infinite products.
  ++ If \spad{f(x)} is a power series with constant coefficient 1 then
  ++ \spad{prod(f(x**n),n = 1..infinity) = exp(lambert(log(f(x))))}.
oddlambert   : ST A -> ST A
  ++ oddlambert(st) computes \spad{f(x) + f(x**3) + f(x**5) + ...}
  ++ if st is a stream representing \spad{f(x)}.
  ++ This function is used for computing infinite products.
  ++ If f(x) is a power series with constant coefficient 1 then
  ++ \spad{prod(f(x**(2*n-1)),n=1..infinity) = exp(oddlambert(log(f(x))))}.
```

```
    evenlambert  : ST A -> ST A
      ++ evenlambert(st) computes \spad{f(x**2) + f(x**4) + f(x**6) + ...}
      ++ if st is a stream representing \spad{f(x)}.
      ++ This function is used for computing infinite products.
      ++ If \spad{f(x)} is a power series with constant coefficient 1, then
      ++ \spad{prod(f(x**(2*n)),n=1..infinity) = exp(evenlambert(log(f(x))))}.
    generalLambert : (ST A,I,I) -> ST A
      ++ generalLambert(f(x),a,d) returns
      ++ \spad{f(x**a) + f(x**(a + d)) + f(x**(a + 2 d)) + ...}.
      ++ \spad{f(x)} should have zero constant
      ++ coefficient and \spad{a} and d should be positive.
    multisect    : (I,I,ST A) -> ST A
      ++ multisect(a,b,st)
      ++ selects the coefficients of \spad{x**((a+b)*n+a)},
      ++ and changes them to \spad{x**n}.
    invmultisect : (I,I,ST A) -> ST A
      ++ invmultisect(a,b,st) substitutes \spad{x**((a+b)*n)} for \spad{x**n}
      ++ and multiplies by \spad{x**b}.
    if A has Algebra RN then
      integrate  : (A,ST A) -> ST A
        ++ integrate(r,a) returns the integral of the power series \spad{a}
        ++ with respect to the power series variableintegration where
        ++ r denotes the constant of integration. Thus
        ++ \spad{integrate(a,[a0,a1,a2,...]) = [a,a0,a1/2,a2/3,...]}.
      lazyIntegrate  : (A,() -> ST A) -> ST A
        ++ lazyIntegrate(r,f) is a local function
        ++ used for fixed point computations.
      nlde       : ST ST A -> ST A
        ++ nlde(u) solves a
        ++ first order non-linear differential equation described by u of the
        ++ form \spad{[[b<0,0>,b<0,1>,...],[b<1,0>,b<1,1>,.],...]}.
        ++ the differential equation has the form
        ++ \spad{y'=sum(i=0 to infinity,j=0 to infinity,b<i,j>*(x**i)*(y**j))}.
      powern : (RN,ST A) -> ST A
        ++ powern(r,f) raises power series f to the power r.
    if A has Field then
      mapdiv     : (ST A,ST A) -> ST A
        ++ mapdiv([a0,a1,..],[b0,b1,..]) returns
        ++ \spad{[a0/b0,a1/b1,..]}.
      lazyGintegrate : (I -> A,A,() -> ST A) -> ST A
        ++ lazyGintegrate(f,r,g) is used for fixed point computations.
      power      : (A,ST A) -> ST A
        ++ power(a,f) returns the power series f raised to the power \spad{a}.

  Implementation ==> add

--% definitions

    zro: () -> ST A
    -- returns a zero power series
```

```
    zro() == empty()$ST(A)

--% arithmetic

    x + y == delay
      empty? y => x
      empty? x => y
      eq?(x,rst x) => map(z +-> frst x+z, y)
      eq?(y,rst y) => map(z +-> frst y+z, x)
      concat(frst x + frst y,rst x + rst y)

    x - y == delay
      empty? y => x
      empty? x => -y
      eq?(x,rst x) => map(z +-> frst x-z, y)
      eq?(y,rst y) => map(z +-> z-frst y, x)
      concat(frst x - frst y,rst x - rst y)

    -y == map(z +-> -z, y)

    (x:ST A) * (y:ST A) == delay
      empty? y => zro()
      empty? x => zro()
      concat(frst x * frst y,frst x * rst y + rst x * y)

    (s:A) * (x:ST A) ==
      zero? s => zro()
      map(z +-> s*z, x)

    (x:ST A) * (s:A) ==
      zero? s => zro()
      map(z +-> z*s, x)

    iDiv: (ST A,ST A,A) -> ST A
    iDiv(x,y,ry0) == delay
      empty? x => empty()
      c0 := frst x * ry0
      concat(c0,iDiv(rst x - c0 * rst y,y,ry0))

    x exquo y ==
      for n in 1.. repeat
        n > 1000 => return "failed"
        empty? y => return "failed"
        empty? x => return empty()
        frst y = 0 =>
          frst x = 0 => (x := rst x; y := rst y)
          return "failed"
        leave "first entry in y is non-zero"
      (ry0 := recip frst y) case "failed" => "failed"
      empty? rst y => map(z +-> z*(ry0 :: A), x)
```

```
      iDiv(x,y,ry0 :: A)

    (x:ST A) / (y:ST A) == delay
      empty? y => error "/: division by zero"
      empty? x => empty()
      (ry0 := recip frst y) case "failed" =>
        error "/: second argument is not invertible"
      empty? rst y => map(z +-> z*(ry0::A),x)
      iDiv(x,y,ry0 :: A)

    recip x ==
      empty? x => "failed"
      rh1 := recip frst x
      rh1 case "failed" => "failed"
      rh := rh1 :: A
      delay
        concat(rh,iDiv(- rh * rst x,x,rh))

--% coefficients

    rp: (I,A) -> L A
    -- rp(z,s) is a list of length z each of whose entries is s.
    rp(z,s) ==
      z <= 0 => empty()
      concat(s,rp(z-1,s))

    rpSt: (I,A) -> ST A
    -- rpSt(z,s) is a stream of length z each of whose entries is s.
    rpSt(z,s) == delay
      z <= 0 => empty()
      concat(s,rpSt(z-1,s))

    monom(s,z) ==
      z < 0 => error "monom: cannot create monomial of negative degree"
      concat(rpSt(z,0),concat(s,zro()))

--% some streams of integers
    nnintegers: NNI -> ST NNI
    nnintegers zz == generate(y +-> y+1, zz)
    integers z    == generate(y +-> y+1, z)
    oddintegers z == generate(y +-> y+2, z)
    int s         == generate(y +-> y+1, s)

--% derivatives

    mapmult(x,y) == delay
      empty? y => zro()
      empty? x => zro()
      concat(frst x * frst y,mapmult(rst x,rst y))
```

```
    deriv x ==
      empty? x => zro()
      mapmult(int 1,rest x)

    gderiv(f,x) ==
      empty? x => zro()
      mapmult(map(f,integers 0)$SP2(I,A),x)

--% coercions

    coerce(s:A) ==
      zero? s => zro()
      concat(s,zro())

--% evaluations and compositions

    eval(x,at) ==
      scan(0,(y,z) +-> y+z,mapmult(x,generate(y +-> at*y,1)))$SP2(A,A)

    compose(x,y) == delay
      empty? y => concat(frst x,zro())
      not zero? frst y =>
        error "compose: 2nd argument should have 0 constant coefficient"
      empty? x => zro()
      concat(frst x,compose(rst x,y) * rst(y))

--% reversion

    lagrangere:(ST A,ST A) -> ST A
    lagrangere(x,c) == delay(concat(0,compose(x,c)))
    lagrange x == YS(y +-> lagrangere(x,y))

    revert x ==
      empty? x => error "revert should start 0,1,..."
      zero? frst x =>
        empty? rst x => error "revert: should start 0,1,..."
--        one? frst rst x => lagrange(recip(rst x) :: (ST A))
        (frst rst x) = 1 => lagrange(recip(rst x) :: (ST A))
      error "revert:should start 0,1,..."

--% lambert functions

    addiag(ststa:ST ST A) == delay
      empty? ststa => zro()
      empty? frst ststa => concat(0,addiag rst ststa)
      concat(frst(frst ststa),rst(frst ststa) + addiag(rst ststa))

-- lambert operates on a series +/[a[i]x**i for i in 1..] , and produces
-- the series +/[a[i](x**i/(1-x**i)) for i in 1..] i.e. forms the
-- coefficients A[n] which is the sum of a[i] for all divisors i of n
```

```
-- (including 1 and n)

    rptg1:(I,A) -> ST A
    --                                 ---------
    -- returns the repeating stream [s,0,...,0]; (there are z zeroes)
    rptg1(z,s) == repeating concat(s,rp(z,0))

    rptg2:(I,A) -> ST A
    --                                        ---------
    -- returns the repeating stream [0,...,0,s,0,...,0]
    -- there are z leading zeroes and z-1 in the period
    rptg2(z,s) == repeating concat(rp(z,0),concat(s,rp(z-1,0)))

    rptg3:(I,I,I,A) -> ST A
    rptg3(a,d,n,s) ==
      concat(rpSt(n*(a-1),0),repeating(concat(s,rp(d*n-1,0))))

    lambert x == delay
      empty? x => zro()
      zero? frst x =>
        concat(0,addiag(map(rptg1,integers 0,rst x)$SP3(I,A,ST A)))
      error "lambert:constant coefficient should be zero"

    oddlambert x == delay
      empty? x => zro()
      zero? frst x =>
        concat(0,addiag(map(rptg1,oddintegers 1,rst x)$SP3(I,A,ST A)))
      error "oddlambert: constant coefficient should be zero"

    evenlambert x == delay
      empty? x => zro()
      zero? frst x =>
        concat(0,addiag(map(rptg2,integers 1,rst x)$SP3(I,A,ST A)))
      error "evenlambert: constant coefficient should be zero"

    generalLambert(st,a,d) == delay
      a < 1 or d < 1 =>
        error "generalLambert: both integer arguments must be positive"
      empty? st => zro()
      zero? frst st =>
        concat(0,addiag(map((x,y) +-> rptg3(a,d,x,y),
                  integers 1,rst st)$SP3(I,A,ST A)))
      error "generalLambert: constant coefficient should be zero"

--% misc. functions

    ms: (I,I,ST A) -> ST A
    ms(m,n,s) == delay
      empty? s => zro()
      zero? n => concat(frst s,ms(m,m-1,rst s))
```

```
      ms(m,n-1,rst s)

    multisect(b,a,x) == ms(a+b,0,rest(x,a :: NNI))

    altn: (ST A,ST A) -> ST A
    altn(zs,s) == delay
      empty? s => zro()
      concat(frst s,concat(zs,altn(zs,rst s)))

    invmultisect(a,b,x) ==
      concat(rpSt(b,0),altn(rpSt(a + b - 1,0),x))

-- comps(ststa,y) forms the composition of +/b[i,j]*y**i*x**j
-- where y is a power series in y.

    cssa ==> concat$(ST ST A)
    mapsa ==> map$SP2(ST A,ST A)
    comps: (ST ST A,ST A) -> ST ST A
    comps(ststa,x) == delay$(ST ST A)
       empty? ststa => empty()$(ST ST A)
       empty? x => cssa(frst ststa,empty()$(ST ST A))
       cssa(frst ststa,mapsa(y +-> (rst x)*y,comps(rst ststa,x)))

    if A has Algebra RN then
      integre: (ST A,I) -> ST A
      integre(x,n) == delay
        empty? x => zro()
        concat((1$I/n) * frst(x),integre(rst x,n + 1))

      integ: ST A -> ST A
      integ x == integre(x,1)

      integrate(a,x) == concat(a,integ x)
      lazyIntegrate(s,xf) == concat(s,integ(delay xf))

      nldere:(ST ST A,ST A) -> ST A
      nldere(lslsa,c) == lazyIntegrate(0,addiag(comps(lslsa,c)))
      nlde lslsa == YS(y +-> nldere(lslsa,y))

      RATPOWERS : Boolean := A has "**": (A,RN) -> A

      smult: (RN,ST A) -> ST A
      smult(rn,x) == map(y +-> rn*y, x)
      powerrn:(RN,ST A,ST A) -> ST A
      powerrn(rn,x,c) == delay
        concat(1,integ(smult(rn + 1,c * deriv x)) - rst x * c)
      powern(rn,x) ==
        order : I := 0
        for n in 0.. repeat
          empty? x => return zro()
```

```
          not zero? frst x => (order := n; leave x)
          x := rst x
          n = 1000 =>
            error "**: series with many leading zero coefficients"
        (ord := (order exquo denom(rn))) case "failed" =>
          error "**: rational power does not exist"
        co := frst x
        (invCo := recip co) case "failed" =>
           error "** rational power of coefficient undefined"
-- This error message is misleading, isn't it? see sups.spad/cRationalPower
        power :=
--          one? co => YS(y +-> powerrn(rn,x,y))
          (co = 1) => YS(y +-> powerrn(rn,x,y))
          (denom rn) = 1 =>
            not negative?(num := numer rn) =>
-- It seems that this cannot happen, but I don't know why
              (co**num::NNI) * YS(y +-> powerrn(rn,(invCo :: A) * x, y))
            (invCo::A)**((-num)::NNI) * YS(y +-> powerrn(rn,(invCo :: A)*x, y))

          RATPOWERS => co**rn * YS(y +-> powerrn(rn,(invCo :: A)*x, y))
          error "** rational power of coefficient undefined"

    if A has Field then
      mapdiv(x,y) == delay
        empty? y => error "stream division by zero"
        empty? x => zro()
        concat(frst x/frst y,mapdiv(rst x,rst y))

      ginteg: (I -> A,ST A) -> ST A
      ginteg(f,x) == mapdiv(x,map(f,integers 1)$SP2(I,A))

      lazyGintegrate(fntoa,s,xf) == concat(s,ginteg(fntoa,delay xf))

      finteg: ST A -> ST A
      finteg x == mapdiv(x,int 1)
      powerre: (A,ST A,ST A) -> ST A
      powerre(s,x,c) == delay
        empty? x => zro()
        frst x^=1 => error "**:constant coefficient should be 1"
        concat(frst x,finteg((s+1)*(c*deriv x))-rst x * c)
      power(s,x) == YS(y +-> powerre(s,x,y))
```

———-

— STTAYLOR.dotabb —

```
"STTAYLOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STTAYLOR"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
```

```
"STTAYLOR" -> "FIELD"
```

———-

20.21 package STNSR StreamTensor

— **StreamTensor.input** —

```
)set break resume
)sys rm -f StreamTensor.output
)spool StreamTensor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamTensor
--R
--R StreamTensor(R: Type)  is a package constructor
--R Abbreviation for StreamTensor is STNSR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STNSR
--R
--R------------------------------ Operations ------------------------------
--R tensorMap : (Stream(R),(R -> List(R))) -> Stream(R)
--R
--E 1

)spool
)lisp (bye)
```

———-

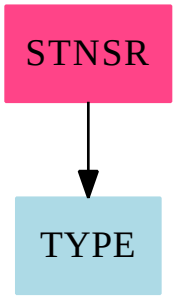
— **StreamTensor.help** —

```
=====================================================================
StreamTensor examples
=====================================================================

See Also:
o )show StreamTensor
```

———-

20.21.1 StreamTensor (STNSR)



Exports:
tensorMap

— **package STNSR StreamTensor** —

```
)abbrev package STNSR StreamTensor
StreamTensor(R: Type): with

        tensorMap: (Stream R, R -> List R) -> Stream R
          ++ tensorMap([s1, s2, ...], f) returns the stream consisting of all
          ++ elements of f(s1) followed by all elements of f(s2) and so on.

    == add

        tensorMap(s, f) ==
            empty? s => empty()
            concat([f first s], delay tensorMap(rest s, f))
```

———-

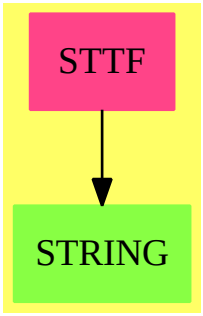
— **STNSR.dotabb** —

```
"STNSR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STNSR"]
"TYPE" [color=lightblue,href="bookvol10.2.pdf#nameddest=TYPE"];
"STNSR" -> "TYPE"
```

———-

20.22 package STTF StreamTranscendentalFunctions

20.22.1 StreamTranscendentalFunctions (STTF)



Exports:

acos	acot	acsc	asec	asin
atan	cos	cot	csc	exp
sec	sin	tan	sincos	sinhcosh
?**?				

— package STTF StreamTranscendentalFunctions —

```
)abbrev package STTF StreamTranscendentalFunctions
++ Author: William Burge, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 6 March 1995
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, elementary function, transcendental function
++ Examples:
++ References:
++ Description:
++ StreamTranscendentalFunctions implements transcendental functions on
++ Taylor series, where a Taylor series is represented by a stream of
++ its coefficients.

StreamTranscendentalFunctions(Coef): Exports == Implementation where
  Coef : Algebra Fraction Integer
  L   ==> List
  I   ==> Integer
  RN  ==> Fraction Integer
  SG  ==> String
  ST  ==> Stream Coef
  STT ==> StreamTaylorSeriesOperations Coef
  YS  ==> Y$ParadoxicalCombinatorsForStreams(Coef)
```

```
  Exports ==> with
--% Exponentials and Logarithms
    exp     : ST -> ST
      ++ exp(st) computes the exponential of a power series st.
    log     : ST -> ST
      ++ log(st) computes the log of a power series.
    "**"    : (ST,ST) -> ST
      ++ st1 ** st2 computes the power of a power series st1 by another
      ++ power series st2.

--% TrigonometricFunctionCategory
    sincos  : ST -> Record(sin:ST, cos:ST)
      ++ sincos(st) returns a record containing the sine and cosine
      ++ of a power series st.
    sin     : ST -> ST
      ++ sin(st) computes sine of a power series st.
    cos     : ST -> ST
      ++ cos(st) computes cosine of a power series st.
    tan     : ST -> ST
      ++ tan(st) computes tangent of a power series st.
    cot     : ST -> ST
      ++ cot(st) computes cotangent of a power series st.
    sec     : ST -> ST
      ++ sec(st) computes secant of a power series st.
    csc     : ST -> ST
      ++ csc(st) computes cosecant of a power series st.
    asin    : ST -> ST
      ++ asin(st) computes arcsine of a power series st.
    acos    : ST -> ST
      ++ acos(st) computes arccosine of a power series st.
    atan    : ST -> ST
      ++ atan(st) computes arctangent of a power series st.
    acot    : ST -> ST
      ++ acot(st) computes arccotangent of a power series st.
    asec    : ST -> ST
      ++ asec(st) computes arcsecant of a power series st.
    acsc    : ST -> ST
      ++ acsc(st) computes arccosecant of a power series st.

--% HyperbloicTrigonometricFunctionCategory
    sinhcosh: ST -> Record(sinh:ST, cosh:ST)
      ++ sinhcosh(st) returns a record containing
      ++ the hyperbolic sine and cosine
      ++ of a power series st.
    sinh    : ST -> ST
      ++ sinh(st) computes the hyperbolic sine of a power series st.
    cosh    : ST -> ST
      ++ cosh(st) computes the hyperbolic cosine of a power series st.
    tanh    : ST -> ST
```

```
      ++ tanh(st) computes the hyperbolic tangent of a power series st.
    coth    : ST -> ST
      ++ coth(st) computes the hyperbolic cotangent of a power series st.
    sech    : ST -> ST
      ++ sech(st) computes the hyperbolic secant of a power series st.
    csch    : ST -> ST
      ++ csch(st) computes the hyperbolic cosecant of a power series st.
    asinh   : ST -> ST
      ++ asinh(st) computes the inverse hyperbolic sine of a power series st.
    acosh   : ST -> ST
      ++ acosh(st) computes the inverse hyperbolic cosine
      ++ of a power series st.
    atanh   : ST -> ST
      ++ atanh(st) computes the inverse hyperbolic tangent
      ++ of a power series st.
    acoth   : ST -> ST
      ++ acoth(st) computes the inverse hyperbolic
      ++ cotangent of a power series st.
    asech   : ST -> ST
      ++ asech(st) computes the inverse hyperbolic secant of a
      ++ power series st.
    acsch   : ST -> ST
      ++ acsch(st) computes the inverse hyperbolic
      ++ cosecant of a power series st.

  Implementation ==> add
    import StreamTaylorSeriesOperations Coef

    TRANSFCN : Boolean := Coef has TranscendentalFunctionCategory

--% Error Reporting

    TRCONST : SG := "series expansion involves transcendental constants"
    NPOWERS : SG := "series expansion has terms of negative degree"
    FPOWERS : SG := "series expansion has terms of fractional degree"
    MAYFPOW : SG := "series expansion may have terms of fractional degree"
    LOGS : SG := "series expansion has logarithmic term"
    NPOWLOG : SG :=
       "series expansion has terms of negative degree or logarithmic term"
    FPOWLOG : SG :=
       "series expansion has terms of fractional degree or logarithmic term"
    NOTINV : SG := "leading coefficient not invertible"

--% Exponentials and Logarithms

    expre:(Coef,ST,ST) -> ST
    expre(r,e,dx) == lazyIntegrate(r,e*dx)

    exp z ==
      empty? z => 1 :: ST
```

```
      (coef := frst z) = 0 => YS(y +-> expre(1,y,deriv z))
      TRANSFCN => YS(y +-> expre(exp coef,y,deriv z))
      error concat("exp: ",TRCONST)

    log z ==
      empty? z => error "log: constant coefficient should not be 0"
      (coef := frst z) = 0 => error "log: constant coefficient should not be 0"
      coef = 1 => lazyIntegrate(0,deriv z/z)
      TRANSFCN => lazyIntegrate(log coef,deriv z/z)
      error concat("log: ",TRCONST)

    z1:ST ** z2:ST == exp(z2 * log z1)

--% Trigonometric Functions

    sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
    sincosre(rs,rc,sc,dx,sign) ==
      [lazyIntegrate(rs,(second sc)*dx),lazyIntegrate(rc,sign*(first sc)*dx)]

    -- When the compiler had difficulties with the above definition,
    -- I did the following to help it:

    -- sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
    -- sincosre(rs,rc,sc,dx,sign) ==
      -- st1 : ST := (second sc) * dx
      -- st2 : ST := (first sc) * dx
      -- st2 := sign * st2
      -- [lazyIntegrate(rs,st1),lazyIntegrate(rc,st2)]

    sincos z ==
      empty? z => [0 :: ST,1 :: ST]
      l :=
        (coef := frst z) = 0 => YS(y +-> sincosre(0,1,y,deriv z,-1),2)
        TRANSFCN => YS(y +-> sincosre(sin coef,cos coef,y,deriv z,-1),2)
        error concat("sincos: ",TRCONST)
      [first l,second l]

    sin z == sincos(z).sin
    cos z == sincos(z).cos

    tanre:(Coef,ST,ST,Coef) -> ST
    tanre(r,t,dx,sign) == lazyIntegrate(r,((1 :: ST) + sign*t*t)*dx)

    -- When the compiler had difficulties with the above definition,
    -- I did the following to help it:

    -- tanre:(Coef,ST,ST,Coef) -> ST
    -- tanre(r,t,dx,sign) ==
      -- st1 : ST := t * t
      -- st1 := sign * st1
```

```
    -- st2 : ST := 1 :: ST
    -- st1 := st2 + st1
    -- st1 := st1 * dx
    -- lazyIntegrate(r,st1)

  tan z ==
    empty? z => 0 :: ST
    (coef := frst z) = 0 => YS(y +-> tanre(0,y,deriv z,1))
    TRANSFCN => YS(y +-> tanre(tan coef,y,deriv z,1))
    error concat("tan: ",TRCONST)

  cotre:(Coef,ST,ST) -> ST
  cotre(r,t,dx) == lazyIntegrate(r,-((1 :: ST) + t*t)*dx)

  -- When the compiler had difficulties with the above definition,
  -- I did the following to help it:

  -- cotre:(Coef,ST,ST) -> ST
  -- cotre(r,t,dx) ==
    -- st1 : ST := t * t
    -- st2 : ST := 1 :: ST
    -- st1 := st2 + st1
    -- st1 := st1 * dx
    -- st1 := -st1
    -- lazyIntegrate(r,st1)

  cot z ==
    empty? z => error "cot: cot(0) is undefined"
    (coef := frst z) = 0 => error concat("cot: ",NPOWERS)
    TRANSFCN => YS(y +-> cotre(cot coef,y,deriv z))
    error concat("cot: ",TRCONST)

  sec z ==
    empty? z => 1 :: ST
    frst z = 0 => recip(cos z) :: ST
    TRANSFCN =>
      cosz := cos z
      first cosz = 0 => error concat("sec: ",NPOWERS)
      recip(cosz) :: ST
    error concat("sec: ",TRCONST)

  csc z ==
    empty? z => error "csc: csc(0) is undefined"
    TRANSFCN =>
      sinz := sin z
      first sinz = 0 => error concat("csc: ",NPOWERS)
      recip(sinz) :: ST
    error concat("csc: ",TRCONST)

  orderOrFailed : ST -> Union(I,"failed")
```

```
orderOrFailed x ==
-- returns the order of x or "failed"
-- if -1 is returned, the series is identically zero
  for n in 0..1000 repeat
    empty? x => return -1
    not zero? frst x => return n :: I
    x := rst x
  "failed"

asin z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 =>
    integrate(0,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
  TRANSFCN =>
    coef = 1 or coef = -1 =>
      x := (1 :: ST) - z*z
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("asin: ",MAYFPOW)
      (order := ord :: I) = -1 => return asin(coef) :: ST
      odd? order => error concat("asin: ",FPOWERS)
      squirt := powern(1/2,x)
      (quot := (deriv z) exquo squirt) case "failed" =>
         error concat("asin: ",NOTINV)
      integrate(asin coef,quot :: ST)
    integrate(asin coef,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
  error concat("asin: ",TRCONST)

acos z ==
  empty? z =>
    TRANSFCN => acos(0)$Coef :: ST
    error concat("acos: ",TRCONST)
  TRANSFCN =>
    coef := frst z
    coef = 1 or coef = -1 =>
      x := (1 :: ST) - z*z
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("acos: ",MAYFPOW)
      (order := ord :: I) = -1 => return acos(coef) :: ST
      odd? order => error concat("acos: ",FPOWERS)
      squirt := powern(1/2,x)
      (quot := (-deriv z) exquo squirt) case "failed" =>
         error concat("acos: ",NOTINV)
      integrate(acos coef,quot :: ST)
    integrate(acos coef,-powern(-1/2,(1 :: ST) - z*z) * (deriv z))
  error concat("acos: ",TRCONST)

atan z ==
  empty? z => 0 :: ST
```

```
      (coef := frst z) = 0 =>
        integrate(0,(recip((1 :: ST) + z*z) :: ST) * (deriv z))
      TRANSFCN =>
        (y := recip((1 :: ST) + z*z)) case "failed" =>
          error concat("atan: ",LOGS)
        integrate(atan coef,(y :: ST) * (deriv z))
      error concat("atan: ",TRCONST)

    acot z ==
      empty? z =>
        TRANSFCN => acot(0)$Coef :: ST
        error concat("acot: ",TRCONST)
      TRANSFCN =>
        (y := recip((1 :: ST) + z*z)) case "failed" =>
          error concat("acot: ",LOGS)
        integrate(acot frst z,-(y :: ST) * (deriv z))
      error concat("acot: ",TRCONST)

    asec z ==
      empty? z => error "asec: constant coefficient should not be 0"
      TRANSFCN =>
        (coef := frst z) = 0 =>
          error "asec: constant coefficient should not be 0"
        coef = 1 or coef = -1 =>
          x := z*z - (1 :: ST)
          -- compute order of 'x'
          (ord := orderOrFailed x) case "failed" =>
            error concat("asec: ",MAYFPOW)
          (order := ord :: I) = -1 => return asec(coef) :: ST
          odd? order => error concat("asec: ",FPOWERS)
          squirt := powern(1/2,x)
          (quot := (deriv z) exquo squirt) case "failed" =>
            error concat("asec: ",NOTINV)
          (quot2 := (quot :: ST) exquo z) case "failed" =>
            error concat("asec: ",NOTINV)
          integrate(asec coef,quot2 :: ST)
        integrate(asec coef,(powern(-1/2,z*z-(1::ST))*(deriv z)) / z)
      error concat("asec: ",TRCONST)

    acsc z ==
      empty? z => error "acsc: constant coefficient should not be zero"
      TRANSFCN =>
        (coef := frst z) = 0 =>
          error "acsc: constant coefficient should not be zero"
        coef = 1 or coef = -1 =>
          x := z*z - (1 :: ST)
          -- compute order of 'x'
          (ord := orderOrFailed x) case "failed" =>
            error concat("acsc: ",MAYFPOW)
          (order := ord :: I) = -1 => return acsc(coef) :: ST
```

```
          odd? order => error concat("acsc: ",FPOWERS)
          squirt := powern(1/2,x)
          (quot := (-deriv z) exquo squirt) case "failed" =>
            error concat("acsc: ",NOTINV)
          (quot2 := (quot :: ST) exquo z) case "failed" =>
            error concat("acsc: ",NOTINV)
          integrate(acsc coef,quot2 :: ST)
        integrate(acsc coef,-(powern(-1/2,z*z-(1::ST))*(deriv z)) / z)
      error concat("acsc: ",TRCONST)

--% Hyperbolic Trigonometric Functions

    sinhcosh z ==
      empty? z => [0 :: ST,1 :: ST]
      l :=
        (coef := frst z) = 0 => YS(y +-> sincosre(0,1,y,deriv z,1),2)
        TRANSFCN => YS(y +-> sincosre(sinh coef,cosh coef,y,deriv z,1),2)
        error concat("sinhcosh: ",TRCONST)
      [first l,second l]

    sinh z == sinhcosh(z).sinh
    cosh z == sinhcosh(z).cosh

    tanh z ==
      empty? z => 0 :: ST
      (coef := frst z) = 0 => YS(y +-> tanre(0,y,deriv z,-1))
      TRANSFCN => YS(y +-> tanre(tanh coef,y,deriv z,-1))
      error concat("tanh: ",TRCONST)

    coth z ==
      tanhz := tanh z
      empty? tanhz => error "coth: coth(0) is undefined"
      (frst tanhz) = 0 => error concat("coth: ",NPOWERS)
      recip(tanhz) :: ST

    sech z ==
      coshz := cosh z
      (empty? coshz) or (frst coshz = 0) => error concat("sech: ",NPOWERS)
      recip(coshz) :: ST

    csch z ==
      sinhz := sinh z
      (empty? sinhz) or (frst sinhz = 0) => error concat("csch: ",NPOWERS)
      recip(sinhz) :: ST

    asinh z ==
      empty? z => 0 :: ST
      (coef := frst z) = 0 => log(z + powern(1/2,(1 :: ST) + z*z))
      TRANSFCN =>
        x := (1 :: ST) + z*z
```

```
    -- compute order of 'x', in case coefficient(z,0) = +- %i
    (ord := orderOrFailed x) case "failed" =>
      error concat("asinh: ",MAYFPOW)
    (order := ord :: I) = -1 => return asinh(coef) :: ST
    odd? order => error concat("asinh: ",FPOWERS)
    -- the argument to 'log' must have a non-zero constant term
    log(z + powern(1/2,x))
  error concat("asinh: ",TRCONST)

acosh z ==
  empty? z =>
    TRANSFCN => acosh(0)$Coef :: ST
    error concat("acosh: ",TRCONST)
  TRANSFCN =>
    coef := frst z
    coef = 1 or coef = -1 =>
      x := z*z - (1 :: ST)
      -- compute order of 'x'
      (ord := orderOrFailed x) case "failed" =>
        error concat("acosh: ",MAYFPOW)
      (order := ord :: I) = -1 => return acosh(coef) :: ST
      odd? order => error concat("acosh: ",FPOWERS)
      -- the argument to 'log' must have a non-zero constant term
      log(z + powern(1/2,x))
    log(z + powern(1/2,z*z - (1 :: ST)))
  error concat("acosh: ",TRCONST)

atanh z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 =>
    (inv(2::RN)::Coef) * log(((1 :: ST) + z)/((1 :: ST) - z))
  TRANSFCN =>
    coef = 1 or coef = -1 => error concat("atanh: ",LOGS)
    (inv(2::RN)::Coef) * log(((1 :: ST) + z)/((1 :: ST) - z))
  error concat("atanh: ",TRCONST)

acoth z ==
  empty? z =>
    TRANSFCN => acoth(0)$Coef :: ST
    error concat("acoth: ",TRCONST)
  TRANSFCN =>
    frst z = 1 or frst z = -1 => error concat("acoth: ",LOGS)
    (inv(2::RN)::Coef) * log((z + (1 :: ST))/(z - (1 :: ST)))
  error concat("acoth: ",TRCONST)

asech z ==
  empty? z => error "asech: asech(0) is undefined"
  TRANSFCN =>
    (coef := frst z) = 0 => error concat("asech: ",NPOWLOG)
    coef = 1 or coef = -1 =>
```

```
          x := (1 :: ST) - z*z
          -- compute order of 'x'
          (ord := orderOrFailed x) case "failed" =>
            error concat("asech: ",MAYFPOW)
          (order := ord :: I) = -1 => return asech(coef) :: ST
          odd? order => error concat("asech: ",FPOWERS)
          log(((1 :: ST) + powern(1/2,x))/z)
        log(((1 :: ST) + powern(1/2,(1 :: ST) - z*z))/z)
      error concat("asech: ",TRCONST)

    acsch z ==
      empty? z => error "acsch: acsch(0) is undefined"
      TRANSFCN =>
        frst z = 0 => error concat("acsch: ",NPOWLOG)
        x := z*z + (1 :: ST)
        -- compute order of 'x'
        (ord := orderOrFailed x) case "failed" =>
          error concat("acsc: ",MAYFPOW)
        (order := ord :: I) = -1 => return acsch(frst z) :: ST
        odd? order => error concat("acsch: ",FPOWERS)
        log(((1 :: ST) + powern(1/2,x))/z)
      error concat("acsch: ",TRCONST)
```

———-

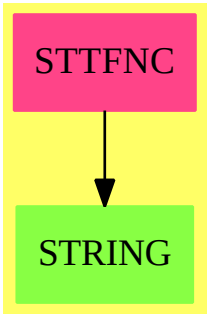
— STTF.dotabb —

```
"STTF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STTF"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"STTF" -> "STRING"
```

———-

20.23 package STTFNC StreamTranscendentalFunctionsNonCommutative

20.23.1 StreamTranscendentalFunctionsNonCommutative (STTFNC)



Exports:

acos	acot	acsc	asec	asin
atan	cos	cot	csc	exp
sec	sin	tan	?**?	

— **package STTFNC StreamTranscendentalFunctionsNonCommutative** —

```
)abbrev package STTFNC StreamTranscendentalFunctionsNonCommutative
++ Author: Clifton J. Williamson
++ Date Created: 26 May 1994
++ Date Last Updated: 26 May 1994
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, transcendental function, non-commutative
++ Examples:
++ References:
++ Description:
++ StreamTranscendentalFunctionsNonCommutative implements transcendental
++ functions on Taylor series over a non-commutative ring, where a Taylor
++ series is represented by a stream of its coefficients.

StreamTranscendentalFunctionsNonCommutative(Coef): _
         Exports == Implementation where
  Coef :   Algebra Fraction Integer
  I    ==> Integer
  SG   ==> String
  ST   ==> Stream Coef
  STTF ==> StreamTranscendentalFunctions Coef
```

```
  Exports ==> with
--% Exponentials and Logarithms
    exp     : ST -> ST
      ++ exp(st) computes the exponential of a power series st.
    log     : ST -> ST
      ++ log(st) computes the log of a power series.
    "**"    : (ST,ST) -> ST
      ++ st1 ** st2 computes the power of a power series st1 by another
      ++ power series st2.

--% TrigonometricFunctionCategory
    sin     : ST -> ST
      ++ sin(st) computes sine of a power series st.
    cos     : ST -> ST
      ++ cos(st) computes cosine of a power series st.
    tan     : ST -> ST
      ++ tan(st) computes tangent of a power series st.
    cot     : ST -> ST
      ++ cot(st) computes cotangent of a power series st.
    sec     : ST -> ST
      ++ sec(st) computes secant of a power series st.
    csc     : ST -> ST
      ++ csc(st) computes cosecant of a power series st.
    asin    : ST -> ST
      ++ asin(st) computes arcsine of a power series st.
    acos    : ST -> ST
      ++ acos(st) computes arccosine of a power series st.
    atan    : ST -> ST
      ++ atan(st) computes arctangent of a power series st.
    acot    : ST -> ST
      ++ acot(st) computes arccotangent of a power series st.
    asec    : ST -> ST
      ++ asec(st) computes arcsecant of a power series st.
    acsc    : ST -> ST
      ++ acsc(st) computes arccosecant of a power series st.

--% HyperbloicTrigonometricFunctionCategory
    sinh    : ST -> ST
      ++ sinh(st) computes the hyperbolic sine of a power series st.
    cosh    : ST -> ST
      ++ cosh(st) computes the hyperbolic cosine of a power series st.
    tanh    : ST -> ST
      ++ tanh(st) computes the hyperbolic tangent of a power series st.
    coth    : ST -> ST
      ++ coth(st) computes the hyperbolic cotangent of a power series st.
    sech    : ST -> ST
      ++ sech(st) computes the hyperbolic secant of a power series st.
    csch    : ST -> ST
      ++ csch(st) computes the hyperbolic cosecant of a power series st.
```

```
    asinh   : ST -> ST
      ++ asinh(st) computes the inverse hyperbolic sine of a power series st.
    acosh   : ST -> ST
      ++ acosh(st) computes the inverse hyperbolic cosine
      ++ of a power series st.
    atanh   : ST -> ST
      ++ atanh(st) computes the inverse hyperbolic tangent
      ++ of a power series st.
    acoth   : ST -> ST
      ++ acoth(st) computes the inverse hyperbolic
      ++ cotangent of a power series st.
    asech   : ST -> ST
      ++ asech(st) computes the inverse hyperbolic secant of a
      ++ power series st.
    acsch   : ST -> ST
      ++ acsch(st) computes the inverse hyperbolic
      ++ cosecant of a power series st.

  Implementation ==> add
    import StreamTaylorSeriesOperations(Coef)

--% Error Reporting

    ZERO    : SG := "series must have constant coefficient zero"
    ONE     : SG := "series must have constant coefficient one"
    NPOWERS : SG := "series expansion has terms of negative degree"

--% Exponentials and Logarithms

    exp z ==
      empty? z => 1 :: ST
      (frst z) = 0 =>
        expx := exp(monom(1,1))$STTF
        compose(expx,z)
      error concat("exp: ",ZERO)

    log z ==
      empty? z => error concat("log: ",ONE)
      (frst z) = 1 =>
        log1PlusX := log(monom(1,0) + monom(1,1))$STTF
        compose(log1PlusX,z - monom(1,0))
      error concat("log: ",ONE)

    (z1:ST) ** (z2:ST) == exp(log(z1) * z2)

--% Trigonometric Functions

    sin z ==
      empty? z => 0 :: ST
      (frst z) = 0 =>
```

```
    sinx := sin(monom(1,1))$STTF
    compose(sinx,z)
  error concat("sin: ",ZERO)

cos z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    cosx := cos(monom(1,1))$STTF
    compose(cosx,z)
  error concat("cos: ",ZERO)

tan z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    tanx := tan(monom(1,1))$STTF
    compose(tanx,z)
  error concat("tan: ",ZERO)

cot z ==
  empty? z => error "cot: cot(0) is undefined"
  (frst z) = 0 => error concat("cot: ",NPOWERS)
  error concat("cot: ",ZERO)

sec z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    secx := sec(monom(1,1))$STTF
    compose(secx,z)
  error concat("sec: ",ZERO)

csc z ==
  empty? z => error "csc: csc(0) is undefined"
  (frst z) = 0 => error concat("csc: ",NPOWERS)
  error concat("csc: ",ZERO)

asin z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    asinx := asin(monom(1,1))$STTF
    compose(asinx,z)
  error concat("asin: ",ZERO)

atan z ==
  empty? z => 0 :: ST
  (frst z) = 0 =>
    atanx := atan(monom(1,1))$STTF
    compose(atanx,z)
  error concat("atan: ",ZERO)

acos z == error "acos: acos undefined on this coefficient domain"
```

```
    acot z == error "acot: acot undefined on this coefficient domain"
    asec z == error "asec: asec undefined on this coefficient domain"
    acsc z == error "acsc: acsc undefined on this coefficient domain"

--% Hyperbolic Trigonometric Functions

    sinh z ==
      empty? z => 0 :: ST
      (frst z) = 0 =>
        sinhx := sinh(monom(1,1))$STTF
        compose(sinhx,z)
      error concat("sinh: ",ZERO)

    cosh z ==
      empty? z => 1 :: ST
      (frst z) = 0 =>
        coshx := cosh(monom(1,1))$STTF
        compose(coshx,z)
      error concat("cosh: ",ZERO)

    tanh z ==
      empty? z => 0 :: ST
      (frst z) = 0 =>
        tanhx := tanh(monom(1,1))$STTF
        compose(tanhx,z)
      error concat("tanh: ",ZERO)

    coth z ==
      empty? z => error "coth: coth(0) is undefined"
      (frst z) = 0 => error concat("coth: ",NPOWERS)
      error concat("coth: ",ZERO)

    sech z ==
      empty? z => 1 :: ST
      (frst z) = 0 =>
        sechx := sech(monom(1,1))$STTF
        compose(sechx,z)
      error concat("sech: ",ZERO)

    csch z ==
      empty? z => error "csch: csch(0) is undefined"
      (frst z) = 0 => error concat("csch: ",NPOWERS)
      error concat("csch: ",ZERO)

    asinh z ==
      empty? z => 0 :: ST
      (frst z) = 0 =>
        asinhx := asinh(monom(1,1))$STTF
        compose(asinhx,z)
      error concat("asinh: ",ZERO)
```

```
    atanh z ==
      empty? z => 0 :: ST
      (frst z) = 0 =>
        atanhx := atanh(monom(1,1))$STTF
        compose(atanhx,z)
      error concat("atanh: ",ZERO)

    acosh z == error "acosh: acosh undefined on this coefficient domain"
    acoth z == error "acoth: acoth undefined on this coefficient domain"
    asech z == error "asech: asech undefined on this coefficient domain"
    acsch z == error "acsch: acsch undefined on this coefficient domain"
```

———-

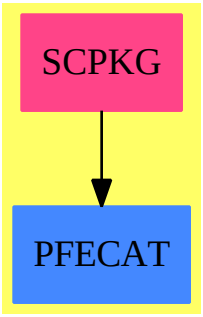
— **STTFNC.dotabb** —

```
"STTFNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STTFNC"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"STTFNC" -> "STRING"
```

———-

20.24 package SCPKG StructuralConstantsPackage

20.24.1 StructuralConstantsPackage (SCPKG)



Exports:
coordinates structuralConstants

— **package SCPKG StructuralConstantsPackage** —

```
)abbrev package SCPKG StructuralConstantsPackage
++ Authors: J. Grabmeier
++ Date Created: 02 April 1992
++ Date Last Updated: 14 April 1992
++ Basic Operations:
++ Related Constructors: AlgebraPackage, AlgebraGivenByStructuralConstants
++ Also See:
++ AMS Classifications:
++ Keywords: structural constants
++ Reference:
++ Description:
++ StructuralConstantsPackage provides functions creating
++ structural constants from a multiplication tables or a basis
++ of a matrix algebra and other useful functions in this context.

StructuralConstantsPackage(R:Field): public == private where

  L  ==> List
  S  ==> Symbol
  FRAC ==> Fraction
  POLY ==> Polynomial
  V  ==> Vector
  M  ==> Matrix
  REC  ==> Record(particular: Union(V R,"failed"),basis: List V R)
  LSMP ==> LinearSystemMatrixPackage(R,V R,V R, M R)

  public ==>  with
      -- what we really want to have here is a matrix over
      -- linear polynomials in the list of symbols, having arbitrary
      -- coefficients from a ring extension of R, e.g. FRAC POLY R.
      structuralConstants : (L S, M FRAC POLY R) -> V M FRAC POLY R
        ++ structuralConstants(ls,mt) determines the structural constants
        ++ of an algebra with generators ls and multiplication table mt, the
        ++ entries of which must be given as linear polynomials in the
        ++ indeterminates given by ls. The result is in particular useful
        ++  as fourth argument for \spadtype{AlgebraGivenByStructuralConstants}
        ++  and \spadtype{GenericNonAssociativeAlgebra}.
      structuralConstants : (L S, M POLY R) -> V M POLY R
        ++ structuralConstants(ls,mt) determines the structural constants
        ++ of an algebra with generators ls and multiplication table mt, the
        ++ entries of which must be given as linear polynomials in the
        ++ indeterminates given by ls. The result is in particular useful
        ++  as fourth argument for \spadtype{AlgebraGivenByStructuralConstants}
        ++  and \spadtype{GenericNonAssociativeAlgebra}.
      structuralConstants: L M R -> V M R
        ++ structuralConstants(basis)  takes the basis of a matrix
        ++ algebra, e.g. the result of \spadfun{basisOfCentroid} and calculates
        ++ the structural constants.
        ++ Note, that the it is not checked, whether basis really is a
        ++ basis of a matrix algebra.
```

```
      coordinates: (M R, L M R) -> V R
        ++ coordinates(a,[v1,...,vn]) returns the coordinates of \spad{a}
        ++ with respect to the \spad{R}-module basis \spad{v1},...,\spad{vn}.

  private ==> add

      matrix2Vector: M R -> V R
      matrix2Vector m ==
        lili : L L R := listOfLists m
        --li : L R  := reduce(concat, listOfLists m)
        li : L R  := reduce(concat, lili)
        construct(li)$(V R)

      coordinates(x,b) ==
        m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
        n : NonNegativeInteger := nrows(b.1) * ncols(b.1)
        transitionMatrix   : Matrix R := new(n,m,0$R)$Matrix(R)
        for i in 1..m repeat
          setColumn_!(transitionMatrix,i,matrix2Vector(b.i))
        res : REC := solve(transitionMatrix,matrix2Vector(x))$LSMP
        if (not every?(zero?$R,first res.basis)) then
          error("coordinates: the second argument is linearly dependent")
        (res.particular  case "failed") =>
          error("coordinates: first argument is not in linear span of _
second argument")
        (res.particular) :: (Vector R)

      structuralConstants b ==
        --n := rank()
        -- be careful with the possibility that b is not a basis
        m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
        sC : Vector Matrix R := [new(m,m,0$R) for k in 1..m]
        for i in 1..m repeat
          for j in 1..m repeat
            covec : Vector R := coordinates(b.i * b.j, b)$%
            for k in 1..m repeat
               setelt( sC.k, i, j, covec.k )
        sC

      structuralConstants(ls:L S, mt: M POLY R)  ==
        nn := #(ls)
        nrows(mt) ^= nn or ncols(mt) ^= nn =>
          error "structuralConstants: size of second argument does not _
agree with number of generators"
        gamma : L M POLY R := []
        lscopy : L S := copy ls
        while not null lscopy repeat
          mat : M POLY R := new(nn,nn,0)
          s : S := first lscopy
          for i in 1..nn repeat
```

```
            for j in 1..nn repeat
              p := qelt(mt,i,j)
              totalDegree(p,ls) > 1 =>
                error "structuralConstants: entries of second argument _
must be linear polynomials in the generators"
              if (c := coefficient(p, s, 1) ) ^= 0 then qsetelt_!(mat,i,j,c)
          gamma := cons(mat, gamma)
          lscopy := rest lscopy
        vector reverse gamma

      structuralConstants(ls:L S, mt: M FRAC POLY R)  ==
        nn := #(ls)
        nrows(mt) ^= nn or ncols(mt) ^= nn =>
          error "structuralConstants: size of second argument does not _
agree with number of generators"
        gamma : L M FRAC(POLY R) := []
        lscopy : L S := copy ls
        while not null lscopy repeat
          mat : M FRAC(POLY R) := new(nn,nn,0)
          s : S := first lscopy
          for i in 1..nn repeat
            for j in 1..nn repeat
              r := qelt(mt,i,j)
              q := denom(r)
              totalDegree(q,ls) ^= 0 =>
                error "structuralConstants: entries of second argument _
must be (linear) polynomials in the generators"
              p := numer(r)
              totalDegree(p,ls) > 1 =>
                error "structuralConstants: entries of second argument _
must be linear polynomials in the generators"
              if (c := coefficient(p, s, 1) ) ^= 0 then qsetelt_!(mat,i,j,c/q)
          gamma := cons(mat, gamma)
          lscopy := rest lscopy
        vector reverse gamma
```

———-

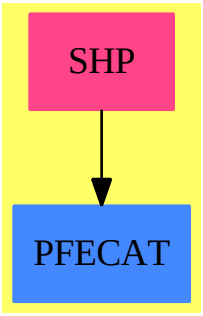
— SCPKG.dotabb —

```
"SCPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SCPKG"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SCPKG" -> "PFECAT"
```

———-

20.25 package SHP SturmHabichtPackage

20.25.1 SturmHabichtPackage (SHP)



Exports:

SturmHabicht	SturmHabichtCoefficients
SturmHabichtMultiple	SturmHabichtSequence
countRealRoots	countRealRootsMultiple
subresultantSequence	

— **package SHP SturmHabichtPackage** —

```
)abbrev package SHP SturmHabichtPackage
++ Author: Lalo Gonzalez-Vega
++ Date Created: 1994?
++ Date Last Updated: 30 January 1996
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords: localization
++ References:
++ Description:
++ This package produces functions for counting etc. real roots of univariate
++ polynomials in x over R, which must be an OrderedIntegralDomain

SturmHabichtPackage(R,x): T == C where
  R: OrderedIntegralDomain
  x: Symbol

  UP ==> UnivariatePolynomial
  L ==> List
  INT ==> Integer
  NNI ==> NonNegativeInteger

  T == with
--     subresultantSequenceBegin:(UP(x,R),UP(x,R)) -> L UP(x,R)
```

```
--       ++ \spad{subresultantSequenceBegin(p1,p2)} computes the  initial terms
--       ++ of the Subresultant sequence Sres(j)(P,deg(P),Q,deg(P)-1)
--       ++ when deg(Q)<deg(P)
--     subresultantSequenceNext:L UP(x,R) -> L UP(x,R)
--     subresultantSequenceInner:(UP(x,R),UP(x,R)) -> L UP(x,R)
     subresultantSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
       ++ subresultantSequence(p1,p2) computes the (standard)
       ++ subresultant sequence of p1 and p2
--     sign:R -> R
--     delta:NNI -> R

--     polsth1:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
--     polsth2:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
--     polsth3:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
     SturmHabichtSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
       ++ SturmHabichtSequence(p1,p2) computes the Sturm-Habicht
       ++ sequence of p1 and p2
     SturmHabichtCoefficients:(UP(x,R),UP(x,R)) -> L R
       ++ SturmHabichtCoefficients(p1,p2) computes the principal
       ++ Sturm-Habicht coefficients of p1 and p2

--     variation:L R -> INT
--     permanence:L R -> INT
--     qzeros:L R -> L R
--     epsil:(NNI,R,R) -> INT
--     numbnce:L R -> NNI
--     numbce:L R -> NNI
--     wfunctaux:L R -> INT
--     wfunct:L R -> INT

     SturmHabicht:(UP(x,R),UP(x,R)) -> INT
       ++ SturmHabicht(p1,p2) computes c_{+}-c_{-} where
       ++ c_{+} is the number of real roots of p1 with p2>0 and c_{-}
       ++ is the number of real roots of p1 with p2<0. If p2=1 what
       ++ you get is the number of real roots of p1.
     countRealRoots:(UP(x,R)) -> INT
       ++ countRealRoots(p) says how many real roots p has
     if R has GcdDomain then
        SturmHabichtMultiple:(UP(x,R),UP(x,R)) -> INT
          ++ SturmHabichtMultiple(p1,p2) computes c_{+}-c_{-} where
          ++ c_{+} is the number of real roots of p1 with p2>0 and c_{-}
          ++ is the number of real roots of p1 with p2<0. If p2=1 what
          ++ you get is the number of real roots of p1.
        countRealRootsMultiple:(UP(x,R)) -> INT
          ++ countRealRootsMultiple(p) says how many real roots p has,
          ++ counted with multiplicity


  C == add
     p1,p2: UP(x,R)
```

```
Ex ==> OutputForm
import OutputForm

subresultantSequenceBegin(p1,p2):L UP(x,R) ==
  d1:NNI:=degree(p1)
  d2:NNI:=degree(p2)
  n:NNI:=(d1-1)::NNI
  d2 = n =>
    Pr:UP(x,R):=pseudoRemainder(p1,p2)
    append([p1,p2]::L UP(x,R),[Pr]::L UP(x,R))
  d2 = (n-1)::NNI =>
    Lc1:UP(x,R):=leadingCoefficient(p1)*leadingCoefficient(p2)*p2
    Lc2:UP(x,R):=-leadingCoefficient(p1)*pseudoRemainder(p1,p2)
    append([p1,p2]::L UP(x,R),[Lc1,Lc2]::L UP(x,R))
  LSubr:L UP(x,R):=[p1,p2]
  in1:INT:=(d2+1)::INT
  in2:INT:=(n-1)::INT
  for i in in1..in2 repeat
    LSubr:L UP(x,R):=append(LSubr::L UP(x,R),[0]::L UP(x,R))
  c1:R:=(leadingCoefficient(p1)*leadingCoefficient(p2))**((n-d2)::NNI)
  Lc1:UP(x,R):=monomial(c1,0)*p2
  Lc2:UP(x,R):=
    (-leadingCoefficient(p1))**((n-d2)::NNI)*pseudoRemainder(p1,p2)
  append(LSubr::L UP(x,R),[Lc1,Lc2]::L UP(x,R))

subresultantSequenceNext(LcsI:L UP(x,R)):L UP(x,R) ==
  p2:UP(x,R):=last LcsI
  p1:UP(x,R):=first rest reverse LcsI
  d1:NNI:=degree(p1)
  d2:NNI:=degree(p2)
  in1:NNI:=(d1-1)::NNI
  d2 = in1 =>
    pr1:UP(x,R):=
      (pseudoRemainder(p1,p2) exquo (leadingCoefficient(p1))**2)::UP(x,R)
    append(LcsI:L UP(x,R),[pr1]:L UP(x,R))
  d2 < in1 =>
    c1:R:=leadingCoefficient(p1)
    pr1:UP(x,R):=
     (leadingCoefficient(p2)**((in1-d2)::NNI)*p2 exquo
         c1**((in1-d2)::NNI))::UP(x,R)
    pr2:UP(x,R):=
      (pseudoRemainder(p1,p2) exquo (-c1)**((in1-d2+2)::NNI))::UP(x,R)
    LSub:L UP(x,R):=[pr1,pr2]
    for k in ((d2+1)::INT)..((in1-1)::INT) repeat
      LSub:L UP(x,R):=append([0]:L UP(x,R),LSub:L UP(x,R))
    append(LcsI:L UP(x,R),LSub:L UP(x,R))

subresultantSequenceInner(p1,p2):L UP(x,R) ==
  Lin:L UP(x,R):=subresultantSequenceBegin(p1:UP(x,R),p2:UP(x,R))
  indf:NNI:= if not(Lin.last::UP(x,R) = 0) then degree(Lin.last::UP(x,R))
```

```
                                          else 0
      while not(indf = 0) repeat
        Lin:L UP(x,R):=subresultantSequenceNext(Lin:L UP(x,R))
        indf:NNI:= if not(Lin.last::UP(x,R)=0) then degree(Lin.last::UP(x,R))
                                          else 0
      for j in #(Lin:L UP(x,R))..degree(p1) repeat
        Lin:L UP(x,R):=append(Lin:L UP(x,R),[0]:L UP(x,R))
      Lin


-- Computation of the subresultant sequence Sres(j)(P,p,Q,q) when:
--             deg(P) = p   and   deg(Q) = q   and   p > q

     subresultantSequence(p1,p2):L UP(x,R) ==
       p:NNI:=degree(p1)
       q:NNI:=degree(p2)
       List1:L UP(x,R):=subresultantSequenceInner(p1,p2)
       List2:L UP(x,R):=[p1,p2]
       c1:R:=leadingCoefficient(p1)
       for j in 3..#(List1) repeat
         Pr0:UP(x,R):=List1.j
         Pr1:UP(x,R):=(Pr0 exquo c1**((p-q-1)::NNI))::UP(x,R)
         List2:L UP(x,R):=append(List2:L UP(x,R),[Pr1]:L UP(x,R))
       List2


-- Computation of the sign (+1,0,-1) of an element in an ordered integral
-- domain

--     sign(r:R):R ==
--       r =$R 0 => 0
--       r >$R 0 => 1
--       -1


-- Computation of the delta function:

     delta(int1:NNI):R ==
       (-1)**((int1*(int1+1) exquo 2)::NNI)


-- Computation of the Sturm-Habicht sequence of two polynomials P and Q
-- in R[x] where R is an ordered integral domaine

     polsth1(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
       sc1:R:=(sign(c1))::R
       Pr1:UP(x,R):=pseudoRemainder(differentiate(p1)*p2,p1)
       Pr2:UP(x,R):=(Pr1 exquo c1**(q::NNI))::UP(x,R)
       c2:R:=leadingCoefficient(Pr2)
       r:NNI:=degree(Pr2)
```

```
    Pr3:UP(x,R):=monomial(sc1**((p-r-1)::NNI),0)*p1
    Pr4:UP(x,R):=monomial(sc1**((p-r-1)::NNI),0)*Pr2
    Listf:L UP(x,R):=[Pr3,Pr4]
    if r < p-1 then
      Pr5:UP(x,R):=monomial(delta((p-r-1)::NNI)*c2**((p-r-1)::NNI),0)*Pr2
      for j in ((r+1)::INT)..((p-2)::INT) repeat
        Listf:L UP(x,R):=append(Listf:L UP(x,R),[0]:L UP(x,R))
      Listf:L UP(x,R):=append(Listf:L UP(x,R),[Pr5]:L UP(x,R))
    if Pr1=0 then List1:L UP(x,R):=Listf
             else List1:L UP(x,R):=subresultantSequence(p1,Pr2)
    List2:L UP(x,R):=[]
    for j in 0..((r-1)::INT) repeat
      Pr6:UP(x,R):=monomial(delta((p-j-1)::NNI),0)*List1.((p-j+1)::NNI)
      List2:L UP(x,R):=append([Pr6]:L UP(x,R),List2:L UP(x,R))
    append(Listf:L UP(x,R),List2:L UP(x,R))

  polsth2(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
    sc1:R:=(sign(c1))::R
    Pr1:UP(x,R):=monomial(sc1,0)*p1
    Pr2:UP(x,R):=differentiate(p1)*p2
    Pr3:UP(x,R):=monomial(sc1,0)*Pr2
    Listf:L UP(x,R):=[Pr1,Pr3]
    List1:L UP(x,R):=subresultantSequence(p1,Pr2)
    List2:L UP(x,R):=[]
    for j in 0..((p-2)::INT) repeat
      Pr4:UP(x,R):=monomial(delta((p-j-1)::NNI),0)*List1.((p-j+1)::NNI)
      Pr5:UP(x,R):=(Pr4 exquo c1)::UP(x,R)
      List2:L UP(x,R):=append([Pr5]:L UP(x,R),List2:L UP(x,R))
    append(Listf:L UP(x,R),List2:L UP(x,R))

  polsth3(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
    sc1:R:=(sign(c1))::R
    q1:NNI:=(q-1)::NNI
    v:NNI:=(p+q1)::NNI
    Pr1:UP(x,R):=monomial(delta(q1::NNI)*sc1**((q+1)::NNI),0)*p1
    Listf:L UP(x,R):=[Pr1]
    List1:L UP(x,R):=subresultantSequence(differentiate(p1)*p2,p1)
    List2:L UP(x,R):=[]
    for j in 0..((p-1)::NNI) repeat
      Pr2:UP(x,R):=monomial(delta((v-j)::NNI),0)*List1.((v-j+1)::NNI)
      Pr3:UP(x,R):=(Pr2 exquo c1)::UP(x,R)
      List2:L UP(x,R):=append([Pr3]:L UP(x,R),List2:L UP(x,R))
    append(Listf:L UP(x,R),List2:L UP(x,R))

  SturmHabichtSequence(p1,p2):L UP(x,R) ==
    p:NNI:=degree(p1)
    q:NNI:=degree(p2)
    c1:R:=leadingCoefficient(p1)
    c1 = 1 or q = 1 => polsth1(p1,p,p2,q,c1)
    q = 0 => polsth2(p1,p,p2,q,c1)
```

```
       polsth3(p1,p,p2,q,c1)


-- Computation of the Sturm-Habicht principal coefficients of two
-- polynomials P and Q in R[x] where R is an ordered integral domain

     SturmHabichtCoefficients(p1,p2):L R ==
       List1:L UP(x,R):=SturmHabichtSequence(p1,p2)
--       List2:L R:=[]
       qp:NNI:=#(List1)::NNI
       [coefficient(p,(qp-j)::NNI) for p in List1 for j in 1..qp]
--       for j in 1..qp repeat
--         Ply:R:=coefficient(List1.j,(qp-j)::NNI)
--         List2:L R:=append(List2,[Ply])
--       List2


-- Computation of the number of sign variations of a list of non zero
-- elements in an ordered integral domain

     variation(Lsig:L R):INT ==
       size?(Lsig,1) => 0
       elt1:R:=first Lsig
       elt2:R:=Lsig.2
       sig1:R:=(sign(elt1*elt2))::R
       List1:L R:=rest Lsig
       sig1 = 1 => variation List1
       1+variation List1


-- Computation of the number of sign permanences of a list of non zero
-- elements in an ordered integral domain

     permanence(Lsig:L R):INT ==
       size?(Lsig,1) => 0
       elt1:R:=first Lsig
       elt2:R:=Lsig.2
       sig1:R:=(sign(elt1*elt2))::R
       List1:L R:=rest Lsig
       sig1 = -1 => permanence List1
       1+permanence List1


-- Computation of the functional W which works over a list of elements
-- in an ordered integral domain, with non zero first element

     qzeros(Lsig:L R):L R ==
       while last Lsig = 0 repeat
         Lsig:L R:=reverse rest reverse Lsig
       Lsig
```

```
    epsil(int1:NNI,elt1:R,elt2:R):INT ==
      int1 = 0 => 0
      odd? int1 => 0
      ct1:INT:=if elt1 > 0 then 1 else -1
      ct2:INT:=if elt2 > 0 then 1 else -1
      ct3:NNI:=(int1 exquo 2)::NNI
      ct4:INT:=(ct1*ct2)::INT
      ((-1)**(ct3::NNI))*ct4

    numbnce(Lsig:L R):NNI ==
      null Lsig => 0
      eltp:R:=Lsig.1
      eltp = 0 => 0
      1 + numbnce(rest Lsig)

    numbce(Lsig:L R):NNI ==
      null Lsig => 0
      eltp:R:=Lsig.1
      not(eltp = 0) => 0
      1 + numbce(rest Lsig)

    wfunctaux(Lsig:L R):INT ==
      null Lsig => 0
      List2:L R:=[]
      List1:L R:=Lsig:L R
      cont1:NNI:=numbnce(List1:L R)
      for j in 1..cont1 repeat
        List2:L R:=append(List2:L R,[first List1]:L R)
        List1:L R:=rest List1
      ind2:INT:=0
      cont2:NNI:=numbce(List1:L R)
      for j in 1..cont2 repeat
        List1:L R:=rest List1
        ind2:INT:=epsil(cont2:NNI,last List2,first List1)
      ind3:INT:=permanence(List2:L R)-variation(List2:L R)
      ind4:INT:=ind2+ind3
      ind4+wfunctaux(List1:L R)

    wfunct(Lsig:L R):INT ==
      List1:L R:=qzeros(Lsig:L R)
      wfunctaux(List1:L R)


-- Computation of the integer number:
--    #[{a in Rc(R)/P(a)=0 Q(a)>0}] - #[{a in Rc(R)/P(a)=0 Q(a)<0}]
-- where:
--    - R is an ordered integral domain,
--    - Rc(R) is the real clousure of R,
--    - P and Q are polynomials in R[x],
```

```
--    - by #[A] we note the cardinal of the set A

-- In particular:
--      - SturmHabicht(P,1) is the number of "real" roots of P,
--      - SturmHabicht(P,Q**2) is the number of "real" roots of P making Q neq 0

     SturmHabicht(p1,p2):INT ==
--     print("+" :: Ex)
       p2 = 0 => 0
       degree(p1:UP(x,R)) = 0 => 0
       List1:L UP(x,R):=SturmHabichtSequence(p1,p2)
       qp:NNI:=#(List1)::NNI
       wfunct [coefficient(p,(qp-j)::NNI) for p in List1 for j in 1..qp]

     countRealRoots(p1):INT == SturmHabicht(p1,1)

     if R has GcdDomain then
        SturmHabichtMultiple(p1,p2):INT ==
   --     print("+" :: Ex)
          p2 = 0 => 0
          degree(p1:UP(x,R)) = 0 => 0
          SH:L UP(x,R):=SturmHabichtSequence(p1,p2)
          qp:NNI:=#(SH)::NNI
          ans:= wfunct [coefficient(p,(qp-j)::NNI) for p in SH for j in 1..qp]
          SH:=reverse SH
          while first SH = 0 repeat SH:=rest SH
          degree first SH = 0 => ans
          -- OK: it probably wasn't square free, so this item is probably the
          -- gcd of p1 and p1'
          -- unless p1 and p2 have a factor in common (naughty!)
          differentiate(p1) exquo first SH case UP(x,R) =>
             -- it was the gcd of p1 and p1'
             ans+SturmHabichtMultiple(first SH,p2)
          sqfr:=factorList squareFree p1
          #sqfr = 1 and sqfr.first.xpnt=1 => ans
          reduce("+",[f.xpnt*SturmHabicht(f.fctr,p2) for f in sqfr])

        countRealRootsMultiple(p1):INT == SturmHabichtMultiple(p1,1)
```

———-

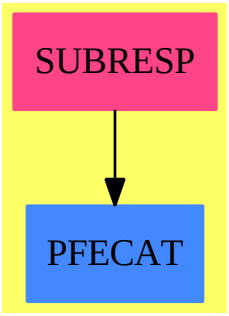
— SHP.dotabb —

```
"SHP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SHP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SHP" -> "PFECAT"
```

———-

20.26 package SUBRESP SubResultantPackage

20.26.1 SubResultantPackage (SUBRESP)



Exports:
primitivePart subresultantVector

— package SUBRESP SubResultantPackage —

```
)abbrev package SUBRESP SubResultantPackage
++ Author: Barry Trager, Renaud Rioboo
++ Date Created: 1987
++ Date Last Updated: August 2000
++ Description:
++ This package computes the subresultants of two polynomials which is needed
++ for the 'Lazard Rioboo' enhancement to Tragers integrations formula
++ For efficiency reasons this has been rewritten to call Lionel Ducos
++ package which is currently the best one.

SubResultantPackage(R, UP): Exports == Implementation where
  R : IntegralDomain
  UP: UnivariatePolynomialCategory R

  Z   ==> Integer
  N   ==> NonNegativeInteger

  Exports ==> with
    subresultantVector: (UP, UP) -> PrimitiveArray UP
      ++ subresultantVector(p, q) returns \spad{[p0,...,pn]}
      ++ where pi is the i-th subresultant of p and q.
      ++ In particular, \spad{p0 = resultant(p, q)}.
    if R has EuclideanDomain then
      primitivePart     : (UP,  R) -> UP
        ++ primitivePart(p, q) reduces the coefficient of p
        ++ modulo q, takes the primitive part of the result,
        ++ and ensures that the leading coefficient of that
```

```
       ++ result is monic.

  Implementation ==> add

    Lionel ==> PseudoRemainderSequence(R,UP)

    if R has EuclideanDomain then
      primitivePart(p, q) ==
         rec := extendedEuclidean(leadingCoefficient p, q,
                                  1)::Record(coef1:R, coef2:R)
         unitCanonical primitivePart map(x1 +-> (rec.coef1 * x1) rem q, p)

    subresultantVector(p1, p2) ==
       F : UP -- auxiliary stuff !
       res : PrimitiveArray(UP) := new(2+max(degree(p1),degree(p2)), 0)
       --
       -- kind of stupid interface to Lionel's  Package !!!!!!!!!!!!
       -- might have been wiser to rewrite the loop ...
       -- But I'm too lazy. [rr]
       --
       l := chainSubResultants(p1,p2)$Lionel
       --
       -- this returns the chain of non null subresultants !
       -- we must  rebuild subresultants from this.
       -- we really hope Lionel Ducos minded what he wrote
       -- since we are fully blind !
       --
       null l =>
         -- Hum it seems that Lionel returns [] when min(|p1|,|p2|) = 0
         zero?(degree(p1)) =>
           res.degree(p2) := p2
           if degree(p2) > 0
           then
             res.((degree(p2)-1)::NonNegativeInteger) := p1
             res.0 := (leadingCoefficient(p1)**(degree p2)) :: UP
           else
             -- both are of degree 0 the resultant is 1 according to Loos
             res.0 := 1
           res
         zero?(degree(p2)) =>
           if degree(p1) > 0
           then
             res.((degree(p1)-1)::NonNegativeInteger) := p2
             res.0 := (leadingCoefficient(p2)**(degree p1)) :: UP
           else
             -- both are of degree 0 the resultant is 1 according to Loos
             res.0 := 1
           res
         error "SUBRESP: strange Subresultant chain from PRS"
       Sn := first(l)
```

```
--
-- as of Loos definitions last subresultant should not be defective
--
l := rest(l)
n := degree(Sn)
F := Sn
null l => error "SUBRESP: strange Subresultant chain from PRS"
zero? Sn => error "SUBRESP: strange Subresultant chain from PRS"
while (l ^= []) repeat
  res.(n) := Sn
  F := first(l)
  l := rest(l)
  -- F is potentially defective
  if degree(F) = n
  then
    --
    -- F is defective
    --
    null l => error "SUBRESP: strange Subresultant chain from PRS"
    Sn := first(l)
    l := rest(l)
    n := degree(Sn)
    res.((n-1)::NonNegativeInteger) := F
  else
    --
    -- F is non defective
    --
    degree(F) < n => error "strange result !"
    Sn := F
    n := degree(Sn)
--
-- Lionel forgets about p1 if |p1| > |p2|
-- forgets about p2 if |p2| > |p1|
-- but he reminds p2 if |p1| = |p2|
-- a glance at Loos should correct this !
--
res.n := Sn
--
-- Loos definition
--
if degree(p1) = degree(p2)
then
  res.((degree p1)+1) := p1
else
  if degree(p1) > degree(p2)
  then
    res.(degree p1) := p1
  else
    res.(degree p2) := p2
res
```

———-

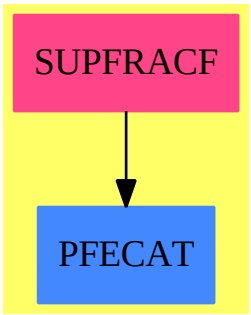
— **SUBRESP.dotabb** —

```
"SUBRESP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUBRESP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUBRESP" -> "PFECAT"
```

———-

20.27 package SUPFRACF SupFractionFactorizer

20.27.1 SupFractionFactorizer (SUPFRACF)



Exports:
factor squareFree

— **package SUPFRACF SupFractionFactorizer** —

```
)abbrev package SUPFRACF SupFractionFactorizer
++ Author: P. Gianni
++ Date Created: October 1993
++ Date Last Updated: March 1995
++ Basic Functions:
++ Related Constructors: MultivariateFactorize
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ SupFractionFactorize  contains the factor function for univariate
```

```
++ polynomials over the quotient field of a ring S such that the package
++ MultivariateFactorize works for S

SupFractionFactorizer(E,OV,R,P) : C == T
 where
  E   :   OrderedAbelianMonoidSup
  OV  :   OrderedSet
  R   :   GcdDomain
  P   :   PolynomialCategory(R,E,OV)
  FP  ==> Fraction P
  SUP ==> SparseUnivariatePolynomial

  C  == with
     factor      : SUP FP   ->  Factored SUP FP
       ++ factor(p) factors the univariate polynomial p with coefficients
       ++ which are fractions of polynomials over R.
     squareFree  : SUP FP   ->  Factored SUP FP
       ++ squareFree(p) returns the square-free factorization of the univariate polynomial p with coefficie
       ++ which are fractions of polynomials over R. Each factor has no repeated roots and the factors are
       ++ pairwise relatively prime.

  T  == add
     MFACT  ==> MultivariateFactorize(OV,E,R,P)
     MSQFR  ==> MultivariateSquareFree(E,OV,R,P)
     UPCF2  ==> UnivariatePolynomialCategoryFunctions2

     factor(p:SUP FP) : Factored SUP FP  ==
       p=0 => 0
       R has CharacteristicZero and R has EuclideanDomain =>
         pden : P := lcm [denom c for c in coefficients p]
         pol  : SUP FP := (pden::FP)*p
         ipol: SUP P := map(numer,pol)$UPCF2(FP,SUP FP,P,SUP P)
         ffact: Factored SUP P := 0
         ffact := factor(ipol)$MFACT
         makeFR((1/pden * map(coerce,unit ffact)$UPCF2(P,SUP P,FP,SUP FP)),
         [["prime",map(coerce,u.factor)$UPCF2(P,SUP P,FP,SUP FP),
            u.exponent] for u in factors ffact])
       squareFree p

     squareFree(p:SUP FP) : Factored SUP FP  ==
       p=0 => 0
       pden : P := lcm [denom c for c in coefficients p]
       pol  : SUP FP := (pden::FP)*p
       ipol: SUP P := map(numer,pol)$UPCF2(FP,SUP FP,P,SUP P)
       ffact: Factored SUP P := 0
       if R has CharacteristicZero and R has EuclideanDomain then
         ffact := squareFree(ipol)$MSQFR
       else ffact := squareFree(ipol)
       makeFR((1/pden * map(coerce,unit ffact)$UPCF2(P,SUP P,FP,SUP FP)),
         [["sqfr",map(coerce,u.factor)$UPCF2(P,SUP P,FP,SUP FP),
```

```
            u.exponent] for u in factors ffact])
```

———-

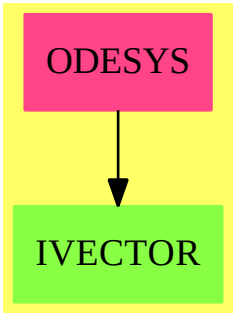
— **SUPFRACF.dotabb** —

```
"SUPFRACF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUPFRACF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUPFRACF" -> "PFECAT"
```

———-

20.28 package ODESYS SystemODESolver

20.28.1 SystemODESolver (ODESYS)



Exports:
solve solveInField triangulate

— **package ODESYS SystemODESolver** —

```
)abbrev package ODESYS SystemODESolver
++ Author: Manuel Bronstein
++ Date Created: 11 June 1991
++ Date Last Updated: 13 April 1994
++ Keywords: differential equation, ODE, system
++ Description:
++ SystemODESolver provides tools for triangulating
++ and solving some systems of linear ordinary differential equations.

SystemODESolver(F, LO): Exports == Implementation where
  F : Field
```

```
LO: LinearOrdinaryDifferentialOperatorCategory F

N   ==> NonNegativeInteger
Z   ==> Integer
MF  ==> Matrix F
M   ==> Matrix LO
V   ==> Vector F
UF  ==> Union(F, "failed")
UV  ==> Union(V, "failed")
REC ==> Record(mat: M, vec: V)
FSL ==> Record(particular: UF, basis: List F)
VSL ==> Record(particular: UV, basis: List V)
SOL ==> Record(particular: F, basis: List F)
USL ==> Union(SOL, "failed")
ER  ==> Record(C: MF, g: V, eq: LO, rh: F)

Exports ==> with
  triangulate: (MF, V) -> Record(A:MF, eqs: List ER)
    ++ triangulate(M,v) returns
    ++ \spad{A,[[C_1,g_1,L_1,h_1],...,[C_k,g_k,L_k,h_k]]}
    ++ such that under the change of variable \spad{y = A z}, the first
    ++ order linear system \spad{D y = M y + v} is uncoupled as
    ++ \spad{D z_i = C_i z_i + g_i} and each \spad{C_i} is a companion
    ++ matrix corresponding to the scalar equation \spad{L_i z_j = h_i}.
  triangulate: (M, V) -> REC
    ++ triangulate(m, v) returns \spad{[m_0, v_0]} such that \spad{m_0}
    ++ is upper triangular and the system \spad{m_0 x = v_0} is equivalent
    ++ to \spad{m x = v}.
  solve: (MF,V,(LO,F)->USL) -> Union(Record(particular:V, basis:MF),"failed")
    ++ solve(m, v, solve) returns \spad{[[v_1,...,v_m], v_p]} such that
    ++ the solutions in \spad{F} of the system \spad{D x = m x + v} are
    ++ \spad{v_p + c_1 v_1 + ... + c_m v_m} where the \spad{c_i's} are
    ++ constants, and the \spad{v_i's} form a basis for the solutions of
    ++ \spad{D x = m x}.
    ++ Argument \spad{solve} is a function for solving a single linear
    ++ ordinary differential equation in \spad{F}.
  solveInField: (M, V, (LO, F) -> FSL) -> VSL
    ++ solveInField(m, v, solve) returns \spad{[[v_1,...,v_m], v_p]} such
    ++ that the solutions in \spad{F} of the system \spad{m x = v} are
    ++ \spad{v_p + c_1 v_1 + ... + c_m v_m} where the \spad{c_i's} are
    ++ constants, and the \spad{v_i's} form a basis for the solutions of
    ++ \spad{m x = 0}.
    ++ Argument \spad{solve} is a function for solving a single linear
    ++ ordinary differential equation in \spad{F}.

Implementation ==> add
  import PseudoLinearNormalForm F

  applyLodo   : (M, Z, V, N) -> F
  applyLodo0  : (M, Z, Matrix F, Z, N) -> F
```

```
backsolve   : (M, V, (LO, F) -> FSL) -> VSL
firstnonzero: (M, Z) -> Z
FSL2USL     : FSL -> USL
M2F         : M -> Union(MF, "failed")

diff := D()$LO

solve(mm, v, solve) ==
  rec  := triangulate(mm, v)
  sols:List(SOL) := empty()
  for e in rec.eqs repeat
      (u := solve(e.eq, e.rh)) case "failed" => return "failed"
      sols := concat(u::SOL, sols)
  n := nrows(rec.A)    -- dimension of original vectorspace
  k:N := 0             -- sum of sizes of visited companionblocks
  i:N := 0             -- number of companionblocks
  m:N := 0             -- number of Solutions
  part:V := new(n, 0)
  -- count first the different solutions
  for sol in sols repeat
    m := m + count((f1:F):Boolean +-> f1 ^= 0, sol.basis)$List(F)
  SolMatrix:MF := new(n, m, 0)
  m := 0
  for sol in reverse_! sols repeat
      i := i+1
      er := rec.eqs.i
      nn := #(er.g)           -- size of active companionblock
      for s in sol.basis repeat
          solVec:V := new(n, 0)
          -- compute corresponding solution base with recursion (24)
          solVec(k+1) := s
          for l in 2..nn repeat solVec(k+l) := diff solVec(k+l-1)
          m := m+1
          setColumn!(SolMatrix, m, solVec)
      -- compute with (24) the corresponding components of the part. sol.
      part(k+1) := sol.particular
      for l in 2..nn repeat part(k+l) := diff part(k+l-1) - (er.g)(l-1)
      k := k+nn
  -- transform these values back to the original system
  [rec.A * part, rec.A * SolMatrix]

triangulate(m:MF, v:V) ==
  k:N := 0       -- sum of companion-dimensions
  rat := normalForm(m, 1, (f1:F):F +-> - diff f1)
  l   := companionBlocks(rat.R, rat.Ainv * v)
  ler:List(ER) := empty()
  for er in l repeat
    n := nrows(er.C)         -- dimension of this companion vectorspace
    op:LO := 0               -- compute homogeneous equation
    for j in 0..n-1 repeat op := op + monomial((er.C)(n, j + 1), j)
```

```
        op := monomial(1, n) - op
        sum:V := new(n::N, 0)    -- compute inhomogen Vector (25)
        for j in 1..n-1 repeat sum(j+1) := diff(sum j) + (er.g) j
        h0:F := 0                 -- compute inhomogenity (26)
        for j in 1..n repeat h0 := h0 - (er.C)(n, j) * sum j
        h0 := h0 + diff(sum n) + (er.g) n
        ler := concat([er.C, er.g, op, h0], ler)
        k := k + n
      [rat.A, ler]

-- like solveInField, but expects a system already triangularized
    backsolve(m, v, solve) ==
      part:V
      r := maxRowIndex m
      offset := minIndex v - (mr := minRowIndex m)
      while r >= mr and every?(zero?, row(m, r))$Vector(LO) repeat r := r - 1
      r < mr => error "backsolve: system has a 0 matrix"
      (c := firstnonzero(m, r)) ^= maxColIndex m =>
        error "backsolve: undetermined system"
      rec := solve(m(r, c), v(r + offset))
      dim := (r - mr + 1)::N
      if (part? := ((u := rec.particular) case F)) then
        part := new(dim, 0)                          -- particular solution
        part(r + offset) :=  u::F
-- hom is the basis for the homogeneous solutions, each column is a solution
      hom:Matrix(F) := new(dim, #(rec.basis), 0)
      for i in minColIndex hom .. maxColIndex hom for b in rec.basis repeat
        hom(r, i) := b
      n:N := 1                -- number of equations already solved
      while r > mr repeat
        r := r - 1
        c := c - 1
        firstnonzero(m, r) ^= c => error "backsolve: undetermined system"
        degree(eq := m(r, c)) > 0 => error "backsolve: pivot of order > 0"
        a := leadingCoefficient(eq)::F
        if part? then
           part(r + offset) := (v(r + offset) - applyLodo(m, r, part, n)) / a
        for i in minColIndex hom .. maxColIndex hom repeat
          hom(r, i) := - applyLodo0(m, r, hom, i, n)
        n := n + 1
      bas:List(V) := [column(hom,i) for i in minColIndex hom..maxColIndex hom]
      part? => [part, bas]
      ["failed", bas]

    solveInField(m, v, solve) ==
      ((n := nrows m) = ncols m) and
         ((u := M2F(diagonalMatrix [diff for i in 1..n] - m)) case MF) =>
             (uu := solve(u::MF, v,
               (l1:LO,f2:F):USL +-> FSL2USL solve(l1, f2))) case "failed" =>
                  ["failed", empty()]
```

```
            rc := uu::Record(particular:V, basis:MF)
            [rc.particular, [column(rc.basis, i) for i in 1..ncols(rc.basis)]]
      rec := triangulate(m, v)
      backsolve(rec.mat, rec.vec, solve)

    M2F m ==
        mf:MF := new(nrows m, ncols m, 0)
        for i in minRowIndex m .. maxRowIndex m repeat
            for j in minColIndex m .. maxColIndex m repeat
                (u := retractIfCan(m(i, j))@Union(F, "failed")) case "failed" =>
                     return "failed"
                mf(i, j) := u::F
        mf

    FSL2USL rec ==
        rec.particular case "failed" => "failed"
        [rec.particular::F, rec.basis]

-- returns the index of the first nonzero entry in row r of m
    firstnonzero(m, r) ==
      for c in minColIndex m .. maxColIndex m repeat
        m(r, c) ^= 0 => return c
      error "firstnonzero: zero row"

-- computes +/[m(r, i) v(i) for i ranging over the last n columns of m]
    applyLodo(m, r, v, n) ==
      ans:F := 0
      c := maxColIndex m
      cv := maxIndex v
      for i in 1..n repeat
        ans := ans + m(r, c) (v cv)
        c := c - 1
        cv := cv - 1
      ans

-- computes +/[m(r, i) mm(i, c) for i ranging over the last n columns of m]
    applyLodo0(m, r, mm, c, n) ==
      ans := 0
      rr := maxRowIndex mm
      cc := maxColIndex m
      for i in 1..n repeat
        ans := ans + m(r, cc) mm(rr, c)
        cc := cc - 1
        rr := rr - 1
      ans

    triangulate(m:M, v:V) ==
      x := copy m
      w := copy v
      nrows := maxRowIndex x
```

```
      ncols := maxColIndex x
      minr  := i := minRowIndex x
      offset := minIndex w - minr
      for j in minColIndex x .. ncols repeat
        if i > nrows then leave x
        rown := minr - 1
        for k in i .. nrows repeat
          if (x(k, j) ^= 0) and ((rown = minr - 1) or
                              degree x(k,j) < degree x(rown,j)) then rown := k
          rown = minr - 1 => "enuf"
          x := swapRows_!(x, i, rown)
          swap_!(w, i + offset, rown + offset)
        for k in i+1 .. nrows | x(k, j) ^= 0 repeat
          l := rightLcm(x(i,j), x(k,j))
          a := rightQuotient(l, x(i, j))
          b := rightQuotient(l, x(k, j))
          -- l = a x(i,j) = b x(k,j)
          for k1 in j+1 .. ncols repeat
            x(k, k1) :=  a * x(i, k1) - b * x(k, k1)
          x(k, j) := 0
          w(k + offset) := a(w(i + offset)) - b(w(k + offset))
        i := i+1
      [x, w]
```

———-

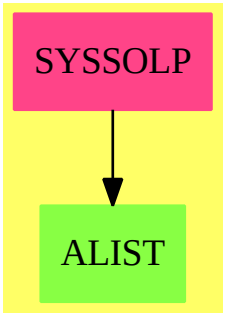
— **ODESYS.dotabb** —

```
"ODESYS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODESYS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ODESYS" -> "IVECTOR"
```

———-

20.29 package SYSSOLP SystemSolvePackage

20.29.1 SystemSolvePackage (SYSSOLP)



Exports:
solve triangularSystems

— package SYSSOLP SystemSolvePackage —

```
)abbrev package SYSSOLP SystemSolvePackage
++ Author: P. Gianni
++ Date Created: summer 1988
++ Date Last Updated: summer 1990
++ Basic Functions:
++ Related Constructors: Fraction, Polynomial, FloatingRealPackage,
++ FloatingComplexPackage, RadicalSolvePackage
++ Also See: LinearSystemMatrixPackage, GroebnerSolve
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Symbolic solver for systems of rational functions with coefficients
++ in an integral domain R.
++ The systems are solved in the field of rational functions over R.
++ Solutions are exact of the form variable = value when the value is
++ a member of the coefficient domain R. Otherwise the solutions
++ are implicitly expressed as roots of univariate polynomial equations over R.
++ Care is taken to guarantee that the denominators of the input
++ equations do not vanish on the solution sets.
++ The arguments to solve can either be given as equations or
++ as rational functions interpreted as equal
++ to zero. The user can specify an explicit list of symbols to
++ be solved for, treating all other symbols appearing as parameters
++ or omit the list of symbols in which case the system tries to
++ solve with respect to all symbols appearing in the input.
```

```
SystemSolvePackage(R): Cat == Cap where
    NNI      ==> NonNegativeInteger
    P        ==> Polynomial
    EQ       ==> Equation
    L        ==> List
    V        ==> Vector
    M        ==> Matrix
    UP       ==> SparseUnivariatePolynomial
    SE       ==> Symbol
    IE       ==> IndexedExponents Symbol
    SUP      ==> SparseUnivariatePolynomial
    R : IntegralDomain
    F   ==> Fraction Polynomial R
    PP2 ==> PolynomialFunctions2(R,F)
    PPR ==> Polynomial Polynomial R

    Cat == with
       solve:    (L F,    L SE) -> L L EQ F
         ++ solve(lp,lv) finds the solutions of the list lp of
         ++ rational functions with respect to the list of symbols lv.

       solve:    (L EQ F, L SE) -> L L EQ F
         ++ solve(le,lv) finds the solutions of the
         ++ list le of equations of rational functions
         ++ with respect to the list of symbols lv.

       solve:         L F       -> L L EQ F
         ++ solve(lp) finds the solutions of the list lp of rational
         ++ functions with respect to all symbols appearing in lp.

       solve:        L EQ F     -> L L EQ F
         ++ solve(le) finds the solutions of the list le of equations of
         ++ rational functions with respect to all symbols appearing in le.

       solve:    (F, SE) ->  L EQ F
         ++ solve(p,v) solves the equation p=0, where p is a rational function
         ++ with respect to the variable v.

       solve:    (EQ F,SE) -> L EQ F
         ++ solve(eq,v) finds the solutions of the equation
         ++ eq with respect to the variable v.

       solve:         F      -> L EQ F
         ++ solve(p) finds the solution of a rational function p = 0
         ++ with respect to the unique variable appearing in p.

       solve:         EQ F     -> L EQ F
         ++ solve(eq) finds the solutions of the equation eq
         ++ with respect to the unique variable appearing in eq.
```

```
    triangularSystems: (L F,    L SE) -> L L P R
      ++ triangularSystems(lf,lv) solves the system of equations
      ++ defined by lf with respect to the list of symbols lv;
      ++ the system of equations is obtaining
      ++ by equating to zero the list of rational functions lf.
      ++ The output is a list of solutions where
      ++ each solution is expressed as a "reduced" triangular system of
      ++ polynomials.

  Cap == add

    import MPolyCatRationalFunctionFactorizer(IE,SE,R,P F)

                  ---- Local Functions ----
    linSolve: (L F,    L SE) -> Union(L EQ F, "failed")
    makePolys :   L EQ F     ->  L F

    makeR2F(r : R) : F == r :: (P R) :: F

    makeP2F(p:P F):F ==
      lv:=variables p
      lv = [] => retract p
      for v in lv repeat p:=pushdown(p,v)
      retract p
                  ---- Local Functions ----
    makeEq(p:P F,lv:L SE): EQ F ==
      z:=last lv
      np:=numer makeP2F p
      lx:=variables np
      for x in lv repeat if member?(x,lx) then leave x
      up:=univariate(np,x)
      (degree up)=1 =>
        equation(x::P(R)::F,-coefficient(up,0)/leadingCoefficient up)
      equation(np::F,0$F)

    varInF(v: SE): F == v::P(R) :: F

    newInF(n: Integer):F==varInF new()$SE

    testDegree(f :P R , lv :L SE) : Boolean ==
      "or"/[degree(f,vv)>0 for vv in lv]
                ---- Exported Functions ----

    -- solve a system of rational functions
    triangularSystems(lf: L F,lv:L SE) : L L P R ==
        empty? lv => empty()
        empty? lf => empty()
        #lf = 1 =>
           p:= numer(first lf)
```

```
        fp:=(factor p)$GeneralizedMultivariateFactorize(SE,IE,R,R,P R)
        [[ff.factor] for ff in factors fp | testDegree(ff.factor,lv)]
      dmp:=DistributedMultivariatePolynomial(lv,P R)
      OV:=OrderedVariableList(lv)
      DP:=DirectProduct(#lv, NonNegativeInteger)
      push:=PushVariables(R,DP,OV,dmp)
      lq : L dmp
      lvv:L OV:=[variable(vv)::OV for vv in lv]
      lq:=[pushup(df::dmp,lvv)$push for f in lf|(df:=denom f)^=1]
      lp:=[pushup(numer(f)::dmp,lvv)$push for f in lf]
      parRes:=groebSolve(lp,lvv)$GroebnerSolve(lv,P R,R)
      if lq^=[] then
        gb:=GroebnerInternalPackage(P R,DirectProduct(#lv,NNI),OV,dmp)
        parRes:=[pr for pr in parRes|
                  and/[(redPol(fq,pr pretend List(dmp))$gb) ^=0
                    for fq in lq]]
      [[retract pushdown(pf,lvv)$push for pf in pr] for pr in parRes]

  -- One polynomial. Implicit variable --
  solve(pol : F) ==
    zero? pol =>
       error "equation is always satisfied"
    lv:=removeDuplicates
        concat(variables numer pol, variables denom pol)
    empty? lv => error "inconsistent equation"
    #lv>1 => error "too many variables"
    solve(pol,first lv)

  -- general solver. Input in equation style. Implicit variables --
  solve(eq : EQ F) ==
    pol:= lhs eq - rhs eq
    zero? pol =>
       error "equation is always satisfied"
    lv:=removeDuplicates
        concat(variables numer pol, variables denom pol)
    empty? lv => error "inconsistent equation"
    #lv>1 => error "too many variables"
    solve(pol,first lv)

  -- general solver. Input in equation style  --
  solve(eq:EQ F,var:SE)  == solve(lhs eq - rhs eq,var)

  -- general solver. Input in polynomial style  --
  solve(pol:F,var:SE) ==
    if R has GcdDomain then
      p:=primitivePart(numer pol,var)
      fp:=(factor p)$GeneralizedMultivariateFactorize(SE,IE,R,R,P R)
      [makeEq(map(makeR2F,ff.factor)$PP2,[var]) for ff in factors fp]
    else empty()
```

```
-- Convert a list of Equations in a list of Polynomials
makePolys(l: L EQ F):L F == [lhs e - rhs e for e in l]

-- linear systems solver. Input as list of polynomials  --
linSolve(lp:L F,lv:L SE) ==
    rec:Record(particular:Union(V F,"failed"),basis:L V F)
    lr : L P R:=[numer f for f in lp]
    rec:=linSolve(lr,lv)$LinearSystemPolynomialPackage(R,IE,SE,P R)
    rec.particular case "failed" => "failed"
    rhs := rec.particular :: V F
    zeron:V F:=zero(#lv)
    for p in rec.basis | p ^= zeron repeat
        sym := newInF(1)
        for i in 1..#lv repeat
            rhs.i := rhs.i + sym*p.i
    eqs: L EQ F := []
    for i in 1..#lv repeat
      eqs := append(eqs,[(lv.i)::(P R)::F = rhs.i])
    eqs

-- general solver. Input in polynomial style. Implicit variables --
solve(lr : L F) ==
  lv :="setUnion"/[setUnion(variables numer p, variables denom p)
        for p in lr]
  solve(lr,lv)

-- general solver. Input in equation style. Implicit variables --
solve(le : L EQ F) ==
  lr:=makePolys le
  lv :="setUnion"/[setUnion(variables numer p, variables denom p)
        for p in lr]
  solve(lr,lv)

-- general solver. Input in equation style  --
solve(le:L EQ F,lv:L SE)  == solve(makePolys le, lv)

checkLinear(lr:L F,vl:L SE):Boolean ==
  ld:=[denom pol for pol in lr]
  for f in ld repeat
    if (or/[member?(x,vl) for x in variables f]) then return false
  and/[totalDegree(numer pol,vl) < 2 for pol in lr]

-- general solver. Input in polynomial style  --
solve(lr:L F,vl:L SE) ==
    empty? vl => empty()
    checkLinear(lr,vl) =>
                    -- linear system --
        soln := linSolve(lr, vl)
        soln case "failed" => []
        eqns: L EQ F := []
```

```
        for i in 1..#vl repeat
            lhs := (vl.i::(P R))::F
            rhs :=  rhs soln.i
            eqns := append(eqns, [lhs = rhs])
        [eqns]

                    -- polynomial system --
    if R has GcdDomain then
      parRes:=triangularSystems(lr,vl)
      [[makeEq(map(makeR2F,f)$PP2,vl) for f in pr]
                                                for pr in parRes]
    else [[]]
```

———-

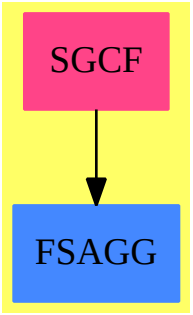
— **SYSSOLP.dotabb** —

```
"SYSSOLP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SYSSOLP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"SYSSOLP" -> "ALIST"
```

———-

20.30 package SGCF SymmetricGroupCombinatoricFunctions

20.30.1 SymmetricGroupCombinatoricFunctions (SGCF)



Exports:

coleman	inverseColeman
listYoungTableaus	makeYoungTableau
nextColeman	nextLatticePermutation
nextPartition	numberOfImproperPartitions
subSet	unrankImproperPartitions0
unrankImproperPartitions1	

— package SGCF SymmetricGroupCombinatoricFunctions —

```
)abbrev package SGCF SymmetricGroupCombinatoricFunctions
++ Authors: Johannes Grabmeier, Thorsten Werther
++ Date Created: 03 September 1988
++ Date Last Updated: 07 June 1990
++ Basic Operations: nextPartition, numberOfImproperPartitions,
++   listYoungTableaus, subSet, unrankImproperPartitions0
++ Related Constructors: IntegerCombinatoricFunctions
++ Also See: RepresentationTheoryPackage1, RepresentationTheoryPackage2,
++   IrrRepSymNatPackage
++ AMS Classifications:
++ Keywords: improper partition, partition, subset, Coleman
++ References:
++   G. James/ A. Kerber: The Representation Theory of the Symmetric
++    Group. Encycl. of Math. and its Appl., Vol. 16., Cambridge
++    Univ. Press 1981, ISBN 0-521-30236-6.
++   S.G. Williamson: Combinatorics for Computer Science,
++    Computer Science Press, Rockville, Maryland, USA, ISBN 0-88175-020-4.
++   A. Nijenhuis / H.S. Wilf: Combinatoral Algorithms, Academic Press 1978.
++    ISBN 0-12-519260-6.
++   H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++    their Realization in the Computer Algebra System Scratchpad,
++    Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23.
++ Description:
++ SymmetricGroupCombinatoricFunctions contains combinatoric
++ functions concerning symmetric groups and representation
++ theory: list young tableaus, improper partitions, subsets
++ bijection of Coleman.

SymmetricGroupCombinatoricFunctions(): public == private where

  NNI  ==> NonNegativeInteger
  I    ==> Integer
  L    ==> List
  M    ==> Matrix
  V    ==> Vector
  B    ==> Boolean
  ICF  ==> IntegerCombinatoricFunctions Integer

  public ==> with

--    IS THERE A WORKING DOMAIN Tableau ??
```

```
--    coerce : M I -> Tableau(I)
--      ++ coerce(ytab) coerces the Young-Tableau ytab to an element of
--      ++ the domain Tableau(I).

    coleman : (L I, L I, L I) -> M I
      ++ coleman(alpha,beta,pi):
      ++ there is a bijection from the set of matrices having nonnegative
      ++ entries and row sums alpha, column sums beta
      ++ to the set of Salpha - Sbeta double cosets of the
      ++ symmetric group Sn. (Salpha is the Young subgroup
      ++ corresponding to the improper partition alpha).
      ++ For a representing element pi of such a double coset,
      ++ coleman(alpha,beta,pi) generates the Coleman-matrix
      ++ corresponding to alpha, beta, pi.
      ++ Note that The permutation pi of {1,2,...,n} has to be given
      ++ in list form.
      ++ Note that the inverse of this map is inverseColeman
      ++ (if pi is the lexicographical smallest permutation
      ++ in the coset). For details see James/Kerber.
    inverseColeman : (L I, L I, M I) -> L I
      ++ inverseColeman(alpha,beta,C):
      ++ there is a bijection from the set of matrices having nonnegative
      ++ entries and row sums alpha, column sums beta
      ++ to the set of Salpha - Sbeta double cosets of the
      ++ symmetric group Sn. (Salpha is the Young subgroup
      ++ corresponding to the improper partition alpha).
      ++ For such a matrix C, inverseColeman(alpha,beta,C)
      ++ calculates the lexicographical smallest pi in the
      ++ corresponding double coset.
      ++ Note that the resulting permutation pi of {1,2,...,n}
      ++ is given in list form.
      ++ Notes: the inverse of this map is coleman.
      ++ For details, see James/Kerber.
    listYoungTableaus : (L I) -> L M I
      ++ listYoungTableaus(lambda) where lambda is a proper partition
      ++ generates the list of all standard tableaus of shape lambda
      ++ by means of lattice permutations. The numbers of the lattice
      ++ permutation are interpreted as column labels. Hence the
      ++ contents of these lattice permutations are the conjugate of
      ++ lambda.
      ++ Notes: the functions nextLatticePermutation and
      ++ makeYoungTableau are used.
      ++ The entries are from 0,...,n-1.
    makeYoungTableau : (L I,L I) -> M I
      ++ makeYoungTableau(lambda,gitter) computes for a given lattice
      ++ permutation gitter and for an improper partition lambda
      ++ the corresponding standard tableau of shape lambda.
      ++ Notes: see listYoungTableaus.
      ++ The entries are from 0,...,n-1.
    nextColeman : (L I, L I, M I) -> M I
```

```
  ++ nextColeman(alpha,beta,C) generates the next Coleman matrix
  ++ of column sums alpha and row sums beta according
  ++ to the lexicographical order from bottom-to-top.
  ++ The first Coleman matrix is achieved by C=new(1,1,0).
  ++ Also, new(1,1,0) indicates that C is the last Coleman matrix.
nextLatticePermutation : (L I, L I, B) -> L I
  ++ nextLatticePermutation(lambda,lattP,constructNotFirst) generates
  ++ the lattice permutation according to the proper partition
  ++ lambda succeeding the lattice permutation lattP in
  ++ lexicographical order as long as constructNotFirst is true.
  ++ If constructNotFirst is false, the first lattice permutation
  ++ is returned.
  ++ The result nil indicates that lattP has no successor.
nextPartition : (V I, V I, I) -> V I
  ++ nextPartition(gamma,part,number) generates the partition of
  ++ number which follows part according to the right-to-left
  ++ lexicographical order. The partition has the property that
  ++ its components do not exceed the corresponding components of
  ++ gamma. The first partition is achieved by part=[].
  ++ Also, [] indicates that part is the last partition.
nextPartition : (L I, V I, I) -> V I
  ++ nextPartition(gamma,part,number) generates the partition of
  ++ number which follows part according to the right-to-left
  ++ lexicographical order. The partition has the property that
  ++ its components do not exceed the corresponding components of
  ++ gamma. the first partition is achieved by part=[].
  ++ Also, [] indicates that part is the last partition.
numberOfImproperPartitions: (I,I) -> I
  ++ numberOfImproperPartitions(n,m) computes the number of partitions
  ++ of the nonnegative integer n in m nonnegative parts with regarding
  ++ the order (improper partitions).
  ++ Example: numberOfImproperPartitions (3,3) is 10,
  ++ since [0,0,3], [0,1,2], [0,2,1], [0,3,0], [1,0,2], [1,1,1],
  ++ [1,2,0], [2,0,1], [2,1,0], [3,0,0] are the possibilities.
  ++ Note that this operation has a recursive implementation.
subSet : (I,I,I) -> L I
  ++ subSet(n,m,k) calculates the k-th m-subset of the set
  ++ 0,1,...,(n-1) in the lexicographic order considered as
  ++ a decreasing map from 0,...,(m-1) into 0,...,(n-1).
  ++ See S.G. Williamson: Theorem 1.60.
  ++ Error: if not (0 <= m <= n and 0 < = k < (n choose m)).
unrankImproperPartitions0 : (I,I,I) -> L I
  ++ unrankImproperPartitions0(n,m,k) computes the k-th improper
  ++ partition of nonnegative n in m nonnegative parts in reverse
  ++ lexicographical order.
  ++ Example: [0,0,3] < [0,1,2] < [0,2,1] < [0,3,0] <
  ++ [1,0,2] < [1,1,1] < [1,2,0] < [2,0,1] < [2,1,0] < [3,0,0].
  ++ Error: if k is negative or too big.
  ++ Note that counting of subtrees is done by
  ++ numberOfImproperPartitions
```

```
  unrankImproperPartitions1: (I,I,I) -> L I
    ++ unrankImproperPartitions1(n,m,k) computes the k-th improper
    ++ partition of nonnegative n in at most m nonnegative parts
    ++ ordered as follows: first, in reverse
    ++ lexicographically according to their non-zero parts, then
    ++ according to their positions (i.e. lexicographical order
    ++ using subSet: [3,0,0] < [0,3,0] < [0,0,3] < [2,1,0] <
    ++ [2,0,1] < [0,2,1] < [1,2,0] < [1,0,2] < [0,1,2] < [1,1,1].
    ++ Note that counting of subtrees is done by
    ++ numberOfImproperPartitionsInternal.

private == add

  import Set I

  -- declaration of local functions


  numberOfImproperPartitionsInternal: (I,I,I) -> I
    -- this is used as subtree counting function in
    -- "unrankImproperPartitions1". For (n,m,cm) it counts
    -- the following set of m-tuples: The  first (from left
    -- to right) m-cm non-zero entries are equal, the remaining
    -- positions sum up to n. Example: (3,3,2) counts
    -- [x,3,0], [x,0,3], [0,x,3], [x,2,1], [x,1,2], x non-zero.


  -- definition of local functions


  numberOfImproperPartitionsInternal(n,m,cm) ==
    n = 0 => binomial(m,cm)$ICF
    cm = 0 and n > 0 => 0
    s := 0
    for i in 0..n-1 repeat
      s := s + numberOfImproperPartitionsInternal(i,m,cm-1)
    s


  -- definition of exported functions

  numberOfImproperPartitions(n,m) ==
    if n < 0 or m < 1 then return 0
    if m = 1 or n = 0 then return 1
    s := 0
    for i in 0..n repeat
      s := s + numberOfImproperPartitions(n-i,m-1)
    s
```

```
unrankImproperPartitions0(n,m,k) ==
  l : L I  := nil$(L I)
  k < 0 => error"counting of partitions is started at 0"
  k >= numberOfImproperPartitions(n,m) =>
    error"there are not so many partitions"
  for t in 0..(m-2) repeat
    s : I := 0
    for y in 0..n repeat
      sOld := s
      s := s + numberOfImproperPartitions(n-y,m-t-1)
      if s > k then leave
    l := append(l,list(y)$(L I))$(L I)
    k := k - sOld
    n := n - y
  l := append(l,list(n)$(L I))$(L I)
  l


unrankImproperPartitions1(n,m,k) ==
  -- we use the counting procedure of the leaves in a tree
  -- having the following structure: First of all non-zero
  -- labels for the sons. If addition along a path gives n,
  -- then we go on creating the subtree for (n choose cm)
  -- where cm is the length of the path. These subsets determine
  -- the positions for the non-zero labels for the partition
  -- to be formeded. The remaining positions are filled by zeros.
  nonZeros   : L I := nil$(L I)
  partition  : V I :=  new(m::NNI,0$I)$(V I)
  k < 0 => nonZeros
  k >= numberOfImproperPartitions(n,m) => nonZeros
  cm : I := m    --cm gives the depth of the tree
  while n ^= 0 repeat
    s : I := 0
    cm := cm - 1
    for y in n..1 by -1 repeat   --determination of the next son
      sOld := s  -- remember old s
      -- this functions counts the number of elements in a subtree
      s := s + numberOfImproperPartitionsInternal(n-y,m,cm)
      if s > k then leave
    -- y is the next son, so put it into the pathlist "nonZero"
    nonZeros := append(nonZeros,list(y)$(L I))$(L I)
    k := k - sOld    --updating
    n := n - y       --updating
  --having found all m-cm non-zero entries we change the structure
  --of the tree and determine the non-zero positions
  nonZeroPos : L I := reverse subSet(m,m-cm,k)
  --building the partition
  for i in 1..m-cm  repeat partition.(1+nonZeroPos.i) := nonZeros.i
  entries partition
```

```
subSet(n,m,k) ==
  k < 0 or n < 0 or m < 0 or m > n =>
    error "improper argument to subSet"
  bin : I := binomial$ICF (n,m)
  k >= bin =>
    error "there are not so many subsets"
  l : L I  := []
  n = 0 => l
  mm : I := k
  s  : I := m
  for t in 0..(m-1) repeat
     for y in (s-1)..(n+1) repeat
        if binomial$ICF (y,s) > mm then leave
     l := append (l,list(y-1)$(L I))
     mm := mm - binomial$ICF (y-1,s)
     s := s-1
  l


nextLatticePermutation(lambda, lattP, constructNotFirst) ==

  lprime  : L I  := conjugate(lambda)$PartitionsAndPermutations
  columns : NNI := (first(lambda)$(L I))::NNI
  rows    : NNI := (first(lprime)$(L I))::NNI
  n       : NNI :=(+/lambda)::NNI

  not constructNotFirst =>   -- first lattice permutation
    lattP := nil$(L I)
    for i in columns..1 by -1 repeat
      for l in 1..lprime(i) repeat
        lattP := cons(i,lattP)
    lattP

  help : V I := new(columns,0) -- entry help(i) stores the number
  -- of occurences of number i on our way from right to left
  rightPosition  : NNI := n
  leftEntry : NNI := lattP(rightPosition)::NNI
  ready  : B  := false
  until (ready or (not constructNotFirst)) repeat
    rightEntry : NNI := leftEntry
    leftEntry := lattP(rightPosition-1)::NNI
    help(rightEntry) := help(rightEntry) + 1
    -- search backward decreasing neighbour elements
    if rightEntry > leftEntry then
      if ((lprime(leftEntry)-help(leftEntry)) >_
        (lprime(rightEntry)-help(rightEntry)+1)) then
        -- the elements may be swapped because the number of occurances
        -- of leftEntry would still be greater than those of rightEntry
        ready := true
```

```
          j : NNI := leftEntry + 1
          -- search among the numbers leftEntry+1..rightEntry for the
          -- smallest one which can take the place of leftEntry.
          -- negation of condition above:
          while (help(j)=0) or ((lprime(leftEntry)-lprime(j))
            < (help(leftEntry)-help(j)+2)) repeat j := j + 1
          lattP(rightPosition-1) := j
          help(j) := help(j)-1
          help(leftEntry) := help(leftEntry) + 1
          -- reconstruct the rest of the list in increasing order
          for l in rightPosition..n repeat
            j := 0
            while help(1+j) = 0 repeat j := j + 1
            lattP(l::NNI) := j+1
            help(1+j) := help(1+j) - 1
        -- end of "if rightEntry > leftEntry"
        rightPosition := (rightPosition-1)::NNI
        if rightPosition = 1 then constructNotFirst := false
      -- end of repeat-loop
      not constructNotFirst =>  nil$(L I)
      lattP


    makeYoungTableau(lambda,gitter) ==
      lprime  : L I  := conjugate(lambda)$PartitionsAndPermutations
      columns : NNI := (first(lambda)$(L I))::NNI
      rows    : NNI := (first(lprime)$(L I))::NNI
      ytab    : M I  := new(rows,columns,0)
      help    : V I  := new(columns,1)
      i : I := -1     -- this makes the entries ranging from 0,..,n-1
                      -- i := 0 would make it from 1,..,n.
      j : I := 0
      for l in 1..maxIndex gitter repeat
        j := gitter(l)
        i := i + 1
        ytab(help(j),j) := i
        help(j) := help(j) + 1
      ytab


--    coerce(ytab) ==
--      lli := listOfLists(ytab)$(M I)
--      -- remove the filling zeros in each row. It is assumed that
--      -- that there are no such in row 0.
--      for i in 2..maxIndex lli repeat
--        THIS IS DEFINIVELY WRONG, I NEED A FUNCTION WHICH DELETES THE
--        0s, in my version there are no mapping facilities yet.
--        deleteInPlace(not zero?,lli i)
--      tableau(lli)$Tableau(I)
```

```
listYoungTableaus(lambda) ==
  lattice   : L I
  ytab      : M I
  younglist : L M I := nil$(L M I)
  lattice   := nextLatticePermutation(lambda,lattice,false)
  until null lattice repeat
    ytab      := makeYoungTableau(lambda,lattice)
    younglist := append(younglist,[ytab]$(L M I))$(L M I)
    lattice   := nextLatticePermutation(lambda,lattice,true)
  younglist


nextColeman(alpha,beta,C) ==
  nrow  : NNI := #beta
  ncol  : NNI := #alpha
  vnull : V I  := vector(nil()$(L I)) -- empty vector
  vzero : V I  := new(ncol,0)
  vrest : V I  := new(ncol,0)
  cnull : M I  := new(1,1,0)
  coleman := copy C
  if coleman ^= cnull then
    -- look for the first row of "coleman" that has a succeeding
    -- partition, this can be atmost row nrow-1
    i : NNI := (nrow-1)::NNI
    vrest := row(coleman,i) + row(coleman,nrow)
    --for k in 1..ncol repeat
    --  vrest(k) := coleman(i,k) + coleman(nrow,k)
    succ := nextPartition(vrest,row(coleman, i),beta(i))
    while (succ = vnull) repeat
      if i = 1 then return cnull -- part is last partition
      i := (i - 1)::NNI
      --for k in 1..ncol repeat
      --  vrest(k) := vrest(k) + coleman(i,k)
      vrest := vrest + row(coleman,i)
      succ := nextPartition(vrest, row(coleman, i), beta(i))
    j : I := i
    coleman := setRow_!(coleman, i, succ)
    --for k in 1..ncol repeat
    --  vrest(k) := vrest(k) - coleman(i,k)
    vrest := vrest - row(coleman,i)
  else
    vrest := vector alpha
    -- for k in 1..ncol repeat
    --  vrest(k) := alpha(k)
    coleman := new(nrow,ncol,0)
    j : I := 0
  for i in (j+1)::NNI..nrow-1 repeat
    succ := nextPartition(vrest,vnull,beta(i))
    coleman := setRow_!(coleman, i, succ)
```

```
      vrest := vrest - succ
      --for k in 1..ncol repeat
      --  vrest(k) := vrest(k) - succ(k)
    setRow_!(coleman, nrow, vrest)


  nextPartition(gamma:V I, part:V I, number:I) ==
    nextPartition(entries gamma, part, number)


  nextPartition(gamma:L I,part:V I,number:I) ==
    n : NNI := #gamma
    vnull : V I := vector(nil()$(L I)) -- empty vector
    if part ^= vnull then
      i : NNI := 2
      sum := part(1)
      while (part(i) = gamma(i)) or (sum = 0) repeat
        sum := sum + part(i)
        i := i + 1
        if i = 1+n then return vnull -- part is last partition
      sum := sum - 1
      part(i) := part(i) + 1
    else
      sum := number
      part := new(n,0)
      i := 1+n
    j : NNI := 1
    while sum > gamma(j) repeat
      part(j) := gamma(j)
      sum := sum - gamma(j)
      j := j + 1
    part(j) := sum
    for k in j+1..i-1 repeat
      part(k) := 0
    part


  inverseColeman(alpha,beta,C) ==
    pi   : L I  := nil$(L I)
    nrow : NNI := #beta
    ncol : NNI := #alpha
    help : V I  := new(nrow,0)
    sum  : I   := 1
    for i in 1..nrow repeat
      help(i) := sum
      sum := sum + beta(i)
    for j in 1..ncol repeat
      for i in 1..nrow repeat
        for k in 2..1+C(i,j) repeat
          pi := append(pi,list(help(i))$(L I))
```

```
            help(i) := help(i) + 1
      pi


    coleman(alpha,beta,pi) ==
      nrow : NNI := #beta
      ncol : NNI := #alpha
      temp : L L I := nil$(L L I)
      help : L I  := nil$(L I)
      colematrix : M I := new(nrow,ncol,0)
      betasum  : NNI := 0
      alphasum : NNI := 0
      for i in 1..ncol repeat
        help := nil$(L I)
        for j in alpha(i)..1 by-1 repeat
          help := cons(pi(j::NNI+alphasum),help)
        alphasum := (alphasum + alpha(i))::NNI
        temp := append(temp,list(help)$(L L I))
      for i in 1..nrow repeat
        help := nil$(L I)
        for j in beta(i)..1 by-1 repeat
          help := cons(j::NNI+betasum, help)
        betasum := (betasum + beta(i))::NNI
        for j in 1..ncol repeat
          colematrix(i,j) := #intersect(brace(help),brace(temp(j)))
      colematrix
```

———-

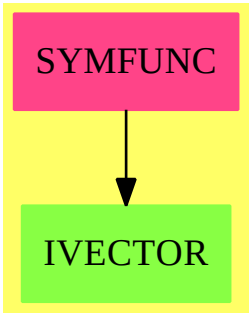
— SGCF.dotabb —

```
"SGCF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SGCF"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"SGCF" -> "FSAGG"
```

———-

20.31 package SYMFUNC SymmetricFunctions

20.31.1 SymmetricFunctions (SYMFUNC)



Exports:
symFunc

— package SYMFUNC SymmetricFunctions —

```
)abbrev package SYMFUNC SymmetricFunctions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 28 Jun 1990
++ Description:
++ Computes all the symmetric functions in n variables.

SymmetricFunctions(R:Ring): Exports == Implementation where
  UP  ==> SparseUnivariatePolynomial R

  Exports ==> with
    symFunc: List R  -> Vector R
      ++ symFunc([r1,...,rn]) returns the vector of the
      ++ elementary symmetric functions in the \spad{ri's}:
      ++ \spad{[r1 + ... + rn, r1 r2 + ... + r(n-1) rn, ..., r1 r2 ... rn]}.
    symFunc: (R, PositiveInteger) -> Vector R
      ++ symFunc(r, n) returns the vector of the elementary
      ++ symmetric functions in \spad{[r,r,...,r]} \spad{n} times.

  Implementation ==> add
    signFix: (UP, NonNegativeInteger) -> Vector R

    symFunc(x, n) == signFix((monomial(1, 1)$UP - x::UP) ** n, 1 + n)

    symFunc l ==
      signFix(*/[monomial(1, 1)$UP - a::UP for a in l], 1 + #l)
```

```
    signFix(p, n) ==
      m := minIndex(v := vectorise(p, n)) + 1
      for i in 0..((#v quo 2) - 1)::NonNegativeInteger repeat
        qsetelt_!(v, 2*i + m, - qelt(v, 2*i + m))
      reverse_! v
```

———-

— **SYMFUNC.dotabb** —

```
"SYMFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SYMFUNC"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"SYMFUNC" -> "IVECTOR"
```

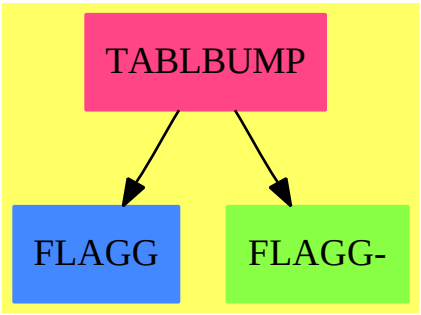
———-

Chapter 21

Chapter T

21.1 package TABLBUMP TableauxBumpers

21.1.1 TableauxBumpers (TABLBUMP)



Exports:

bat	bat1	bumprow	bumptab	bumptab1
inverse	lex	maxrow	mr	slex
tab	tab1	untab		

— package TABLBUMP TableauxBumpers —

```
)abbrev package TABLBUMP TableauxBumpers
++ Author: William H. Burge
++ Date Created: 1987
++ Date Last Updated: 23 Sept 1991
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
```

```
++ Keywords: Young tableau
++ References:
++ Description:
++ TableauBumpers implements the Schenstead-Knuth
++ correspondence between sequences and pairs of Young tableaux.
++ The 2 Young tableaux are represented as a single tableau with
++ pairs as components.

TableauxBumpers(S:OrderedSet):T==C where
     L==>List
     ST==>Stream
     B==>Boolean
     ROW==>Record(fs:B,sd:L S,td:L L S)
     RC==>Record(f1:L S,f2:L L L S,f3:L L S,f4:L L L S)
     PAIR==>L S
     T== with
       bumprow:((S,S)->B,PAIR,L PAIR)->ROW
         ++ bumprow(cf,pr,r) is an auxiliary function which
         ++ bumps a row r with a pair pr
         ++ using comparison function cf, and returns a record
       bumptab:((S,S)->B,PAIR,L L PAIR)->L L PAIR
         ++ bumptab(cf,pr,t) bumps a tableau t with a pair pr
         ++ using comparison function cf, returning a new tableau
       bumptab1:(PAIR,L L PAIR)->L L PAIR
         ++ bumptab1(pr,t) bumps a tableau t with a pair pr
         ++ using comparison function \spadfun{<},
         ++ returning a new tableau
       untab: (L PAIR,L L PAIR)->L PAIR
         ++ untab(lp,llp) is an auxiliary function
         ++ which unbumps a tableau llp,
         ++ using lp to accumulate pairs
       bat1:L L PAIR->L PAIR
         ++ bat1(llp) unbumps a tableau llp.
         ++ Operation bat1 is the inverse of tab1.
       bat:Tableau(L S)->L L S
         ++ bat(ls) unbumps a tableau ls
       tab1:L PAIR->L L PAIR
         ++ tab1(lp) creates a tableau from a list of pairs lp
       tab:L S->Tableau(L S)
         ++ tab(ls) creates a tableau from ls by first creating
         ++ a list of pairs using slex,
         ++ then creating a tableau using tab1.
       lex:L PAIR->L PAIR
         ++ lex(ls) sorts a list of pairs to lexicographic order
       slex:L S->L PAIR
         ++ slex(ls) sorts the argument sequence ls, then zips (see map) the
         ++ original argument sequence with the sorted result to
         ++ a list of pairs
       inverse:L S->L S
         ++ inverse(ls) forms the inverse of a sequence ls
```

```
    maxrow:(PAIR,L L PAIR,L PAIR,L L PAIR,L L PAIR,L L PAIR)->RC
      ++ maxrow(a,b,c,d,e) is an auxiliary function for mr
    mr:L L PAIR->RC
      ++ mr(t) is an auxiliary function which
      ++ finds the position of the maximum element of a tableau t
      ++ which is in the lowest row, producing a record of results
  C== add
    cf:(S,S)->B
    bumprow(cf,x:(PAIR),lls:(L PAIR))==
      if null lls
      then [false,x,[x]]$ROW
      else (y:(PAIR):=first lls;
            if cf(x.2,y.2)
            then [true,[x.1,y.2],cons([y.1,x.2],rest lls)]$ROW
            else (rw:ROW:=bumprow(cf,x,rest lls);
                    [rw.fs,rw.sd,cons(first lls,rw.td)]$ROW ))

    bumptab(cf,x:(PAIR),llls:(L L PAIR))==
        if null llls
        then [[x]]
        else (rw:ROW:= bumprow(cf,x,first llls);
              if rw.fs
              then cons(rw.td, bumptab(cf,rw.sd,rest llls))
              else cons(rw.td,rest llls))

    bumptab1(x,llls)==bumptab((s1,s2) +-> s1<s2, x, llls)

    rd==> reduce$StreamFunctions2(PAIR,L L PAIR)
    tab1(lls:(L PAIR))== rd([],bumptab1,lls::(ST PAIR))

    srt==>sort$(PAIR)
    lexorder:(PAIR,PAIR)->B
    lexorder(p1,p2)==if p1.1=p2.1 then p1.2<p2.2 else p1.1<p2.1
    lex lp==(sort$(L PAIR))((s1,s2) +-> lexorder(s1,s2), lp)
    slex ls==lex([[i,j] for i in srt((s1, s2) +-> s1<s2, ls) for j in ls])
    inverse ls==[lss.2 for lss in
                lex([[j,i] for i in srt((s1,s2) +-> s1<s2, ls)
                           for j in ls])]

    tab(ls:(PAIR))==(tableau tab1 slex ls )

    maxrow(n,a,b,c,d,llls)==
     if null llls or null(first llls)
     then [n,a,b,c]$RC
     else (fst:=first first llls;rst:=rest first llls;
           if fst.1>n.1
           then maxrow(fst,d,rst,rest llls,cons(first llls,d),rest llls)
           else maxrow(n,a,b,c,cons(first llls,d),rest llls))

    mr llls==maxrow(first first llls,[],rest first llls,rest llls,
```

```
                                                    [],llls)

      untab(lp, llls)==
        if null llls
        then lp
        else (rc:RC:=mr llls;
              rv:=reverse (bumptab((s1:S,s2:S):B +-> s2<s1, rc. f1, rc. f2));
              untab(cons(first first rv,lp)
                    ,append(rest rv,
                                        if null rc.f3
                                        then []
                                        else cons(rc.f3,rc.f4))))

      bat1 llls==untab([],[reverse lls for lls in llls])
      bat tb==bat1(listOfLists tb)
```

———-

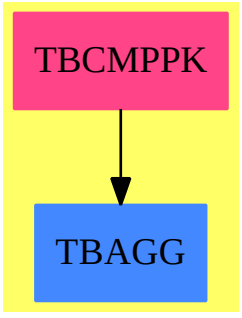
— **TABLBUMP.dotabb** —

```
"TABLBUMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TABLBUMP"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"TABLBUMP" -> "FLAGG-"
"TABLBUMP" -> "FLAGG"
```

———-

21.2 package TBCMPPK TabulatedComputationPackage

21.2.1 TabulatedComputationPackage (TBCMPPK)



Exports:

clearTable!	extractIfCan	initTable!	insert!	makingStats?
printInfo!	printStats!	printingInfo?	startStats!	usingTable?

— package TBCMPPK TabulatedComputationPackage —

```
)abbrev package TBCMPPK TabulatedComputationPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/09/1998
++ Date Last Updated: 12/16/1998
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ \axiom{TabulatedComputationPackage(Key ,Entry)} provides some modest support
++ for dealing with operations with type \axiom{Key -> Entry}. The result of
++ such operations can be stored and retrieved with this package by using
++ a hash-table. The user does not need to worry about the management of
++ this hash-table. However, onnly one hash-table is built by calling
++ \axiom{TabulatedComputationPackage(Key ,Entry)}.

TabulatedComputationPackage(Key ,Entry): Exports == Implementation where
  Key: SetCategory
  Entry: SetCategory
  N ==> NonNegativeInteger
  H ==> HashTable(Key, Entry, "UEQUAL")
  iprintpack ==> InternalPrintPackage()

  Exports ==  with
     initTable!: () -> Void
       ++ \axiom{initTable!()} initializes the hash-table.
     printInfo!: (String, String) -> Void
       ++ \axiom{printInfo!(x,y)} initializes the mesages to be printed
       ++ when manipulating items from the hash-table. If
       ++ a key is retrieved then \axiom{x} is displayed. If an item is
       ++ stored then \axiom{y} is displayed.
     startStats!: (String) -> Void
       ++ \axiom{startStats!(x)} initializes the statisitics process and
       ++ sets the comments to display when statistics are printed
     printStats!: () -> Void
       ++ \axiom{printStats!()} prints the statistics.
     clearTable!: () -> Void
       ++ \axiom{clearTable!()} clears the hash-table and assumes that
       ++ it will no longer be used.
     usingTable?: () -> Boolean
       ++ \axiom{usingTable?()} returns true iff the hash-table is used
     printingInfo?: () -> Boolean
```

```
      ++ \axiom{printingInfo?()} returns true iff messages are printed
      ++ when manipulating items from the hash-table.
    makingStats?: () -> Boolean
      ++ \axiom{makingStats?()} returns true iff the statisitics process
      ++ is running.
    extractIfCan: Key -> Union(Entry,"failed")
      ++ \axiom{extractIfCan(x)} searches the item whose key is \axiom{x}.
    insert!: (Key, Entry) -> Void
      ++ \axiom{insert!(x,y)} stores the item whose key is \axiom{x} and whose
      ++ entry is \axiom{y}.

  Implementation == add
    table?: Boolean := false
    t: H := empty()
    info?: Boolean := false
    stats?: Boolean := false
    used: NonNegativeInteger := 0
    ok: String := "o"
    ko: String := "+"
    domainName: String := empty()$String

    initTable!(): Void ==
      table? := true
      t := empty()
      void()
    printInfo!(s1: String, s2: String): Void ==
      (empty? s1) or (empty? s2) => void()
      not usingTable? =>
        error "in printInfo!()$TBCMPPK: not allowed to use hashtable"
      info? := true
      ok := s1
      ko := s2
      void()
    startStats!(s: String): Void ==
      empty? s => void()
      not table? =>
        error "in startStats!()$TBCMPPK: not allowed to use hashtable"
      stats? := true
      used := 0
      domainName := s
      void()
    printStats!(): Void ==
      not table? =>
        error "in printStats!()$TBCMPPK: not allowed to use hashtable"
      not stats? =>
        error "in printStats!()$TBCMPPK: statistics not started"
      output(" ")$OutputPackage
      title: String := concat("*** ", concat(domainName," Statistics ***"))
      output(title)$OutputPackage
      n: N := #t
```

```
      output("   Table     size: ", n::OutputForm)$OutputPackage
      output("   Entries reused: ", used::OutputForm)$OutputPackage
    clearTable!(): Void ==
      not table? =>
        error "in clearTable!()$TBCMPPK: not allowed to use hashtable"
      t := empty()
      table? := false
      info? := false
      stats? := false
      domainName := empty()$String
      void()
    usingTable?() == table?
    printingInfo?() == info?
    makingStats?() == stats?
    extractIfCan(k: Key): Union(Entry,"failed") ==
      not table? => "failed" :: Union(Entry,"failed")
      s: Union(Entry,"failed") := search(k,t)
      s case Entry =>
        if info? then iprint(ok)$iprintpack
        if stats? then used := used + 1
        return s
      "failed" :: Union(Entry,"failed")
    insert!(k: Key, e:Entry): Void ==
      not table? => void()
      t.k := e
      if info? then iprint(ko)$iprintpack
      void()
```

———-

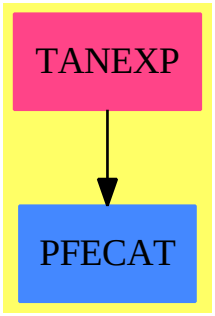
— TBCMPPK.dotabb —

```
"TBCMPPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TBCMPPK"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"TBCMPPK" -> "TBAGG"
```

———-

21.3 package TANEXP TangentExpansions

21.3.1 TangentExpansions (TANEXP)



Exports:
tanAn tanNa tanSum

— package TANEXP TangentExpansions —

```
)abbrev package TANEXP TangentExpansions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 20 Apr 1990
++ Description:
++ Expands tangents of sums and scalar products.

TangentExpansions(R:Field): Exports == Implementation where
  PI ==> PositiveInteger
  Z  ==> Integer
  UP ==> SparseUnivariatePolynomial R
  QF ==> Fraction UP

  Exports ==> with
    tanSum: List R -> R
      ++ tanSum([a1,...,an]) returns \spad{f(a1,...,an)} such that
      ++ if \spad{ai = tan(ui)} then \spad{f(a1,...,an) = tan(u1 + ... + un)}.
    tanAn : (R, PI) -> UP
      ++ tanAn(a, n) returns \spad{P(x)} such that
      ++ if \spad{a = tan(u)} then \spad{P(tan(u/n)) = 0}.
    tanNa : (R,  Z) -> R
      ++ tanNa(a, n) returns \spad{f(a)} such that
      ++ if \spad{a = tan(u)} then \spad{f(a) = tan(n * u)}.

  Implementation ==> add
    import SymmetricFunctions(R)
    import SymmetricFunctions(UP)
```

```
    m1toN : Integer -> Integer
    tanPIa: PI -> QF

    m1toN n     == (odd? n => -1; 1)
    tanAn(a, n) == a * denom(q := tanPIa n) - numer q

    tanNa(a, n) ==
      zero? n => 0
      negative? n => - tanNa(a, -n)
      (numer(t := tanPIa(n::PI)) a) / ((denom t) a)

    tanSum l ==
      m := minIndex(v := symFunc l)
      +/[m1toN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
        / +/[m1toN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]

-- tanPIa(n) returns P(a)/Q(a) such that
-- if a = tan(u) then P(a)/Q(a) = tan(n * u);
    tanPIa n ==
      m := minIndex(v := symFunc(monomial(1, 1)$UP, n))
      +/[m1toN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
        / +/[m1toN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]
```

———-

— **TANEXP.dotabb** —

```
"TANEXP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TANEXP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"TANEXP" -> "PFECAT"
```

———-

21.4 package UTSSOL TaylorSolve

UTSSOL is a facility to compute the first few coefficients of a Taylor series given only implicitely by a function f that vanishes when applied to the Taylor series.

It uses the method of undetermined coefficients.

```
  Could I either
  \begin{itemize}
  \item take a function UTSCAT F $->$ UTSCAT F and still be able to compute
    with undetermined coefficients, or
```

```
\item take a function F $->$ F, and do likewise?
\end{itemize}

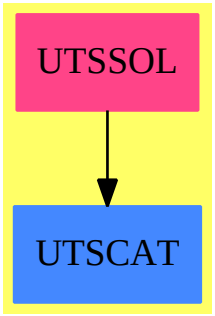
Let's see.

Try to compute the equation without resorting to power series. I.e., %
c: SUP SUP F, and f: SUP SUP F $->$ SUP SUP F. Won't this make the
computation of coefficients terribly slow?

I could also try to replace transcendental kernels with variables\dots

Unfortunately, SUP F does not have TRANFUN -- well, it can't, of
course. However, I'd like to be able to compute %
sin(1+monomial(1,1)$UFPS SUP EXPR INT).
```

21.4.1 TaylorSolve (UTSSOL)



Exports:
seriesSolve

— package UTSSOL TaylorSolve —

```
)abbrev package UTSSOL TaylorSolve
++ Description:
++ This package has no description

TaylorSolve(F, UTSF, UTSSUPF): Exports == Implementation where
    F: Field
    SUP  ==> SparseUnivariatePolynomialExpressions
    UTSF: UnivariateTaylorSeriesCategory F
    UTSSUPF: UnivariateTaylorSeriesCategory SUP F
    NNI  ==> NonNegativeInteger

    Exports == with
        seriesSolve: (UTSSUPF -> UTSSUPF, List F) -> UTSF
```

```
    Implementation == add
\getchunk{implementation: UTSSOL TaylorSolve}
```

———-

— **implementation: UTSSOL TaylorSolve** —

```
        seriesSolve(f, l) ==
            c1 :=
             map((x:F):SUP F +-> x::(SUP F), l)
                  $ListFunctions2(F, SUP F)::(Stream SUP F)
            coeffs: Stream SUP F := concat(c1, generate(monomial(1$F,1$NNI)))
--            coeffs: Stream SUP F := concat(c1, monomial(1$F,1$NNI))
```

———-

coeffs is the stream of the already computed coefficients of the solution, plus one which is so far undetermined. We store in st.2 the complete stream and in st.1 the stream starting with the first coefficient that has possibly not yet been computed.

```
  The mathematics is not quite worked out. If coeffs is initialized as
  stream with all coefficients set to the \emph{same} transcendental value,
  and not enough initial values are given, then the missing ones are
  implicitely assumed to be all identical. It may well happen that a solution
  is produced, although it is not uniquely determined\dots
```

— **implementation: UTSSOL TaylorSolve** —

```
            st: List Stream SUP F := [coeffs, coeffs]
```

———-

Consider an arbitrary equation $f\big(x, y(x)\big) = 0$. When setting $x = 0$, we obtain $f\big(0, y(0)\big) = 0$. It is not necessarily the case that this determines $y(0)$ uniquely, so we need one initial value that satisfies this equation.

```
  seriesSolve should check that the given initial values satisfy $f\big(0, y(0),
  y'(0),...\big) = 0$.
```

Now consider the derivatives of f, where we write y instead of $y(x)$ for better readability:

$$\frac{d}{dx} f(x, y) = f_1(x, y) + f_2(x, y)y'$$

and

```
\frac{d^2}{dx^2}f(x, y)&=f_{1,1}(x, y)\\
                       &+f_{1,2}(x, y)y^\prime\\
                       &+f_{2,1}(x, y)y^\prime\\
                       &+f_{2,2}(x, y)(y^\prime)^2\\
                       &+f_2(x, y)y^{\prime\prime}.
```

In general, $\frac{d^2}{dx^2}f(x,y)$ depends only linearly on y''.

```
This points to another possibility: Since we know that we only need to solve
linear equations, we could compute two values and then use interpolation.
This might be a bit slower, but more importantly: can we still check that we
have enough initial values? Furthermore, we then really need that $f$ is
analytic, i.e., operators are not necessarily allowed anymore. However, it
seems that composition is allowed.
```

— **implementation: UTSSOL TaylorSolve** —

```
next: () -> F :=
    nr := st.1
    res: F

    if ground?(coeff: SUP F := nr.1)$(SUP F)
```

———-

If the next element was already calculated, we can simply return it:

— **implementation: UTSSOL TaylorSolve** —

```
    then
        res := ground coeff
        st.1 := rest nr
    else
```

———-

Otherwise, we have to find the first non-satisfied relation and solve it. It should be linear, or a single non-constant monomial. That is, the solution should be unique.

— **implementation: UTSSOL TaylorSolve** —

```
        ns := st.2
        eqs: Stream SUP F := coefficients f series ns
        while zero? first eqs repeat eqs := rest eqs
        eq: SUP F := first eqs
        if degree eq > 1 then
```

```
                        if monomial? eq then res := 0
                        else
                            output(hconcat("The equation is: ", eq::OutputForm))
                                  $OutputPackage
                            error "seriesSolve: equation for coefficient not linear"
                    else res := (-coefficient(eq, 0$NNI)$(SUP F)
                                 /coefficient(eq, 1$NNI)$(SUP F))

                    nr.1 := res::SUP F
--                    concat!(st.2, monomial(1$F,1$NNI))
                    st.1 := rest nr

                res

            series generate next
```

———-

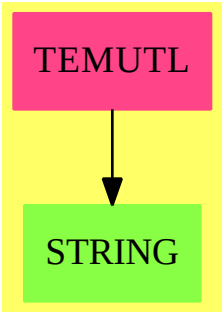
— **UTSSOL.dotabb** —

```
"UTSSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSSOL"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTSSOL" -> "UTSCAT"
```

———-

21.5 package TEMUTL TemplateUtilities

21.5.1 TemplateUtilities (TEMUTL)



Exports:
interpretString stripCommentsAndBlanks

— package TEMUTL TemplateUtilities —

```
)abbrev package TEMUTL TemplateUtilities
++ Author: Mike Dewar
++ Date Created:  October 1992
++ Date Last Updated:
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides functions for template manipulation

TemplateUtilities(): Exports == Implementation where

  Exports == with
    interpretString : String -> Any
      ++ interpretString(s) treats a string as a piece of AXIOM input, by
      ++ parsing and interpreting it.
    stripCommentsAndBlanks : String -> String
      ++ stripCommentsAndBlanks(s) treats s as a piece of AXIOM input, and
      ++ removes comments, and leading and trailing blanks.

  Implementation == add

    import InputForm

    stripC(s:String,u:String):String ==
      i : Integer := position(u,s,1)
      i = 0 => s
      delete(s,i..)

    stripCommentsAndBlanks(s:String):String ==
      trim(stripC(stripC(s,"++"),"--"),char " ")

    parse(s:String):InputForm ==
      ncParseFromString(s)$Lisp::InputForm

    interpretString(s:String):Any ==
      interpret parse s
```

———-

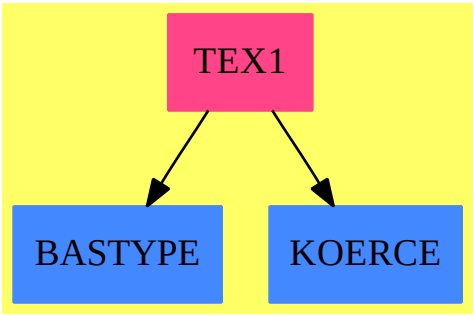
— TEMUTL.dotabb —

```
"TEMUTL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TEMUTL"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"TEMUTL" -> "STRING"
```

———-

21.6 package TEX1 TexFormat1

21.6.1 TexFormat1 (TEX1)



Exports:
coerce

— package TEX1 TexFormat1 —

```
)abbrev package TEX1 TexFormat1
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ Change History:
++ Basic Operations: coerce
++ Related Constructors: TexFormat
++ Also See: ScriptFormulaFormat, ScriptFormulaFormat1
++ AMS Classifications:
++ Keywords: TeX, output, format
++ Description:
++ \spadtype{TexFormat1} provides a utility coercion for changing
++ to TeX format anything that has a coercion to the standard output format.

TexFormat1(S : SetCategory): public == private where
  public  ==  with
    coerce: S -> TexFormat()
      ++ coerce(s) provides a direct coercion from a domain S to
      ++ TeX format. This allows the user to skip the step of first
      ++ manually coercing the object to standard output format before
```

```
      ++ it is coerced to TeX format.

  private == add
    import TexFormat()

    coerce(s : S): TexFormat ==
      coerce(s :: OutputForm)$TexFormat
```

———-

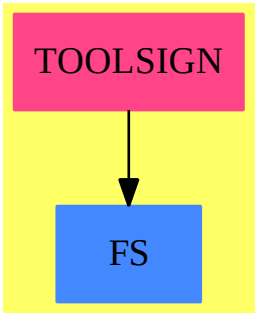
— **TEX1.dotabb** —

```
"TEX1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TEX1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"TEX1" -> "BASTYPE"
"TEX1" -> "KOERCE"
```

———-

21.7 package TOOLSIGN ToolsForSign

21.7.1 ToolsForSign (TOOLSIGN)



Exports:
direction nonQsign sign

— **package TOOLSIGN ToolsForSign** —

```
)abbrev package TOOLSIGN ToolsForSign
++ Author: Manuel Bronstein
++ Date Created: 25 August 1989
```

```
++ Date Last Updated: 26 November 1991
++ Description:
++ Tools for the sign finding utilities.

ToolsForSign(R:Ring): with
    sign     : R      -> Union(Integer, "failed")
      ++ sign(r) \undocumented
    nonQsign : R      -> Union(Integer, "failed")
      ++ nonQsign(r) \undocumented
    direction: String -> Integer
      ++ direction(s) \undocumented
 == add

    if R is AlgebraicNumber then
      nonQsign r ==
        sign((r pretend AlgebraicNumber)::Expression(
                  Integer))$ElementaryFunctionSign(Integer, Expression Integer)
    else
      nonQsign r == "failed"

    if R has RetractableTo Fraction Integer then
      sign r ==
        (u := retractIfCan(r)@Union(Fraction Integer, "failed"))
          case Fraction(Integer) => sign(u::Fraction Integer)
        nonQsign r

    else
      if R has RetractableTo Integer then
        sign r ==
          (u := retractIfCan(r)@Union(Integer, "failed"))
            case "failed" => "failed"
          sign(u::Integer)

      else
        sign r ==
          zero? r => 0
--          one? r => 1
          r = 1 => 1
          r = -1 => -1
          "failed"

    direction st ==
      st = "right" => 1
      st = "left" => -1
      error "Unknown option"
```

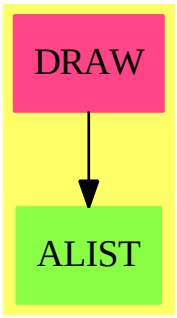
— **TOOLSIGN.dotabb** —

```
"TOOLSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOOLSIGN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"TOOLSIGN" -> "FS"
```

———-

21.8 package DRAW TopLevelDrawFunctions

21.8.1 TopLevelDrawFunctions (DRAW)



Exports:
draw makeObject

— **package DRAW TopLevelDrawFunctions** —

```
)abbrev package DRAW TopLevelDrawFunctions
++ Author: Clifton J. Williamson
++ Date Created: 23 January 1990
++ Date Last Updated: October 1991 by Jon Steinbach
++ Basic Operations: draw
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ TopLevelDrawFunctions provides top level functions for
++ drawing graphics of expressions.

TopLevelDrawFunctions(Ex:Join(ConvertibleTo InputForm,SetCategory)):
 Exports == Implementation where
  B    ==> Boolean
```

```
  BIND ==> SegmentBinding Float
  L    ==> List
  SF   ==> DoubleFloat
  DROP ==> DrawOption

  PPC  ==> ParametricPlaneCurve Ex
  PPCF ==> ParametricPlaneCurve(SF -> SF)
  PSC  ==> ParametricSpaceCurve Ex
  PSCF ==> ParametricSpaceCurve(SF -> SF)
  PSF  ==> ParametricSurface Ex
  PSFF ==> ParametricSurface((SF,SF) -> SF)
  SPACE3 ==> ThreeSpace(SF)
  VIEW2 ==> TwoDimensionalViewport
  VIEW3 ==> ThreeDimensionalViewport

  Exports ==> with

--% Two Dimensional Function Plots

    draw: (Ex,BIND,L DROP) -> VIEW2
      ++ draw(f(x),x = a..b,l) draws the graph of \spad{y = f(x)} as x
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}; \spad{f(x)} is the
      ++ default title, and the options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    draw: (Ex,BIND) -> VIEW2
      ++ draw(f(x),x = a..b) draws the graph of \spad{y = f(x)} as x
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}; \spad{f(x)} appears
      ++ in the title bar.

--% Parametric Plane Curves

    draw: (PPC,BIND,L DROP) -> VIEW2
      ++ draw(curve(f(t),g(t)),t = a..b,l) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)}; \spad{(f(t),g(t))} is the default title, and the
      ++ options contained in the list l of the domain \spad{DrawOption}
      ++ are applied.
    draw: (PPC,BIND) -> VIEW2
      ++ draw(curve(f(t),g(t)),t = a..b) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)}; \spad{(f(t),g(t))} appears in the title bar.

--% Parametric Space Curves

    draw: (PSC,BIND,L DROP) -> VIEW3
      ++ draw(curve(f(t),g(t),h(t)),t = a..b,l) draws the graph of the
      ++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
      ++ as t ranges from \spad{min(a,b)} to \spad{max(a,b)}; \spad{h(t)}
      ++ is the default title, and the options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
```

```
    draw: (PSC,BIND) -> VIEW3
      ++ draw(curve(f(t),g(t),h(t)),t = a..b) draws the graph of the parametric
      ++ curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)} as t ranges
      ++ from \spad{min(a,b)} to \spad{max(a,b)}; \spad{h(t)} is the default
      ++ title.
    makeObject: (PSC,BIND,L DROP) -> SPACE3
      ++ makeObject(curve(f(t),g(t),h(t)),t = a..b,l) returns a space of
      ++ the domain \spadtype{ThreeSpace} which contains the graph of the
      ++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
      ++ as t ranges from \spad{min(a,b)} to \spad{max(a,b)}; \spad{h(t)}
      ++ is the default title, and the options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    makeObject: (PSC,BIND) -> SPACE3
      ++ makeObject(curve(f(t),g(t),h(t)),t = a..b) returns a space of the
      ++ domain \spadtype{ThreeSpace} which contains the graph of the
      ++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
      ++ as t ranges from \spad{min(a,b)} to \spad{max(a,b)}; \spad{h(t)} is
      ++ the default title.

--% Three Dimensional Function Plots

    draw: (Ex,BIND,BIND,L DROP) -> VIEW3
      ++ draw(f(x,y),x = a..b,y = c..d,l) draws the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}; \spad{f(x,y)} is the default
      ++ title, and the options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    draw: (Ex,BIND,BIND) -> VIEW3
      ++ draw(f(x,y),x = a..b,y = c..d) draws the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}; \spad{f(x,y)} appears in the title bar.
    makeObject: (Ex,BIND,BIND,L DROP) -> SPACE3
      ++ makeObject(f(x,y),x = a..b,y = c..d,l) returns a space of the
      ++ domain \spadtype{ThreeSpace} which contains the graph of
      ++ \spad{z = f(x,y)} as x ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and y ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{f(x,y)}
      ++ is the default title, and the options contained in the list l of the
      ++ domain \spad{DrawOption} are applied.
    makeObject: (Ex,BIND,BIND) -> SPACE3
      ++ makeObject(f(x,y),x = a..b,y = c..d) returns a space of the domain
      ++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}; \spad{f(x,y)} appears as the
      ++ default title.

--% Parametric Surfaces

    draw: (PSF,BIND,BIND,L DROP) -> VIEW3
      ++ draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) draws the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
```

```
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)}
      ++ is the default title, and the options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    draw: (PSF,BIND,BIND) -> VIEW3
      ++ draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) draws the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
      ++ the default title.
    makeObject: (PSF,BIND,BIND,L DROP) -> SPACE3
      ++ makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) returns
      ++ a space of the domain \spadtype{ThreeSpace} which contains the graph
      ++ of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
      ++ the default title, and the options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    makeObject: (PSF,BIND,BIND) -> SPACE3
      ++ makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) returns
      ++ a space of the domain \spadtype{ThreeSpace} which contains the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
      ++ the default title.

  Implementation ==> add
    import TopLevelDrawFunctionsForCompiledFunctions
    import MakeFloatCompiledFunction(Ex)
    import ParametricPlaneCurve(SF -> SF)
    import ParametricSpaceCurve(SF -> SF)
    import ParametricSurface((SF,SF) -> SF)
    import ThreeSpace(SF)

------------------------------------------------------------------------
--                    2D - draw's  (given by formulae)
------------------------------------------------------------------------

--% Two Dimensional Function Plots

    draw(f:Ex,bind:BIND,l:L DROP) ==
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
          l := concat(title "AXIOM2D",l)
        else l := concat(title s,l)
      -- call 'draw'
      draw(makeFloatFunction(f,variable bind),segment bind,l)
```

```
    draw(f:Ex,bind:BIND) == draw(f,bind,nil())

--% Parametric Plane Curves

    draw(ppc:PPC,bind:BIND,l:L DROP) ==
      f := coordinate(ppc,1); g := coordinate(ppc,2)
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
          l := concat(title "AXIOM2D",l)
        else l := concat(title s,l)
      -- create curve with functions as coordinates
      curve : PPCF := curve(makeFloatFunction(f,variable bind),_
                            makeFloatFunction(g,variable bind))$PPCF
      -- call 'draw'
      draw(curve,segment bind,l)

    draw(ppc:PPC,bind:BIND) == draw(ppc,bind,nil())

------------------------------------------------------------------------
--                    3D - Curves  (given by formulas)
------------------------------------------------------------------------

    makeObject(psc:PSC,tBind:BIND,l:L DROP) ==
      -- obtain dependent variable and coordinate functions
      t := variable tBind; tSeg := segment tBind
      f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
          l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
      -- indicate draw style if necessary
      if not option?(l,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
      -- create curve with functions as coordinates
      curve : PSCF := curve(makeFloatFunction(f,t),_
                            makeFloatFunction(g,t),_
                            makeFloatFunction(h,t))
      -- call 'draw'
      makeObject(curve,tSeg,l)

    makeObject(psc:PSC,tBind:BIND) ==
      makeObject(psc,tBind,nil())

    draw(psc:PSC,tBind:BIND,l:L DROP) ==
      -- obtain dependent variable and coordinate functions
      t := variable tBind; tSeg := segment tBind
```

```
      f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
          l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
      -- indicate draw style if necessary
      if not option?(l,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
      -- create curve with functions as coordinates
      curve : PSCF := curve(makeFloatFunction(f,t),_
                            makeFloatFunction(g,t),_
                            makeFloatFunction(h,t))
      -- call 'draw'
      draw(curve,tSeg,l)

    draw(psc:PSC,tBind:BIND) ==
      draw(psc,tBind,nil())

------------------------------------------------------------------------
--                    3D - Surfaces  (given by formulas)
------------------------------------------------------------------------

--% Three Dimensional Function Plots

    makeObject(f:Ex,xBind:BIND,yBind:BIND,l:L DROP) ==
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
          l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
      -- indicate draw style if necessary
      if not option?(l,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
      -- obtain dependent variables and their ranges
      x := variable xBind; xSeg := segment xBind
      y := variable yBind; ySeg := segment yBind
      -- call 'draw'
      makeObject(makeFloatFunction(f,x,y),xSeg,ySeg,l)

    makeObject(f:Ex,xBind:BIND,yBind:BIND) ==
      makeObject(f,xBind,yBind,nil())

    draw(f:Ex,xBind:BIND,yBind:BIND,l:L DROP) ==
      -- create title if necessary
      if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
```

```
        l := concat(title "AXIOM3D",l)
      else l := concat(title s,l)
    -- indicate draw style if necessary
    if not option?(l,"style" :: Symbol) then
      l := concat(style unparse(convert(f)@InputForm),l)
    -- obtain dependent variables and their ranges
    x := variable xBind; xSeg := segment xBind
    y := variable yBind; ySeg := segment yBind
    -- call 'draw'
    draw(makeFloatFunction(f,x,y),xSeg,ySeg,l)

  draw(f:Ex,xBind:BIND,yBind:BIND) ==
    draw(f,xBind,yBind,nil())

--% parametric surface

  makeObject(s:PSF,uBind:BIND,vBind:BIND,l:L DROP) ==
    f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)
    if not option?(l,"title" :: Symbol) then
      s:String := unparse(convert(f)@InputForm)
      if sayLength(s)$DisplayPackage > 50 then
        l := concat(title "AXIOM3D",l)
      else l := concat(title s,l)
    if not option?(l,"style" :: Symbol) then
      l := concat(style unparse(convert(f)@InputForm),l)
    u := variable uBind; uSeg := segment uBind
    v := variable vBind; vSeg := segment vBind
    surf : PSFF := surface(makeFloatFunction(f,u,v),_
                           makeFloatFunction(g,u,v),_
                           makeFloatFunction(h,u,v))
    makeObject(surf,uSeg,vSeg,l)

  makeObject(s:PSF,uBind:BIND,vBind:BIND) ==
    makeObject(s,uBind,vBind,nil())

  draw(s:PSF,uBind:BIND,vBind:BIND,l:L DROP) ==
    f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)
    -- create title if necessary
    if not option?(l,"title" :: Symbol) then
      s:String := unparse(convert(f)@InputForm)
      if sayLength(s)$DisplayPackage > 50 then
        l := concat(title "AXIOM3D",l)
      else l := concat(title s,l)
    -- indicate draw style if necessary
    if not option?(l,"style" :: Symbol) then
      l := concat(style unparse(convert(f)@InputForm),l)
    -- obtain dependent variables and their ranges
    u := variable uBind; uSeg := segment uBind
    v := variable vBind; vSeg := segment vBind
    -- create surface with functions as coordinates
```

```
      surf : PSFF := surface(makeFloatFunction(f,u,v),_
                             makeFloatFunction(g,u,v),_
                             makeFloatFunction(h,u,v))
      -- call 'draw'
      draw(surf,uSeg,vSeg,l)

    draw(s:PSF,uBind:BIND,vBind:BIND) ==
      draw(s,uBind,vBind,nil())
```

———-

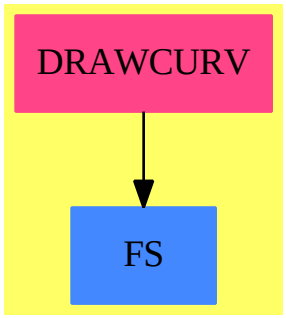
— **DRAW.dotabb** —

```
"DRAW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAW"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DRAW" -> "ALIST"
```

———-

21.9 package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves

21.9.1 TopLevelDrawFunctionsForAlgebraicCurves (DRAWCURV)



Exports:
draw

— **package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves** —

```
)abbrev package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves
```

```
++ Author: Clifton J. Williamson
++ Date Created: 26 June 1990
++ Date Last Updated:  October 1991 by Jon Steinbach
++ Basic Operations: draw
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ TopLevelDrawFunctionsForAlgebraicCurves provides top level
++ functions for drawing non-singular algebraic curves.

TopLevelDrawFunctionsForAlgebraicCurves(R,Ex): Exports == Implementation where
  R  : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
  Ex : FunctionSpace(R)

  ANY1  ==> AnyFunctions1
  DROP  ==> DrawOption
  EQ    ==> Equation
  F     ==> Float
  FRAC  ==> Fraction
  I     ==> Integer
  L     ==> List
  P     ==> Polynomial
  RN    ==> Fraction Integer
  SEG   ==> Segment
  SY    ==> Symbol
  VIEW2 ==> TwoDimensionalViewport

  Exports ==> with

    draw: (EQ Ex,SY,SY,L DROP) -> VIEW2
      ++ draw(f(x,y) = g(x,y),x,y,l) draws the graph of a polynomial
      ++ equation.  The list l of draw options must specify a region
      ++ in the plane in which the curve is to sketched.

  Implementation ==> add
    import ViewportPackage
    import PlaneAlgebraicCurvePlot
    import ViewDefaultsPackage
    import GraphicsDefaults
    import DrawOptionFunctions0
    import SegmentFunctions2(RN,F)
    import SegmentFunctions2(F,RN)
    import AnyFunctions1(L SEG RN)

    drawToScaleRanges: (SEG F,SEG F) -> L SEG F
    drawToScaleRanges(xVals,yVals) ==
      -- warning: assumes window is square
```

```
    xHi := hi xVals; xLo := lo xVals
    yHi := hi yVals; yLo := lo yVals
    xDiff := xHi - xLo; yDiff := yHi - yLo
    pad := abs(yDiff - xDiff)/2
    yDiff > xDiff =>
      [segment(xLo - pad,xHi + pad),yVals]
    [xVals,segment(yLo - pad,yHi + pad)]

  intConvert: R -> I
  intConvert r ==
    (nn := retractIfCan(r)@Union(I,"failed")) case "failed" =>
      error "draw: polynomial must have rational coefficients"
    nn :: I

  polyEquation: EQ Ex -> P I
  polyEquation eq ==
    ff := lhs(eq) - rhs(eq)
    (r := retractIfCan(ff)@Union(FRAC P R,"failed")) case "failed" =>
      error "draw: not a polynomial equation"
    rat := r :: FRAC P R
    retractIfCan(denom rat)@Union(R,"failed") case "failed" =>
      error "draw: non-constant denominator"
    map(intConvert,numer rat)$PolynomialFunctions2(R,I)

  draw(eq,x,y,l) ==
    -- obtain polynomial equation
    p := polyEquation eq
    -- extract ranges from option list
    floatRange := option(l,"rangeFloat" :: Symbol)
    ratRange := option(l,"rangeRat" :: Symbol)
    (floatRange case "failed") and (ratRange case "failed") =>
      error "draw: you must specify ranges for an implicit plot"
    ranges : L SEG RN := nil()             -- dummy value
    floatRanges : L SEG F := nil()         -- dummy value
    xRange : SEG RN := segment(0,0)        -- dummy value
    yRange : SEG RN := segment(0,0)        -- dummy value
    xRangeFloat : SEG F := segment(0,0)    -- dummy value
    yRangeFloat : SEG F := segment(0,0)    -- dummy value
    if not ratRange case "failed" then
      ranges := retract(ratRange :: Any)$ANY1(L SEG RN)
      not size?(ranges,2) => error "draw: you must specify two ranges"
      xRange := first ranges; yRange := second ranges
      xRangeFloat := map((s:RN):F+->convert(s)@Float,xRange)@(SEG F)
      yRangeFloat := map((s:RN):F+->convert(s)@Float,yRange)@(SEG F)
      floatRanges := [xRangeFloat,yRangeFloat]
    else
      floatRanges := retract(floatRange :: Any)$ANY1(L SEG F)
      not size?(floatRanges,2) =>
        error "draw: you must specify two ranges"
      xRangeFloat := first floatRanges
```

```
      yRangeFloat := second floatRanges
      xRange := map((s:F):RN+->retract(s)@RN,xRangeFloat)@(SEG RN)
      yRange := map((s:F):RN+->retract(s)@RN,yRangeFloat)@(SEG RN)
      ranges := [xRange,yRange]
    -- create curve plot
    acplot := makeSketch(p,x,y,xRange,yRange)
    -- process scaling information
    if toScale(l,drawToScale()) then
      scaledRanges := drawToScaleRanges(xRangeFloat,yRangeFloat)
      -- add scaled ranges to list of options
      l := concat(ranges scaledRanges,l)
    else
      -- add ranges to list of options
      l := concat(ranges floatRanges,l)
    -- process color information
    ptCol := pointColorPalette(l,pointColorDefault())
    crCol := curveColorPalette(l,lineColorDefault())
    -- draw
    drawCurves(listBranches acplot,ptCol,crCol,pointSizeDefault(),l)
```

———-

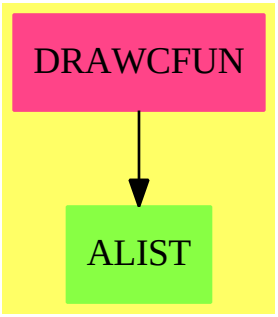
— DRAWCURV.dotabb —

```
"DRAWCURV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWCURV"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"DRAWCURV" -> "FS"
```

———-

21.10 package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions

21.10.1 TopLevelDrawFunctionsForCompiledFunctions (DRAWCFUN)



Exports:
draw makeObject recolor

— **package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions** —

```
)abbrev package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions
++ Author: Clifton J. Williamson
++ Date Created: 22 June 1990
++ Date Last Updated: January 1992 by Scott Morrison
++ Basic Operations: draw, recolor
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ TopLevelDrawFunctionsForCompiledFunctions provides top level
++ functions for drawing graphics of expressions.

TopLevelDrawFunctionsForCompiledFunctions():
 Exports == Implementation where
  ANY1 ==> AnyFunctions1
  B    ==> Boolean
  F    ==> Float
  L    ==> List
  SEG  ==> Segment Float
  SF   ==> DoubleFloat
  DROP ==> DrawOption
  PLOT ==> Plot
```

```
  PPC  ==> ParametricPlaneCurve(SF -> SF)
  PSC  ==> ParametricSpaceCurve(SF -> SF)
  PSF  ==> ParametricSurface((SF,SF) -> SF)
  Pt   ==> Point SF
  PSFUN ==> (SF, SF) -> Pt
  PCFUN ==> SF -> Pt
  SPACE3 ==> ThreeSpace(SF)
  VIEW2 ==> TwoDimensionalViewport
  VIEW3 ==> ThreeDimensionalViewport

  Exports ==> with

--% Two Dimensional Function Plots

    draw: (SF -> SF,SEG,L DROP) -> VIEW2
      ++ draw(f,a..b,l) draws the graph of \spad{y = f(x)} as x
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.
      ++ The options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    draw: (SF -> SF,SEG) -> VIEW2
      ++ draw(f,a..b) draws the graph of \spad{y = f(x)} as x
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.

--% Parametric Plane Curves

    draw: (PPC,SEG,L DROP) -> VIEW2
      ++ draw(curve(f,g),a..b,l) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)}.
      ++ The options contained in the list l of the domain \spad{DrawOption}
      ++ are applied.
    draw: (PPC,SEG) -> VIEW2
      ++ draw(curve(f,g),a..b) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)}.

--% Parametric Space Curves

    draw: (PSC,SEG,L DROP) -> VIEW3
      ++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)}.
      ++ The options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    draw: (PSC,SEG) -> VIEW3
      ++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
      ++ curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)}.
    draw: (PCFUN,SEG,L DROP) -> VIEW3
      ++ draw(f,a..b,l) draws the graph of the parametric
```

```
      ++ curve \spad{f} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)}.
      ++ The options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    draw: (PCFUN,SEG) -> VIEW3
      ++ draw(f,a..b,l) draws the graph of the parametric
      ++ curve \spad{f} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)}.

    makeObject: (PSC,SEG,L DROP) -> SPACE3
      ++ makeObject(curve(f,g,h),a..b,l) returns a space of the
      ++ domain \spadtype{ThreeSpace} which contains the graph of the
      ++ parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)};
      ++ The options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    makeObject: (PSC,SEG) -> SPACE3
      ++ makeObject(sp,curve(f,g,h),a..b) returns the space \spad{sp}
      ++ of the domain \spadtype{ThreeSpace} with the addition of the graph
      ++ of the parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.
    makeObject: (PCFUN,SEG,L DROP) -> SPACE3
      ++ makeObject(curve(f,g,h),a..b,l) returns a space of the
      ++ domain \spadtype{ThreeSpace} which contains the graph of the
      ++ parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
      ++ \spad{min(a,b)} to \spad{max(a,b)}.
      ++ The options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    makeObject: (PCFUN,SEG) -> SPACE3
      ++ makeObject(sp,curve(f,g,h),a..b) returns the space \spad{sp}
      ++ of the domain \spadtype{ThreeSpace} with the addition of the graph
      ++ of the parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t
      ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.

--% Three Dimensional Function Plots

    draw: ((SF,SF) -> SF,SEG,SEG,L DROP) -> VIEW3
      ++ draw(f,a..b,c..d,l) draws the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}.
      ++ and the options contained in the list l of the domain
      ++ \spad{DrawOption} are applied.
    draw: ((SF,SF) -> SF,SEG,SEG) -> VIEW3
      ++ draw(f,a..b,c..d) draws the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}.
    makeObject: ((SF,SF) -> SF,SEG,SEG,L DROP) -> SPACE3
      ++ makeObject(f,a..b,c..d,l) returns a space of the domain
      ++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
```

```
      ++ \spad{min(c,d)} to \spad{max(c,d)}, and the options contained in the
      ++ list l of the domain \spad{DrawOption} are applied.
    makeObject: ((SF,SF) -> SF,SEG,SEG) -> SPACE3
      ++ makeObject(f,a..b,c..d) returns a space of the domain
      ++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
      ++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
      ++ \spad{min(c,d)} to \spad{max(c,d)}.

--% Parametric Surfaces

    draw: (PSFUN, SEG, SEG, L DROP) -> VIEW3
      ++ draw(f,a..b,c..d) draws the
      ++ graph of the parametric surface \spad{f(u,v)}
      ++ as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}.
      ++ The options contained in the
      ++ list l of the domain \spad{DrawOption} are applied.
    draw: (PSFUN, SEG, SEG) -> VIEW3
      ++ draw(f,a..b,c..d) draws the
      ++ graph of the parametric surface \spad{f(u,v)}
      ++ as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}
      ++ The options contained in the list
      ++ l of the domain \spad{DrawOption} are applied.
    makeObject: (PSFUN, SEG, SEG, L DROP) -> SPACE3
      ++ makeObject(f,a..b,c..d,l) returns a
      ++ space of the domain \spadtype{ThreeSpace} which contains the
      ++ graph of the parametric surface \spad{f(u,v)}
      ++ as u ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)} and v ranges from \spad{min(c,d)} to \spad{max(c,d)};
      ++ The options contained in the
      ++ list l of the domain \spad{DrawOption} are applied.
    makeObject: (PSFUN, SEG, SEG) -> SPACE3
      ++ makeObject(f,a..b,c..d,l) returns a
      ++ space of the domain \spadtype{ThreeSpace} which contains the
      ++ graph of the parametric surface \spad{f(u,v)}
      ++ as u ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)} and v ranges from \spad{min(c,d)} to \spad{max(c,d)}.
    draw: (PSF,SEG,SEG,L DROP) -> VIEW3
      ++ draw(surface(f,g,h),a..b,c..d) draws the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)};
      ++ The options contained in the
      ++ list l of the domain \spad{DrawOption} are applied.
    draw: (PSF,SEG,SEG) -> VIEW3
      ++ draw(surface(f,g,h),a..b,c..d) draws the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
      ++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)};
```

```
    makeObject: (PSF,SEG,SEG,L DROP) -> SPACE3
      ++ makeObject(surface(f,g,h),a..b,c..d,l) returns a
      ++ space of the domain \spadtype{ThreeSpace} which contains the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)} and v ranges from \spad{min(c,d)} to \spad{max(c,d)}.
      ++ The options contained in the
      ++ list l of the domain \spad{DrawOption} are applied.
    makeObject: (PSF,SEG,SEG) -> SPACE3
      ++ makeObject(surface(f,g,h),a..b,c..d,l) returns a
      ++ space of the domain \spadtype{ThreeSpace} which contains the
      ++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
      ++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to
      ++ \spad{max(a,b)} and v ranges from \spad{min(c,d)} to \spad{max(c,d)}.
    recolor: ((SF,SF) -> Pt,(SF,SF,SF) -> SF) -> ((SF,SF) -> Pt)
      ++ recolor(), uninteresting to top level user; exported in order to
      ++ compile package.

  Implementation ==> add
```

———-

I have had to work my way around the following bug in the compiler: When a local variable is given a mapping as a value, e.g.

```
   foo : SF -> SF := makeFloatFunction(f,t),
```

the compiler cannot distinguish that local variable from a local function defined elsewhere in the package. Thus, when 'foo' is passed to a function, e.g.

```
  bird := fcn(foo),
```

foo will often be compiled as DRAW;foo rather than foo. This, of course, causes a run-time error.

To avoid this problem, local variables are not given mappings as values, but rather (singleton) lists of mappings. The first element of the list can always be extracted and everything goes through as before. There is no major loss in efficiency, as the computation of points will always dominate the computation time.
- cjw, 22 June MCMXC

— **package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions** —

```
    import PLOT
    import TwoDimensionalPlotClipping
    import GraphicsDefaults
    import ViewportPackage
    import ThreeDimensionalViewport
    import DrawOptionFunctions0
    import MakeFloatCompiledFunction(Ex)
```

```
    import MeshCreationRoutinesForThreeDimensions
    import SegmentFunctions2(SF,Float)
    import ViewDefaultsPackage
    import AnyFunctions1(Pt -> Pt)
    import AnyFunctions1((SF,SF,SF) -> SF)
    import DrawOptionFunctions0
    import SPACE3

    EXTOVARERROR : String := _
      "draw: when specifying function, left hand side must be a variable"
    SMALLRANGEERROR : String := _
      "draw: range is in interval with only one point"
    DEPVARERROR : String := _
      "draw: independent variable appears on lhs of function definition"

------------------------------------------------------------------------
--                    2D - draw's
------------------------------------------------------------------------

    drawToScaleRanges: (Segment SF,Segment SF) -> L SEG
    drawToScaleRanges(xVals,yVals) ==
      -- warning: assumes window is square
      xHi := convert(hi xVals)@Float; xLo := convert(lo xVals)@Float
      yHi := convert(hi yVals)@Float; yLo := convert(lo yVals)@Float
      xDiff := xHi - xLo; yDiff := yHi - yLo
      pad := abs(yDiff - xDiff)/2
      yDiff > xDiff =>
        [segment(xLo - pad,xHi + pad),map(x +-> convert(x)@Float,yVals)]
      [map(x +-> convert(x)@Float,xVals),segment(yLo - pad,yHi + pad)]

    drawPlot: (PLOT,L DROP) -> VIEW2
    drawPlot(plot,l) ==
      branches := listBranches plot
      xRange := xRange plot; yRange := yRange plot
      -- process clipping information
      if (cl := option(l,"clipSegment" :: Symbol)) case "failed" then
        if clipBoolean(l,clipPointsDefault()) then
          clipInfo :=
            parametric? plot => clipParametric plot
            clip plot
          branches := clipInfo.brans
          xRange := clipInfo.xValues; yRange := clipInfo.yValues
        else
          "No explicit user-specified clipping"
      else
        segList := retract(cl :: Any)$ANY1(L SEG)
        empty? segList =>
          error "draw: you may specify at least 1 segment for 2D clipping"
        more?(segList,2) =>
          error "draw: you may specify at most 2 segments for 2D clipping"
```

```
      xLo : SF := 0; xHi : SF := 0; yLo : SF := 0; yHi : SF := 0
      if empty? rest segList then
        xLo := lo xRange; xHi := hi xRange
        yRangeF := first segList
        yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
      else
        xRangeF := first segList
        xLo := convert(lo xRangeF)@SF; xHi := convert(hi xRangeF)@SF
        yRangeF := second segList
        yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
      clipInfo := clipWithRanges(branches,xLo,xHi,yLo,yHi)
      branches := clipInfo.brans
      xRange := clipInfo.xValues; yRange := clipInfo.yValues
    -- process scaling information
    if toScale(l,drawToScale()) then
      scaledRanges := drawToScaleRanges(xRange,yRange)
      -- add scaled ranges to list of options
      l := concat(ranges scaledRanges,l)
    else
      xRangeFloat : SEG := map(x +-> convert(x)@Float,xRange)
      yRangeFloat : SEG := map(x +-> convert(x)@Float,yRange)
      -- add ranges to list of options
      l := concat(ranges(ll : L SEG := [xRangeFloat,yRangeFloat]),l)
    -- process color information
    ptCol := pointColorPalette(l,pointColorDefault())
    crCol := curveColorPalette(l,lineColorDefault())
    -- draw
    drawCurves(branches,ptCol,crCol,pointSizeDefault(),l)

  normalize: SEG -> Segment SF
  normalize seg ==
    -- normalize [a,b]:
    -- error if a = b, returns [a,b] if a < b, returns [b,a] if b > a
    a := convert(lo seg)@SF; b := convert(hi seg)@SF
    a = b => error SMALLRANGEERROR
    a < b => segment(a,b)
    segment(b,a)
```

———-

The function `myTrap1` is a local function for used in creating maps SF -¿ Point SF (two dimensional). The range of this function is SingleFloat. As originally coded it would return `$NaNvalue$Lisp` which is outside the range. Since this function is only used internallly by the draw package we handle the "failed" case by returning zero. We handle the out-of-range case by returning the maximum or minimum SingleFloat value.

— package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions —

```
myTrap1: (SF-> SF, SF) -> SF
```

```
    myTrap1(ff:SF-> SF, f:SF):SF ==
      s := trapNumericErrors(ff(f))$Lisp :: Union(SF, "failed")
      s case "failed" => 0
      r:=s::SF
      r >max()$SF => max()$SF
      r < min()$SF => min()$SF
      r

    makePt2: (SF,SF) -> Point SF
    makePt2(x,y) == point(l : List SF := [x,y])

--% Two Dimensional Function Plots

    draw(f:SF -> SF,seg:SEG,l:L DROP) ==
      -- set adaptive plotting off or on
      oldAdaptive := adaptive?()$PLOT
      setAdaptive(adaptive(l,oldAdaptive))$PLOT
      -- create function SF -> Point SF
      ff : L(SF -> Point SF) := [x +-> makePt2(myTrap1(f,x),x)]
      -- process change of coordinates
      if (c := option(l,"coordinates" :: Symbol)) case "failed" then
        -- default coordinate transformation
        ff := [x +-> makePt2(x,myTrap1(f,x))]
      else
        cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
        ff := [x +-> (first cc)((first ff)(x))]
      -- create PLOT
      pl := pointPlot(first ff,normalize seg)
      -- reset adaptive plotting
      setAdaptive(oldAdaptive)$PLOT
      -- draw
      drawPlot(pl,l)

    draw(f:SF -> SF,seg:SEG) == draw(f,seg,nil())

--% Parametric Plane Curves

    draw(ppc:PPC,seg:SEG,l:L DROP) ==
      -- set adaptive plotting off or on
      oldAdaptive := adaptive?()$PLOT
      setAdaptive(adaptive(l,oldAdaptive))$PLOT
      -- create function SF -> Point SF
      f := coordinate(ppc,1); g := coordinate(ppc,2)
      fcn : L(SF -> Pt) := [x +-> makePt2(myTrap1(f,x),myTrap1(g,x))]
      -- process change of coordinates
      if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
        cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
        fcn := [x +-> (first cc)((first fcn)(x))]
      -- create PLOT
      pl := pointPlot(first fcn,normalize seg)
```

```
      -- reset adaptive plotting
      setAdaptive(oldAdaptive)$PLOT
      -- draw
      drawPlot(pl,l)

    draw(ppc:PPC,seg:SEG) == draw(ppc,seg,nil())

------------------------------------------------------------------------
--                     3D - Curves
------------------------------------------------------------------------

--% functions for creation of maps SF -> Point SF (three dimensional)

    makePt4: (SF,SF,SF,SF) -> Point SF
    makePt4(x,y,z,c) == point(l : List SF := [x,y,z,c])

--% Parametric Space Curves

    id: SF -> SF
    id x == x

    zCoord: (SF,SF,SF) -> SF
    zCoord(x,y,z) == z

    colorPoints: (List List Pt,(SF,SF,SF) -> SF) -> List List Pt
    colorPoints(llp,func) ==
      for lp in llp repeat for p in lp repeat
        p.4 := func(p.1,p.2,p.3)
      llp

    makeObject(psc:PSC,seg:SEG,l:L DROP) ==
      sp := space l
      -- obtain dependent variable and coordinate functions
      f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
      -- create function SF -> Point SF with default or user-specified
      -- color function
      fcn : L(SF -> Pt) := [x +-> makePt4(myTrap1(f,x),myTrap1(g,x),
                            myTrap1(h,x), myTrap1(id,x))]
      pointsColored? : Boolean := false
      if not (c1 := option(l,"colorFunction1" :: Symbol)) case "failed" then
        pointsColored? := true
        fcn := [x +-> makePt4(myTrap1(f,x),myTrap1(g,x),myTrap1(h,x),
                retract(c1 :: Any)$ANY1(SF -> SF)(x))]
      -- process change of coordinates
      if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
        cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
        fcn := [x +-> (first cc)((first fcn)(x))]
      -- create PLOT
      pl := pointPlot(first fcn,normalize seg)$Plot3D
      -- create ThreeSpace
```

```
      s := sp
      -- draw Tube
--      print(pl::OutputForm)
      option?(l,"tubeRadius" :: Symbol) =>
        pts := tubePoints(l,8)
        rad := convert(tubeRadius(l,0.25))@DoubleFloat
        tub := tube(pl,rad,pts)$NumericTubePlot(Plot3D)
        loops := listLoops tub
        -- color points if this has not been done already
        if not pointsColored? then
          if (c3 := option(l,"colorFunction3" :: Symbol)) case "failed"
            then colorPoints(loops,zCoord)  -- default color function
            else colorPoints(loops,retract(c3 :: Any)$ANY1((SF,SF,SF) -> SF))
        mesh(s,loops,false,false)
        s
      -- draw curve
      br := listBranches pl
      for b in br repeat curve(s,b)
      s

    makeObject(psc:PCFUN,seg:SEG,l:L DROP) ==
      sp := space l
      -- create function SF -> Point SF with default or user-specified
      -- color function
      fcn : L(SF -> Pt) := [psc]
      pointsColored? : Boolean := false
      if not (c1 := option(l,"colorFunction1" :: Symbol)) case "failed" then
        pointsColored? := true
        fcn := [x +-> concat(psc(x), retract(c1 :: Any)$ANY1(SF -> SF)(x))]
      -- process change of coordinates
      if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
        cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
        fcn := [x +-> (first cc)((first fcn)(x))]
      -- create PLOT
      pl := pointPlot(first fcn,normalize seg)$Plot3D
      -- create ThreeSpace
      s := sp
      -- draw Tube
      option?(l,"tubeRadius" :: Symbol) =>
        pts := tubePoints(l,8)
        rad := convert(tubeRadius(l,0.25))@DoubleFloat
        tub := tube(pl,rad,pts)$NumericTubePlot(Plot3D)
        loops := listLoops tub
        -- color points if this has not been done already
        mesh(s,loops,false,false)
        s
      -- draw curve
      br := listBranches pl
      for b in br repeat curve(s,b)
      s
```

```
    makeObject(psc:PSC,seg:SEG) ==
      makeObject(psc,seg,nil())

    makeObject(psc:PCFUN,seg:SEG) ==
      makeObject(psc,seg,nil())

    draw(psc:PSC,seg:SEG,l:L DROP) ==
      sp := makeObject(psc,seg,l)
      makeViewport3D(sp, l)

    draw(psc:PSC,seg:SEG) ==
      draw(psc,seg,nil())

    draw(psc:PCFUN,seg:SEG,l:L DROP) ==
      sp := makeObject(psc,seg,l)
      makeViewport3D(sp, l)

    draw(psc:PCFUN,seg:SEG) ==
      draw(psc,seg,nil())

------------------------------------------------------------------------
--                     3D - Surfaces
------------------------------------------------------------------------
```

———-

The function `myTrap2` is a local function for used in creating maps SF -¿ Point SF (three dimensional). The range of this function is SingleFloat. As originally coded it would return `$NaNvalue$Lisp` which is outside the range. Since this function is only used internallly by the draw package we handle the "failed" case by returning zero. We handle the out-of-range case by returning the maximum or minimum SingleFloat value.

— package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions —

```
    myTrap2: ((SF, SF) -> SF, SF, SF) -> SF
    myTrap2(ff:(SF, SF) -> SF, u:SF, v:SF):SF ==
      s := trapNumericErrors(ff(u, v))$Lisp :: Union(SF, "failed")
      s case "failed" => 0
      r:SF := s::SF
      r >max()$SF => max()$SF
      r < min()$SF => min()$SF
      r

    recolor(ptFunc,colFunc) ==
     (f1,f2) +->
       pt := ptFunc(f1,f2)
       pt.4 := colFunc(pt.1,pt.2,pt.3)
       pt
```

```
    xCoord: (SF,SF) -> SF
    xCoord(x,y) == x

--% Three Dimensional Function Plots

    makeObject(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG,l:L DROP) ==
      sp := space l
      -- process color function of two variables
      col2 : L((SF,SF) -> SF) := [xCoord]     -- dummy color function
      pointsColored? : Boolean := false
      if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
        pointsColored? := true
        col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
      fcn : L((SF,SF) -> Pt) :=
        [(x,y) +-> makePt4(myTrap2(f,x,y),x,y,(first col2)(x,y))]
      -- process change of coordinates
      if (c := option(l,"coordinates" :: Symbol)) case "failed" then
        -- default coordinate transformation
        fcn := [(x,y) +-> makePt4(x,y,myTrap2(f,x,y),(first col2)(x,y))]
      else
        cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
        fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
      -- process color function of three variables, if there was no
      -- color function of two variables
      if not pointsColored? then
        c := option(l,"colorFunction3" :: Symbol)
        fcn :=
          c case "failed" => [recolor((first fcn),zCoord)]
          [recolor((first fcn),retract(c :: Any)$ANY1((SF,SF,SF) -> SF))]
      -- create mesh
      mesh := meshPar2Var(sp,first fcn,normalize xSeg,normalize ySeg,l)
      mesh

    makeObject(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG) ==
      makeObject(f,xSeg,ySeg,nil())

    draw(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG,l:L DROP) ==
      sp := makeObject(f, xSeg, ySeg, l)
      makeViewport3D(sp, l)

    draw(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG) ==
      draw(f,xSeg,ySeg,nil())

--% parametric surface

    makeObject(s:PSF,uSeg:SEG,vSeg:SEG,l:L DROP) ==
      sp := space l
      -- create functions from expressions
      f : L((SF,SF) -> SF) := [coordinate(s,1)]
```

```
    g : L((SF,SF) -> SF) := [coordinate(s,2)]
    h : L((SF,SF) -> SF) := [coordinate(s,3)]
    -- process color function of two variables
    col2 : L((SF,SF) -> SF) := [xCoord]     -- dummy color function
    pointsColored? : Boolean := false
    if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
      pointsColored? := true
      col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
    fcn : L((SF,SF) -> Pt) :=
      [(x,y)+->makePt4(myTrap2((first f),x,y),myTrap2((first g),x,y),
                myTrap2((first h),x,y), myTrap2((first col2),x,y))]
    -- process change of coordinates
    if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
      cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
      fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
    -- process color function of three variables, if there was no
    -- color function of two variables
    if not pointsColored? then
      col3 : L((SF,SF,SF) -> SF) := [zCoord]  -- default color function
      if not (c := option(l,"colorFunction3" :: Symbol)) case "failed" then
        col3 := [retract(c :: Any)$ANY1((SF,SF,SF) -> SF)]
      fcn := [recolor((first fcn),(first col3))]
    -- create mesh
    mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
    mesh

  makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG,l:L DROP) ==
    sp := space l
    -- process color function of two variables
    col2 : L((SF,SF) -> SF) := [xCoord]     -- dummy color function
    pointsColored? : Boolean := false
    if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
      pointsColored? := true
      col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
    fcn : L((SF,SF) -> Pt) :=
      pointsColored? => [(x,y) +-> concat(s(x, y), (first col2)(x, y))]
      [s]
    -- process change of coordinates
    if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
      cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
      fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
    -- create mesh
    mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
    mesh

  makeObject(s:PSF,uSeg:SEG,vSeg:SEG) ==
    makeObject(s,uSeg,vSeg,nil())

  draw(s:PSF,uSeg:SEG,vSeg:SEG,l:L DROP) ==
    -- draw
```

```
      mesh := makeObject(s,uSeg,vSeg,l)
      makeViewport3D(mesh,l)

    draw(s:PSF,uSeg:SEG,vSeg:SEG) ==
      draw(s,uSeg,vSeg,nil())

    makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
      makeObject(s,uSeg,vSeg,nil())

    draw(s:PSFUN,uSeg:SEG,vSeg:SEG,l:L DROP) ==
      -- draw
      mesh := makeObject(s,uSeg,vSeg,l)
      makeViewport3D(mesh,l)

    draw(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
      draw(s,uSeg,vSeg,nil())
```

———-

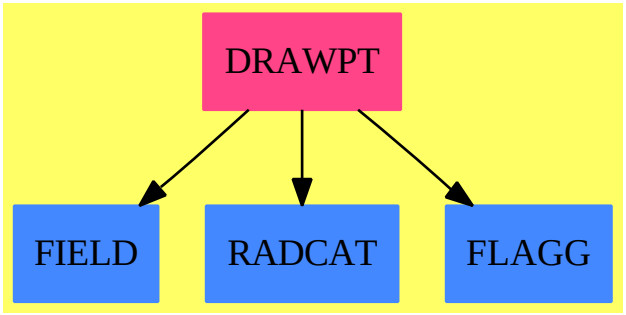
— **DRAWCFUN.dotabb** —

```
"DRAWCFUN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWCFUN"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DRAWCFUN" -> "ALIST"
```

———-

21.11 package DRAWPT TopLevelDrawFunctionsForPoints

21.11.1 TopLevelDrawFunctionsForPoints (DRAWPT)



Exports:
draw

— package DRAWPT TopLevelDrawFunctionsForPoints —

```
)abbrev package DRAWPT TopLevelDrawFunctionsForPoints
++ Author: Mike Dewar
++ Date Created: 24 May 1995
++ Date Last Updated: 25 November 1996
++ Basic Operations: draw
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ TopLevelDrawFunctionsForPoints provides top level functions
++ for drawing curves and surfaces described by sets of points.

TopLevelDrawFunctionsForPoints(): Exports == Implementation where

  DROP  ==> DrawOption
  L     ==> List
  SF    ==> DoubleFloat
  Pt    ==> Point SF
  VIEW2 ==> TwoDimensionalViewport
  VIEW3 ==> ThreeDimensionalViewport

  Exports ==> with
    draw: (L SF,L SF) -> VIEW2
      ++ draw(lx,ly) plots the curve constructed of points (x,y) for x
      ++ in \spad{lx} for y in \spad{ly}.
    draw: (L SF,L SF,L DROP) -> VIEW2
      ++ draw(lx,ly,l) plots the curve constructed of points (x,y) for x
      ++ in \spad{lx} for y in \spad{ly}.
      ++ The options contained in the list l of
      ++ the domain \spad{DrawOption} are applied.
    draw: (L Pt) -> VIEW2
      ++ draw(lp) plots the curve constructed from the list of points lp.
    draw: (L Pt,L DROP) -> VIEW2
      ++ draw(lp,l) plots the curve constructed from the list of points lp.
      ++ The options contained in the list l of the domain \spad{DrawOption}
      ++ are applied.
    draw: (L SF, L SF, L SF) -> VIEW3
      ++ draw(lx,ly,lz) draws the surface constructed by projecting the values
      ++ in the \axiom{lz} list onto the rectangular grid formed by the
      ++ \axiom{lx x ly}.
    draw: (L SF, L SF, L SF, L DROP) -> VIEW3
      ++ draw(lx,ly,lz,l) draws the surface constructed by
      ++ projecting the values
      ++ in the \axiom{lz} list onto the rectangular grid formed by the
      ++ The options contained in the list l of the domain \spad{DrawOption}
```

```
      ++ are applied.

  Implementation ==> add

    draw(lp:L Pt,l:L DROP):VIEW2 ==
      makeViewport2D(makeGraphImage([lp])$GraphImage,l)$VIEW2

    draw(lp:L Pt):VIEW2 == draw(lp,[])

    draw(lx: L SF, ly: L SF, l:L DROP):VIEW2 ==
      draw([point([x,y])$Pt for x in lx for y in ly],l)

    draw(lx: L SF, ly: L SF):VIEW2 == draw(lx,ly,[])

    draw(x:L SF,y:L SF,z:L SF):VIEW3 == draw(x,y,z,[])

    draw(x:L SF,y:L SF,z:L SF,l:L DROP):VIEW3 ==
      m  : Integer := #x
      zero? m => error "No X values"
      n  : Integer := #y
      zero? n => error "No Y values"
      zLen : Integer := #z
      zLen ~= (m*n) =>
        zLen > (m*n) => error "Too many Z-values to fit grid"
        error "Not enough Z-values to fit grid"
      points : L L Pt := []
      for j in n..1 by -1 repeat
        row : L Pt := []
        for i in m..1 by -1 repeat
          zval := (j-1)*m+i
          row := cons(point([x.i,y.j,z.zval,z.zval]),row)
        points := cons(row,points)
      makeViewport3D(mesh points,l)
```

———-

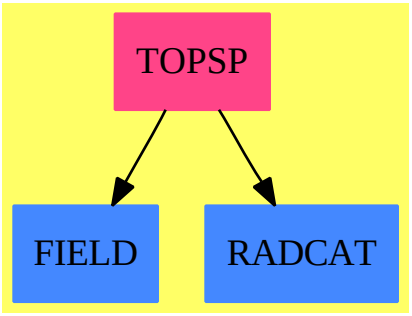
— DRAWPT.dotabb —

```
"DRAWPT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWPT"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"DRAWPT" -> "FIELD"
"DRAWPT" -> "RADCAT"
"DRAWPT" -> "FLAGG"
```

———-

21.12 package TOPSP TopLevelThreeSpace

21.12.1 TopLevelThreeSpace (TOPSP)



Exports:
createThreeSpace

— **package TOPSP TopLevelThreeSpace** —

```
)abbrev package TOPSP TopLevelThreeSpace
++ Description:
++ This package exports a function for making a \spadtype{ThreeSpace}

TopLevelThreeSpace(): with
    createThreeSpace: () -> ThreeSpace DoubleFloat
      ++ createThreeSpace() creates a \spadtype{ThreeSpace(DoubleFloat)} object
      ++ capable of holding point, curve, mesh components and any combination.
  == add
    createThreeSpace() == create3Space()$ThreeSpace(DoubleFloat)
```

————-

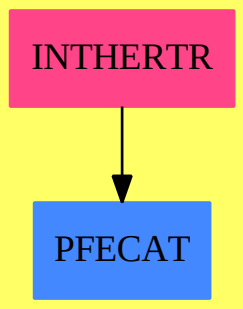
— **TOPSP.dotabb** —

```
"TOPSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOPSP"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"TOPSP" -> "FIELD"
"TOPSP" -> "RADCAT"
```

————-

21.13 package INTHERTR TranscendentalHermiteIntegration

21.13.1 TranscendentalHermiteIntegration (INTHERTR)



Exports:
HermiteIntegrate

— package INTHERTR TranscendentalHermiteIntegration —

```
)abbrev package INTHERTR TranscendentalHermiteIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 August 1992
++ Description:
++ Hermite integration, transcendental case.

TranscendentalHermiteIntegration(F, UP): Exports == Implementation where
  F  : Field
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  RF  ==> Fraction UP
  REC ==> Record(answer:RF, lognum:UP, logden:UP)
  HER ==> Record(answer:RF, logpart:RF, specpart:RF, polypart:UP)

  Exports ==> with
    HermiteIntegrate: (RF, UP -> UP) -> HER
         ++ HermiteIntegrate(f, D) returns \spad{[g, h, s, p]}
         ++ such that \spad{f = Dg + h + s + p},
         ++ h has a squarefree denominator normal w.r.t. D,
         ++ and all the squarefree factors of the denominator of s are
         ++ special w.r.t. D. Furthermore, h and s have no polynomial parts.
         ++ D is the derivation to use on \spadtype{UP}.
```

```
  Implementation ==> add
    import MonomialExtensionTools(F, UP)

    normalHermiteIntegrate: (RF,UP->UP) -> Record(answer:RF,lognum:UP,logden:UP)

    HermiteIntegrate(f, derivation) ==
      rec := decompose(f, derivation)
      hi  := normalHermiteIntegrate(rec.normal, derivation)
      qr  := divide(hi.lognum, hi.logden)
      [hi.answer, qr.remainder / hi.logden, rec.special, qr.quotient + rec.poly]

-- Hermite Reduction on f, every squarefree factor of denom(f) is normal wrt D
-- this is really a "parallel" Hermite reduction, in the sense that
-- every multiple factor of the denominator gets reduced at each pass
-- so if the denominator is P1 P2**2 ... Pn**n, this requires O(n)
-- reduction steps instead of O(n**2), like Mack's algorithm
-- (D.Mack, On Rational Integration, Univ. of Utah C.S. Tech.Rep. UCP-38,1975)
-- returns [g, b, d] s.t. f = g' + b/d and d is squarefree and normal wrt D
    normalHermiteIntegrate(f, derivation) ==
      a := numer f
      q := denom f
      p:UP    := 0
      mult:UP := 1
      qhat := (q exquo (g0 := g := gcd(q, differentiate q)))::UP
      while(degree(qbar := g) > 0) repeat
        qbarhat := (qbar exquo (g := gcd(qbar, differentiate qbar)))::UP
        qtil:= - ((qhat * (derivation qbar)) exquo qbar)::UP
        bc :=
         extendedEuclidean(qtil, qbarhat, a)::Record(coef1:UP, coef2:UP)
        qr := divide(bc.coef1, qbarhat)
        a  := bc.coef2 + qtil * qr.quotient - derivation(qr.remainder)
               * (qhat exquo qbarhat)::UP
        p  := p + mult * qr.remainder
        mult:= mult * qbarhat
      [p / g0, a, qhat]
```

———-

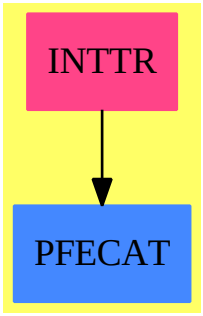
— **INTHERTR.dotabb** —

```
"INTHERTR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTHERTR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTHERTR" -> "PFECAT"
```

———-

21.14 package INTTR TranscendentalIntegration

21.14.1 TranscendentalIntegration (INTTR)



Exports:

expextendedint	expintegrate
expintfldpoly	explimitedint
monomialIntPoly	monomialIntegrate
primextendedint	primextintfrac
primintegrate	primintfldpoly
primlimintfrac	primlimitedint
tanintegrate	

— **package INTTR TranscendentalIntegration** —

```
)abbrev package INTTR TranscendentalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ This package provides functions for the transcendental
++ case of the Risch algorithm.
-- Internally used by the integrator

TranscendentalIntegration(F, UP): Exports == Implementation where
  F  : Field
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Z
  GP  ==> LaurentPolynomial(F, UP)
  UP2 ==> SparseUnivariatePolynomial UP
  RF  ==> Fraction UP
  UPR ==> SparseUnivariatePolynomial RF
  IR  ==> IntegrationResult RF
```

```
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
LLG ==> List Record(coeff:RF, logand:RF)
NE  ==> Record(integrand:RF, intvar:RF)
NL  ==> Record(mainpart:RF, limitedlogs:LLG)
UPF ==> Record(answer:UP, a0:F)
RFF ==> Record(answer:RF, a0:F)
IRF ==> Record(answer:IR, a0:F)
NLF ==> Record(answer:NL, a0:F)
GPF ==> Record(answer:GP, a0:F)
UPUP==> Record(elem:UP, notelem:UP)
GPGP==> Record(elem:GP, notelem:GP)
RFRF==> Record(elem:RF, notelem:RF)
FF  ==> Record(ratpart:F,  coeff:F)
FFR ==> Record(ratpart:RF, coeff:RF)
UF  ==> Union(FF,  "failed")
UF2 ==> Union(List F, "failed")
REC ==> Record(ir:IR, specpart:RF, polypart:UP)
PSOL==> Record(ans:F, right:F, sol?:Boolean)
FAIL==> error "Sorry - cannot handle that integrand yet"

Exports ==> with
  primintegrate  : (RF, UP -> UP, F -> UF)          -> IRF
    ++ primintegrate(f, ', foo) returns \spad{[g, a]} such that
    ++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in UP.
    ++ Argument foo is an extended integration function on F.
  expintegrate   : (RF, UP -> UP, (Z, F) -> PSOL) -> IRF
    ++ expintegrate(f, ', foo) returns \spad{[g, a]} such that
    ++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F;
    ++ Argument foo is a Risch differential equation solver on F;
  tanintegrate   : (RF, UP -> UP, (Z, F, F) -> UF2) -> IRF
    ++ tanintegrate(f, ', foo) returns \spad{[g, a]} such that
    ++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F;
    ++ Argument foo is a Risch differential system solver on F;
  primextendedint:(RF, UP -> UP, F->UF, RF) -> Union(RFF,FFR,"failed")
    ++ primextendedint(f, ', foo, g) returns either \spad{[v, c]} such that
    ++ \spad{f = v' + c g} and \spad{c' = 0}, or \spad{[v, a]} such that
    ++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in UP.
    ++ Returns "failed" if neither case can hold.
    ++ Argument foo is an extended integration function on F.
  expextendedint:(RF,UP->UP,(Z,F)->PSOL, RF) -> Union(RFF,FFR,"failed")
    ++ expextendedint(f, ', foo, g) returns either \spad{[v, c]} such that
    ++ \spad{f = v' + c g} and \spad{c' = 0}, or \spad{[v, a]} such that
    ++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in F.
    ++ Returns "failed" if neither case can hold.
    ++ Argument foo is a Risch differential equation function on F.
  primlimitedint:(RF, UP -> UP, F->UF, List RF) -> Union(NLF,"failed")
    ++ primlimitedint(f, ', foo, [u1,...,un]) returns
    ++ \spad{[v, [c1,...,cn], a]} such that \spad{ci' = 0},
    ++ \spad{f = v' + a + reduce(+,[ci * ui'/ui])},
    ++ and \spad{a = 0} or \spad{a} has no integral in UP.
```

```
    ++ Returns "failed" if no such v, ci, a exist.
    ++ Argument foo is an extended integration function on F.
  explimitedint:(RF, UP->UP,(Z,F)->PSOL,List RF) -> Union(NLF,"failed")
    ++ explimitedint(f, ', foo, [u1,...,un]) returns
    ++ \spad{[v, [c1,...,cn], a]} such that \spad{ci' = 0},
    ++ \spad{f = v' + a + reduce(+,[ci * ui'/ui])},
    ++ and \spad{a = 0} or \spad{a} has no integral in F.
    ++ Returns "failed" if no such v, ci, a exist.
    ++ Argument foo is a Risch differential equation function on F.
  primextintfrac    : (RF, UP -> UP, RF)  -> Union(FFR, "failed")
    ++ primextintfrac(f, ', g) returns \spad{[v, c]} such that
    ++ \spad{f = v' + c g} and \spad{c' = 0}.
    ++ Error: if \spad{degree numer f >= degree denom f} or
    ++ if \spad{degree numer g >= degree denom g} or
    ++ if \spad{denom g} is not squarefree.
  primlimintfrac    : (RF, UP -> UP, List RF) -> Union(NL, "failed")
    ++ primlimintfrac(f, ', [u1,...,un]) returns \spad{[v, [c1,...,cn]]}
    ++ such that \spad{ci' = 0} and \spad{f = v' + +/[ci * ui'/ui]}.
    ++ Error: if \spad{degree numer f >= degree denom f}.
  primintfldpoly    : (UP, F -> UF, F)    -> Union(UP, "failed")
    ++ primintfldpoly(p, ', t') returns q such that \spad{p' = q} or
    ++ "failed" if no such q exists. Argument \spad{t'} is the derivative of
    ++ the primitive generating the extension.
  expintfldpoly     : (GP, (Z, F) -> PSOL) -> Union(GP, "failed")
    ++ expintfldpoly(p, foo) returns q such that \spad{p' = q} or
    ++ "failed" if no such q exists.
    ++ Argument foo is a Risch differential equation function on F.
  monomialIntegrate : (RF, UP -> UP) -> REC
    ++ monomialIntegrate(f, ') returns \spad{[ir, s, p]} such that
    ++ \spad{f = ir' + s + p} and all the squarefree factors of the
    ++ denominator of s are special w.r.t the derivation '.
  monomialIntPoly   : (UP, UP -> UP) -> Record(answer:UP, polypart:UP)
    ++ monomialIntPoly(p, ') returns [q, r] such that
    ++ \spad{p = q' + r} and \spad{degree(r) < degree(t')}.
    ++ Error if \spad{degree(t') < 2}.

Implementation ==> add
  import SubResultantPackage(UP, UP2)
  import MonomialExtensionTools(F, UP)
  import TranscendentalHermiteIntegration(F, UP)
  import CommuteUnivariatePolynomialCategory(F, UP, UP2)

  primintegratepoly  : (UP, F -> UF, F) -> Union(UPF, UPUP)
  expintegratepoly   : (GP, (Z, F) -> PSOL) -> Union(GPF, GPGP)
  expextintfrac      : (RF, UP -> UP, RF) -> Union(FFR, "failed")
  explimintfrac      : (RF, UP -> UP, List RF) -> Union(NL, "failed")
  limitedLogs        : (RF, RF -> RF, List RF) -> Union(LLG, "failed")
  logprmderiv        : (RF, UP -> UP)    -> RF
  logexpderiv        : (RF, UP -> UP, F) -> RF
  tanintegratespecial: (RF, RF -> RF, (Z, F, F) -> UF2) -> Union(RFF, RFRF)
```

```
    UP2UP2             : UP -> UP2
    UP2UPR             : UP -> UPR
    UP22UPR            : UP2 -> UPR
    notelementary      : REC -> IR
    kappa              : (UP, UP -> UP) -> UP

    dummy:RF := 0

    logprmderiv(f, derivation) == differentiate(f, derivation) / f

    UP2UP2 p ==
      map(x+->x::UP, p)$UnivariatePolynomialCategoryFunctions2(F, UP, UP, UP2)

    UP2UPR p ==
      map(x+->x::UP::RF,p)$UnivariatePolynomialCategoryFunctions2(F,UP,RF,UPR)

    UP22UPR p ==
      map(x+->x::RF, p)$SparseUnivariatePolynomialFunctions2(UP, RF)

-- given p in k[z] and a derivation on k[t] returns the coefficient lifting
-- in k[z] of the restriction of D to k.
    kappa(p, derivation) ==
      ans:UP := 0
      while p ^= 0 repeat
        ans := ans + derivation(leadingCoefficient(p)::UP)*monomial(1,degree p)
        p := reductum p
      ans

-- works in any monomial extension
    monomialIntegrate(f, derivation) ==
      zero? f => [0, 0, 0]
      r := HermiteIntegrate(f, derivation)
      zero?(inum := numer(r.logpart)) => [r.answer::IR, r.specpart, r.polypart]
      iden  := denom(r.logpart)
      x := monomial(1, 1)$UP
      resultvec := subresultantVector(UP2UP2 inum -
                                (x::UP2) * UP2UP2 derivation iden, UP2UP2 iden)
      respoly := primitivePart leadingCoefficient resultvec 0
      rec := splitSquarefree(respoly, x1 +-> kappa(x1, derivation))
      logs:List(LOG) := [
            [1, UP2UPR(term.factor),
             UP22UPR swap primitivePart(resultvec(term.exponent),term.factor)]
                    for term in factors(rec.special)]
      dlog :=
--           one? derivation x => r.logpart
           ((derivation x) = 1) => r.logpart
           differentiate(mkAnswer(0, logs, empty()),
                         (x1:RF):RF +-> differentiate(x1, derivation))
      (u := retractIfCan(p := r.logpart - dlog)@Union(UP, "failed")) case UP =>
        [mkAnswer(r.answer, logs, empty), r.specpart, r.polypart + u::UP]
```

```
      [mkAnswer(r.answer, logs, [[p, dummy]]), r.specpart, r.polypart]

-- returns [q, r] such that p = q' + r and degree(r) < degree(dt)
-- must have degree(derivation t) >= 2
    monomialIntPoly(p, derivation) ==
      (d := degree(dt := derivation monomial(1,1))::Z) < 2 =>
        error "monomIntPoly: monomial must have degree 2 or more"
      l := leadingCoefficient dt
      ans:UP := 0
      while (n := 1 + degree(p)::Z - d) > 0 repeat
        ans := ans + (term := monomial(leadingCoefficient(p) / (n * l), n::N))
        p   := p - derivation term      -- degree(p) must drop here
      [ans, p]

-- returns either
--   (q in GP, a in F)  st p = q' + a, and a=0 or a has no integral in F
-- or (q in GP, r in GP) st p = q' + r, and r has no integral elem/UP
    expintegratepoly(p, FRDE) ==
      coef0:F := 0
      notelm := answr := 0$GP
      while p ^= 0 repeat
        ans1 := FRDE(n := degree p, a := leadingCoefficient p)
        answr := answr + monomial(ans1.ans, n)
        if ~ans1.sol? then         -- Risch d.e. has no complete solution
               missing := a - ans1.right
               if zero? n then coef0 := missing
                          else notelm := notelm + monomial(missing, n)
        p   := reductum p
      zero? notelm => [answr, coef0]
      [answr, notelm]

-- f is either 0 or of the form p(t)/(1 + t**2)**n
-- returns either
--   (q in RF, a in F)  st f = q' + a, and a=0 or a has no integral in F
-- or (q in RF, r in RF) st f = q' + r, and r has no integral elem/UP
    tanintegratespecial(f, derivation, FRDE) ==
      ans:RF := 0
      p := monomial(1, 2)$UP + 1
      while (n := degree(denom f) quo 2) ^= 0 repeat
        r := numer(f) rem p
        a := coefficient(r, 1)
        b := coefficient(r, 0)
        (u := FRDE(n, a, b)) case "failed" => return [ans, f]
        l := u::List(F)
        term:RF := (monomial(first l, 1)$UP + second(l)::UP) / denom f
        ans := ans + term
        f   := f - derivation term    -- the order of the pole at 1+t^2 drops
      zero?(c0 := retract(retract(f)@UP)@F) or
        (u := FRDE(0, c0, 0)) case "failed" => [ans, c0]
      [ans + first(u::List(F))::UP::RF, 0::F]
```

```
-- returns (v in RF, c in RF) s.t. f = v' + cg, and c' = 0, or "failed"
-- g must have a squarefree denominator (always possible)
-- g must have no polynomial part and no pole above t = 0
-- f must have no polynomial part and no pole above t = 0
    expextintfrac(f, derivation, g) ==
      zero? f => [0, 0]
      degree numer f >= degree denom f => error "Not a proper fraction"
      order(denom f,monomial(1,1)) ^= 0 => error "Not integral at t = 0"
      r := HermiteIntegrate(f, derivation)
      zero? g =>
        r.logpart ^= 0 => "failed"
        [r.answer, 0]
      degree numer g >= degree denom g => error "Not a proper fraction"
      order(denom g,monomial(1,1)) ^= 0 => error "Not integral at t = 0"
      differentiate(c := r.logpart / g, derivation) ^= 0 => "failed"
      [r.answer, c]

    limitedLogs(f, logderiv, lu) ==
      zero? f => empty()
      empty? lu => "failed"
      empty? rest lu =>
        logderiv(c0 := f / logderiv(u0 := first lu)) ^= 0 => "failed"
        [[c0, u0]]
      num := numer f
      den := denom f
      l1:List Record(logand2:RF, contrib:UP) :=
--        [[u, numer v] for u in lu | one? denom(v := den * logderiv u)]
        [[u, numer v] for u in lu | (denom(v := den * logderiv u) = 1)]
      rows := max(degree den,
                  1 + reduce(max, [degree(u.contrib) for u in l1], 0)$List(N))
      m:Matrix(F) := zero(rows, cols := 1 + #l1)
      for i in 0..rows-1 repeat
        for pp in l1 for j in minColIndex m .. maxColIndex m - 1 repeat
          qsetelt_!(m, i + minRowIndex m, j, coefficient(pp.contrib, i))
        qsetelt_!(m,i+minRowIndex m, maxColIndex m, coefficient(num, i))
      m := rowEchelon m
      ans := empty()$LLG
      for i in minRowIndex m .. maxRowIndex m |
       qelt(m, i, maxColIndex m) ^= 0 repeat
        OK := false
        for pp in l1 for j in minColIndex m .. maxColIndex m - 1
         while not OK repeat
          if qelt(m, i, j) ^= 0 then
            OK := true
            c := qelt(m, i, maxColIndex m) / qelt(m, i, j)
            logderiv(c0 := c::UP::RF) ^= 0 => return "failed"
            ans := concat([c0, pp.logand2], ans)
        not OK => return "failed"
      ans
```

```
-- returns q in UP s.t. p = q', or "failed"
    primintfldpoly(p, extendedint, t') ==
      (u := primintegratepoly(p, extendedint, t')) case UPUP => "failed"
      u.a0 ^= 0 => "failed"
      u.answer

-- returns q in GP st p = q', or "failed"
    expintfldpoly(p, FRDE) ==
      (u := expintegratepoly(p, FRDE)) case GPGP => "failed"
      u.a0 ^= 0 => "failed"
      u.answer

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + +/[ci * ui'/ui]
--                                  and a = 0 or a has no integral in UP
    primlimitedint(f, derivation, extendedint, lu) ==
      qr := divide(numer f, denom f)
      (u1 := primlimintfrac(qr.remainder / (denom f), derivation, lu))
        case "failed" => "failed"
      (u2 := primintegratepoly(qr.quotient, extendedint,
               retract derivation monomial(1, 1))) case UPUP => "failed"
      [[u1.mainpart + u2.answer::RF, u1.limitedlogs], u2.a0]

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + +/[ci * ui'/ui]
--                                   and a = 0 or a has no integral in F
    explimitedint(f, derivation, FRDE, lu) ==
      qr := separate(f)$GP
      (u1 := explimintfrac(qr.fracPart,derivation, lu)) case "failed" =>
                                                              "failed"
      (u2 := expintegratepoly(qr.polyPart, FRDE)) case GPGP => "failed"
      [[u1.mainpart + convert(u2.answer)@RF, u1.limitedlogs], u2.a0]

-- returns [v, c1...cn] s.t. f = v' + +/[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
    primlimintfrac(f, derivation, lu) ==
      zero? f => [0, empty()]
      degree numer f >= degree denom f => error "Not a proper fraction"
      r := HermiteIntegrate(f, derivation)
      zero?(r.logpart) => [r.answer, empty()]
      (u := limitedLogs(r.logpart, x1 +-> logprmderiv(x1, derivation), lu))
        case "failed" => "failed"
      [r.answer, u::LLG]

-- returns [v, c1...cn] s.t. f = v' + +/[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
-- f must be integral above t = 0
    explimintfrac(f, derivation, lu) ==
      zero? f => [0, empty()]
```

```
      degree numer f >= degree denom f => error "Not a proper fraction"
      order(denom f, monomial(1,1)) > 0 => error "Not integral at t = 0"
      r  := HermiteIntegrate(f, derivation)
      zero?(r.logpart) => [r.answer, empty()]
      eta' := coefficient(derivation monomial(1, 1), 1)
      (u := limitedLogs(r.logpart, x1 +-> logexpderiv(x1,derivation,eta'), lu))
        case "failed" => "failed"
      [r.answer - eta'::UP *
        +/[((degree numer(v.logand))::Z - (degree denom(v.logand))::Z) *
                                          v.coeff for v in u], u::LLG]

    logexpderiv(f, derivation, eta') ==
      (differentiate(f, derivation) / f) -
            (((degree numer f)::Z - (degree denom f)::Z) * eta')::UP::RF

    notelementary rec ==
      rec.ir + integral(rec.polypart::RF + rec.specpart, monomial(1,1)$UP :: RF)

-- returns
--   (g in IR, a in F)  st f = g'+ a, and a=0 or a has no integral in UP
    primintegrate(f, derivation, extendedint) ==
      rec := monomialIntegrate(f, derivation)
      not elem?(i1 := rec.ir) => [notelementary rec, 0]
      (u2 := primintegratepoly(rec.polypart, extendedint,
                        retract derivation monomial(1, 1))) case UPUP =>
             [i1 + u2.elem::RF::IR
                 + integral(u2.notelem::RF, monomial(1,1)$UP :: RF), 0]
      [i1 + u2.answer::RF::IR, u2.a0]

-- returns
--   (g in IR, a in F)  st f = g' + a, and a = 0 or a has no integral in F
    expintegrate(f, derivation, FRDE) ==
      rec := monomialIntegrate(f, derivation)
      not elem?(i1 := rec.ir) => [notelementary rec, 0]
-- rec.specpart is either 0 or of the form p(t)/t**n
      special := rec.polypart::GP +
                   (numer(rec.specpart)::GP exquo denom(rec.specpart)::GP)::GP
      (u2 := expintegratepoly(special, FRDE)) case GPGP =>
        [i1 + convert(u2.elem)@RF::IR + integral(convert(u2.notelem)@RF,
                                                 monomial(1,1)$UP :: RF), 0]
      [i1 + convert(u2.answer)@RF::IR, u2.a0]

-- returns
--   (g in IR, a in F)  st f = g' + a, and a = 0 or a has no integral in F
    tanintegrate(f, derivation, FRDE) ==
      rec := monomialIntegrate(f, derivation)
      not elem?(i1 := rec.ir) => [notelementary rec, 0]
      r := monomialIntPoly(rec.polypart, derivation)
      t := monomial(1, 1)$UP
      c := coefficient(r.polypart, 1) / leadingCoefficient(derivation t)
```

```
      derivation(c::UP) ^= 0 =>
        [i1 + mkAnswer(r.answer::RF, empty(),
                       [[r.polypart::RF + rec.specpart, dummy]$NE]), 0]
      logs:List(LOG) :=
        zero? c => empty()
        [[1, monomial(1,1)$UPR - (c/(2::F))::UP::RF::UPR, (1 + t**2)::RF::UPR]]
      c0 := coefficient(r.polypart, 0)
      (u := tanintegratespecial(rec.specpart, x+->differentiate(x, derivation),
       FRDE)) case RFRF =>
        [i1+mkAnswer(r.answer::RF + u.elem, logs, [[u.notelem,dummy]$NE]), c0]
      [i1 + mkAnswer(r.answer::RF + u.answer, logs, empty()), u.a0 + c0]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
--             or (v in RF, a in F)  s.t. f = v' + a
--                                   and a = 0 or a has no integral in UP
    primextendedint(f, derivation, extendedint, g) ==
      fqr := divide(numer f, denom f)
      gqr := divide(numer g, denom g)
      (u1 := primextintfrac(fqr.remainder / (denom f), derivation,
                  gqr.remainder / (denom g))) case "failed" => "failed"
      zero?(gqr.remainder) =>
      -- the following FAIL cannot occur if the primitives are all logs
         degree(gqr.quotient) > 0 => FAIL
         (u3 := primintegratepoly(fqr.quotient, extendedint,
               retract derivation monomial(1, 1))) case UPUP => "failed"
         [u1.ratpart + u3.answer::RF, u3.a0]
      (u2 := primintfldpoly(fqr.quotient - retract(u1.coeff)@UP *
        gqr.quotient, extendedint, retract derivation monomial(1, 1)))
          case "failed" => "failed"
      [u2::UP::RF + u1.ratpart, u1.coeff]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
--             or (v in RF, a in F)  s.t. f = v' + a
--                                   and a = 0 or a has no integral in F
    expextendedint(f, derivation, FRDE, g) ==
      qf := separate(f)$GP
      qg := separate g
      (u1 := expextintfrac(qf.fracPart, derivation, qg.fracPart))
         case "failed" => "failed"
      zero?(qg.fracPart) =>
      --the following FAIL's cannot occur if the primitives are all logs
        retractIfCan(qg.polyPart)@Union(F,"failed") case "failed"=> FAIL
        (u3 := expintegratepoly(qf.polyPart,FRDE)) case GPGP => "failed"
        [u1.ratpart + convert(u3.answer)@RF, u3.a0]
      (u2 := expintfldpoly(qf.polyPart - retract(u1.coeff)@UP :: GP
        * qg.polyPart, FRDE)) case "failed" => "failed"
      [convert(u2::GP)@RF + u1.ratpart, u1.coeff]

-- returns either
--   (q in UP, a in F)  st p = q'+ a, and a=0 or a has no integral in UP
```

```
-- or (q in UP, r in UP) st p = q'+ r, and r has no integral elem/UP
    primintegratepoly(p, extendedint, t') ==
      zero? p => [0, 0$F]
      ans:UP := 0
      while (d := degree p) > 0 repeat
        (ans1 := extendedint leadingCoefficient p) case "failed" =>
          return([ans, p])
        p   := reductum p - monomial(d * t' * ans1.ratpart, (d - 1)::N)
        ans := ans + monomial(ans1.ratpart, d)
                           + monomial(ans1.coeff / (d + 1)::F, d + 1)
      (ans1:= extendedint(rp := retract(p)@F)) case "failed" => [ans,rp]
      [monomial(ans1.coeff, 1) + ans1.ratpart::UP + ans, 0$F]

-- returns (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- g must have a squarefree denominator (always possible)
-- g must have no polynomial part (degree numer g < degree denom g)
-- f must have no polynomial part (degree numer f < degree denom f)
    primextintfrac(f, derivation, g) ==
      zero? f => [0, 0]
      degree numer f >= degree denom f => error "Not a proper fraction"
      r := HermiteIntegrate(f, derivation)
      zero? g =>
        r.logpart ^= 0 => "failed"
        [r.answer, 0]
      degree numer g >= degree denom g => error "Not a proper fraction"
      differentiate(c := r.logpart / g, derivation) ^= 0 => "failed"
      [r.answer, c]
```

———-

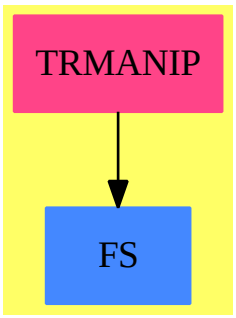
— INTTR.dotabb —

```
"INTTR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTTR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTTR" -> "PFECAT"
```

———-

21.15 package TRMANIP TranscendentalManipulations

21.15.1 TranscendentalManipulations (TRMANIP)



Exports:

cos2sec	cosh2sech	cot2tan
cot2trig	coth2tanh	coth2trigh
csc2sin	csch2sinh	expand
expandLog	expandPower	htrigs
removeCosSq	removeCoshSq	removeSinSq
removeSinhSq	sec2cos	sech2cosh
simplify	simplifyExp	simplifyLog
sin2csc	sinh2csch	tan2cot
tan2trig	tanh2coth	tanh2trigh
expandTrigProducts		

— package TRMANIP TranscendentalManipulations —

```
)abbrev package TRMANIP TranscendentalManipulations
++ Author: Bob Sutor, Manuel Bronstein
++ Date Created: Way back
++ Date Last Updated: 22 January 1996, added simplifyLog MCD.
++ Keywords: transcendental, manipulation.
++ Description:
++ TranscendentalManipulations provides functions to simplify and
++ expand expressions involving transcendental operators.

TranscendentalManipulations(R, F): Exports == Implementation where
  R : Join(OrderedSet, GcdDomain)
  F : Join(FunctionSpace R, TranscendentalFunctionCategory)

  Z       ==> Integer
  K       ==> Kernel F
  P       ==> SparseMultivariatePolynomial(R, K)
  UP      ==> SparseUnivariatePolynomial P
  POWER   ==> "%power"::Symbol
```

```
POW     ==> Record(val: F,exponent: Z)
PRODUCT ==> Record(coef : Z, var : K)
FPR     ==> Fraction Polynomial R

Exports ==> with
  expand     : F -> F
    ++ expand(f) performs the following expansions on f:\begin{items}
    ++ \item 1. logs of products are expanded into sums of logs,
    ++ \item 2. trigonometric and hyperbolic trigonometric functions
    ++ of sums are expanded into sums of products of trigonometric
    ++ and hyperbolic trigonometric functions.
    ++ \item 3. formal powers of the form \spad{(a/b)**c} are expanded into
    ++ \spad{a**c * b**(-c)}.
    ++ \end{items}
  simplify   : F -> F
    ++ simplify(f) performs the following simplifications on f:\begin{items}
    ++ \item 1. rewrites trigs and hyperbolic trigs in terms
    ++ of \spad{sin} ,\spad{cos}, \spad{sinh}, \spad{cosh}.
    ++ \item 2. rewrites \spad{sin**2} and \spad{sinh**2} in terms
    ++ of \spad{cos} and \spad{cosh},
    ++ \item 3. rewrites \spad{exp(a)*exp(b)} as \spad{exp(a+b)}.
    ++ \item 4. rewrites \spad{(a**(1/n))**m * (a**(1/s))**t} as a single
    ++ power of a single radical of \spad{a}.
    ++ \end{items}
  htrigs     : F -> F
    ++ htrigs(f) converts all the exponentials in f into
    ++ hyperbolic sines and cosines.
  simplifyExp: F -> F
    ++ simplifyExp(f) converts every product \spad{exp(a)*exp(b)}
    ++ appearing in f into \spad{exp(a+b)}.
  simplifyLog : F -> F
    ++ simplifyLog(f) converts every \spad{log(a) - log(b)} appearing in f
    ++ into \spad{log(a/b)}, every \spad{log(a) + log(b)} into \spad{log(a*b)}
    ++ and every \spad{n*log(a)} into \spad{log(a^n)}.
  expandPower: F -> F
    ++ expandPower(f) converts every power \spad{(a/b)**c} appearing
    ++ in f into \spad{a**c * b**(-c)}.
  expandLog  : F -> F
    ++ expandLog(f) converts every \spad{log(a/b)} appearing in f into
    ++ \spad{log(a) - log(b)}, and every \spad{log(a*b)} into
    ++ \spad{log(a) + log(b)}..
  cos2sec    : F -> F
    ++ cos2sec(f) converts every \spad{cos(u)} appearing in f into
    ++ \spad{1/sec(u)}.
  cosh2sech  : F -> F
    ++ cosh2sech(f) converts every \spad{cosh(u)} appearing in f into
    ++ \spad{1/sech(u)}.
  cot2trig   : F -> F
    ++ cot2trig(f) converts every \spad{cot(u)} appearing in f into
    ++ \spad{cos(u)/sin(u)}.
```

```
coth2trigh : F -> F
  ++ coth2trigh(f) converts every \spad{coth(u)} appearing in f into
  ++ \spad{cosh(u)/sinh(u)}.
csc2sin    : F -> F
  ++ csc2sin(f) converts every \spad{csc(u)} appearing in f into
  ++ \spad{1/sin(u)}.
csch2sinh  : F -> F
  ++ csch2sinh(f) converts every \spad{csch(u)} appearing in f into
  ++ \spad{1/sinh(u)}.
sec2cos    : F -> F
  ++ sec2cos(f) converts every \spad{sec(u)} appearing in f into
  ++ \spad{1/cos(u)}.
sech2cosh  : F -> F
  ++ sech2cosh(f) converts every \spad{sech(u)} appearing in f into
  ++ \spad{1/cosh(u)}.
sin2csc    : F -> F
  ++ sin2csc(f) converts every \spad{sin(u)} appearing in f into
  ++ \spad{1/csc(u)}.
sinh2csch  : F -> F
  ++ sinh2csch(f) converts every \spad{sinh(u)} appearing in f into
  ++ \spad{1/csch(u)}.
tan2trig   : F -> F
  ++ tan2trig(f) converts every \spad{tan(u)} appearing in f into
  ++ \spad{sin(u)/cos(u)}.
tanh2trigh : F -> F
  ++ tanh2trigh(f) converts every \spad{tanh(u)} appearing in f into
  ++ \spad{sinh(u)/cosh(u)}.
tan2cot    : F -> F
  ++ tan2cot(f) converts every \spad{tan(u)} appearing in f into
  ++ \spad{1/cot(u)}.
tanh2coth  : F -> F
  ++ tanh2coth(f) converts every \spad{tanh(u)} appearing in f into
  ++ \spad{1/coth(u)}.
cot2tan    : F -> F
  ++ cot2tan(f) converts every \spad{cot(u)} appearing in f into
  ++ \spad{1/tan(u)}.
coth2tanh  : F -> F
  ++ coth2tanh(f) converts every \spad{coth(u)} appearing in f into
  ++ \spad{1/tanh(u)}.
removeCosSq: F -> F
  ++ removeCosSq(f) converts every \spad{cos(u)**2} appearing in f into
  ++ \spad{1 - sin(x)**2}, and also reduces higher
  ++ powers of \spad{cos(u)} with that formula.
removeSinSq: F -> F
  ++ removeSinSq(f) converts every \spad{sin(u)**2} appearing in f into
  ++ \spad{1 - cos(x)**2}, and also reduces higher powers of
  ++ \spad{sin(u)} with that formula.
removeCoshSq:F -> F
  ++ removeCoshSq(f) converts every \spad{cosh(u)**2} appearing in f into
  ++ \spad{1 - sinh(x)**2}, and also reduces higher powers of
```

```
    ++ \spad{cosh(u)} with that formula.
  removeSinhSq:F -> F
    ++ removeSinhSq(f) converts every \spad{sinh(u)**2} appearing in f into
    ++ \spad{1 - cosh(x)**2}, and also reduces higher powers
    ++ of \spad{sinh(u)} with that formula.
  if R has PatternMatchable(R) and R has ConvertibleTo(Pattern(R))
   and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then
    expandTrigProducts : F -> F
      ++ expandTrigProducts(e) replaces \axiom{sin(x)*sin(y)} by
      ++ \spad{(cos(x-y)-cos(x+y))/2}, \axiom{cos(x)*cos(y)} by
      ++ \spad{(cos(x-y)+cos(x+y))/2}, and \axiom{sin(x)*cos(y)} by
      ++ \spad{(sin(x-y)+sin(x+y))/2}.  Note that this operation uses
      ++ the pattern matcher and so is relatively expensive.  To avoid
      ++ getting into an infinite loop the transformations are applied
      ++ at most ten times.

Implementation ==> add
  import FactoredFunctions(P)
  import PolynomialCategoryLifting(IndexedExponents K, K, R, P, F)
  import
    PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,P,F)

  smpexp    : P -> F
  termexp   : P -> F
  exlog     : P -> F
  smplog    : P -> F
  smpexpand : P -> F
  smp2htrigs: P -> F
  kerexpand : K -> F
  expandpow : K -> F
  logexpand : K -> F
  sup2htrigs: (UP, F) -> F
  supexp    : (UP, F, F, Z) -> F
  ueval     : (F, String, F -> F) -> F
  ueval2    : (F, String, F -> F) -> F
  powersimp : (P, List K) -> F
  t2t       : F -> F
  c2t       : F -> F
  c2s       : F -> F
  s2c       : F -> F
  s2c2      : F -> F
  th2th     : F -> F
  ch2th     : F -> F
  ch2sh     : F -> F
  sh2ch     : F -> F
  sh2ch2    : F -> F
  simplify0 : F -> F
  simplifyLog1 : F -> F
  logArgs   : List F -> F
```

```
import F
import List F

if R has PatternMatchable R and R has ConvertibleTo Pattern R
 and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then
  XX : F := coerce new()$Symbol
  YY : F := coerce new()$Symbol
  sinCosRule : RewriteRule(R,R,F) :=
    rule(cos(XX)*sin(YY),(sin(XX+YY)-sin(XX-YY))/2::F)
  sinSinRule : RewriteRule(R,R,F) :=
    rule(sin(XX)*sin(YY),(cos(XX-YY)-cos(XX+YY))/2::F)
  cosCosRule : RewriteRule(R,R,F) :=
    rule(cos(XX)*cos(YY),(cos(XX-YY)+cos(XX+YY))/2::F)
  expandTrigProducts(e:F):F ==
    applyRules([sinCosRule,sinSinRule,cosCosRule],e,10)$ApplyRules(R,R,F)

logArgs(l:List F):F ==
  -- This function will take a list of Expressions (implicitly a sum) and
  -- add them up, combining log terms.  It also replaces n*log(x) by
  -- log(x^n).
  import K
  sum  : F := 0
  arg  : F := 1
  for term in l repeat
    is?(term,"log"::Symbol) =>
      arg := arg * simplifyLog(first(argument(first(kernels(term)))))
    -- Now look for multiples, including negative ones.
    prod : Union(PRODUCT, "failed") := isMult(term)
    (prod case PRODUCT) and is?(prod.var,"log"::Symbol) =>
        arg := arg * simplifyLog ((first argument(prod.var))**(prod.coef))
    sum := sum+term
  sum+log(arg)

simplifyLog(e:F):F ==
  simplifyLog1(numerator e)/simplifyLog1(denominator e)

simplifyLog1(e:F):F ==
  freeOf?(e,"log"::Symbol) => e

  -- Check for n*log(u)
  prod : Union(PRODUCT, "failed") := isMult(e)
  (prod case PRODUCT) and is?(prod.var,"log"::Symbol) =>
    log simplifyLog ((first argument(prod.var))**(prod.coef))

  termList : Union(List(F),"failed") := isTimes(e)
  -- I'm using two variables, termList and terms, to work round a
  -- bug in the old compiler.
  not (termList case "failed") =>
    -- We want to simplify each log term in the product and then multiply
    -- them together.  However, if there is a constant or arithmetic
```

```
        -- expression (i.e. somwthing which looks like a Polynomial) we would
        -- like to combine it with a log term.
        terms :List F := [simplifyLog(term) for term in termList::List(F)]
        exprs :List F := []
        for i in 1..#terms repeat
          if retractIfCan(terms.i)@Union(FPR,"failed") case FPR then
            exprs := cons(terms.i,exprs)
            terms := delete!(terms,i)
        if not empty? exprs then
          foundLog := false
          i : NonNegativeInteger := 0
          while (not(foundLog) and (i < #terms)) repeat
            i := i+1
            if is?(terms.i,"log"::Symbol) then
              args : List F := argument(retract(terms.i)@K)
              setelt(terms,i, log simplifyLog1(first(args)**(*/exprs)))
              foundLog := true
          -- The next line deals with a situation which shouldn't occur,
          -- since we have checked whether we are freeOf log already.
          if not foundLog then terms := append(exprs,terms)
        */terms

      terms : Union(List(F),"failed") := isPlus(e)
      not (terms case "failed") => logArgs(terms)

      expt : Union(POW, "failed") := isPower(e)
--      (expt case POW) and not one? expt.exponent =>
      (expt case POW) and not (expt.exponent = 1) =>
        simplifyLog(expt.val)**(expt.exponent)

      kers : List K := kernels e
--      not(one?(#kers)) => e -- Have a constant
      not(((#kers) = 1)) => e -- Have a constant
      kernel(operator first kers,[simplifyLog(u) for u in argument first kers])


    if R has RetractableTo Integer then
      simplify x == rootProduct(simplify0 x)$AlgebraicManipulations(R,F)

    else simplify x == simplify0 x

    expandpow k ==
      a := expandPower first(arg := argument k)
      b := expandPower second arg
--      ne:F := (one? numer a => 1; numer(a)::F ** b)
      ne:F := (((numer a) = 1) => 1; numer(a)::F ** b)
--      de:F := (one? denom a => 1; denom(a)::F ** (-b))
      de:F := (((denom a) = 1) => 1; denom(a)::F ** (-b))
      ne * de
```

```
    termexp p ==
      exponent:F := 0
      coef := (leadingCoefficient p)::P
      lpow := select((z:K):Boolean+->is?(z,POWER)$K, lk := variables p)$List(K)
      for k in lk repeat
        d := degree(p, k)
        if is?(k, "exp"::Symbol) then
          exponent := exponent + d * first argument k
        else if not is?(k, POWER) then
          -- Expand arguments to functions as well ... MCD 23/1/97
          --coef := coef * monomial(1, k, d)
          coef := coef *
           monomial(1,
             kernel(operator k,
               [simplifyExp u for u in argument k], height k), d)
      coef::F * exp exponent * powersimp(p, lpow)

    expandPower f ==
      l := select((z:K):Boolean +-> is?(z, POWER)$K, kernels f)$List(K)
      eval(f, l, [expandpow k for k in l])

-- l is a list of pure powers appearing as kernels in p
    powersimp(p, l) ==
      empty? l => 1
      k := first l                          -- k = a**b
      a := first(arg := argument k)
      exponent := degree(p, k) * second arg
      empty?(lk := select((z:K):Boolean +-> a = first argument z, rest l)) =>
        (a ** exponent) * powersimp(p, rest l)
      for k0 in lk repeat
        exponent := exponent + degree(p, k0) * second argument k0
      (a ** exponent) * powersimp(p, setDifference(rest l, lk))

    t2t x         == sin(x) / cos(x)
    c2t x         == cos(x) / sin(x)
    c2s x         == inv sin x
    s2c x         == inv cos x
    s2c2 x        == 1 - cos(x)**2
    th2th x       == sinh(x) / cosh(x)
    ch2th x       == cosh(x) / sinh(x)
    ch2sh x       == inv sinh x
    sh2ch x       == inv cosh x
    sh2ch2 x      == cosh(x)**2 - 1
    ueval(x, s,f) == eval(x, s::Symbol, f)
    ueval2(x,s,f) == eval(x, s::Symbol, 2, f)
    cos2sec x     == ueval(x, "cos", (z1:F):F +-> inv sec z1)
    sin2csc x     == ueval(x, "sin", (z1:F):F +-> inv csc z1)
    csc2sin x     == ueval(x, "csc", c2s)
    sec2cos x     == ueval(x, "sec", s2c)
    tan2cot x     == ueval(x, "tan", (z1:F):F +-> inv cot z1)
```

```
cot2tan x     == ueval(x, "cot", (z1:F):F +-> inv tan z1)
tan2trig x    == ueval(x, "tan", t2t)
cot2trig x    == ueval(x, "cot", c2t)
cosh2sech x   == ueval(x, "cosh", (z1:F):F +-> inv sech z1)
sinh2csch x   == ueval(x, "sinh", (z1:F):F +-> inv csch z1)
csch2sinh x   == ueval(x, "csch", ch2sh)
sech2cosh x   == ueval(x, "sech", sh2ch)
tanh2coth x   == ueval(x, "tanh", (z1:F):F +-> inv coth z1)
coth2tanh x   == ueval(x, "coth", (z1:F):F +-> inv tanh z1)
tanh2trigh x  == ueval(x, "tanh", th2th)
coth2trigh x  == ueval(x, "coth", ch2th)
removeCosSq x == ueval2(x, "cos", (z1:F):F +-> 1 - (sin z1)**2)
removeSinSq x == ueval2(x, "sin", s2c2)
removeCoshSq x== ueval2(x, "cosh", (z1:F):F +-> 1 + (sinh z1)**2)
removeSinhSq x== ueval2(x, "sinh", sh2ch2)
expandLog x   == smplog(numer x) / smplog(denom x)
simplifyExp x == (smpexp numer x) / (smpexp denom x)
expand x      == (smpexpand numer x) / (smpexpand denom x)
smpexpand p   == map(kerexpand, (r1:R):F +-> r1::F, p)
smplog p      == map(logexpand, (r1:R):F +-> r1::F, p)
smp2htrigs p  == map((k1:K):F +-> htrigs(k1::F), (r1:R):F +-> r1::F, p)

htrigs f ==
  (m := mainKernel f) case "failed" => f
  op  := operator(k := m::K)
  arg := [htrigs x for x in argument k]$List(F)
  num := univariate(numer f, k)
  den := univariate(denom f, k)
  is?(op, "exp"::Symbol) =>
    g1 := cosh(a := first arg) + sinh(a)
    g2 := cosh(a) - sinh(a)
    supexp(num,g1,g2,b:= (degree num)::Z quo 2)/supexp(den,g1,g2,b)
  sup2htrigs(num, g1:= op arg) / sup2htrigs(den, g1)

supexp(p, f1, f2, bse) ==
  ans:F := 0
  while p ^= 0 repeat
    g := htrigs(leadingCoefficient(p)::F)
    if ((d := degree(p)::Z - bse) >= 0) then
         ans := ans + g * f1 ** d
    else ans := ans + g * f2 ** (-d)
    p := reductum p
  ans

sup2htrigs(p, f) ==
  (map(smp2htrigs, p)$SparseUnivariatePolynomialFunctions2(P, F)) f

exlog p == +/[r.coef * log(r.logand::F) for r in log squareFree p]

logexpand k ==
```

```
    nullary?(op := operator k) => k::F
    is?(op, "log"::Symbol) =>
       exlog(numer(x := expandLog first argument k)) - exlog denom x
    op [expandLog x for x in argument k]$List(F)

  kerexpand k ==
    nullary?(op := operator k) => k::F
    is?(op, POWER) => expandpow k
    arg := first argument k
    is?(op, "sec"::Symbol) => inv expand cos arg
    is?(op, "csc"::Symbol) => inv expand sin arg
    is?(op, "log"::Symbol) =>
       exlog(numer(x := expand arg)) - exlog denom x
    num := numer arg
    den := denom arg
    (b := (reductum num) / den) ^= 0 =>
      a := (leadingMonomial num) / den
      is?(op, "exp"::Symbol) => exp(expand a) * expand(exp b)
      is?(op, "sin"::Symbol) =>
         sin(expand a) * expand(cos b) + cos(expand a) * expand(sin b)
      is?(op, "cos"::Symbol) =>
         cos(expand a) * expand(cos b) - sin(expand a) * expand(sin b)
      is?(op, "tan"::Symbol) =>
        ta := tan expand a
        tb := expand tan b
        (ta + tb) / (1 - ta * tb)
      is?(op, "cot"::Symbol) =>
        cta := cot expand a
        ctb := expand cot b
        (cta * ctb - 1) / (ctb + cta)
      op [expand x for x in argument k]$List(F)
    op [expand x for x in argument k]$List(F)

  smpexp p ==
    ans:F := 0
    while p ^= 0 repeat
      ans := ans + termexp leadingMonomial p
      p   := reductum p
    ans

  -- this now works in 3 passes over the expression:
  --   pass1 rewrites trigs and htrigs in terms of sin,cos,sinh,cosh
  --   pass2 rewrites sin**2 and sinh**2 in terms of cos and cosh.
  --   pass3 groups exponentials together
  simplify0 x ==
    simplifyExp eval(eval(x,
        ["tan"::Symbol,"cot"::Symbol,"sec"::Symbol,"csc"::Symbol,
         "tanh"::Symbol,"coth"::Symbol,"sech"::Symbol,"csch"::Symbol],
            [t2t,c2t,s2c,c2s,th2th,ch2th,sh2ch,ch2sh]),
              ["sin"::Symbol, "sinh"::Symbol], [2, 2], [s2c2, sh2ch2])
```

———-

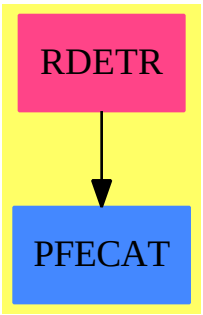
— **TRMANIP.dotabb** —

```
"TRMANIP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TRMANIP"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"TRMANIP" -> "FS"
```

———-

21.16 package RDETR TranscendentalRischDE

21.16.1 TranscendentalRischDE (RDETR)



Exports:
baseRDE monomRDE polyRDE

— **package RDETR TranscendentalRischDE** —

```
)abbrev package RDETR TranscendentalRischDE
++ Author: Manuel Bronstein
++ Date Created: Jan 1988
++ Date Last Updated: 2 November 1995
++ Description:
++ Risch differential equation, transcendental case.

TranscendentalRischDE(F, UP): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
```

```
  Z   ==> Integer
  RF  ==> Fraction UP
  REC ==> Record(a:UP, b:UP, c:UP, t:UP)
  SPE ==> Record(b:UP, c:UP, m:Z, alpha:UP, beta:UP)
  PSOL==> Record(ans:UP, nosol:Boolean)
  ANS ==> Union(ans:PSOL, eq:SPE)
  PSQ ==> Record(ans:RF, nosol:Boolean)

  Exports ==> with
    monomRDE: (RF,RF,UP->UP) -> Union(Record(a:UP,b:RF,c:RF,t:UP), "failed")
      ++ monomRDE(f,g,D) returns \spad{[A, B, C, T]} such that
      ++ \spad{y' + f y = g} has a solution if and only if \spad{y = Q / T},
      ++ where Q satisfies \spad{A Q' + B Q = C} and has no normal pole.
      ++ A and T are polynomials and B and C have no normal poles.
      ++ D is the derivation to use.
    baseRDE : (RF, RF) -> PSQ
      ++ baseRDE(f, g) returns a \spad{[y, b]} such that \spad{y' + fy = g}
      ++ if \spad{b = true}, y is a partial solution otherwise (no solution
      ++ in that case).
      ++ D is the derivation to use.
    polyRDE : (UP, UP, UP, Z, UP -> UP) -> ANS
      ++ polyRDE(a, B, C, n, D) returns either:
      ++ 1. \spad{[Q, b]} such that \spad{degree(Q) <= n} and
      ++    \spad{a Q'+ B Q = C} if \spad{b = true}, Q is a partial solution
      ++    otherwise.
      ++ 2. \spad{[B1, C1, m, \alpha, \beta]} such that any polynomial solution
      ++    of degree at most n of \spad{A Q' + BQ = C} must be of the form
      ++    \spad{Q = \alpha H + \beta} where \spad{degree(H) <= m} and
      ++    H satisfies \spad{H' + B1 H = C1}.
      ++ D is the derivation to use.

  Implementation ==> add
    import MonomialExtensionTools(F, UP)

    getBound     : (UP, UP, Z) -> Z
    SPDEnocancel1: (UP, UP, Z, UP -> UP) -> PSOL
    SPDEnocancel2: (UP, UP, Z, Z, F, UP -> UP) -> ANS
    SPDE         : (UP, UP, UP, Z, UP -> UP) -> Union(SPE, "failed")

-- cancellation at infinity is possible, A is assumed nonzero
-- needs tagged union because of branch choice problem
-- always returns a PSOL in the base case (never a SPE)
    polyRDE(aa, bb, cc, d, derivation) ==
      n:Z
      (u := SPDE(aa, bb, cc, d, derivation)) case "failed" => [[0, true]]
      zero?(u.c) => [[u.beta, false]]
--      baseCase? := one?(dt := derivation monomial(1, 1))
      baseCase? := ((dt := derivation monomial(1, 1)) = 1)
      n := degree(dt)::Z - 1
      b0? := zero?(u.b)
```

```
      (~b0?) and (baseCase? or degree(u.b) > max(0, n)) =>
          answ := SPDEnocancel1(u.b, u.c, u.m, derivation)
          [[u.alpha * answ.ans + u.beta, answ.nosol]]
      (n > 0) and (b0? or degree(u.b) < n) =>
          uansw := SPDEnocancel2(u.b,u.c,u.m,n,leadingCoefficient dt,derivation)
          uansw case ans=> [[u.alpha * uansw.ans.ans + u.beta, uansw.ans.nosol]]
          [[uansw.eq.b, uansw.eq.c, uansw.eq.m,
            u.alpha * uansw.eq.alpha, u.alpha * uansw.eq.beta + u.beta]]
      b0? and baseCase? =>
          degree(u.c) >= u.m => [[0, true]]
          [[u.alpha * integrate(u.c) + u.beta, false]]
      [u::SPE]

-- cancellation at infinity is possible, A is assumed nonzero
-- if u.b = 0 then u.a = 1 already, but no degree check is done
-- returns "failed" if a p' + b p = c has no soln of degree at most d,
-- otherwise [B, C, m, \alpha, \beta] such that any soln p of degree at
-- most d of  a p' + b p = c  must be of the form p = \alpha h + \beta,
-- where h' + B h = C and h has degree at most m
    SPDE(aa, bb, cc, d, derivation) ==
      zero? cc => [0, 0, 0, 0, 0]
      d < 0 => "failed"
      (u := cc exquo (g := gcd(aa, bb))) case "failed" => "failed"
      aa := (aa exquo g)::UP
      bb := (bb exquo g)::UP
      cc := u::UP
      (ra := retractIfCan(aa)@Union(F, "failed")) case F =>
        a1 := inv(ra::F)
        [a1 * bb, a1 * cc, d, 1, 0]
      bc := extendedEuclidean(bb, aa, cc)::Record(coef1:UP, coef2:UP)
      qr := divide(bc.coef1, aa)
      r  := qr.remainder         -- z = bc.coef2 + b * qr.quotient
      (v  := SPDE(aa, bb + derivation aa,
                  bc.coef2 + bb * qr.quotient - derivation r,
                   d - degree(aa)::Z, derivation)) case "failed" => "failed"
      [v.b, v.c, v.m, aa * v.alpha, aa * v.beta + r]

-- solves q' + b q = c  with deg(q) <= d
-- case (B <> 0) and (D = d/dt or degree(B) > max(0, degree(Dt) - 1))
-- this implies no cancellation at infinity, BQ term dominates
-- returns [Q, flag] such that Q is a solution if flag is false,
-- a partial solution otherwise.
    SPDEnocancel1(bb, cc, d, derivation) ==
      q:UP := 0
      db := (degree bb)::Z
      lb := leadingCoefficient bb
      while cc ^= 0 repeat
        d < 0 or (n := (degree cc)::Z - db) < 0 or n > d => return [q, true]
        r := monomial((leadingCoefficient cc) / lb, n::N)
        cc := cc - bb * r - derivation r
```

```
        d := n - 1
        q := q + r
      [q, false]

-- case (t is a nonlinear monomial) and (B = 0 or degree(B) < degree(Dt) - 1)
-- this implies no cancellation at infinity, DQ term dominates or degree(Q) = 0
-- dtm1 = degree(Dt) - 1
    SPDEnocancel2(bb, cc, d, dtm1, lt, derivation) ==
      q:UP := 0
      while cc ^= 0 repeat
        d < 0 or (n := (degree cc)::Z - dtm1) < 0 or n > d => return [[q, true]]
        if n > 0 then
          r  := monomial((leadingCoefficient cc) / (n * lt), n::N)
          cc := cc - bb * r - derivation r
          d  := n - 1
          q  := q + r
        else        -- n = 0 so solution must have degree 0
          db:N := (zero? bb => 0; degree bb);
          db ^= degree(cc) => return [[q, true]]
          zero? db => return [[bb, cc, 0, 1, q]]
          r  := leadingCoefficient(cc) / leadingCoefficient(bb)
          cc := cc - r * bb - derivation(r::UP)
          d  := - 1
          q := q + r::UP
      [[q, false]]

    monomRDE(f, g, derivation) ==
      gg := gcd(d := normalDenom(f,derivation), e := normalDenom(g,derivation))
      tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
      (u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
      [aa, aa * f - (d * derivation tt)::RF, u::UP * e * g, tt]

-- solve y' + f y = g for y in RF
-- assumes that f is weakly normalized (no finite cancellation)
-- base case: F' = 0
    baseRDE(f, g) ==
      (u := monomRDE(f, g, differentiate)) case "failed" => [0, true]
      n := getBound(u.a,bb := retract(u.b)@UP,degree(cc := retract(u.c)@UP)::Z)
      v := polyRDE(u.a, bb, cc, n, differentiate).ans
      [v.ans / u.t, v.nosol]

-- return an a bound on the degree of a solution of A P'+ B P = C,A ^= 0
-- cancellation at infinity is possible
-- base case: F' = 0
    getBound(a, b, dc) ==
      da := (degree a)::Z
      zero? b => max(0, dc - da + 1)
      db := (degree b)::Z
      da > (db + 1) => max(0, dc - da + 1)
      da < (db + 1) => dc - db
```

```
      (n := retractIfCan(- leadingCoefficient(b) / leadingCoefficient(a)
                    )@Union(Z, "failed")) case Z => max(n::Z, dc - db)
      dc - db
```

———-

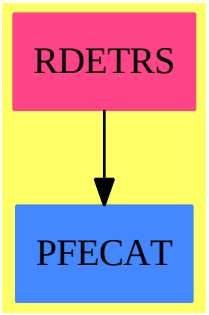
— **RDETR.dotabb** —

```
"RDETR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDETR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RDETR" -> "PFECAT"
```

———-

21.17 package RDETRS TranscendentalRischDESystem

21.17.1 TranscendentalRischDESystem (RDETRS)



Exports:
baseRDEsys monomRDEsys

— **package RDETRS TranscendentalRischDESystem** —

```
)abbrev package RDETRS TranscendentalRischDESystem
++ Author: Manuel Bronstein
++ Date Created: 17 August 1992
++ Date Last Updated: 3 February 1994
++ Description:
++ Risch differential equation system, transcendental case.

TranscendentalRischDESystem(F, UP): Exports == Implementation where
  F  : Join(Field, CharacteristicZero, RetractableTo Integer)
```

```
  UP : UnivariatePolynomialCategory F

  N   ==> NonNegativeInteger
  Z   ==> Integer
  RF  ==> Fraction UP
  V   ==> Vector UP
  U   ==> Union(List UP, "failed")
  REC ==> Record(z1:UP, z2:UP, r1:UP, r2:UP)

  Exports ==> with
    monomRDEsys: (RF, RF, RF, UP -> UP) -> _
                  Union(Record(a:UP, b:RF, h:UP, c1:RF, c2:RF, t:UP),"failed")
      ++ monomRDEsys(f,g1,g2,D) returns \spad{[A, B, H, C1, C2, T]} such that
      ++ \spad{(y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)} has a solution
      ++ if and only if \spad{y1 = Q1 / T, y2 = Q2 / T},
      ++ where \spad{B,C1,C2,Q1,Q2} have no normal poles and satisfy
      ++ A \spad{(Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)}
      ++ D is the derivation to use.
    baseRDEsys: (RF, RF, RF)   -> Union(List RF, "failed")
      ++ baseRDEsys(f, g1, g2) returns fractions \spad{y_1.y_2} such that
      ++ \spad{(y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)}
      ++ if \spad{y_1,y_2} exist, "failed" otherwise.

  Implementation ==> add
    import MonomialExtensionTools(F, UP)
    import SmithNormalForm(UP, V, V, Matrix UP)

    diophant: (UP, UP, UP, UP, UP) -> Union(REC, "failed")
    getBound: (UP, UP, UP, UP, UP) -> Z
    SPDEsys : (UP, UP, UP, UP, UP, Z, UP -> UP, (F, F, F, UP, UP, Z) -> U) -> U
    DSPDEsys: (F, UP, UP, UP, UP, Z, UP -> UP) -> U
    DSPDEmix: (UP, UP, F, F, N, Z, F) -> U
    DSPDEhdom: (UP, UP, F, F, N, Z) -> U
    DSPDEbdom: (UP, UP, F, F, N, Z) -> U
    DSPDEsys0: (F, UP, UP, UP, UP, F, F, Z, UP -> UP, (UP,UP,F,F,N) -> U) -> U

-- reduces (y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2) to
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2), Q1 = y1 T, Q2 = y2 T
-- where A and H are polynomials, and B,C1,C2,Q1 and Q2 have no normal poles.
-- assumes that f is weakly normalized (no finite cancellation)
    monomRDEsys(f, g1, g2, derivation) ==
      gg := gcd(d := normalDenom(f, derivation),
                e := lcm(normalDenom(g1,derivation),normalDenom(g2,derivation)))
      tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
      (u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
      [aa, tt * d * f, - d * derivation tt, u::UP * e * g1, u::UP * e * g2, tt]

-- solve (y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2) for y1,y2 in RF
-- assumes that f is weakly normalized (no finite cancellation) and nonzero
-- base case: F' = 0
```

```
    baseRDEsys(f, g1, g2) ==
      zero? f => error "baseRDEsys: f must be nonzero"
      zero? g1 and zero? g2 => [0, 0]
      (u := monomRDEsys(f, g1, g2, differentiate)) case "failed" => "failed"
      n := getBound(u.a, bb := retract(u.b), u.h,
                    cc1 := retract(u.c1), cc2 := retract(u.c2))
      (v := SPDEsys(u.a, bb, u.h, cc1, cc2, n, differentiate,
                   (z1,z2,z3,z4,z5,z6) +->
                    DSPDEsys(z1, z2::UP, z3::UP, z4, z5, z6, differentiate)))
                          case "failed" => "failed"
      l := v::List(UP)
      [first(l) / u.t, second(l) / u.t]

-- solve
--   D1 = A Z1 + B R1 - C R2
--   D2 = A Z2 + C R1 + B R2
-- i.e. (D1,D2) = ((A, 0, B, -C), (0, A, C, B)) (Z1, Z2, R1, R2)
-- for R1, R2 with degree(Ri) < degree(A)
-- assumes (A,B,C) = (1) and A and C are nonzero
    diophant(a, b, c, d1, d2) ==
      (u := diophantineSystem(matrix [[a,0,b,-c], [0,a,c,b]],
                          vector [d1,d2]).particular) case "failed" => "failed"
      v := u::V
      qr1 := divide(v 3, a)
      qr2 := divide(v 4, a)
      [v.1 + b * qr1.quotient - c * qr2.quotient,
       v.2 + c * qr1.quotient + b * qr2.quotient, qr1.remainder, qr2.remainder]

-- solve
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)
-- for polynomials Q1 and Q2 with degree <= n
-- A and B are nonzero
-- cancellation at infinity is possible
    SPDEsys(a, b, h, c1, c2, n, derivation, degradation) ==
      zero? c1 and zero? c2 => [0, 0]
      n < 0 => "failed"
      g := gcd(a, gcd(b, h))
      ((u1 := c1 exquo g) case "failed") or
        ((u2 := c2 exquo g) case "failed") => "failed"
      a := (a exquo g)::UP
      b := (b exquo g)::UP
      h := (h exquo g)::UP
      c1 := u1::UP
      c2 := u2::UP
      (da := degree a) > 0 =>
        (u := diophant(a, h, b, c1, c2)) case "failed" => "failed"
        rec := u::REC
        v := SPDEsys(a, b, h + derivation a, rec.z1 - derivation(rec.r1),
                     rec.z2 - derivation(rec.r2),n-da::Z,derivation,degradation)
        v case "failed" => "failed"
```

```
        l := v::List(UP)
        [a * first(l) + rec.r1, a * second(l) + rec.r2]
      ra := retract(a)@F
      ((rb := retractIfCan(b)@Union(F, "failed")) case "failed") or
        ((rh := retractIfCan(h)@Union(F, "failed")) case "failed") =>
                              DSPDEsys(ra, b, h, c1, c2, n, derivation)
      degradation(ra, rb::F, rh::F, c1, c2, n)

-- solve
-- a (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)
-- for polynomials Q1 and Q2 with degree <= n
-- a and B are nonzero, either B or H has positive degree
-- cancellation at infinity is not possible
    DSPDEsys(a, b, h, c1, c2, n, derivation) ==
      bb := degree(b)::Z
      hh:Z :=
        zero? h => 0
        degree(h)::Z
      lb := leadingCoefficient b
      lh := leadingCoefficient h
      bb < hh =>
        DSPDEsys0(a,b,h,c1,c2,lb,lh,n,derivation,
                  (z1,z2,z3,z4,z5) +-> DSPDEhdom(z1,z2,z3,z4,z5,hh))
      bb > hh =>
        DSPDEsys0(a,b,h,c1,c2,lb,lh,n,derivation,
                  (z1,z2,z3,z4,z5) +-> DSPDEbdom(z1,z2,z3,z4,z5,bb))
      det := lb * lb + lh * lh
      DSPDEsys0(a,b,h,c1,c2,lb,lh,n,derivation,
                (z1,z2,z3,z4,z5) +-> DSPDEmix(z1,z2,z3,z4,z5,bb,det))

    DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, getlc) ==
      ans1 := ans2 := 0::UP
      repeat
        zero? c1 and zero? c2 => return [ans1, ans2]
        n < 0 or (u := getlc(c1,c2,lb,lh,n::N)) case "failed" => return "failed"
        lq := u::List(UP)
        q1 := first lq
        q2 := second lq
        c1 := c1 - a * derivation(q1) - h * q1 + b * q2
        c2 := c2 - a * derivation(q2) - b * q1 - h * q2
        n := n - 1
        ans1 := ans1 + q1
        ans2 := ans2 + q2

    DSPDEmix(c1, c2, lb, lh, n, d, det) ==
      rh1:F :=
        zero? c1 => 0
        (d1 := degree(c1)::Z - d) < n => 0
        d1 > n => return "failed"
        leadingCoefficient c1
```

```
      rh2:F :=
        zero? c2 => 0
        (d2 := degree(c2)::Z - d) < n => 0
        d2 > n => return "failed"
        leadingCoefficient c2
      q1 := (rh1 * lh + rh2 * lb) / det
      q2 := (rh2 * lh - rh1 * lb) / det
      [monomial(q1, n), monomial(q2, n)]


    DSPDEhdom(c1, c2, lb, lh, n, d) ==
      q1:UP :=
        zero? c1 => 0
        (d1 := degree(c1)::Z - d) < n => 0
        d1 > n => return "failed"
        monomial(leadingCoefficient(c1) / lh, n)
      q2:UP :=
        zero? c2 => 0
        (d2 := degree(c2)::Z - d) < n => 0
        d2 > n => return "failed"
        monomial(leadingCoefficient(c2) / lh, n)
      [q1, q2]

    DSPDEbdom(c1, c2, lb, lh, n, d) ==
      q1:UP :=
        zero? c2 => 0
        (d2 := degree(c2)::Z - d) < n => 0
        d2 > n => return "failed"
        monomial(leadingCoefficient(c2) / lb, n)
      q2:UP :=
        zero? c1 => 0
        (d1 := degree(c1)::Z - d) < n => 0
        d1 > n => return "failed"
        monomial(- leadingCoefficient(c1) / lb, n)
      [q1, q2]

-- return a common bound on the degrees of a solution of
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2), Q1 = y1 T, Q2 = y2 T
-- cancellation at infinity is possible
-- a and b are nonzero
-- base case: F' = 0
    getBound(a, b, h, c1, c2) ==
      da := (degree a)::Z
      dc :=
        zero? c1 => degree(c2)::Z
        zero? c2 => degree(c1)::Z
        max(degree c1, degree c2)::Z
      hh:Z :=
        zero? h => 0
        degree(h)::Z
```

```
      db := max(hh, bb := degree(b)::Z)
      da < db + 1 => dc - db
      da > db + 1 => max(0, dc - da + 1)
      bb >= hh => dc - db
      (n := retractIfCan(leadingCoefficient(h) / leadingCoefficient(a)
                     )@Union(Z, "failed")) case Z => max(n::Z, dc - db)
      dc - db
```

———-

— **RDETRS.dotabb** —

```
"RDETRS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDETRS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RDETRS" -> "PFECAT"
```

———-

21.18 package SOLVETRA TransSolvePackage

— **TransSolvePackage.input** —

```
)set break resume
)sys rm -f TransSolvePackage.output
)spool TransSolvePackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 4
solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
--R
--R
--R   (1)
--R   [phi= 2atan(%phi0) - theta, phi= 2atan(%phi1) - theta,
--R
--R     phi =
--R           2
--R        *
--R           atan
--R                  ROOT
--R                                4    theta 4        4          2     theta 2
--R                             - 3v tan(-----)  + (- 6v  + 24g l v )tan(-----)
--R                                       2                                2
--R                         +
```

```
--R                              4          2      2 2
--R                           - 3v  + 24g l v  - 48g l
--R                        *
--R                              2
--R                           %phi1
--R                        +
--R                                   4    theta 4        4          2     theta 2
--R                                 - 2v tan(-----)  + (- 4v  + 16g l v )tan(-----)
--R                                          2                                2
--R                               +
--R                                    4          2      2 2
--R                                 - 2v  + 16g l v  - 32g l
--R                             *
--R                                %phi0
--R                           +
--R                                    2    theta 3             2      2 2     theta
--R                           - 16g l v tan(-----)  + (- 16g l v  + 64g l )tan(-----)
--R                                           2                                  2
--R                        *
--R                           %phi1
--R                        +
--R                                    4    theta 4        4          2     theta 2
--R                             - 3v tan(-----)  + (- 6v  + 24g l v )tan(-----)
--R                                          2                                2
--R                           +
--R                                    4          2      2 2
--R                             - 3v  + 24g l v  - 48g l
--R                        *
--R                                    2
--R                           %phi0
--R                        +
--R                                           2    theta 3
--R                               - 16g l v tan(-----)
--R                                                 2
--R                             +
--R                                           2      2 2     theta
--R                               (- 16g l v  + 64g l )tan(-----)
--R                                                        2
--R                        *
--R                           %phi0
--R                        +
--R                             4          2     theta 4      4    theta 2     4
--R                          (8v  + 16g l v )tan(-----)  + 16v tan(-----)  + 8v
--R                                                 2                2
--R                        +
--R                                     2      2 2
--R                             - 16g l v  - 64g l
--R                     +
--R                           2    theta 2    2
--R                      (- v tan(-----)  - v  + 4g l)%phi1
```

```
--R                            2
--R               +
--R                     2    theta 2    2                          theta
--R                 (- v tan(-----)  - v  + 4g l)%phi0 - 8g l tan(-----)
--R                            2                                     2
--R             /
--R                  2    theta 2     2
--R                2v tan(-----)  + 2v  - 8g l
--R                        2
--R       +
--R         - theta
--R     ,
--R
--R     phi =
--R         -
--R              2
--R           *
--R              atan
--R                     ROOT
--R                                    4    theta 4        4          2     theta 2
--R                              - 3v tan(-----)  + (- 6v  + 24g l v )tan(-----)
--R                                         2                                  2
--R                            +
--R                                  4          2      2 2
--R                              - 3v  + 24g l v  - 48g l
--R                          *
--R                                 2
--R                            %phi1
--R                         +
--R                                      4    theta 4
--R                                  - 2v tan(-----)
--R                                             2
--R                                +
--R                                       4          2     theta 2     4          2
--R                                  (- 4v  + 16g l v )tan(-----)  - 2v  + 16g l v
--R                                                        2
--R                                +
--R                                       2 2
--R                                  - 32g l
--R                               *
--R                                  %phi0
--R                            +
--R                                      2    theta 3
--R                              - 16g l v tan(-----)
--R                                           2
--R                            +
--R                                       2      2 2     theta
--R                              (- 16g l v  + 64g l )tan(-----)
--R                                                      2
--R                          *
```

```
--R                          %phi1
--R                       +
--R                                4    theta 4        4          2     theta 2
--R                         - 3v tan(-----)  + (- 6v  + 24g l v )tan(-----)
--R                                    2                                 2
--R                          %phi1
--R                       +
--R                                4          2      2 2
--R                         - 3v  + 24g l v  - 48g l
--R                        *
--R                                2
--R                          %phi0
--R                       +
--R                                          2    theta 3
--R                         - 16g l v tan(-----)
--R                                           2
--R                          %phi0
--R                       +
--R                                       2      2 2     theta
--R                         (- 16g l v  + 64g l )tan(-----)
--R                                                    2
--R                        *
--R                          %phi0
--R                       +
--R                            4          2     theta 4      4    theta 2     4
--R                       (8v  + 16g l v )tan(-----)  + 16v tan(-----)  + 8v
--R                                         2                 2
--R                       +
--R                                    2      2 2
--R                         - 16g l v  - 64g l
--R                  +
--R                      2    theta 2    2
--R                    (v tan(-----)  + v  - 4g l)%phi1
--R                             2
--R                  +
--R                      2    theta 2    2                          theta
--R                    (v tan(-----)  + v  - 4g l)%phi0 + 8g l tan(-----)
--R                             2                                     2
--R               /
--R                    2    theta 2     2
--R                  2v tan(-----)  + 2v  - 8g l
--R                             2
--R      +
--R        - theta
--R    ]
--R                                    Type: List(Equation(Expression(Integer)))
--E 1

--S 2 of 4
definingPolynomial %phi0
--R
--R
```

```
--R   (2)
--R              4         2      2            2             theta 2
--R       ((%phi0  - 2%phi0  + 1)v  + (- 4%phi0  - 4)g l)tan(-----)
--R                                                            2
--R     +
--R              3                  theta          4         2      2
--R       (8%phi0  + 8%phi0)g l tan(-----) + (%phi0  - 2%phi0  + 1)v
--R                                  2
--R     +
--R                4         2
--R       (- 4%phi0  - 4%phi0 )g l
--R  /
--R      2    theta 2    2
--R     v tan(-----)  + v  - 4g l
--R             2
--R                                                    Type: Expression(Integer)
--E 2

--S 3 of 4
definingPolynomial %phi1
--R
--R
--R   (3)
--R         2    theta 2    2             3
--R       (v tan(-----)  + v  - 4g l)%phi0
--R                2
--R     +
--R               2    theta 2            theta           2                   2
--R       (%phi1 v tan(-----)  + 8g l tan(-----) + %phi1 v  - 4%phi1 g l)%phi0
--R                      2                  2
--R     +
--R                  2      2            theta 2                  theta
--R           ((%phi1  - 2)v  - 4g l)tan(-----)  + 8%phi1 g l tan(-----)
--R                                       2                        2
--R         +
--R                 2      2            2
--R           (%phi1  - 2)v  + (- 4%phi1  - 4)g l
--R      *
--R         %phi0
--R     +
--R               3           2                  theta 2
--R       ((%phi1  - 2%phi1)v  - 4%phi1 g l)tan(-----)
--R                                               2
--R     +
--R            2             theta          3           2            3
--R     (8%phi1  + 8)g l tan(-----) + (%phi1  - 2%phi1)v  + (- 4%phi1  - 4%phi1)g l
--R                            2
--R  /
--R      2    theta 2    2
--R     v tan(-----)  + v  - 4g l
```

```
--R             2
--R                                                  Type: Expression(Integer)
--E 3

--S 4 of 4
)show TransSolvePackage
--R
--R TransSolvePackage(R: Join(OrderedSet,EuclideanDomain,RetractableTo(Integer),LinearlyExplicitRingOver(Int
--R Abbreviation for TransSolvePackage is SOLVETRA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SOLVETRA
--R
--R------------------------------ Operations --------------------------------
--R solve : Expression(R) -> List(Equation(Expression(R)))
--R solve : Equation(Expression(R)) -> List(Equation(Expression(R)))
--R solve : (Equation(Expression(R)),Symbol) -> List(Equation(Expression(R)))
--R solve : (Expression(R),Symbol) -> List(Equation(Expression(R)))
--R solve : (List(Equation(Expression(R))),List(Symbol)) -> List(List(Equation(Expression(R))))
--R
--E 4

)spool
```

———-

— **TransSolvePackage.help** —

```
=====================================================================
TransSolvePackage examples
=====================================================================

solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)

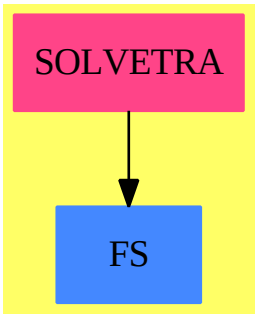
definingPolynomial %phi0

definingPolynomial %phi1

See Also:
o )show TransSolvePackage
```

———-

21.18.1 TransSolvePackage (SOLVETRA)



Exports:
solve

— package SOLVETRA TransSolvePackage —

```
)abbrev package SOLVETRA TransSolvePackage
++ Author: W. Wiwianka, Martin Rubey
++ Date Created: Summer 1991
++ Change History: 9/91
++ Basic Operations: solve
++ Related Constructors: RadicalSolvePackage, FloatingRealPackage
++ Keywords:
++ Description:
++ This package tries to find solutions of equations of type Expression(R).
++ This means expressions involving transcendental, exponential, logarithmic
++ and nthRoot functions.
++ After trying to transform different kernels to one kernel by applying
++ several rules, it calls zerosOf for the SparseUnivariatePolynomial in
++ the remaining kernel.
++ For example the expression  \spad{sin(x)*cos(x)-2}  will be transformed to
++ \spad{-2 tan(x/2)**4 -2 tan(x/2)**3 -4 tan(x/2)**2 +2 tan(x/2) -2}
++ by  using the function  normalize  and then to
++ \spad{-2 tan(x)**2 + tan(x) -2}
++ with help of subsTan. This function tries to express the given function
++ in terms of \spad{tan(x/2)}  to express in terms of \spad{tan(x)} .
++ Other examples are the expressions    \spad{sqrt(x+1)+sqrt(x+7)+1}   or
++ \spad{sqrt(sin(x))+1} .

TransSolvePackage(R) : Exports == Implementation where
   R : Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
            LinearlyExplicitRingOver Integer, CharacteristicZero)

   I   ==> Integer
   NNI ==> NonNegativeInteger
   RE  ==> Expression R
```

```
   EQ  ==> Equation
   S   ==> Symbol
   V   ==> Variable
   L   ==> List
   K   ==> Kernel RE
   SUP ==> SparseUnivariatePolynomial
   C   ==> Complex
   F   ==> Float
   INT ==> Interval
   SMP ==> SparseMultivariatePolynomial


   Exports == with

     solve :  RE           ->  L EQ RE
       ++ solve(expr) finds the solutions of the equation expr = 0
       ++ where expr is a function of type Expression(R)
       ++ with respect to the unique symbol x appearing in eq.
     solve :  EQ RE        ->  L EQ RE
       ++ solve(eq) finds the solutions of the equation  eq
       ++ where  eq  is  an equation of functions of type Expression(R)
       ++ with respect to the unique symbol x appearing in eq.
     solve : ( EQ RE , S ) ->  L EQ RE
       ++ solve(eq,x) finds the solutions of the equation  eq
       ++ where  eq  is  an equation of functions of type Expression(R)
       ++ with respect to the symbol x.
     solve : ( RE , S)     ->  L EQ RE
       ++ solve(expr,x) finds the solutions of the equation expr = 0
       ++ with respect to the symbol x where expr is a function
       ++ of type Expression(R).
       ++
       ++X solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
       ++X definingPolynomial %phi0
       ++X definingPolynomial %phi1

     solve :  (L EQ RE, L S) -> L L EQ RE
       ++ solve(leqs, lvar) returns a list of solutions to the list of
       ++ equations leqs with respect to the list of symbols lvar.
--     solve :  (L EQ RE, L Kernel RE) -> L L EQ RE
--       ++ solve(leqs, lker) returns a list of solutions to the list
--       ++ of equations leqs with respect to the list of kernels lker.

   Implementation == add
     import ACF
     import HomogeneousAggregate(R)
     import AlgebraicManipulations(R, RE)
     import TranscendentalManipulations(R, RE)
     import TrigonometricManipulations(R, RE)
     import ElementaryFunctionStructurePackage(R, RE)
     import SparseUnivariatePolynomial(R)
```

```
import LinearSystemMatrixPackage(RE,Vector RE,Vector RE,Matrix RE)
import TransSolvePackageService(R)
import MultivariateFactorize(K, IndexedExponents K, R, SMP(R, K))



   ---- Local Function Declarations ----

solveInner : (RE, S) -> L EQ RE
tryToTrans : ( RE , S)     ->  RE

eliminateKernRoot: (RE , K) -> RE
eliminateRoot: (RE , S) -> RE

combineLog : ( RE , S ) -> RE
testLog : ( RE , S ) -> Boolean
splitExpr : ( RE ) -> L RE
buildnexpr : ( RE , S ) -> L RE
logsumtolog : RE -> RE
logexpp : ( RE , RE ) -> RE

testRootk : ( RE, S) -> Boolean
testkernel : ( RE , S ) -> Boolean
funcinv : ( RE , RE ) -> Union(RE,"failed")
testTrig : ( RE , S ) -> Boolean
testHTrig : ( RE , S ) -> Boolean
tableXkernels : ( RE , S ) -> L RE
subsTan : ( RE , S )  -> RE


-- exported functions


solve(oside: RE) : L EQ RE ==
  zero? oside => error "equation is always satisfied"
  lv := variables oside
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  solve(oside,lv.first)

solve(equ:EQ RE) : L EQ RE ==
  solve(lhs(equ)-rhs(equ))

solve(equ:EQ RE, x:S) : L EQ RE ==
  oneside:=lhs(equ)-rhs(equ)
  solve(oneside,x)

testZero?(lside:RE,sol:EQ RE):Boolean ==
  if R has QuotientFieldCategory(Integer) then
    retractIfCan(rhs sol)@Union(Integer,"failed") case "failed" => true
```

```
    else
      retractIfCan(rhs sol)@Union(Fraction Integer,"failed") case "failed" => true
    zero? eval(lside,sol) => true
    false

  solve(lside: RE, x:S) : L EQ RE ==
    [sol for sol in solveInner(lside,x) | testZero?(lside,sol)]

  solveInner(lside: RE, x:S) : L EQ RE ==
    lside:=eliminateRoot(lside,x)
    ausgabe1:=tableXkernels(lside,x)

    X:=new()@Symbol
    Y:=new()@Symbol::RE
    (#ausgabe1) = 1 =>
       bigX:= (first ausgabe1)::RE
       eq1:=eval(lside,bigX=(X::RE))
           -- Type  :  Expression R
       f:=univariate(eq1,first kernels (X::RE))
           -- Type  :  Fraction SparseUnivariatePolynomial Expression R
       lfatt:= factors factorPolynomial numer f
       lr:L RE := "append" /[zerosOf(fatt.factor,x) for fatt in lfatt]
           -- Type  :  List Expression R
       r1:=[]::L RE
       for i in 1..#lr repeat
          finv := funcinv(bigX,lr(i))
          if finv case RE then r1:=cons(finv::RE,r1)
       bigX_back:=funcinv(bigX,bigX)::RE
       if not testkernel(bigX_back,x) then
         if bigX = bigX_back then return []::L EQ RE
         return
           "append"/[solve(bigX_back-ri, x) for ri in r1]
       newlist:=[]::L EQ RE

       for i in 1..#r1 repeat
          elR :=  eliminateRoot((numer(bigX_back - r1(i))::RE ),x)
          f:=univariate(elR, kernel(x))
           -- Type  :  Fraction SparseUnivariatePolynomial Expression R
          lfatt:= factors factorPolynomial numer f
          secondsol:="append" /[zerosOf(ff.factor,x) for ff in lfatt]
          for j in 1..#secondsol repeat
             newlist:=cons((x::RE)=rootSimp( secondsol(j) ),newlist)
       newlist
    newlside:=tryToTrans(lside,x) ::RE
    listofkernels:=tableXkernels(newlside,x)
    (#listofkernels) = 1 => solve(newlside,x)
    lfacts := factors factor(numer lside)
    #lfacts > 1 =>
       sols : L EQ RE := []
       for frec in lfacts repeat
```

```
            sols := append(solve(frec.factor :: RE, x), sols)
         sols
    return []::L EQ RE

  -- local functions

   --  This function was suggested by Manuel Bronstein as a simpler
   --  alternative to normalize.
   simplifyingLog(f:RE):RE ==
     (u:=isExpt(f,"exp"::Symbol)) case Record(var:Kernel RE,exponent:Integer) =>
       rec := u::Record(var:Kernel RE,exponent:Integer)
       rec.exponent * first argument(rec.var)
     log f


   testkernel(var1:RE,y:S) : Boolean ==
     var1:=eliminateRoot(var1,y)
     listvar1:=tableXkernels(var1,y)
     if (#listvar1 = 1) and ((listvar1(1) = (y::RE))@Boolean ) then
          true
     else if #listvar1 = 0 then true
          else false

   solveRetract(lexpr:L RE, lvar:L S):Union(L L EQ RE, "failed") ==
      nlexpr : L Fraction Polynomial R := []
      for expr in lexpr repeat
         rf:Union(Fraction Polynomial R, "failed") := retractIfCan(expr)$RE
         rf case "failed" => return "failed"
         nlexpr := cons(rf, nlexpr)
      radicalSolve(nlexpr, lvar)$RadicalSolvePackage(R)

   tryToTrans(lside: RE, x:S) : RE ==
     if testTrig(lside,x) or testHTrig(lside,x) then
        convLside:=( simplify(lside) )::RE
        resultLside:=convLside
        listConvLside:=tableXkernels(convLside,x)
        if (#listConvLside) > 1  then
          NormConvLside:=normalize(convLside,x)
          NormConvLside:=( NormConvLside ) :: RE
          resultLside:=subsTan(NormConvLside , x)

     else if testLog(lside,x) then
            numlside:=numer(lside)::RE
            resultLside:=combineLog(numlside,x)
          else
            NormConvLside:=normalize(lside,x)
            NormConvLside:=( NormConvLside ) :: RE
            resultLside:=NormConvLside
            listConvLside:=tableXkernels(NormConvLside,x)
            if  (#listConvLside) > 1  then
```

```
            cnormConvLside:=complexNormalize(lside,x)
            cnormConvLside:=cnormConvLside::RE
            resultLside:=cnormConvLside
            listcnorm:=tableXkernels(cnormConvLside,x)
            if (#listcnorm) > 1 then
              if testLog(cnormConvLside,x) then
                numlside:=numer(cnormConvLside)::RE
                resultLside:=combineLog(numlside,x)
    resultLside


  subsTan(exprvar:RE,y:S) : RE ==
    Z:=new()@Symbol
    listofkern:=tableXkernels(exprvar,y)
    varkern:=(first listofkern)::RE
    Y:=(numer first argument first (kernels(varkern)))::RE
    test : Boolean := varkern=tan(((Y::RE)/(2::RE))::RE)
    if not( (#listofkern=1) and test) then
      return exprvar
    fZ:=eval(exprvar,varkern=(Z::RE))
    fN:=(numer fZ)::RE
    f:=univariate(fN, first kernels(Z::RE))
    secondfun:=(-2*(Y::RE)/((Y::RE)**2-1) )::RE
    g:=univariate(secondfun,first kernels(y::RE))
    H:=(new()@Symbol)::RE
    newH:=univariate(H,first kernels(Z::RE))
    result:=decomposeFunc(f,g,newH)
    if not ( result = f ) then
      result1:=result( H::RE )
      resultnew:=eval(result1,H=(( tan((Y::RE))::RE ) ))
    else return exprvar


  eliminateKernRoot(var: RE, varkern: K) : RE ==
    X:=new()@Symbol
    var1:=eval(var, (varkern::RE)=(X::RE) )
    var2:=numer univariate(var1, first kernels(X::RE))
    var3:= monomial(1, ( retract( second argument varkern)@I )::NNI)@SUP RE_
           - monomial(first argument varkern, 0::NNI)@SUP RE
    resultvar:=resultant(var2, var3)

  eliminateRoot(var:RE, y:S) : RE ==
    var1:=var
    while testRootk(var1,y) repeat
      varlistk1:=tableXkernels(var1,y)
      for i in varlistk1 repeat
         if is?(i, "nthRoot"::S) then
           var1:=eliminateKernRoot(var1,first kernels(i::RE))
    var1
```

```
logsumtolog(var:RE) : RE ==
  (listofexpr:=isPlus(var)) case "failed" => var
  listofexpr:= listofexpr ::L RE
  listforgcd:=[]::L R
  for i in listofexpr repeat
     exprcoeff:=leadingCoefficient(numer(i))
     listforgcd:=cons(exprcoeff, listforgcd)
  gcdcoeff:=gcd(listforgcd)::RE
  newexpr:RE :=0
  for i in listofexpr repeat
     exprlist:=splitExpr(i::RE)
     newexpr:=newexpr + logexpp(exprlist.2, exprlist.1/gcdcoeff)
  kernelofvar:=kernels(newexpr)
  var2:=1::RE
  for i in kernelofvar repeat
     var2:=var2*(first argument i)
  gcdcoeff * log(var2)


testLog(expr:RE,Z:S) : Boolean ==
  testList:=[log]::L S
  kernelofexpr:=tableXkernels(expr,Z)
  if #kernelofexpr = 0 then
    return false
  for i in kernelofexpr repeat
     if not member?(name(first kernels(i)),testList) or _
        not testkernel( (first argument first kernels(i)) ,Z) then
       return false
  true

splitExpr(expr:RE) : L RE ==
  lcoeff:=leadingCoefficient((numer expr))
  exprwcoeff:=expr
  listexpr:=isTimes(exprwcoeff)
  if listexpr case "failed" then
    [1::RE , expr]
  else
    listexpr:=remove_!(lcoeff::RE , listexpr)
    cons(lcoeff::RE , listexpr)

buildnexpr(expr:RE, Z:S) : L RE ==
  nlist:=splitExpr(expr)
  n2list:=remove_!(nlist.1, nlist)
  anscoeff:RE:=1
  ansmant:RE:=0
  for i in n2list repeat
     if freeOf?(i::RE,Z) then
       anscoeff:=(i::RE)*anscoeff
     else
```

```
        ansmant:=(i::RE)
    [anscoeff, ansmant * nlist.1 ]

  logexpp(expr1:RE, expr2:RE) : RE ==
    log( (first argument first kernels(expr1))**expr2 )

  combineLog(expr:RE,Y:S) : RE ==
    exprtable:Table(RE,RE):=table()
    (isPlus(expr)) case "failed" => expr
    ans:RE:=0
    while expr ^= 0 repeat
      loopexpr:RE:=leadingMonomial(numer(expr))::RE
      if testLog(loopexpr,Y) and (#tableXkernels(loopexpr,Y)=1) then
        exprr:=buildnexpr(loopexpr,Y)
        if search(exprr.1,exprtable) case "failed" then
          exprtable.(exprr.1):=0
        exprtable.(exprr.1):= exprtable.(exprr.1) + exprr.2
      else
        ans:=ans+loopexpr
      expr:=(reductum(numer expr))::RE
    ansexpr:RE:=0
    for i in keys(exprtable) repeat
       ansexpr:=ansexpr + logsumtolog(exprtable.i) * (i::RE)
    ansexpr:=ansexpr + ans


  testRootk(varlistk:RE,y:S) : Boolean ==
    testList:=[nthRoot]::L S
    kernelofeqnvar:=tableXkernels(varlistk,y)
    if #kernelofeqnvar = 0 then
      return false
    for i in kernelofeqnvar repeat
       if member?(name(first kernels(i)),testList) then
         return true
    false

  tableXkernels(evar:RE,Z:S) : L RE ==
    kOfvar:=kernels(evar)
    listkOfvar:=[]::L RE
    for i in kOfvar repeat
       if not freeOf?(i::RE,Z) then
           listkOfvar:=cons(i::RE,listkOfvar)
    listkOfvar

  testTrig(eqnvar:RE,Z:S) : Boolean ==
    testList:=[sin , cos , tan , cot , sec , csc]::L S
    kernelofeqnvar:=tableXkernels(eqnvar,Z)
    if #kernelofeqnvar = 0 then
      return false
    for i in kernelofeqnvar repeat
```

```
      if not member?(name(first kernels(i)),testList) or _
         not testkernel( (first argument first kernels(i)) ,Z) then
        return false
    true


testHTrig(eqnvar:RE,Z:S) : Boolean ==
  testList:=[sinh , cosh , tanh , coth , sech , csch]::L S
  kernelofeqnvar:=tableXkernels(eqnvar,Z)
  if #kernelofeqnvar = 0 then
    return false
  for i in kernelofeqnvar repeat
     if not member?(name(first kernels(i)),testList) or _
        not testkernel( (first argument first kernels(i)) ,Z) then
       return false
  true

-- Auxiliary local function for use in funcinv.
makeInterval(l:R):C INT F ==
  if R has complex and R has ConvertibleTo(C F) then
    map(interval$INT(F),convert(l)$R)$ComplexFunctions2(F,INT F)
  else
    error "This should never happen"

funcinv(k:RE,l:RE) : Union(RE,"failed") ==
  is?(k, "sin"::Symbol)   => asin(l)
  is?(k, "cos"::Symbol)   => acos(l)
  is?(k, "tan"::Symbol)   => atan(l)
  is?(k, "cot"::Symbol)   => acot(l)
  is?(k, "sec"::Symbol)   =>
      l = 0 => "failed"
      asec(l)
  is?(k, "csc"::Symbol)   =>
      l = 0 => "failed"
      acsc(l)
  is?(k, "sinh"::Symbol)  => asinh(l)
  is?(k, "cosh"::Symbol)  => acosh(l)
  is?(k, "tanh"::Symbol)  => atanh(l)
  is?(k, "coth"::Symbol)  => acoth(l)
  is?(k, "sech"::Symbol)  => asech(l)
  is?(k, "csch"::Symbol)  => acsch(l)
  is?(k, "atan"::Symbol)  => tan(l)
  is?(k, "acot"::Symbol)  =>
      l = 0 => "failed"
      cot(l)
  is?(k, "asin"::Symbol)  => sin(l)
  is?(k, "acos"::Symbol)  => cos(l)
  is?(k, "asec"::Symbol)  => sec(l)
  is?(k, "acsc"::Symbol)  =>
      l = 0 => "failed"
```

```
      csc(l)
  is?(k, "asinh"::Symbol) => sinh(l)
  is?(k, "acosh"::Symbol) => cosh(l)
  is?(k, "atanh"::Symbol) => tanh(l)
  is?(k, "acoth"::Symbol) =>
      l = 0 => "failed"
      coth(l)
  is?(k, "asech"::Symbol) => sech(l)
  is?(k, "acsch"::Symbol) =>
      l = 0 => "failed"
      csch(l)
  is?(k, "exp"::Symbol)   =>
      l = 0 => "failed"
      simplifyingLog l
  is?(k, "log"::Symbol)   =>
    if R has complex and R has ConvertibleTo(C F) then
      -- We will check to see if the imaginary part lies in [-Pi,Pi)
      ze : Expression C INT F
      ze := map(makeInterval,l)$ExpressionFunctions2(R,C INT F)
      z : Union(C INT F,"failed") := retractIfCan ze
      z case "failed" => exp l
      im := imag z
      fpi : Float := pi()
      (-fpi < inf(im)) and (sup(im) <= fpi) => exp l
      "failed"
    else -- R not Complex or something which doesn't map to Complex Floats
      exp l
  is?(k, "%power"::Symbol)   =>
       (t:=normalize(l)) = 0 => "failed"
       log t
  l

import SystemSolvePackage(RE)

ker2Poly(k:Kernel RE, lvar:L S):Polynomial RE ==
   member?(nm:=name k, lvar) => nm :: Polynomial RE
   k :: RE :: Polynomial RE

smp2Poly(pol:SMP(R,Kernel RE), lvar:L S):Polynomial RE ==
   map(x +-> ker2Poly(x, lvar),
       y +-> y::RE::Polynomial RE, pol)$PolynomialCategoryLifting(
         IndexedExponents Kernel RE, Kernel RE, R, SMP(R, Kernel RE),
                 Polynomial RE)

makeFracPoly(expr:RE, lvar:L S):Fraction Polynomial RE ==
   smp2Poly(numer expr, lvar) / smp2Poly(denom expr, lvar)

makeREpol(pol:Polynomial RE):RE ==
   lvar := variables pol
   lval : List RE := [v::RE for v in lvar]
```

```
        ground eval(pol,lvar,lval)

     makeRE(frac:Fraction Polynomial RE):RE ==
        makeREpol(numer frac)/makeREpol(denom frac)

     solve1Pol(pol:Polynomial RE, var: S, sol:L EQ RE):L L EQ RE ==
        repol := eval(makeREpol pol, sol)
        vsols := solve(repol, var)
        [cons(vsol, sol) for vsol in vsols]

     solve1Sys(plist:L Polynomial RE, lvar:L S):L L EQ RE ==
        rplist := reverse plist
        rlvar := reverse lvar
        sols : L L EQ RE := list(empty())
        for p in rplist for v in rlvar repeat
           sols := "append"/[solve1Pol(p,v,sol) for sol in sols]
        sols
```

The input

```
solve(sinh(z)=cosh(z),z)
```

generates the error (reported as bug # 102):

```
>> Error detected within library code:
   No identity element for reduce of empty list using operation append
```

— package SOLVETRA TransSolvePackage —

```
     solveList(lexpr:L RE, lvar:L S):L L EQ RE ==
        ans1 := solveRetract(lexpr, lvar)
        not(ans1 case "failed") => ans1 :: L L EQ RE
        lfrac:L Fraction Polynomial RE :=
           [makeFracPoly(expr, lvar) for expr in lexpr]
        trianglist := triangularSystems(lfrac, lvar)
--        "append"/[solve1Sys(plist, lvar) for plist in trianglist]
        l: L L L EQ RE := [solve1Sys(plist, lvar) for plist in trianglist]
        reduce(append, l, [])

     solve(leqs:L EQ RE, lvar:L S):L L EQ RE ==
        lexpr:L RE := [lhs(eq)-rhs(eq) for eq in leqs]
        solveList(lexpr, lvar)

--     solve(leqs:L EQ RE, lker:L Kernel RE):L L EQ RE ==
--        lexpr:L RE := [lhs(eq)-rhs(eq) for eq in leqs]
--        lvar :L S := [new()$S for k in lker]
```

```
--        lval :L RE := [kernel v for v in lvar]
--        nlexpr := [eval(expr,lker,lval) for expr in lexpr]
--        ans := solveList(nlexpr, lvar)
--        lker2 :L Kernel RE := [v::Kernel(RE) for v in lvar]
--        lval2 := [k::RE for k in lker]
--        [[map(eval(#1,lker2,lval2), neq) for neq in sol] for sol in ans]
```

———-

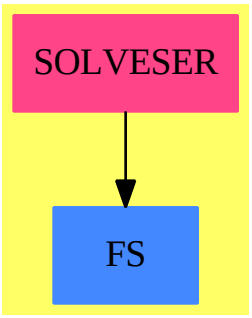
— **SOLVETRA.dotabb** —

```
"SOLVETRA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVETRA"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SOLVETRA" -> "FS"
```

———-

21.19 package SOLVESER TransSolvePackageService

21.19.1 TransSolvePackageService (SOLVESER)



Exports:
decomposeFunc unvectorise

— **package SOLVESER TransSolvePackageService** —

```
)abbrev package SOLVESER TransSolvePackageService
++ Author: W. Wiwianka
++ Date Created: Summer 1991
++ Change History: 9/91
++ Basic Operations: decomposeFunc, unvectorise
++ Related Constructors:
```

```
++ Keywords:
++ Description:
++ This package finds the function func3 where  func1 and func2
++ are given and  func1 = func3(func2) .  If there is no solution then
++ function func1 will be returned.
++ An example would be  \spad{func1:= 8*X**3+32*X**2-14*X ::EXPR INT} and
++ \spad{func2:=2*X ::EXPR INT} convert them via univariate
++ to FRAC SUP EXPR INT and then the solution is \spad{func3:=X**3+X**2-X}
++ of type FRAC SUP EXPR INT

TransSolvePackageService(R) : Exports == Implementation where
   R   :   Join(IntegralDomain, OrderedSet)

   RE  ==> Expression R
   EQ  ==> Equation
   S   ==> Symbol
   V   ==> Variable
   L   ==> List
   SUP ==> SparseUnivariatePolynomial
   ACF      ==> AlgebraicallyClosedField()


   Exports == with

     decomposeFunc : ( Fraction SUP RE , Fraction SUP RE, Fraction SUP RE )  -> Fraction SUP RE
       ++ decomposeFunc(func1, func2, newvar)  returns a function  func3 where
       ++ func1 = func3(func2)  and expresses it in the new variable  newvar.
       ++ If there is no solution then func1 will be returned.
     unvectorise : ( Vector RE , Fraction SUP RE , Integer ) -> Fraction SUP RE
       ++ unvectorise(vect, var, n) returns
       ++ \spad{vect(1) + vect(2)*var + ... + vect(n+1)*var**(n)} where
       ++ vect  is the vector of the coefficients of the polynomail , var
       ++ the new variable and n the degree.


   Implementation == add
     import ACF
     import TranscendentalManipulations(R, RE)
     import ElementaryFunctionStructurePackage(R, RE)
     import SparseUnivariatePolynomial(R)
     import LinearSystemMatrixPackage(RE,Vector RE,Vector RE,Matrix RE)
     import HomogeneousAggregate(R)

        ---- Local Function Declarations ----

     subsSolve : ( SUP RE, NonNegativeInteger, SUP RE, SUP RE, Integer, Fraction SUP RE) -> Unio
       --++ subsSolve(f, degf, g1, g2, m, h)


    -- exported functions
```

```
 unvectorise(vect:Vector RE, var:Fraction SUP RE,n:Integer) : Fraction SUP RE ==
   Z:=new()@Symbol
   polyvar: Fraction SUP RE :=0
   for i in 1..((n+1)::Integer) repeat
      vecti:=univariate(vect( i ),first kernels(Z::RE))
      polyvar:=polyvar + ( vecti )*( var )**( (n-i+1)::NonNegativeInteger )
   polyvar


 decomposeFunc(exprf:Fraction SUP RE , exprg:Fraction SUP RE, newH:Fraction SUP RE ) : Fraction SUP RE =
   X:=new()@Symbol
   f1:=numer(exprf)
   f2:=denom(exprf)
   g1:=numer(exprg)
   g2:=denom(exprg)
   degF:=max(degree(numer(exprf)),degree(denom(exprf)))
   degG:=max(degree(g1),degree(g2))
   newF1,newF2 : Union(SUP RE, "failed")
   N:= degF exquo degG
   if not ( N case "failed" ) then
     m:=N::Integer
     newF1:=subsSolve(f1,degF,g1,g2,m,newH)
     if f2 = 1 then
       newF2:= 1 :: SUP RE
     else newF2:=subsSolve(f2,degF,g1,g2,m,newH)
     if ( not ( newF1 case "failed" ) ) and ( not ( newF2 case "failed" ) ) then
       newF:=newF1/newF2
     else return exprf
   else return exprf


 -- local functions


 subsSolve(F:SUP RE, DegF:NonNegativeInteger, G1:SUP RE, G2:SUP RE, M:Integer, HH: Fraction SUP RE) : U
   coeffmat:=new((DegF+1),1,0)@Matrix RE
   for i in 0..M repeat
      coeffmat:=horizConcat(coeffmat, (vectorise( ( ( G1**((M-i)::NonNegativeInteger) )*G2**i ), (DegF+
   vec:= vectorise(F,DegF+1)
   coeffma:=subMatrix(coeffmat,1,(DegF+1),2,(M+2))
   solvar:=solve(coeffma,vec)
   if not ( solvar.particular  case  "failed" ) then
     solvevarlist:=(solvar.particular)::Vector RE
     resul:= numer(unvectorise(solvevarlist,( HH ),M))
     resul
   else return "failed"
```

———-

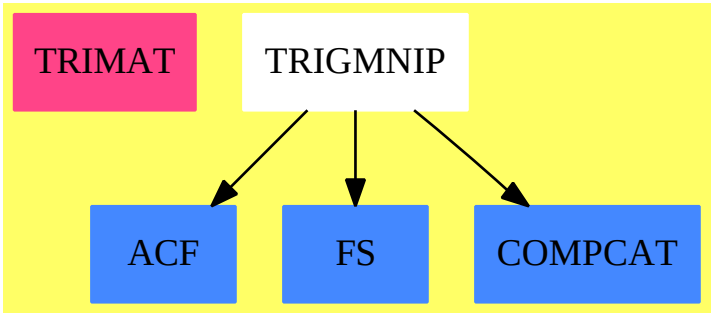
— **SOLVESER.dotabb** —

```
"SOLVESER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVESER"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SOLVESER" -> "FS"
```

———-

21.20 package TRIMAT TriangularMatrixOperations

21.20.1 TriangularMatrixOperations (TRIMAT)



Exports:
LowTriBddDenomInv UpTriBddDenomInv

— **package TRIMAT TriangularMatrixOperations** —

```
)abbrev package TRIMAT TriangularMatrixOperations
++ Author: Victor Miller
++ Date Created:
++ Date Last Updated: 24 Jul 1990
++ Keywords:
++ Examples:
++ References:
++ Description:
++ This package provides functions that compute "fraction-free"
++ inverses of upper and lower triangular matrices over a integral
++ domain.  By "fraction-free inverses" we mean the following:
++ given a matrix B with entries in R and an element d of R such that
++ d * inv(B) also has entries in R, we return d * inv(B).  Thus,
++ it is not necessary to pass to the quotient field in any of our
```

```
++ computations.

TriangularMatrixOperations(R,Row,Col,M): Exports == Implementation where
  R   : IntegralDomain
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M   : MatrixCategory(R,Row,Col)

  Exports ==> with

    UpTriBddDenomInv: (M,R) -> M
      ++ UpTriBddDenomInv(B,d) returns M, where
      ++ B is a non-singular upper triangular matrix and d is an
      ++ element of R such that \spad{M = d * inv(B)} has entries in R.
    LowTriBddDenomInv:(M,R) -> M
      ++ LowTriBddDenomInv(B,d) returns M, where
      ++ B is a non-singular lower triangular matrix and d is an
      ++ element of R such that \spad{M = d * inv(B)} has entries in R.

  Implementation ==> add

    UpTriBddDenomInv(A,denom) ==
      AI := zero(nrows A, nrows A)$M
      offset := minColIndex AI - minRowIndex AI
      for i in minRowIndex AI .. maxRowIndex AI
        for j in minColIndex AI .. maxColIndex AI repeat
          qsetelt_!(AI,i,j,(denom exquo qelt(A,i,j))::R)
      for i in minRowIndex AI .. maxRowIndex AI repeat
        for j in offset + i + 1 .. maxColIndex AI repeat
          qsetelt_!(AI,i,j, - (((+/[qelt(AI,i,k) * qelt(A,k-offset,j)
                                  for k in i+offset..(j-1)])
                                    exquo qelt(A, j-offset, j))::R))
      AI

    LowTriBddDenomInv(A, denom) ==
      AI := zero(nrows A, nrows A)$M
      offset := minColIndex AI - minRowIndex AI
      for i in minRowIndex AI .. maxRowIndex AI
        for j in minColIndex AI .. maxColIndex AI repeat
          qsetelt_!(AI,i,j,(denom exquo qelt(A,i,j))::R)
      for i in minColIndex AI .. maxColIndex AI repeat
        for j in i - offset + 1 .. maxRowIndex AI repeat
          qsetelt_!(AI,j,i, - (((+/[qelt(A,j,k+offset) * qelt(AI,k,i)
                                   for k in i-offset..(j-1)])
                                     exquo qelt(A, j, j+offset))::R))
      AI
```

———————

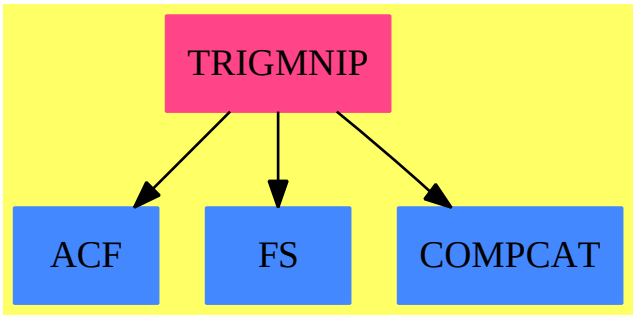
— TRIMAT.dotabb —

```
"TRIMAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TRIMAT"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"TRIGMNIP" -> "ACF"
"TRIGMNIP" -> "FS"
"TRIGMNIP" -> "COMPCAT"
```

———-

21.21 package TRIGMNIP TrigonometricManipulations

21.21.1 TrigonometricManipulations (TRIGMNIP)



Exports:

complexElementary	complexForm	complexNormalize	imag	real
real?	trigs			

— package TRIGMNIP TrigonometricManipulations —

```
)abbrev package TRIGMNIP TrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 14 February 1994
++ Keywords: trigonometric, function, manipulation.
++ Description:
++ \spadtype{TrigonometricManipulations} provides transformations from
++ trigonometric functions to complex exponentials and logarithms, and back.

TrigonometricManipulations(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, RetractableTo Integer,
```

```
          LinearlyExplicitRingOver Integer)
 F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
          FunctionSpace R)

 Z   ==> Integer
 SY  ==> Symbol
 K   ==> Kernel F
 FG  ==> Expression Complex R

 Exports ==> with
   complexNormalize: F -> F
     ++ complexNormalize(f) rewrites \spad{f} using the least possible number
     ++ of complex independent kernels.
   complexNormalize: (F, SY) -> F
     ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
     ++ number of complex independent kernels involving \spad{x}.
   complexElementary: F -> F
     ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
     ++ complex transcendental elementary functions: \spad{log, exp}.
   complexElementary: (F, SY) -> F
     ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
     ++ \spad{x} in terms of the 2 fundamental complex
     ++ transcendental elementary functions: \spad{log, exp}.
   trigs  : F -> F
     ++ trigs(f) rewrites all the complex logs and exponentials
     ++ appearing in \spad{f} in terms of trigonometric functions.
   real   : F -> F
     ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
     ++ function.
   imag   : F -> F
     ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
     ++ is a complex function.
   real?  : F -> Boolean
     ++ real?(f) returns \spad{true} if \spad{f = real f}.
   complexForm: F -> Complex F
     ++ complexForm(f) returns \spad{[real f, imag f]}.

 Implementation ==> add
   import ElementaryFunctionSign(R, F)
   import InnerTrigonometricManipulations(R,F,FG)
   import ElementaryFunctionStructurePackage(R, F)
   import ElementaryFunctionStructurePackage(Complex R, FG)

   s1  := sqrt(-1::F)
   ipi := pi()$F * s1

   K2KG          : K -> Kernel FG
   kcomplex      : K -> Union(F, "failed")
   locexplogs    : F -> FG
   localexplogs  : (F, F, List SY) -> FG
```

```
    complexKernels: F -> Record(ker: List K, val: List F)

    K2KG k           == retract(tan F2FG first argument k)@Kernel(FG)
    real? f          == empty?(complexKernels(f).ker)
    real f           == real complexForm f
    imag f           == imag complexForm f

-- returns [[k1,...,kn], [v1,...,vn]] such that ki should be replaced by vi
    complexKernels f ==
      lk:List(K) := empty()
      lv:List(F) := empty()
      for k in tower f repeat
        if (u := kcomplex k) case F then
           lk := concat(k, lk)
           lv := concat(u::F, lv)
      [lk, lv]

-- returns f if it is certain that k is not a real kernel and k = f,
-- "failed" otherwise
    kcomplex k ==
      op := operator k
      is?(k, "nthRoot"::SY) =>
        arg := argument k
        even?(retract(n := second arg)@Z) and ((u := sign(first arg)) case Z)
          and (u::Z < 0) => op(s1, n / 2::F) * op(- first arg, n)
        "failed"
      is?(k, "log"::SY) and ((u := sign(a := first argument k)) case Z)
          and (u::Z < 0) => op(- a) + ipi
      "failed"

    complexForm f ==
      empty?((l := complexKernels f).ker) => complex(f, 0)
      explogs2trigs locexplogs eval(f, l.ker, l.val)

    locexplogs f ==
      any?(x +-> has?(x, "rtrig"),
           operators(g := realElementary f))$List(BasicOperator) =>
              localexplogs(f, g, variables g)
      F2FG g

    complexNormalize(f, x) ==
      any?(y +-> has?(operator y, "rtrig"),
       [k for k in tower(g := realElementary(f, x))
               | member?(x, variables(k::F))]$List(K))$List(K) =>
                   FG2F(rischNormalize(localexplogs(f, g, [x]), x).func)
      rischNormalize(g, x).func

    complexNormalize f ==
      l := variables(g := realElementary f)
      any?(x +-> has?(x, "rtrig"), operators g)$List(BasicOperator) =>
```

```
        h := localexplogs(f, g, l)
        for x in l repeat h := rischNormalize(h, x).func
        FG2F h
      for x in l repeat g := rischNormalize(g, x).func
      g

    complexElementary(f, x) ==
      any?(y +-> has?(operator y, "rtrig"),
       [k for k in tower(g := realElementary(f, x))
                 | member?(x, variables(k::F))]$List(K))$List(K) =>
                     FG2F localexplogs(f, g, [x])
      g

    complexElementary f ==
      any?(x +-> has?(x, "rtrig"),
        operators(g := realElementary f))$List(BasicOperator) =>
          FG2F localexplogs(f, g, variables g)
      g

    localexplogs(f, g, lx) ==
      trigs2explogs(F2FG g, [K2KG k for k in tower f
                           | is?(k, "tan"::SY) or is?(k, "cot"::SY)], lx)

    trigs f ==
      real? f => f
      g := explogs2trigs F2FG f
      real g + s1 * imag g
```

———-

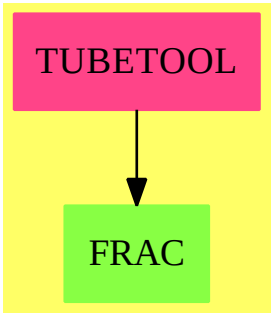
— **TRIGMNIP.dotabb** —

```
"TRIGMNIP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TRIGMNIP"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"TRIGMNIP" -> "ACF"
"TRIGMNIP" -> "FS"
"TRIGMNIP" -> "COMPCAT"
```

———-

21.22 package TUBETOOL TubePlotTools

21.22.1 TubePlotTools (TUBETOOL)



Exports:

cosSinInfo	cross	dot	loopPoints	point
unitVector	?*?	?+?	?-?	

— package TUBETOOL TubePlotTools —

```
)abbrev package TUBETOOL TubePlotTools
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Keywords:
++ Examples:
++ Description:
++ Tools for constructing tubes around 3-dimensional parametric curves.

TubePlotTools(): Exports == Implementation where
  I   ==> Integer
  SF  ==> DoubleFloat
  L   ==> List
  Pt  ==> Point SF

  Exports ==> with
    point: (SF,SF,SF,SF) -> Pt
      ++ point(x1,x2,x3,c) creates and returns a point from the three
      ++ specified coordinates \spad{x1}, \spad{x2}, \spad{x3}, and also
      ++ a fourth coordinate, c, which is generally used to specify the
      ++ color of the point.
    "*" : (SF,Pt) -> Pt
      ++ s * p returns a point whose coordinates are the scalar multiple
      ++ of the point p by the scalar s, preserving the color, or fourth
      ++ coordinate, of p.
    "+" : (Pt,Pt) -> Pt
      ++ p + q computes and returns a point whose coordinates are the sums
```

```
    ++ of the coordinates of the two points \spad{p} and \spad{q}, using
    ++ the color, or fourth coordinate, of the first point \spad{p}
    ++ as the color also of the point \spad{q}.
  "-" : (Pt,Pt) -> Pt
    ++ p - q computes and returns a point whose coordinates are the
    ++ differences of the coordinates of two points \spad{p} and \spad{q},
    ++ using the color, or fourth coordinate, of the first point \spad{p}
    ++ as the color also of the point \spad{q}.
  dot : (Pt,Pt) -> SF
    ++ dot(p,q) computes the dot product of the two points \spad{p}
    ++ and \spad{q} using only the first three coordinates, and returns
    ++ the resulting \spadtype{DoubleFloat}.
  cross : (Pt,Pt) -> Pt
    ++ cross(p,q) computes the cross product of the two points \spad{p}
    ++ and \spad{q} using only the first three coordinates, and keeping
    ++ the color of the first point p.  The result is returned as a point.
  unitVector: Pt -> Pt
    ++ unitVector(p) creates the unit vector of the point p and returns
    ++ the result as a point.  Note that \spad{unitVector(p) = p/|p|}.
  cosSinInfo: I -> L L SF
    ++ cosSinInfo(n) returns the list of lists of values for n, in the form
    ++ \spad{[[cos(n-1) a,sin(n-1) a],...,[cos 2 a,sin 2 a],[cos a,sin a]]}
    ++ where \spad{a = 2 pi/n}.  Note that n should be greater than 2.
  loopPoints: (Pt,Pt,Pt,SF,L L SF) -> L Pt
    ++ loopPoints(p,n,b,r,lls) creates and returns a list of points
    ++ which form the loop with radius r, around the center point
    ++ indicated by the point p, with the principal normal vector of
    ++ the space curve at point p given by the point(vector) n, and the
    ++ binormal vector given by the point(vector) b, and a list of lists,
    ++ lls, which is the \spadfun{cosSinInfo} of the number of points
    ++ defining the loop.

Implementation ==> add
  import PointPackage(SF)

  point(x,y,z,c) == point(l : L SF := [x,y,z,c])

  getColor: Pt -> SF
  getColor pt == (maxIndex pt > 3 => color pt; 0)

  getColor2: (Pt,Pt) -> SF
  getColor2(p0,p1) ==
    maxIndex p0 > 3 => color p0
    maxIndex p1 > 3 => color p1
    0

  a * p ==
    l : L SF := [a * xCoord p,a * yCoord p,a * zCoord p,getColor p]
    point l
```

```
    p0 + p1 ==
      l : L SF := [xCoord p0 + xCoord p1,yCoord p0 + yCoord p1,_
                   zCoord p0 + zCoord p1,getColor2(p0,p1)]
      point l

    p0 - p1 ==
      l : L SF := [xCoord p0 - xCoord p1,yCoord p0 - yCoord p1,_
                   zCoord p0 - zCoord p1,getColor2(p0,p1)]
      point l

    dot(p0,p1) ==
      (xCoord p0 * xCoord p1) + (yCoord p0 * yCoord p1) +_
        (zCoord p0 * zCoord p1)

    cross(p0,p1) ==
      x0 := xCoord p0; y0 := yCoord p0; z0 := zCoord p0;
      x1 := xCoord p1; y1 := yCoord p1; z1 := zCoord p1;
      l : L SF := [y0 * z1 - y1 * z0,z0 * x1 - z1 * x0,_
                   x0 * y1 - x1 * y0,getColor2(p0,p1)]
      point l

    unitVector p == (inv sqrt dot(p,p)) * p

    cosSinInfo n ==
      ans : L L SF := nil()
      theta : SF := 2 * pi()/n
      for i in 1..(n-1) repeat             --!! make more efficient
        angle := i * theta
        ans := concat([cos angle,sin angle],ans)
      ans

    loopPoints(ctr,pNorm,bNorm,rad,cosSin) ==
      ans : L Pt := nil()
      while not null cosSin repeat
        cossin := first cosSin; cos := first cossin; sin := second cossin
        ans := cons(ctr + rad * (cos * pNorm + sin * bNorm),ans)
        cosSin := rest cosSin
      pt := ctr + rad * pNorm
      concat(pt,concat(ans,pt))
```

———-

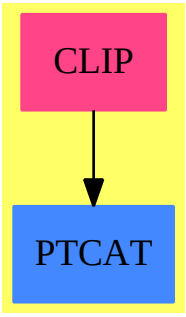
— TUBETOOL.dotabb —

```
"TUBETOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TUBETOOL"]
"FRAC" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FRAC"]
"TUBETOOL" -> "FRAC"
```

———-

21.23 package CLIP TwoDimensionalPlotClipping

21.23.1 TwoDimensionalPlotClipping (CLIP)



Exports:
clip clipParametric clipWithRanges

— package CLIP TwoDimensionalPlotClipping —

```
)abbrev package CLIP TwoDimensionalPlotClipping
++ Author: Clifton J. Williamson
++ Date Created: 22 December 1989
++ Date Last Updated: 10 July 1990
++ Keywords: plot, singularity
++ Examples:
++ References:
++ Description:
++ Automatic clipping for 2-dimensional plots
++ The purpose of this package is to provide reasonable plots of
++ functions with singularities.

TwoDimensionalPlotClipping(): Exports == Implementation where
  B      ==> Boolean
  L      ==> List
  SEG    ==> Segment
  RN     ==> Fraction Integer
  SF     ==> DoubleFloat
  Pt     ==> Point DoubleFloat
  PLOT   ==> Plot
  CLIPPED ==> Record(brans: L L Pt,xValues: SEG SF,yValues: SEG SF)

  Exports ==> with
    clip: PLOT -> CLIPPED
```

```
      ++ clip(p) performs two-dimensional clipping on a plot, p, from
      ++ the domain \spadtype{Plot} for the graph of one variable,
      ++ \spad{y = f(x)}; the default parameters \spad{1/4} for the fraction
      ++ and \spad{5/1} for the scale are used in the \spadfun{clip} function.
    clip: (PLOT,RN,RN) -> CLIPPED
      ++ clip(p,frac,sc) performs two-dimensional clipping on a plot, p,
      ++ from the domain \spadtype{Plot} for the graph of one variable
      ++ \spad{y = f(x)}; the fraction parameter is specified by \spad{frac}
      ++ and the scale parameter is specified by \spad{sc} for use in the
      ++ \spadfun{clip} function.
    clipParametric: PLOT -> CLIPPED
      ++ clipParametric(p) performs two-dimensional clipping on a plot,
      ++ p, from the domain \spadtype{Plot} for the parametric curve
      ++ \spad{x = f(t)}, \spad{y = g(t)}; the default parameters \spad{1/2}
      ++ for the fraction and \spad{5/1} for the scale are used in the
      ++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
      ++ function.
    clipParametric: (PLOT,RN,RN) -> CLIPPED
      ++ clipParametric(p,frac,sc) performs two-dimensional clipping on a
      ++ plot, p, from the domain \spadtype{Plot} for the parametric curve
      ++ \spad{x = f(t)}, \spad{y = g(t)}; the fraction parameter is
      ++ specified by \spad{frac} and the scale parameter is specified
      ++ by \spad{sc} for use in the \fakeAxiomFun{iClipParametric} subroutine,
      ++ which is called by this function.
    clipWithRanges: (L L Pt,SF,SF,SF,SF) -> CLIPPED
      ++ clipWithRanges(pointLists,xMin,xMax,yMin,yMax) performs clipping
      ++ on a list of lists of points, \spad{pointLists}.  Clipping is
      ++ done within the specified ranges of \spad{xMin}, \spad{xMax} and
      ++ \spad{yMin}, \spad{yMax}.  This function is used internally by
      ++ the \fakeAxiomFun{iClipParametric} subroutine in this package.
    clip: L Pt -> CLIPPED
      ++ clip(l) performs two-dimensional clipping on a curve l, which is
      ++ a list of points; the default parameters \spad{1/2} for the
      ++ fraction and \spad{5/1} for the scale are used in the
      ++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
      ++ function.
    clip: L L Pt -> CLIPPED
      ++ clip(ll) performs two-dimensional clipping on a list of lists
      ++ of points, \spad{ll}; the default parameters \spad{1/2} for
      ++ the fraction and \spad{5/1} for the scale are used in the
      ++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
      ++ function.

  Implementation ==> add
    import PointPackage(DoubleFloat)
    import ListFunctions2(Point DoubleFloat,DoubleFloat)

    point:(SF,SF) -> Pt
    intersectWithHorizLine:(SF,SF,SF,SF,SF) -> Pt
    intersectWithVertLine:(SF,SF,SF,SF,SF) -> Pt
```

```
intersectWithBdry:(SF,SF,SF,SF,Pt,Pt) -> Pt
discardAndSplit: (L Pt,Pt -> B,SF,SF,SF,SF) -> L L Pt
norm: Pt -> SF
iClipParametric: (L L Pt,RN,RN) -> CLIPPED
findPt: L L Pt -> Union(Pt,"failed")
Fnan?: SF ->Boolean
Pnan?:Pt ->Boolean

Fnan? x == x~=x
Pnan? p == any?(Fnan?,p)

iClipParametric(pointLists,fraction,scale) ==
  -- error checks and special cases
  (fraction < 0) or (fraction > 1) =>
    error "clipDraw: fraction should be between 0 and 1"
  empty? pointLists => [nil(),segment(0,0),segment(0,0)]
  -- put all points together , sort them according to norm
  sortedList := sort((x:Pt,y:Pt):Boolean +-> norm(x) < norm(y),
                select((z:Pt):Boolean +-> not Pnan? z,concat pointLists))
  empty? sortedList => [nil(),segment(0,0),segment(0,0)]
  n := # sortedList
  num := numer fraction
  den := denom fraction
  clipNum := (n * num) quo den
  lastN := n - 1 - clipNum
  firstPt := first sortedList
  xMin : SF := xCoord firstPt
  xMax : SF := xCoord firstPt
  yMin : SF := yCoord firstPt
  yMax : SF := yCoord firstPt
  -- calculate min/max for the first (1-fraction)*N points
  -- this contracts the range
  -- this unnecessarily clips monotonic functions (step-function, x^(high power),etc.)
  for k in 0..lastN  for pt in rest sortedList repeat
    xMin := min(xMin,xCoord pt)
    xMax := max(xMax,xCoord pt)
    yMin := min(yMin,yCoord pt)
    yMax := max(yMax,yCoord pt)
  xDiff := xMax - xMin; yDiff := yMax - yMin
  xDiff = 0 =>
    yDiff = 0 =>
      [pointLists,segment(xMin-1,xMax+1),segment(yMin-1,yMax+1)]
    [pointLists,segment(xMin-1,xMax+1),segment(yMin,yMax)]
  yDiff = 0 =>
    [pointLists,segment(xMin,xMax),segment(yMin-1,yMax+1)]
  numm := numer scale; denn := denom scale
  -- now expand the range by scale
  xMin := xMin - (numm :: SF) * xDiff / (denn :: SF)
  xMax := xMax + (numm :: SF) * xDiff / (denn :: SF)
  yMin := yMin - (numm :: SF) * yDiff / (denn :: SF)
```

```
  yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
  -- clip with the calculated range
  newclip:=clipWithRanges(pointLists,xMin,xMax,yMin,yMax)
  -- if we split the lists use the new clip
  # (newclip.brans) > # pointLists   => newclip
  -- calculate extents
  xs :L SF:= map (xCoord,sortedList)
  ys :L SF:= map (yCoord,sortedList)
  xMin :SF :=reduce (min,xs)
  yMin :SF :=reduce (min,ys)
  xMax :SF :=reduce (max,xs)
  yMax :SF :=reduce (max,ys)
  xseg:SEG SF :=xMin..xMax
  yseg:SEG SF :=yMin..yMax
  -- return original
  [pointLists,xseg,yseg]@CLIPPED



point(xx,yy) == point(l : L SF := [xx,yy])

intersectWithHorizLine(x1,y1,x2,y2,yy) ==
  x1 = x2 => point(x1,yy)
  point(x1 + (x2 - x1)*(yy - y1)/(y2 - y1),yy)

intersectWithVertLine(x1,y1,x2,y2,xx) ==
  y1 = y2 => point(xx,y1)
  point(xx,y1 + (y2 - y1)*(xx - x1)/(x2 - x1))

intersectWithBdry(xMin,xMax,yMin,yMax,pt1,pt2) ==
  -- pt1 is in rectangle, pt2 is not
  x1 := xCoord pt1; y1 := yCoord pt1
  x2 := xCoord pt2; y2 := yCoord pt2
  if y2 > yMax then
    pt2 := intersectWithHorizLine(x1,y1,x2,y2,yMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if y2 < yMin then
    pt2 := intersectWithHorizLine(x1,y1,x2,y2,yMin)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 > xMax then
    pt2 := intersectWithVertLine(x1,y1,x2,y2,xMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 < xMin then
    pt2 := intersectWithVertLine(x1,y1,x2,y2,xMin)
  pt2

discardAndSplit(pointList,pred,xMin,xMax,yMin,yMax) ==
  ans : L L Pt := nil()
  list : L Pt  := nil()
```

```
      lastPt? : B  := false
      lastPt : Pt  := point(0,0)
      while not empty? pointList repeat
        pt := first pointList
        pointList := rest pointList
        pred(pt) =>
          if (empty? list) and lastPt? then
            bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,pt,lastPt)
            -- print bracket [ coerce bdryPt ,coerce pt ]
            --list := cons(bdryPt,list)
          list := cons(pt,list)
        if not empty? list then
          bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,first list,pt)
          -- print bracket [ coerce bdryPt,coerce first list]
          --list := cons(bdryPt,list)
          ans := cons( list,ans)
        lastPt := pt
        lastPt? := true
        list := nil()
      empty? list => ans
      reverse_! cons(reverse_! list,ans)

    clip(plot,fraction,scale) ==
--      sayBrightly(["   clip: "::OutputForm]$List(OutputForm))$Lisp
      (fraction < 0) or (fraction > 1/2) =>
        error "clipDraw: fraction should be between 0 and 1/2"
      xVals := xRange plot
      empty?(pointLists := listBranches plot) =>
        [nil(),xVals,segment(0,0)]
      more?(pointLists := listBranches plot,1) =>
        error "clipDraw: plot has more than one branch"
      empty?(pointList := first pointLists) =>
        [nil(),xVals,segment(0,0)]
      sortedList := sort((x,y)+->yCoord(x) < yCoord(y),pointList)
      n := # sortedList; num := numer fraction; den := denom fraction
      clipNum := (n * num) quo den
      -- throw out points with large and small y-coordinates
      yMin := yCoord(sortedList.clipNum)
      yMax := yCoord(sortedList.(n - 1 - clipNum))
      if Fnan? yMin then yMin : SF := 0
      if Fnan? yMax then yMax : SF := 0
      (yDiff := yMax - yMin) = 0 =>
        [pointLists,xRange plot,segment(yMin - 1,yMax + 1)]
      numm := numer scale; denn := denom scale
      xMin := lo xVals; xMax := hi xVals
      yMin := yMin - (numm :: SF) * yDiff / (denn :: SF)
      yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
      lists := discardAndSplit(pointList,_
         x +-> (yCoord(x) < yMax) and (yCoord(x) > yMin),
           xMin,xMax,yMin,yMax)
```

```
    yMin := yCoord(sortedList.clipNum)
    yMax := yCoord(sortedList.(n - 1 - clipNum))
    if Fnan? yMin then yMin : SF := 0
    if Fnan? yMax then yMax : SF := 0
    for list in lists repeat
      for pt in list repeat
        if not Fnan?(yCoord pt) then
          yMin := min(yMin,yCoord pt)
          yMax := max(yMax,yCoord pt)
    [lists,xVals,segment(yMin,yMax)]

  clip(plot:PLOT) == clip(plot,1/4,5/1)

  norm(pt) ==
    x := xCoord(pt); y := yCoord(pt)
    if Fnan? x then
      if Fnan? y then
        r:SF := 0
      else
        r:SF := y**2
    else
      if Fnan? y then
        r:SF := x**2
      else
        r:SF := x**2 + y**2
    r

  findPt lists ==
    for list in lists repeat
      not empty? list =>
           for p in list repeat
             not Pnan? p => return p
    "failed"

  clipWithRanges(pointLists,xMin,xMax,yMin,yMax) ==
    lists : L L Pt := nil()
    for pointList in pointLists repeat
      lists := concat(lists,discardAndSplit(pointList,
        (x:Pt):Boolean +-> (xCoord(x) <= xMax) and (xCoord(x) >= xMin) and
          (yCoord(x) <= yMax) and (yCoord(x) >= yMin),
           xMin,xMax,yMin,yMax))
    (pt := findPt lists) case "failed" =>
      [nil(),segment(0,0),segment(0,0)]
    firstPt := pt :: Pt
    xMin : SF := xCoord firstPt; xMax : SF := xCoord firstPt
    yMin : SF := yCoord firstPt; yMax : SF := yCoord firstPt
    for list in lists repeat
      for pt in list repeat
        if not Pnan? pt then
          xMin := min(xMin,xCoord pt)
```

```
            xMax := max(xMax,xCoord pt)
            yMin := min(yMin,yCoord pt)
            yMax := max(yMax,yCoord pt)
      [lists,segment(xMin,xMax),segment(yMin,yMax)]

    clipParametric(plot,fraction,scale) ==
      iClipParametric(listBranches plot,fraction,scale)

    clipParametric plot == clipParametric(plot,1/2,5/1)

    clip(l: L Pt)   == iClipParametric(list l,1/2,5/1)
    clip(l: L L Pt) == iClipParametric(l,1/2,5/1)
```

———-

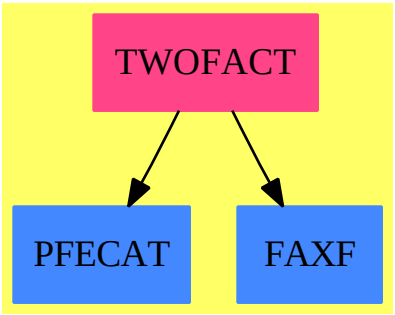
— **CLIP.dotabb** —

```
"CLIP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CLIP"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"CLIP" -> "PTCAT"
```

———-

21.24 package TWOFACT TwoFactorize

21.24.1 TwoFactorize (TWOFACT)



Exports:
generalSqFr generalTwoFactor twoFactor

— **package TWOFACT TwoFactorize** —

```
)abbrev package TWOFACT TwoFactorize
++ Authors : P.Gianni, J.H.Davenport
++ Date Created : May 1990
++ Date Last Updated: March 1992
++ Description:
++ A basic package for the factorization of bivariate polynomials
++ over a finite field.
++ The functions here represent the base step for the multivariate factorizer.

TwoFactorize(F) : C == T
 where
  F          :   FiniteFieldCategory
  SUP       ==>  SparseUnivariatePolynomial
  R         ==>  SUP F
  P         ==>  SUP R
  UPCF2     ==>  UnivariatePolynomialCategoryFunctions2

  C == with
    generalTwoFactor    : (P)  ->  Factored P
      ++ generalTwoFactor(p) returns the factorisation of polynomial p,
      ++ a sparse univariate polynomial (sup) over a
      ++ sup over F.
    generalSqFr    : (P)  ->  Factored P
      ++ generalSqFr(p) returns the square-free factorisation of polynomial p,
      ++ a sparse univariate polynomial (sup) over a
      ++ sup over F.
    twoFactor    : (P,Integer)  ->  Factored P
      ++ twoFactor(p,n) returns the factorisation of polynomial p,
      ++ a sparse univariate polynomial (sup) over a
      ++ sup over F.
      ++ Also, p is assumed primitive and square-free and n is the
      ++ degree of the inner variable of p (maximum of the degrees
      ++ of the coefficients of p).

  T == add
    PI ==> PositiveInteger
    NNI ==> NonNegativeInteger
    import CommuteUnivariatePolynomialCategory(F,R,P)

                  ----  Local Functions  ----
    computeDegree  :  (P,Integer,Integer) -> PI
    exchangeVars   :           P          -> P
    exchangeVarTerm:        (R, NNI)      -> P
    pthRoot        :     (R, NNI, NNI)    -> R

    -- compute the degree of the extension to reduce the polynomial to a
    -- univariate one
    computeDegree(m : P,mx:Integer,q:Integer): PI ==
      my:=degree m
      n1:Integer:=length(10*mx*my)
```

```
      n2:Integer:=length(q)-1
      n:=(n1 quo n2)+1
      n::PI
--      n=1 => 1$PositiveInteger
--      (nextPrime(max(n,min(mx,my)))$IntegerPrimesPackage(Integer))::PI

    exchangeVars(p : P) : P ==
       p = 0 => 0
       exchangeVarTerm(leadingCoefficient p, degree p) +
          exchangeVars(reductum p)

    exchangeVarTerm(c:R, e:NNI) : P ==
       c = 0 => 0
       monomial(monomial(leadingCoefficient c, e)$R, degree c)$P +
          exchangeVarTerm(reductum c, e)

    pthRoot(poly:R,p:NonNegativeInteger,PthRootPow:NonNegativeInteger):R ==
       tmp:=divideExponents(map((x:F):F+->(x::F)**PthRootPow,poly),p)
       tmp case "failed" => error "consistency error in TwoFactor"
       tmp

    fUnion ==> Union("nil", "sqfr", "irred", "prime")
    FF     ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

    generalSqFr(m:P): Factored P ==
       m = 0 => 0
       degree m = 0 =>
         l:=squareFree(leadingCoefficient m)
         makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
       cont := content m
       m := (m exquo cont)::P
       sqfrm := squareFree m
       pfaclist : List FF := empty()
       unitPart := unit sqfrm
       for u in factorList sqfrm repeat
          u.flg = "nil" =>
             uexp:NNI:=(u.xpnt):NNI
             nfacs:=squareFree(exchangeVars u.fctr)
             for v in factorList nfacs repeat
                pfaclist:=cons([v.flg, exchangeVars v.fctr, v.xpnt*uexp],
                               pfaclist)
             unitPart := unit(nfacs)**uexp * unitPart
          pfaclist := cons(u,pfaclist)
       cont ^= 1 =>
           sqp := squareFree cont
           contlist:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList sqp]
           pfaclist:= append(contlist, pfaclist)
           makeFR(unit(sqp)*unitPart,pfaclist)
       makeFR(unitPart,pfaclist)
```

```
generalTwoFactor(m:P): Factored P ==
   m = 0 => 0
   degree m = 0 =>
     l:=factor(leadingCoefficient m)$DistinctDegreeFactorize(F,R)
     makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
   ll:List FF
   ll:=[]
   unitPart:P
   cont:=content m
   if degree(cont)>0 then
      m1:=m exquo cont
      m1 case "failed" => error "content doesn't divide"
      m:=m1
      contfact:=factor(cont)$DistinctDegreeFactorize(F,R)
      unitPart:=(unit contfact)::P
      ll:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList contfact]
   else
      unitPart:=cont::P
   sqfrm:=squareFree m
   for u in factors sqfrm repeat
       expo:=u.exponent
       if expo < 0 then error "negative exponent in a factorisation"
       expon:NonNegativeInteger:=expo::NonNegativeInteger
       fac:=u.factor
       degree fac = 1 => ll:=[["irred",fac,expon],:ll]
       differentiate fac = 0 =>
          -- the polynomial is  inseparable w.r.t. its main variable
          map(differentiate,fac) = 0 =>
            p:=characteristic$F
            PthRootPow:=(size$F exquo p)::NonNegativeInteger
            m1:=divideExponents(map(x+->pthRoot(x,p,PthRootPow),fac),p)
            m1 case "failed" => error "consistency error in TwoFactor"
            res:=generalTwoFactor m1
            unitPart:=unitPart*unit(res)**((p*expon)::NNI)
            ll:=
             [:[[v.flg,v.fctr,expon*p*v.xpnt] for v in factorList res],:ll]
          m2:=generalTwoFactor swap fac
          unitPart:=unitPart*unit(m2)**(expon::NNI)
          ll:=
           [:[[v.flg,swap v.fctr,expon*v.xpnt] for v in factorList m2],:ll]
       ydeg:="max"/[degree w for w in coefficients fac]
       twoF:=twoFactor(fac,ydeg)
       unitPart:=unitPart*unit(twoF)**expon
       ll:=[:[[v.flg,v.fctr,expon*v.xpnt] for v in factorList twoF],
            :ll]
   makeFR(unitPart,ll)

-- factorization of a primitive square-free bivariate polynomial --
twoFactor(m:P,dx:Integer):Factored P ==
```

```
-- choose the degree for the extension
n:PI:=computeDegree(m,dx,size()$F)
-- extend the field
-- find the substitution for x
look:Boolean:=true
dm:=degree m
try:Integer:=min(5,size()$F)
i:Integer:=0
lcm := leadingCoefficient m
umv : R
while look and i < try repeat
   vval := random()$F
   i:=i+1
   zero? elt(lcm, vval) => "next value"
   umv := map(x +-> elt(x,vval), m)$UPCF2(R, P, F, R)
   degree(gcd(umv,differentiate umv))^=0 => "next val"
   n := 1
   look := false
extField:=FiniteFieldExtension(F,n)
SUEx:=SUP extField
TP:=SparseUnivariatePolynomial SUEx
mm:TP:=0
m1:=m
while m1^=0 repeat
  mm:=mm+monomial(map(coerce,leadingCoefficient m1)$UPCF2(F,R,
         extField,SUEx),degree m1)
  m1:=reductum m1
lcmm := leadingCoefficient mm
val : extField
umex : SUEx
if not look then
   val := vval :: extField
   umex := map(coerce, umv)$UPCF2(F, R, extField, SUEx)
while look repeat
  val:=random()$extField
  i:=i+1
  elt(lcmm,val)=0 => "next value"
  umex := map(x +-> elt(x,val), mm)$UPCF2(SUEx, TP, extField, SUEx)
  degree(gcd(umex,differentiate umex))^=0 => "next val"
  look:=false
prime:SUEx:=monomial(1,1)-monomial(val,0)
fumex:=factor(umex)$DistinctDegreeFactorize(extField,SUEx)
lfact1:=factors fumex

#lfact1=1 => primeFactor(m,1)
lfact : List TP :=
   [map(coerce,lf.factor)$UPCF2(extField,SUEx,SUEx,TP)
    for lf in lfact1]
lfact:=cons(map(coerce,unit fumex)$UPCF2(extField,SUEx,SUEx,TP),
            lfact)
```

```
import GeneralHenselPackage(SUEx,TP)
dx1:PI:=(dx+1)::PI
lfacth:=completeHensel(mm,lfact,prime,dx1)
lfactk: List P :=[]
Normp := NormRetractPackage(F, extField, SUEx, TP, n)

while not empty? lfacth repeat
  ff := first lfacth
  lfacth := rest lfacth
  if (c:=leadingCoefficient leadingCoefficient ff) ^=1 then
    ff:=((inv c)::SUEx)* ff
  not ((ffu:= retractIfCan(ff)$Normp) case "failed") =>
             lfactk := cons(ffu::P, lfactk)
  normfacs := normFactors(ff)$Normp
  lfacth := [g for g in lfacth | not member?(g, normfacs)]
  ffn := */normfacs
  lfactk:=cons(retractIfCan(ffn)$Normp :: P, lfactk)
*/[primeFactor(ff1,1) for ff1 in lfactk]
```

———-

— TWOFACT.dotabb —

```
"TWOFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TWOFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"TWOFACT" -> "PFECAT"
"TWOFACT" -> "FAXF"
```

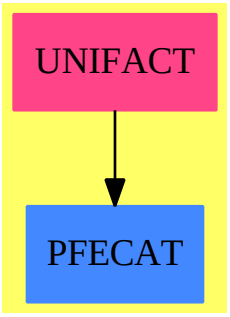
———-

Chapter 22

Chapter U

22.1 package UNIFACT UnivariateFactorize

22.1.1 UnivariateFactorize (UNIFACT)



Exports:
factor factorSquareFree henselFact

— package UNIFACT UnivariateFactorize —

```
)abbrev package UNIFACT UnivariateFactorize
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: December 1993
++ Description:
++ Package for the factorization of univariate polynomials with integer
++ coefficients. The factorization is done by "lifting" (HENSEL) the
++ factorization over a finite field.
```

```
UnivariateFactorize(ZP) : public == private where
  Z    ==>  Integer
  PI   ==>  PositiveInteger
  NNI  ==>  NonNegativeInteger
  SUPZ ==>  SparseUnivariatePolynomial Z

  ZP : UnivariatePolynomialCategory Z

  FR        ==>  Factored ZP
  fUnion    ==>  Union("nil", "sqfr", "irred", "prime")
  FFE       ==>  Record(flg:fUnion, fctr:ZP, xpnt:Z)
  ParFact   ==>  Record(irr: ZP,pow: Z)
  FinalFact ==>  Record(contp: Z,factors:List(ParFact))


  public == with
     factor            :           ZP              -> FR
       ++ factor(m) returns the factorization of m
     factorSquareFree  :           ZP              -> FR
       ++ factorSquareFree(m) returns the factorization of m square free
       ++ polynomial
     henselFact        :      (ZP,Boolean)         -> FinalFact
       ++ henselFact(m,flag) returns the factorization of m,
       ++ FinalFact is a Record s.t. FinalFact.contp=content m,
       ++ FinalFact.factors=List of irreducible factors
       ++ of m with exponent , if flag =true the polynomial is
       ++ assumed square free.

  private == add
                --- local functions ---

     henselfact  :           ZP      -> List(ZP)
     quadratic   :           ZP      -> List(ZP)
     remp        :        (Z, PI)    -> Z
     negShiftz   :        (Z, PI)    -> Z
     negShiftp   :        (ZP,PI)    -> ZP
     bound       :           ZP      -> PI
     choose      :           ZP      -> FirstStep
     eisenstein  :           ZP      -> Boolean
     isPowerOf2  :           Z       -> Boolean
     subMinusX   :          SUPZ     -> ZP
     sqroot      :           Z       -> Z

                 ---   declarations  ---
     CYC       ==> CyclotomicPolynomialPackage()
     DDRecord  ==> Record(factor: ZP,degree: Z)
     DDList    ==> List DDRecord
     FirstStep ==> Record(prime:PI,factors:DDList)
     ContPrim  ==> Record(cont: Z,prim: ZP)
```

```
import GeneralHenselPackage(Z,ZP)
import ModularDistinctDegreeFactorizer ZP


factor(m: ZP) ==
  flist := henselFact(m,false)
  ctp:=unitNormal flist.contp
  makeFR((ctp.unit)::ZP,cons(["nil",ctp.canonical::ZP,1$Z]$FFE,
                [["prime",u.irr,u.pow]$FFE for u in flist.factors]))

factorSquareFree(m: ZP) ==
  flist := henselFact(m,true)
  ctp:=unitNormal flist.contp
  makeFR((ctp.unit)::ZP,cons(["nil",ctp.canonical::ZP,1$Z]$FFE,
               [["prime",u.irr,u.pow]$FFE for u in flist.factors]))


-- Integer square root: returns 0 if t is non-positive
sqroot(t: Z): Z  ==
 t <= 0 => 0
 s:Integer:=t::Integer
 s:=approxSqrt(s)$IntegerRoots(Integer)
 t:=s::Z
 t

-- Eisenstein criterion: returns true if polynomial is
-- irreducible. Result of false in inconclusive.
eisenstein(m : ZP): Boolean ==
  -- calculate the content of the terms after the first
  c := content reductum m
  c = 0 => false
  c = 1 => false
  -- factor the content
  -- if there is a prime in the factorization that does not divide
  -- the leading term and appears to multiplicity 1, and the square
  -- of this does not divide the last coef, return true.
  -- Otherwise reurn false.
  lead := leadingCoefficient m
  trail := lead
  m := reductum m
  while m ^= 0 repeat
    trail := leadingCoefficient m
    m:= reductum m
  fc := factor(c) :: Factored(Z)
  for r in factors fc repeat
    if (r.exponent = 1) and (0 ^= (lead rem r.factor)) and
      (0 ^= (trail rem (r.factor ** 2))) then return true
  false

negShiftz(n: Z,Modulus:PI): Z ==
```

```
  if n < 0 then n := n+Modulus
  n > (Modulus quo 2) => n-Modulus
  n

negShiftp(pp: ZP,Modulus:PI): ZP ==
  map(x +-> negShiftz(x,Modulus),pp)

-- Choose the bound for the coefficients of factors
bound(m: ZP):PI ==
  nm,nmq2,lcm,bin0,bin1:NNI
  cbound,j : PI
  k:NNI
  lcm := abs(leadingCoefficient m)::NNI
  nm := (degree m)::NNI
  nmq2:NNI := nm quo 2
  norm: Z := sqroot(+/[coefficient(m,k)**2 for k in 0..nm])
  if nmq2^=1 then nm := (nmq2-1):NNI
  else nm := nmq2
  bin0 := nm
  cbound := (bin0*norm+lcm)::PI
  for i in 2..(nm-1)::NNI repeat
    bin1 := bin0
    bin0 := (bin0*(nm+1-i):NNI) quo i
    j := (bin0*norm+bin1*lcm)::PI
    if cbound<j then cbound := j
  (2*cbound*lcm)::PI -- adjusted by lcm to prepare for exquo in ghensel

remp(t: Z,q:PI): Z == ((t := t rem q)<0 => t+q ;t)

numFactors(ddlist:DDList): Z ==
  ans: Z := 0
  for dd in ddlist repeat
    (d := degree(dd.factor)) = 0 => nil
    ans := ans + ((d pretend Z) exquo dd.degree):: Z
  ans

-- select the prime,try up to 4 primes,
-- choose the one yielding the fewest factors, but stopping if
-- fewer than 9 factors
choose(m: ZP):FirstStep ==
  qSave:PI := 1
  ddSave:DDList := []
  numberOfFactors: Z := 0
  lcm := leadingCoefficient m
  k: Z := 1
  ddRep := 5
  disc:ZP:=0
  q:PI:=2
  while k<ddRep repeat
    -- q must be a new prime number at each iteration
```

```
          q:=nextPrime(q)$IntegerPrimesPackage(Z) pretend PI
          (rr:=lcm rem q) = 0$Z => "next prime"
          disc:=gcd(m,differentiate m,q)
          (degree disc)^=0 => "next prime"
          k := k+1
          newdd := ddFact(m,q)
          ((n := numFactors(newdd)) < 9) =>
            ddSave := newdd
            qSave := q
            k := 5
          (numberOfFactors = 0) or (n < numberOfFactors) =>
            ddSave := newdd
            qSave := q
            numberOfFactors := n
        [qSave,ddSave]$FirstStep

      -- Find the factors of m,primitive, square-free, with lc positive
      -- and mindeg m = 0
      henselfact1(m: ZP):List(ZP) ==
       zero? degree m =>
--           one? m => []
           (m = 1) => []
           [m]
       selected := choose(m)
       (numFactors(selected.factors) = 1$Z) => [m]
       q := selected.prime
       fl := separateFactors(selected.factors,q)
       --choose the bound
       cbound := bound(m)
       completeHensel(m,fl,q,cbound)

      -- check for possible degree reduction
      -- could use polynomial decomposition ?
      henselfact(m: ZP):List ZP ==
       deggcd:=degree m
       mm:= m
       while not zero? mm repeat (deggcd:=gcd(deggcd, degree mm); mm:=reductum mm)
       deggcd>1 and deggcd<degree m =>
          faclist := henselfact1(divideExponents(m, deggcd)::ZP)
          "append"/[henselfact1 multiplyExponents(mm, deggcd) for mm in faclist]
       henselfact1 m

      quadratic(m: ZP):List(ZP) ==
        d,d2: Z
        d := coefficient(m,1)**2-4*coefficient(m,0)*coefficient(m,2)
        d2 := sqroot(d)
        (d-d2**2)^=0 => [m]
        alpha: Z := coefficient(m,1)+d2
        beta: Z := 2*coefficient(m,2)
        d := gcd(alpha,beta)
```

```
      if d ^=1 then
        alpha := alpha quo d
        beta := beta quo d
      m0: ZP := monomial(beta,1)+monomial(alpha,0)
      cons(m0,[(m exquo m0):: ZP])

    isPowerOf2(n : Z): Boolean ==
      n = 1 => true
      qr : Record(quotient: Z, remainder: Z) := divide(n,2)
      qr.remainder = 1 => false
      isPowerOf2 qr.quotient

    subMinusX(supPol : SUPZ): ZP ==
      minusX : SUPZ := monomial(-1,1)$SUPZ
      (elt(supPol,minusX)$SUPZ) : ZP

--  Factorize the polynomial m, test=true if m is known to be
--  square-free, false otherwise.
--  FinalFact.contp=content m, FinalFact.factors=List of irreducible
--  factors with exponent .
    henselFact(m: ZP,test:Boolean):FinalFact ==
     factorlist : List(ParFact) := []
     c : Z

     -- make m primitive
     c := content m
     m := (m exquo c)::ZP

     -- make the lc m positive
     if leadingCoefficient m < 0 then
       c := -c
       m := -m

     -- is x**d factor of m?
     if (d := minimumDegree m) >0 then
       m := (monicDivide(m,monomial(1,d))).quotient
       factorlist := [[monomial(1,1),d]$ParFact]

     d := degree m

     -- is m constant?
     d=0 => [c,factorlist]$FinalFact

     -- is m linear?
     d=1 => [c,cons([m,1]$ParFact,factorlist)]$FinalFact

     -- does m satisfy Eisenstein's criterion?
     eisenstein m => [c,cons([m,1]$ParFact,factorlist)]$FinalFact

     lcPol : ZP := leadingCoefficient(m) :: ZP
```

```
-- is m cyclotomic (x**n - 1)?
-lcPol = reductum(m) =>    -- if true, both will = 1
  for fac in
    (cyclotomicDecomposition(degree m)$CYC : List ZP) repeat
      factorlist := cons([fac,1]$ParFact,factorlist)
  [c,factorlist]$FinalFact

-- is m odd cyclotomic (x**(2*n+1) + 1)?
odd?(d) and (lcPol = reductum(m)) =>
  for sfac in cyclotomicDecomposition(degree m)$CYC repeat
     fac:=subMinusX sfac
     if leadingCoefficient fac < 0 then fac := -fac
     factorlist := cons([fac,1]$ParFact,factorlist)
  [c,factorlist]$FinalFact

-- is the poly of the form x**n + 1 with n a power of 2?
-- if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(m)) =>
  factorlist := cons([m,1]$ParFact,factorlist)
  [c,factorlist]$FinalFact

-- is m quadratic?
d=2 =>
 lfq:List(ZP) := quadratic m
 #lfq=1 => [c,cons([lfq.first,1]$ParFact,factorlist)]$FinalFact
 (lf0,lf1) := (lfq.first,second lfq)
 if lf0=lf1 then factorlist := cons([lf0,2]$ParFact,factorlist)
 else factorlist := append([[v,1]$ParFact for v in lfq],factorlist)
 [c,factorlist]$FinalFact

-- m is square-free
test =>
  fln := henselfact(m)
  [c,append(factorlist,[[pf,1]$ParFact for pf in fln])]$FinalFact

-- find the square-free decomposition of m
irrFact := squareFree(m)
llf := factors irrFact

-- factorize the square-free primitive terms
for l1 in llf repeat
  d1 := l1.exponent
  pol := l1.factor
  degree pol=1 => factorlist := cons([pol,d1]$ParFact,factorlist)
  degree pol=2 =>
    fln := quadratic(pol)
    factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
  fln := henselfact(pol)
  factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
```

```
[c,factorlist]$FinalFact
```

———-

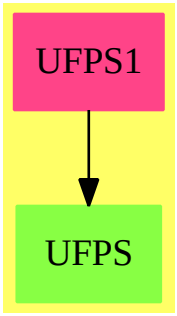
— **UNIFACT.dotabb** —

```
"UNIFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UNIFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UNIFACT" -> "PFECAT"
```

———-

22.2 package UFPS1 UnivariateFormalPowerSeriesFunctions

22.2.1 UnivariateFormalPowerSeriesFunctions (UFPS1)



Exports:
hadamard

— **package UFPS1 UnivariateFormalPowerSeriesFunctions** —

```
)abbrev package UFPS1 UnivariateFormalPowerSeriesFunctions
++ Description:
++ This package has no description

UnivariateFormalPowerSeriesFunctions(Coef: Ring): Exports == Implementation
  where

    UFPS ==> UnivariateFormalPowerSeries Coef
```

```
    Exports == with

      hadamard: (UFPS, UFPS) -> UFPS

    Implementation == add

      hadamard(f, g) ==
        series map((z1:Coef,z2:Coef):Coef +-> z1*z2,
                    coefficients f, coefficients g)
                     $StreamFunctions3(Coef, Coef, Coef)
```

———-

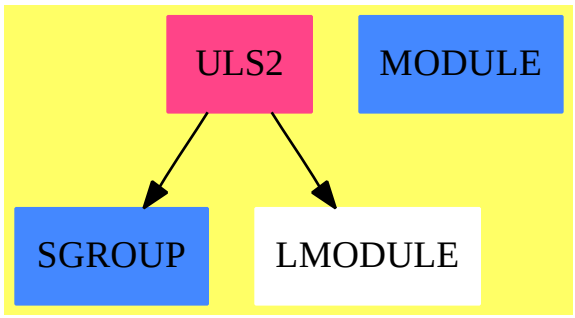
— **UFPS1.dotabb** —

```
"UFPS1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UFPS1"]
"UFPS" [color="#88FF44",href="bookvol10.3.pdf#nameddest=UFPS"]
"UFPS1" -> "UFPS"
```

———-

22.3 package ULS2 UnivariateLaurentSeriesFunctions2

22.3.1 UnivariateLaurentSeriesFunctions2 (ULS2)



Exports:
map

— **package ULS2 UnivariateLaurentSeriesFunctions2** —

```
)abbrev package ULS2 UnivariateLaurentSeriesFunctions2
++ Author: Clifton J. Williamson
```

```
++ Date Created: 5 March 1990
++ Date Last Updated: 5 March 1990
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: Laurent series, map
++ Examples:
++ References:
++ Description:
++ Mapping package for univariate Laurent series
++ This package allows one to apply a function to the coefficients of
++ a univariate Laurent series.

UnivariateLaurentSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):_
 Exports == Implementation where
  Coef1 : Ring
  Coef2 : Ring
  var1: Symbol
  var2: Symbol
  cen1: Coef1
  cen2: Coef2
  ULS1  ==> UnivariateLaurentSeries(Coef1, var1, cen1)
  ULS2  ==> UnivariateLaurentSeries(Coef2, var2, cen2)
  UTS1  ==> UnivariateTaylorSeries(Coef1, var1, cen1)
  UTS2  ==> UnivariateTaylorSeries(Coef2, var2, cen2)
  UTSF2 ==> UnivariateTaylorSeriesFunctions2(Coef1, Coef2, UTS1, UTS2)

  Exports ==> with
    map: (Coef1 -> Coef2,ULS1) -> ULS2
      ++ \spad{map(f,g(x))} applies the map f to the coefficients of the Laurent
      ++ series \spad{g(x)}.

  Implementation ==> add

    map(f,ups) == laurent(degree ups, map(f, taylorRep ups)$UTSF2)
```

———-

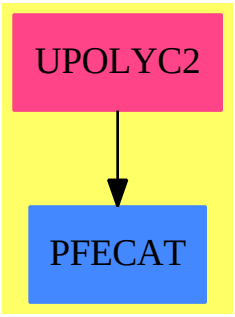
— ULS2.dotabb —

```
"ULS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ULS2"]
"MODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"ULS2" -> "LMODULE"
"ULS2" -> "SGROUP"
```

———-

22.4 package UPOLYC2 UnivariatePolynomialCategoryFunctions2

22.4.1 UnivariatePolynomialCategoryFunctions2 (UPOLYC2)



Exports:
map

— package UPOLYC2 UnivariatePolynomialCategoryFunctions2 —

```
)abbrev package UPOLYC2 UnivariatePolynomialCategoryFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ Mapping from polynomials over R to polynomials over S
++ given a map from R to S assumed to send zero to zero.

UnivariatePolynomialCategoryFunctions2(R,PR,S,PS): Exports == Impl where
  R, S: Ring
  PR  : UnivariatePolynomialCategory R
  PS  : UnivariatePolynomialCategory S

  Exports ==> with
    map: (R -> S, PR) -> PS
     ++ map(f, p) takes a function f from R to S,
     ++ and applies it to each (non-zero) coefficient of a polynomial p
     ++ over R, getting a new polynomial over S.
     ++ Note that since the map is not applied to zero elements, it may map
```

```
    ++ zero to zero.

  Impl ==> add
    map(f, p) ==
      ans:PS := 0
      while p ^= 0 repeat
        ans := ans + monomial(f leadingCoefficient p, degree p)
        p   := reductum p
      ans
```

———-

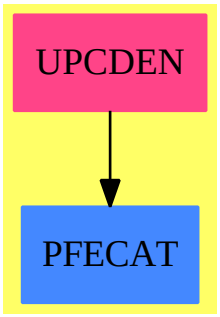
— **UPOLYC2.dotabb** —

```
"UPOLYC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPOLYC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPOLYC2" -> "PFECAT"
```

———-

22.5 package UPCDEN UnivariatePolynomialCommonDenominator

22.5.1 UnivariatePolynomialCommonDenominator (UPCDEN)



Exports:
clearDenominator commonDenominator splitDenominator

— **package UPCDEN UnivariatePolynomialCommonDenominator** —

```
)abbrev package UPCDEN UnivariatePolynomialCommonDenominator
```

```
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Feb 1990
++ Keywords: gcd, quotient, common, denominator, polynomial.
++ Description:
++ UnivariatePolynomialCommonDenominator provides
++ functions to compute the common denominator of the coefficients of
++ univariate polynomials over the quotient field of a gcd domain.

UnivariatePolynomialCommonDenominator(R, Q, UP): Exports == Impl where
  R : IntegralDomain
  Q : QuotientFieldCategory R
  UP: UnivariatePolynomialCategory Q

  Exports ==> with
    commonDenominator: UP -> R
      ++ commonDenominator(q) returns a common denominator d for
      ++ the coefficients of q.
    clearDenominator : UP -> UP
      ++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
      ++ a common denominator for the coefficients of q.
    splitDenominator : UP -> Record(num: UP, den: R)
      ++ splitDenominator(q) returns \spad{[p, d]} such that \spad{q = p/d} and d
      ++ is a common denominator for the coefficients of q.

  Impl ==> add
    import CommonDenominator(R, Q, List Q)

    commonDenominator p == commonDenominator coefficients p

    clearDenominator p ==
      d := commonDenominator p
      map(x +-> numer(d*x)::Q, p)

    splitDenominator p ==
      d := commonDenominator p
      [map(x +-> numer(d*x)::Q, p), d]
```

———-

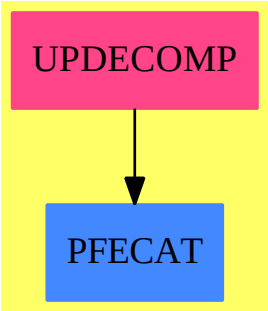
— UPCDEN.dotabb —

```
"UPCDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPCDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPCDEN" -> "PFECAT"
```

———-

22.6 package UPDECOMP UnivariatePolynomialDecompositionPackage

22.6.1 UnivariatePolynomialDecompositionPackage (UPDECOMP)



Exports:

leftFactorIfCan	monicCompleteDecompose	monicDecomposeIfCan
monicRightFactorIfCan	rightFactorIfCan	

— **package UPDECOMP UnivariatePolynomialDecompositionPackage** —

```
)abbrev package UPDECOMP UnivariatePolynomialDecompositionPackage
++ Author: Frederic Lehobey
++ Date Created: 17 June 1996
++ Date Last Updated: 4 June 1997
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keyword:
++ Exemples:
++ References:
++ [1] Peter Henrici, Automatic Computations with Power Series,
++ Journal of the Association for Computing Machinery, Volume 3, No. 1,
++ January 1956, 10-15
++ [2] Dexter Kozen and Susan Landau, Polynomial Decomposition
++ Algorithms, Journal of Symbolic Computation (1989) 7, 445-456
-- Decomposition would be speeded up (O(n log n) instead of O(n^2)) by
-- implementing the algorithm described in [3] based on [4] and [5].
++ [3] Joachim von zur Gathen, Functional Decomposition Polynomials:
++ the Tame Case, Journal of Symbolic Computation (1990) 9, 281-299
++ [4] R. P. Brent and H. T. Kung, Fast Algorithms for Manipulating
++ Formal Power Series, Journal of the Association for Computing
++ Machinery, Vol. 25, No. 4, October 1978, 581-595
++ [5] R. P. Brent, Multiple-Precision Zero-Finding Methods and the
```

```
++ Complexity of Elementary Function Evaluation, Analytic
++ Computational Complexity, J. F. Traub, Ed., Academic Press,
++ New York 1975, 151-176
++ Description:
++ UnivariatePolynomialDecompositionPackage implements
++ functional decomposition of univariate polynomial with coefficients
++ in an \spad{IntegralDomain} of \spad{CharacteristicZero}.

UnivariatePolynomialDecompositionPackage(R,UP): Exports == Implementation where
  R : Join(IntegralDomain,CharacteristicZero)
  UP : UnivariatePolynomialCategory(R)
  N ==> NonNegativeInteger
  LR ==> Record(left: UP, right: UP)
  QR ==> Record(quotient: UP, remainder: UP)


  Exports ==> with

    monicRightFactorIfCan: (UP,N) -> Union(UP,"failed")
      ++ monicRightFactorIfCan(f,d) returns a candidate to be the
      ++ monic right factor (h in f = g o h) of degree d of a
      ++ functional decomposition of the polynomial f or
      ++ \spad{"failed"} if no such candidate.
    rightFactorIfCan: (UP,N,R) -> Union(UP,"failed")
      ++ rightFactorIfCan(f,d,c) returns a candidate to be the
      ++ right factor (h in f = g o h) of degree d with leading
      ++ coefficient c of a functional decomposition of the
      ++ polynomial f or \spad{"failed"} if no such candidate.
    leftFactorIfCan: (UP,UP) -> Union(UP,"failed")
      ++ leftFactorIfCan(f,h) returns the left factor (g in f = g o h)
      ++ of the functional decomposition of the polynomial f with
      ++ given h or \spad{"failed"} if g does not exist.
    monicDecomposeIfCan: UP -> Union(LR,"failed")
      ++ monicDecomposeIfCan(f) returns a functional decomposition
      ++ of the monic polynomial f of "failed" if it has not found any.
    monicCompleteDecompose: UP -> List UP
      ++ monicCompleteDecompose(f) returns a list of factors of f for
      ++ the functional decomposition ([ f1, ..., fn ] means
      ++ f = f1 o ... o fn).

  Implementation ==> add

    rightFactorIfCan(p,dq,lcq) ==
      dp := degree p
      zero? lcq =>
       error "rightFactorIfCan: leading coefficient may not be zero"
      (zero? dp) or (zero? dq) => "failed"
      nc := dp exquo dq
      nc case "failed" => "failed"
      n := nc::N
```

```
    s := subtractIfCan(dq,1)::N
    lcp := leadingCoefficient p
    q: UP := monomial(lcq,dq)
    k: N
    for k in 1..s repeat
      c: R := 0
      i: N
      for i in 0..subtractIfCan(k,1)::N repeat
       c := c+(k::R-(n::R+1)*(i::R))*
        coefficient(q,subtractIfCan(dq,i)::N)*
         coefficient(p,subtractIfCan(dp+i,k)::N)
      cquo := c exquo ((k*n)::R*lcp)
      cquo case "failed" => return "failed"
      q := q+monomial(cquo::R,subtractIfCan(dq,k)::N)
    q

  monicRightFactorIfCan(p,dq) == rightFactorIfCan(p,dq,1$R)

  import UnivariatePolynomialDivisionPackage(R,UP)

  leftFactorIfCan(f,h) ==
    g: UP := 0
    zero? degree h => "failed"
    for i in 0.. while not zero? f repeat
      qrf := divideIfCan(f,h)
      qrf case "failed" => return "failed"
      qr := qrf :: QR
      r := qr.remainder
      not ground? r => return "failed"
      g := g+monomial(ground(r),i)
      f := qr.quotient
    g

  monicDecomposeIfCan f ==
    df := degree f
    zero? df => "failed"
    for dh in 2..subtractIfCan(df,1)::N | zero?(df rem dh) repeat
      h := monicRightFactorIfCan(f,dh)
      h case UP =>
       g := leftFactorIfCan(f,h::UP)
       g case UP => return [g::UP,h::UP]
    "failed"

  monicCompleteDecompose f ==
    cf := monicDecomposeIfCan f
    cf case "failed" => [ f ]
    lr := cf :: LR
    append(monicCompleteDecompose lr.left,[lr.right])
```

———-

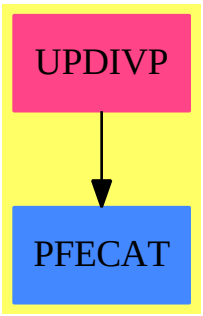
— **UPDECOMP.dotabb** —

```
"UPDECOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPDECOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPDECOMP" -> "PFECAT"
```

———-

22.7 package UPDIVP UnivariatePolynomialDivisionPackage

22.7.1 UnivariatePolynomialDivisionPackage (UPDIVP)



Exports:
divideIfCan

— **package UPDIVP UnivariatePolynomialDivisionPackage** —

```
)abbrev package UPDIVP UnivariatePolynomialDivisionPackage
++ Author: Frederic Lehobey
++ Date Created: 3 June 1997
++ Date Last Updated: 3 June 1997
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keyword:
++ Exemples:
++ References:
++ Description:
```

```
++ UnivariatePolynomialDivisionPackage provides a
++ division for non monic univarite polynomials with coefficients in
++ an \spad{IntegralDomain}.

UnivariatePolynomialDivisionPackage(R,UP): Exports == Implementation where
  R : IntegralDomain
  UP : UnivariatePolynomialCategory(R)
  N ==> NonNegativeInteger
  QR ==> Record(quotient: UP, remainder: UP)

  Exports ==> with

    divideIfCan: (UP,UP) -> Union(QR,"failed")
      ++ divideIfCan(f,g) returns quotient and remainder of the
      ++ division of f by g or "failed" if it has not succeeded.

  Implementation ==> add

    divideIfCan(p1:UP,p2:UP):Union(QR,"failed") ==
      zero? p2 => error "divideIfCan: division by zero"
--      one? (lc := leadingCoefficient p2) => monicDivide(p1,p2)
      ((lc := leadingCoefficient p2) = 1) => monicDivide(p1,p2)
      q: UP := 0
      while not ((e := subtractIfCan(degree(p1),degree(p2))) case "failed")
       repeat
        c := leadingCoefficient(p1) exquo lc
        c case "failed" => return "failed"
        ee := e::N
        q := q+monomial(c::R,ee)
        p1 := p1-c*mapExponents(x +-> x+ee, p2)
      [q,p1]
```

———-

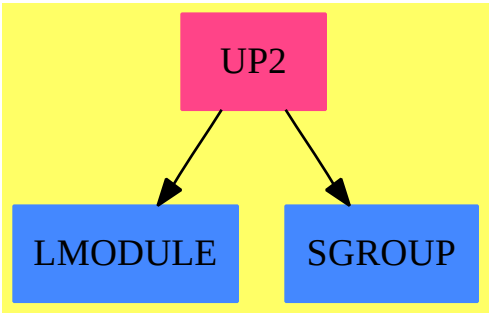
— UPDIVP.dotabb —

```
"UPDIVP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPDIVP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPDIVP" -> "PFECAT"
```

———-

22.8 package UP2 UnivariatePolynomialFunctions2

22.8.1 UnivariatePolynomialFunctions2 (UP2)



Exports:
map

— package UP2 UnivariatePolynomialFunctions2 —

```
)abbrev package UP2 UnivariatePolynomialFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package lifts a mapping from coefficient rings R to S to
++ a mapping from \spadtype{UnivariatePolynomial}(x,R) to
++ \spadtype{UnivariatePolynomial}(y,S). Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

UnivariatePolynomialFunctions2(x:Symbol, R:Ring, y:Symbol, S:Ring): with
  map: (R -> S, UnivariatePolynomial(x,R)) -> UnivariatePolynomial(y,S)
    ++ map(func, poly) creates a new polynomial by applying func to
    ++ every non-zero coefficient of the polynomial poly.
 == add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
               UnivariatePolynomial(x, R), S, UnivariatePolynomial(y, S))
```

———-

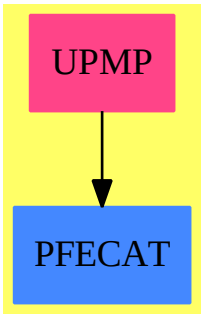
— **UP2.dotabb** —

```
"UP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"UP2" -> "LMODULE"
"UP2" -> "SGROUP"
```

———-

22.9 package UPMP UnivariatePolynomialMultiplicationPackage

22.9.1 UnivariatePolynomialMultiplicationPackage (UPMP)



Exports:
karatsuba karatsubaOnce noKaratsuba

— **package UPMP UnivariatePolynomialMultiplicationPackage** —

```
)abbrev package UPMP UnivariatePolynomialMultiplicationPackage
++ Author: Marc Moreno Maza
++ Date Created: 14.08.2000
++ Description:
++ This package implements Karatsuba's trick for multiplying
++ (large) univariate polynomials. It could be improved with
++ a version doing the work on place and also with a special
++ case for squares. We've done this in Basicmath, but we
++ believe that this out of the scope of AXIOM.

UnivariatePolynomialMultiplicationPackage(R: Ring, U: UnivariatePolynomialCategory(R)): C == T
  where
```

```
HL ==> Record(quotient:U,remainder:U)
C == with
  noKaratsuba: (U, U) -> U
    ++ \spad{noKaratsuba(a,b)} returns \spad{a*b} without
    ++ using Karatsuba's trick at all.
  karatsubaOnce: (U, U) -> U
    ++ \spad{karatsuba(a,b)} returns \spad{a*b} by applying
    ++ Karatsuba's trick once. The other multiplications
    ++ are performed by calling \spad{*} from \spad{U}.
  karatsuba: (U, U, NonNegativeInteger, NonNegativeInteger) -> U;
    ++ \spad{karatsuba(a,b,l,k)} returns \spad{a*b} by applying
    ++ Karatsuba's trick provided that both \spad{a} and \spad{b}
    ++ have at least \spad{l} terms and \spad{k > 0} holds
    ++ and by calling \spad{noKaratsuba} otherwise. The other
    ++ multiplications are performed by recursive calls with
    ++ the same third argument and \spad{k-1} as fourth argument.

T == add
  noKaratsuba(a,b) ==
    zero? a => a
    zero? b => b
    zero?(degree(a)) => leadingCoefficient(a) * b
    zero?(degree(b)) => a * leadingCoefficient(b)
    lu: List(U) := reverse monomials(a)
    res: U := 0;
    for u in lu repeat
      res := pomopo!(res, leadingCoefficient(u), degree(u), b)
    res
  karatsubaOnce(a:U,b:U): U ==
    da := minimumDegree(a)
    db := minimumDegree(b)
    if not zero? da then a := shiftRight(a,da)
    if not zero? db then b := shiftRight(b,db)
    d := da + db
    n: NonNegativeInteger := min(degree(a),degree(b)) quo 2
    rec: HL := karatsubaDivide(a, n)
    ha := rec.quotient
    la := rec.remainder
    rec := karatsubaDivide(b, n)
    hb := rec.quotient
    lb := rec.remainder
    w: U := (ha - la) * (lb - hb)
    u: U := la * lb
    v: U := ha * hb
    w := w + (u + v)
    w := shiftLeft(w,n) + u
    zero? d => shiftLeft(v,2*n) + w
    shiftLeft(v,2*n + d) + shiftLeft(w,d)
  karatsuba(a:U,b:U,l:NonNegativeInteger,k:NonNegativeInteger): U ==
    zero? k => noKaratsuba(a,b)
```

```
        degree(a) < l => noKaratsuba(a,b)
        degree(b) < l => noKaratsuba(a,b)
        numberOfMonomials(a) < l => noKaratsuba(a,b)
        numberOfMonomials(b) < l => noKaratsuba(a,b)
        da := minimumDegree(a)
        db := minimumDegree(b)
        if not zero? da then a := shiftRight(a,da)
        if not zero? db then b := shiftRight(b,db)
        d := da + db
        n: NonNegativeInteger := min(degree(a),degree(b)) quo 2
        k := subtractIfCan(k,1)::NonNegativeInteger
        rec: HL := karatsubaDivide(a, n)
        ha := rec.quotient
        la := rec.remainder
        rec := karatsubaDivide(b, n)
        hb := rec.quotient
        lb := rec.remainder
        w: U := karatsuba(ha - la, lb - hb, l, k)
        u: U := karatsuba(la, lb, l, k)
        v: U := karatsuba(ha, hb, l, k)
        w := w + (u + v)
        w := shiftLeft(w,n) + u
        zero? d => shiftLeft(v,2*n) + w
        shiftLeft(v,2*n + d) + shiftLeft(w,d)
```

———-

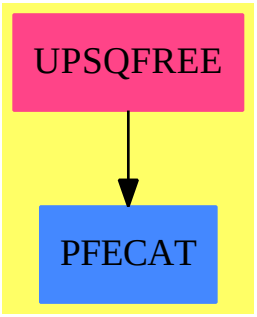
— UPMP.dotabb —

```
"UPMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPMP" -> "PFECAT"
```

———-

22.10 package UPSQFREE UnivariatePolynomialSquareFree

22.10.1 UnivariatePolynomialSquareFree (UPSQFREE)



Exports:
squareFree squareFreePart BumInSepFFE

— package UPSQFREE UnivariatePolynomialSquareFree —

```
)abbrev package UPSQFREE UnivariatePolynomialSquareFree
++ Author: Dave Barton, Barry Trager
++ Date Created:
++ Date Last Updated:
++ Basic Functions: squareFree, squareFreePart
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides for square-free decomposition of
++ univariate polynomials over arbitrary rings, i.e.
++ a partial factorization such that each factor is a product
++ of irreducibles with multiplicity one and the factors are
++ pairwise relatively prime. If the ring
++ has characteristic zero, the result is guaranteed to satisfy
++ this condition. If the ring is an infinite ring of
++ finite characteristic, then it may not be possible to decide when
++ polynomials contain factors which are pth powers. In this
++ case, the flag associated with that polynomial is set to "nil"
++ (meaning that that polynomials are not guaranteed to be square-free).

UnivariatePolynomialSquareFree(RC:IntegralDomain,P):C == T
  where
```

```
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF     ==> Record(flg:fUnion, fctr:P, xpnt:Integer)
P:Join(UnivariatePolynomialCategory(RC),IntegralDomain) with
  gcd: (%,%) -> %
    ++ gcd(p,q) computes the greatest-common-divisor of p and q.

C == with
  squareFree: P -> Factored(P)
    ++ squareFree(p) computes the square-free factorization of the
    ++ univariate polynomial p. Each factor has no repeated roots, and the
    ++ factors are pairwise relatively prime.
  squareFreePart: P -> P
    ++ squareFreePart(p) returns a polynomial which has the same
    ++ irreducible factors as the univariate polynomial p, but each
    ++ factor has multiplicity one.
  BumInSepFFE: FF -> FF
    ++ BumInSepFFE(f) is a local function, exported only because
    ++ it has multiple conditional definitions.

T == add

  if RC has CharacteristicZero then
    squareFreePart(p:P) == (p exquo gcd(p, differentiate p))::P
  else
    squareFreePart(p:P) ==
      unit(s := squareFree(p)$%) * */[f.factor for f in factors s]

  if RC has FiniteFieldCategory then
    BumInSepFFE(ffe:FF) ==
       ["sqfr", map(charthRoot,ffe.fctr), characteristic$P*ffe.xpnt]
  else if RC has CharacteristicNonZero then
     BumInSepFFE(ffe:FF) ==
        np := multiplyExponents(ffe.fctr,characteristic$P:NonNegativeInteger)
        (nthrp := charthRoot(np)) case "failed" =>
           ["nil", np, ffe.xpnt]
        ["sqfr", nthrp, characteristic$P*ffe.xpnt]

  else
    BumInSepFFE(ffe:FF) ==
      ["nil",
       multiplyExponents(ffe.fctr,characteristic$P:NonNegativeInteger),
        ffe.xpnt]


  if RC has CharacteristicZero then
    squareFree(p:P) ==             --Yun's algorithm - see SYMSAC '76, p.27
       --Note ci primitive is, so GCD's don't need to %do contents.
       --Change gcd to return cofctrs also?
       ci:=p; di:=differentiate(p); pi:=gcd(ci,di)
       degree(pi)=0 =>
```

```
            (u,c,a):=unitNormal(p)
            makeFR(u,[["sqfr",c,1]])
          i:NonNegativeInteger:=0; lffe:List FF:=[]
          lcp := leadingCoefficient p
          while degree(ci)^=0 repeat
             ci:=(ci exquo pi)::P
             di:=(di exquo pi)::P - differentiate(ci)
             pi:=gcd(ci,di)
             i:=i+1
             degree(pi) > 0 =>
                lcp:=(lcp exquo (leadingCoefficient(pi)**i))::RC
                lffe:=[["sqfr",pi,i],:lffe]
          makeFR(lcp::P,lffe)

      else
        squareFree(p:P) ==           --Musser's algorithm - see SYMSAC '76, p.27
            --p MUST BE PRIMITIVE, Any characteristic.
            --Note ci primitive, so GCD's don't need to %do contents.
            --Change gcd to return cofctrs also?
          ci := gcd(p,differentiate(p))
          degree(ci)=0 =>
            (u,c,a):=unitNormal(p)
            makeFR(u,[["sqfr",c,1]])
          di := (p exquo ci)::P
          i:NonNegativeInteger:=0; lffe:List FF:=[]
          dunit : P := 1
          while degree(di)^=0 repeat
             diprev := di
             di := gcd(ci,di)
             ci:=(ci exquo di)::P
             i:=i+1
             degree(diprev) = degree(di) =>
                lc := (leadingCoefficient(diprev) exquo leadingCoefficient(di))::RC
                dunit := lc**i * dunit
             pi:=(diprev exquo di)::P
             lffe:=[["sqfr",pi,i],:lffe]
          dunit := dunit * di ** (i+1)
          degree(ci)=0 => makeFR(dunit*ci,lffe)
          redSqfr:=squareFree(divideExponents(ci,characteristic$P)::P)
          lsnil:= [BumInSepFFE(ffe) for ffe in factorList redSqfr]
          lffe:=append(lsnil,lffe)
          makeFR(dunit*(unit redSqfr),lffe)
```

———-

— UPSQFREE.dotabb —

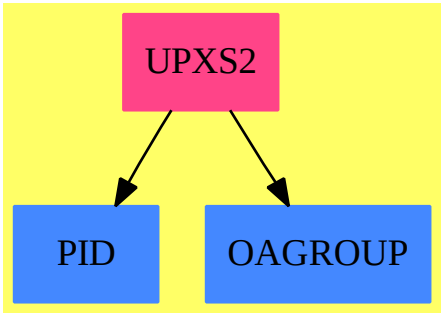
```
"UPSQFREE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPSQFREE"]
```

```
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPSQFREE" -> "PFECAT"
```

———-

22.11 package UPXS2 UnivariatePuiseuxSeriesFunctions2

22.11.1 UnivariatePuiseuxSeriesFunctions2 (UPXS2)



Exports:
map

— package UPXS2 UnivariatePuiseuxSeriesFunctions2 —

```
)abbrev package UPXS2 UnivariatePuiseuxSeriesFunctions2
++ Author: Scott C. Morrison
++ Date Created: 5 April 1991
++ Date Last Updated: 5 April 1991
++ Keywords: Puiseux series, map
++ Examples:
++ References:
++ Description:
++ Mapping package for univariate Puiseux series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Puiseux series.

UnivariatePuiseuxSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):_
 Exports == Implementation where
  Coef1 : Ring
  Coef2 : Ring
  var1: Symbol
  var2: Symbol
  cen1: Coef1
  cen2: Coef2
```

```
  UPS1  ==> UnivariatePuiseuxSeries(Coef1, var1, cen1)
  UPS2  ==> UnivariatePuiseuxSeries(Coef2, var2, cen2)
  ULSP2 ==> UnivariateLaurentSeriesFunctions2(Coef1, Coef2, var1, var2, cen1, cen2)

  Exports ==> with
    map: (Coef1 -> Coef2,UPS1) -> UPS2
      ++ \spad{map(f,g(x))} applies the map f to the coefficients of the
      ++ Puiseux series \spad{g(x)}.

  Implementation ==> add

    map(f,ups) == puiseux(rationalPower ups, map(f, laurentRep ups)$ULSP2)
```

———-

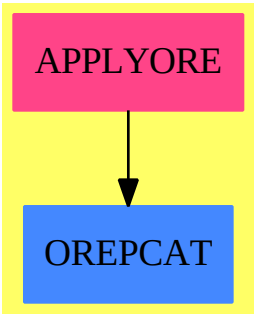
— **UPXS2.dotabb** —

```
"UPXS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPXS2"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"UPXS2" -> "PID"
"UPXS2" -> "OAGROUP"
```

———-

22.12 package OREPCTO UnivariateSkewPolynomialCategoryOps

22.12.1 UnivariateSkewPolynomialCategoryOps (OREPCTO)



Exports:

apply	leftDivide	monicLeftDivide
monicRightDivide	rightDivide	times

— package OREPCTO UnivariateSkewPolynomialCategoryOps —

```
)abbrev package OREPCTO UnivariateSkewPolynomialCategoryOps
++ Author: Manuel Bronstein
++ Date Created: 1 February 1994
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{UnivariateSkewPolynomialCategoryOps} provides products and
++ divisions of univariate skew polynomials.
-- Putting those operations here rather than defaults in OREPCAT allows
-- OREPCAT to be defined independently of sigma and delta.
-- MB 2/94

UnivariateSkewPolynomialCategoryOps(R, C): Exports == Implementation where
    R: Ring
    C: UnivariateSkewPolynomialCategory R

    N   ==> NonNegativeInteger
    MOR ==> Automorphism R
    QUOREM ==> Record(quotient: C, remainder: C)

    Exports ==> with
        times: (C, C, MOR, R -> R) -> C
           ++ times(p, q, sigma, delta) returns \spad{p * q}.
           ++ \spad{\sigma} and \spad{\delta} are the maps to use.
        apply: (C, R, R, MOR, R -> R) -> R
          ++ apply(p, c, m, sigma, delta) returns \spad{p(m)} where the action
          ++ is given by \spad{x m = c sigma(m) + delta(m)}.
        if R has IntegralDomain then
            monicLeftDivide: (C, C, MOR) -> QUOREM
                ++ monicLeftDivide(a, b, sigma) returns the pair \spad{[q,r]}
                ++ such that \spad{a = b*q + r} and the degree of \spad{r} is
                ++ less than the degree of \spad{b}.
                ++ \spad{b} must be monic.
                ++ This process is called ``left division''.
                ++ \spad{\sigma} is the morphism to use.
            monicRightDivide: (C, C, MOR) -> QUOREM
                ++ monicRightDivide(a, b, sigma) returns the pair \spad{[q,r]}
                ++ such that \spad{a = q*b + r} and the degree of \spad{r} is
                ++ less than the degree of \spad{b}.
                ++ \spad{b} must be monic.
                ++ This process is called ``right division''.
                ++ \spad{\sigma} is the morphism to use.
        if R has Field then
            leftDivide: (C, C, MOR) -> QUOREM
                ++ leftDivide(a, b, sigma) returns the pair \spad{[q,r]} such
                ++ that \spad{a = b*q + r} and the degree of \spad{r} is
                ++ less than the degree of \spad{b}.
                ++ This process is called ``left division''.
```

```
            ++ \spad{\sigma} is the morphism to use.
        rightDivide: (C, C, MOR) -> QUOREM
            ++ rightDivide(a, b, sigma) returns the pair \spad{[q,r]} such
            ++ that \spad{a = q*b + r} and the degree of \spad{r} is
            ++ less than the degree of \spad{b}.
            ++ This process is called ``right division''.
            ++ \spad{\sigma} is the morphism to use.

Implementation ==> add
    termPoly:         (R, N, C, MOR, R -> R) -> C
    localLeftDivide : (C, C, MOR, R) -> QUOREM
    localRightDivide: (C, C, MOR, R) -> QUOREM

    times(x, y, sigma, delta) ==
      zero? y => 0
      z:C := 0
      while x ^= 0 repeat
        z := z + termPoly(leadingCoefficient x, degree x, y, sigma, delta)
        x := reductum x
      z

    termPoly(a, n, y, sigma, delta) ==
      zero? y => 0
      (u := subtractIfCan(n, 1)) case "failed" => a * y
      n1 := u::N
      z:C := 0
      while y ^= 0 repeat
        m := degree y
        b := leadingCoefficient y
        z := z + termPoly(a, n1, monomial(sigma b, m + 1), sigma, delta)
               + termPoly(a, n1, monomial(delta b, m), sigma, delta)
        y := reductum y
      z

    apply(p, c, x, sigma, delta) ==
      w:R  := 0
      xn:R := x
      for i in 0..degree p repeat
        w  := w + coefficient(p, i) * xn
        xn := c * sigma xn + delta xn
      w

    -- localLeftDivide(a, b) returns [q, r] such that a = q b + r
    -- b1 is the inverse of the leadingCoefficient of b
    localLeftDivide(a, b, sigma, b1) ==
        zero? b => error "leftDivide: division by 0"
        zero? a or
         (n := subtractIfCan(degree(a),(m := degree b))) case "failed" =>
                [0,a]
        q  := monomial((sigma**(-m))(b1 * leadingCoefficient a), n::N)
```

```
            qr := localLeftDivide(a - b * q, b, sigma, b1)
            [q + qr.quotient, qr.remainder]

        -- localRightDivide(a, b) returns [q, r] such that a = q b + r
        -- b1 is the inverse of the leadingCoefficient of b
        localRightDivide(a, b, sigma, b1) ==
            zero? b => error "rightDivide: division by 0"
            zero? a or
              (n := subtractIfCan(degree(a),(m := degree b))) case "failed" =>
                    [0,a]
            q := monomial(leadingCoefficient(a) * (sigma**n) b1, n::N)
            qr := localRightDivide(a - q * b, b, sigma, b1)
            [q + qr.quotient, qr.remainder]

        if R has IntegralDomain then
            monicLeftDivide(a, b, sigma) ==
                unit?(u := leadingCoefficient b) =>
                    localLeftDivide(a, b, sigma, recip(u)::R)
                error "monicLeftDivide: divisor is not monic"

            monicRightDivide(a, b, sigma) ==
                unit?(u := leadingCoefficient b) =>
                    localRightDivide(a, b, sigma, recip(u)::R)
                error "monicRightDivide: divisor is not monic"

        if R has Field then
            leftDivide(a, b, sigma) ==
                localLeftDivide(a, b, sigma, inv leadingCoefficient b)

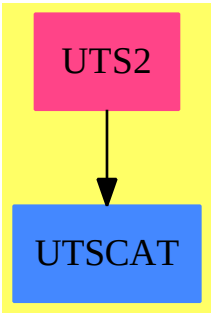
            rightDivide(a, b, sigma) ==
                localRightDivide(a, b, sigma, inv leadingCoefficient b)
```

— OREPCTO.dotabb —

```
"OREPCTO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OREPCTO"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"OREPCTO" -> "OREPCAT"
```

22.13 package UTS2 UnivariateTaylorSeriesFunctions2

22.13.1 UnivariateTaylorSeriesFunctions2 (UTS2)



Exports:
map

— package UTS2 UnivariateTaylorSeriesFunctions2 —

```
)abbrev package UTS2 UnivariateTaylorSeriesFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 9 February 1990
++ Date Last Updated: 9 February 1990
++ Basic Operations:
++ Related Domains: UnivariateTaylorSeries(Coef1,var,cen)
++ Also See:
++ AMS Classifications:
++ Keywords: Taylor series, map
++ Examples:
++ References:
++ Description:
++ Mapping package for univariate Taylor series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Taylor series.

UnivariateTaylorSeriesFunctions2(Coef1,Coef2,UTS1,UTS2):_
 Exports == Implementation where
  Coef1 : Ring
  Coef2 : Ring
  UTS1  : UnivariateTaylorSeriesCategory Coef1
  UTS2  : UnivariateTaylorSeriesCategory Coef2
  ST2 ==> StreamFunctions2(Coef1,Coef2)

  Exports ==> with
    map: (Coef1 -> Coef2,UTS1) -> UTS2
      ++\spad{map(f,g(x))} applies the map f to the coefficients of
```

```
      ++ the Taylor series \spad{g(x)}.

  Implementation ==> add

    map(f,uts) == series map(f,coefficients uts)$ST2
```

———-

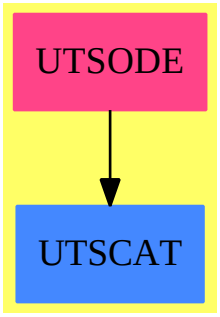
— **UTS2.dotabb** —

```
"UTS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTS2"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTS2" -> "UTSCAT"
```

———-

22.14 package UTSODE UnivariateTaylorSeriesODESolver

22.14.1 UnivariateTaylorSeriesODESolver (UTSODE)



Exports:

fixedPointExquo	mpsode	ode	ode1	ode2
stFunc1	stFunc2	stFuncN		

— **package UTSODE UnivariateTaylorSeriesODESolver** —

```
)abbrev package UTSODE UnivariateTaylorSeriesODESolver
++ Author: Stephen Watt (revised by Clifton J. Williamson)
++ Date Created: February 1988
++ Date Last Updated: 30 September 1993
++ Keywords: differential equation, ODE, Taylor series
```

```
++ Examples:
++ References:
++ Description:
++ Taylor series solutions of explicit ODE's.
++ This package provides Taylor series solutions to regular
++ linear or non-linear ordinary differential equations of
++ arbitrary order.

UnivariateTaylorSeriesODESolver(Coef,UTS):_
 Exports == Implementation where
  Coef  : Algebra Fraction Integer
  UTS   : UnivariateTaylorSeriesCategory Coef
  L   ==> List
  L2  ==> ListFunctions2
  FN  ==> (L UTS) -> UTS
  ST  ==> Stream Coef
  YS  ==> Y$ParadoxicalCombinatorsForStreams(Coef)
  STT ==> StreamTaylorSeriesOperations(Coef)

  Exports ==> with
    stFunc1: (UTS -> UTS) -> (ST -> ST)
      ++ stFunc1(f) is a local function exported due to compiler problem.
      ++ This function is of no interest to the top-level user.
    stFunc2: ((UTS,UTS) -> UTS) -> ((ST,ST) -> ST)
      ++ stFunc2(f) is a local function exported due to compiler problem.
      ++ This function is of no interest to the top-level user.
    stFuncN: FN -> ((L ST) -> ST)
      ++ stFuncN(f) is a local function xported due to compiler problem.
      ++ This function is of no interest to the top-level user.
    fixedPointExquo: (UTS,UTS) -> UTS
      ++ fixedPointExquo(f,g) computes the exact quotient of \spad{f} and
      ++ \spad{g} using a fixed point computation.
    ode1: ((UTS -> UTS),Coef) -> UTS
      ++ ode1(f,c) is the solution to \spad{y' = f(y)}
      ++ such that \spad{y(a) = c}.
    ode2: ((UTS, UTS) -> UTS,Coef,Coef) -> UTS
      ++ ode2(f,c0,c1) is the solution to \spad{y'' = f(y,y')} such that
      ++ \spad{y(a) = c0} and \spad{y'(a) = c1}.
    ode: (FN,List Coef) -> UTS
      ++ ode(f,cl) is the solution to \spad{y<n>=f(y,y',..,y<n-1>)} such that
      ++ \spad{y<i>(a) = cl.i} for i in 1..n.
    mpsode:(L Coef,L FN) -> L UTS
      ++ mpsode(r,f) solves the system of differential equations
      ++ \spad{dy[i]/dx =f[i] [x,y[1],y[2],...,y[n]]},
      ++ \spad{y[i](a) = r[i]} for i in 1..n.

  Implementation ==> add

    stFunc1 f == s +-> coefficients f series(s)
    stFunc2 f == (s1,s2) +-> coefficients f(series(s1),series(s2))
```

```
    stFuncN f == ls +-> coefficients f map(series,ls)$ListFunctions2(ST,UTS)

    import StreamTaylorSeriesOperations(Coef)
    divloopre:(Coef,ST,Coef,ST,ST) -> ST
    divloopre(hx,tx,hy,ty,c) == delay(concat(hx*hy,hy*(tx-(ty*c))))
    divloop: (Coef,ST,Coef,ST) -> ST
    divloop(hx,tx,hy,ty) == YS(s +-> divloopre(hx,tx,hy,ty,s))

    sdiv:(ST,ST) -> ST
    sdiv(x,y) == delay
      empty? x => empty()
      empty? y => error "stream division by zero"
      hx := frst x; tx := rst x
      hy := frst y; ty := rst y
      zero? hy =>
        zero? hx => sdiv(tx,ty)
        error "stream division by zero"
      rhy := recip hy
      rhy case "failed" => error "stream division:no reciprocal"
      divloop(hx,tx,rhy::Coef,ty)

    fixedPointExquo(f,g) == series sdiv(coefficients f,coefficients g)

-- first order

    ode1re: (ST -> ST,Coef,ST) -> ST
    ode1re(f,c,y) == lazyIntegrate(c,f y)$STT

    iOde1: ((ST -> ST),Coef) -> ST
    iOde1(f,c) == YS(s +-> ode1re(f,c,s))

    ode1(f,c) == series iOde1(stFunc1 f,c)

-- second order

    ode2re: ((ST,ST)-> ST,Coef,Coef,ST) -> ST
    ode2re(f,c0,c1,y)==
      yi := lazyIntegrate(c1,f(y,deriv(y)$STT))$STT
      lazyIntegrate(c0,yi)$STT

    iOde2: ((ST,ST) -> ST,Coef,Coef) -> ST
    iOde2(f,c0,c1) == YS(s +-> ode2re(f,c0,c1,s))

    ode2(f,c0,c1) == series iOde2(stFunc2 f,c0,c1)

-- nth order

    odeNre: (List ST -> ST,List Coef,List ST) -> List ST
    odeNre(f,cl,yl) ==
      -- yl is [y, y', ..., y<n>]
```

```
      -- integrate [y',..,y<n>] to get [y,..,y<n-1>]
      yil := [lazyIntegrate(c,y)$STT for c in cl for y in rest yl]
      -- use y<n> = f(y,..,y<n-1>)
      concat(yil,[f yil])

    iOde: ((L ST) -> ST,List Coef) -> ST
    iOde(f,cl) == first YS(ls +-> odeNre(f,cl,ls),#cl + 1)

    ode(f,cl) == series iOde(stFuncN f,cl)

    simulre:(L Coef,L ((L ST) -> ST),L ST) -> L ST
    simulre(cst,lsf,c) ==
      [lazyIntegrate(csti,lsfi concat(monom(1,1)$STT,c))_
          for csti in cst for lsfi in lsf]
    iMpsode:(L Coef,L ((L ST) -> ST)) -> L ST
    iMpsode(cs,lsts) == YS(ls +-> simulre(cs,lsts,ls),# cs)
    mpsode(cs,lsts) ==
--       stSol := iMpsode(cs,map(stFuncN,lsts)$L2(FN,(L ST) -> ST))
      stSol := iMpsode(cs,[stFuncN(lst) for lst in lsts])
      map(series,stSol)$L2(ST,UTS)
```

———-

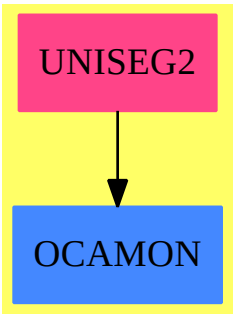
— **UTSODE.dotabb** —

```
"UTSODE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSODE"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTSODE" -> "UTSCAT"
```

———-

22.15 package UNISEG2 UniversalSegmentFunctions2

22.15.1 UniversalSegmentFunctions2 (UNISEG2)



Exports:
map

— package UNISEG2 UniversalSegmentFunctions2 —

```
)abbrev package UNISEG2 UniversalSegmentFunctions2
++ Author:
++ Date Created:
++ Date Last Updated: June 4, 1991
++ Basic Operations:
++ Related Domains: Segment, UniversalSegment
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ This package provides operations for mapping functions onto segments.

UniversalSegmentFunctions2(R:Type, S:Type): with
    map: (R -> S, UniversalSegment R) -> UniversalSegment S
        ++ map(f,seg) returns the new segment obtained by applying
        ++ f to the endpoints of seg.

    if R has OrderedRing then
      map: (R -> S, UniversalSegment R) -> Stream S
        ++ map(f,s) expands the segment s, applying \spad{f} to each value.


  == add
    map(f:R -> S, u:UniversalSegment R):UniversalSegment S ==
      s := f lo u
      hasHi u => segment(s, f hi u)
      segment s

    if R has OrderedRing then
      map(f:R -> S, u:UniversalSegment R): Stream S ==
        map(f, expand u)$StreamFunctions2(R, S)
```

———-

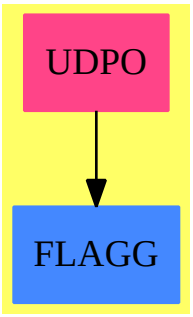
— UNISEG2.dotabb —

```
"UNISEG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UNISEG2"]
"OCAMON" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OCAMON"]
"UNISEG2" -> "OCAMON"
```

———-

22.16 package UDPO UserDefinedPartialOrdering

22.16.1 UserDefinedPartialOrdering (UDPO)



Exports:
setOrder userOrdered? getOrder largest less? more?

— package UDPO UserDefinedPartialOrdering —

```
)abbrev package UDPO UserDefinedPartialOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Provides functions to force a partial ordering on any set.

UserDefinedPartialOrdering(S:SetCategory): with
  setOrder    : List S -> Void
    ++ setOrder([a1,...,an]) defines a partial ordering on S given by:
    ++    (1)  \spad{a1 < a2 < ... < an}.
    ++    (2)  \spad{b < ai   for i = 1..n} and b not among the ai's.
    ++    (3)  undefined on \spad{(b, c)} if neither is among the ai's.
  setOrder    : (List S, List S) -> Void
    ++ setOrder([b1,...,bm], [a1,...,an]) defines a partial
    ++ ordering on S given by:
    ++    (1)  \spad{b1 < b2 < ... < bm < a1 < a2 < ... < an}.
    ++    (2)  \spad{bj < c < ai}  for c not among the ai's and bj's.
    ++    (3)  undefined on \spad{(c,d)} if neither is among the ai's,bj's.
  getOrder    : () -> Record(low: List S, high: List S)
    ++ getOrder() returns \spad{[[b1,...,bm], [a1,...,an]]} such that the
    ++ partial ordering on S was given by
```

```
    ++ \spad{setOrder([b1,...,bm],[a1,...,an])}.
  less?       : (S, S) -> Union(Boolean, "failed")
    ++ less?(a, b) compares \spad{a} and b in the partial ordering induced by
    ++ setOrder.
  less?       : (S, S, (S, S) -> Boolean) -> Boolean
    ++ less?(a, b, fn) compares \spad{a} and b in the partial ordering induced
    ++ by setOrder, and returns \spad{fn(a, b)} if \spad{a}
    ++ and b are not comparable
    ++ in that ordering.
  largest     : (List S, (S, S) -> Boolean) -> S
    ++ largest(l, fn) returns the largest element of l where the partial
    ++ ordering induced by setOrder is completed into a total one by fn.
  userOrdered?: () -> Boolean
    ++ userOrdered?() tests if the partial ordering induced by
    ++ setOrder is not empty.
  if S has OrderedSet then
    largest: List S -> S
      ++ largest l returns the largest element of l where the partial
      ++ ordering induced by setOrder is completed into a total one by
      ++ the ordering on S.
    more?  : (S, S) -> Boolean
      ++ more?(a, b) compares a and b in the partial ordering induced
      ++ by setOrder, and uses the ordering on S if a and b are not
      ++ comparable in the partial ordering.

== add
  llow :Reference List S := ref nil()
  lhigh:Reference List S := ref nil()

  userOrdered?() == not(empty? deref llow) or not(empty? deref lhigh)
  getOrder()     == [deref llow, deref lhigh]
  setOrder l     == setOrder(nil(), l)

  setOrder(l, h) ==
    setref(llow, removeDuplicates l)
    setref(lhigh, removeDuplicates h)
    void

  less?(a, b, f) ==
    (u := less?(a, b)) case "failed" => f(a, b)
    u::Boolean

  largest(x, f) ==
    empty? x => error "largest: empty list"
    empty? rest x => first x
    a := largest(rest x, f)
    less?(first x, a, f) => a
    first x

  less?(a, b) ==
```

```
    for x in deref llow repeat
      x = a => return(a ^= b)
      x = b => return false
    aa := bb := false$Boolean
    for x in deref lhigh repeat
      if x = a then
        bb => return false
        aa := true
      if x = b then
        aa => return(a ^= b)
        bb := true
    aa => false
    bb => true
    "failed"

  if S has OrderedSet then
    more?(a, b) == not less?(a, b, (y,z) +-> y <$S z)
    largest x   == largest(x, (y,z) +-> y <$S z)
```

———-

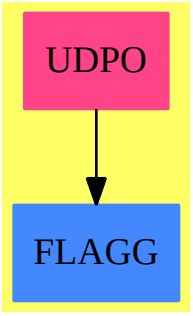
— **UDPO.dotabb** —

```
"UDPO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UDPO"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"UDPO" -> "FLAGG"
```

———-

22.17 package UDVO UserDefinedVariableOrdering

22.17.1 UserDefinedVariableOrdering (UDVO)



Exports:
resetVariableOrder getVariableOrder setVariableOrder

— package UDVO UserDefinedVariableOrdering —

```
)abbrev package UDVO UserDefinedVariableOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ This packages provides functions to allow the user to select the ordering
++ on the variables and operators for displaying polynomials,
++ fractions and expressions. The ordering affects the display
++ only and not the computations.

UserDefinedVariableOrdering(): with
  setVariableOrder  : List Symbol -> Void
    ++ setVariableOrder([a1,...,an]) defines an ordering on the
    ++ variables given by \spad{a1 > a2 > ... > an > other variables}.
  setVariableOrder  : (List Symbol, List Symbol) -> Void
    ++ setVariableOrder([b1,...,bm], [a1,...,an]) defines an ordering
    ++ on the variables given by
    ++ \spad{b1 > b2 > ... > bm >} other variables \spad{>  a1 > a2 > ... > an}.
  getVariableOrder  : () -> Record(high:List Symbol, low:List Symbol)
    ++ getVariableOrder() returns \spad{[[b1,...,bm], [a1,...,an]]} such that
    ++ the ordering on the variables was given by
    ++ \spad{setVariableOrder([b1,...,bm], [a1,...,an])}.
  resetVariableOrder: () -> Void
    ++ resetVariableOrder() cancels any previous use of
    ++ setVariableOrder and returns to the default system ordering.
 == add
  import UserDefinedPartialOrdering(Symbol)

  setVariableOrder l       == setOrder reverse l
  setVariableOrder(l1, l2) == setOrder(reverse l2, reverse l1)
  resetVariableOrder()     == setVariableOrder(nil(), nil())

  getVariableOrder() ==
    r := getOrder()
    [reverse(r.high), reverse(r.low)]
```

———-

— UDVO.dotabb —

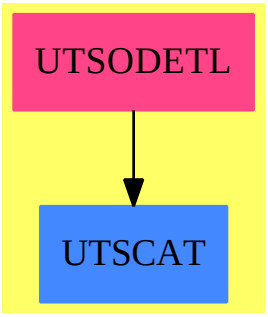
```
"UDVO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UDVO"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
```

```
"UDVO" -> "BASTYPE"
"UDVO" -> "KOERCE"
```

———-

22.18 package UTSODETL UTSodetools

22.18.1 UTSodetools (UTSODETL)



Exports:
LODO2FUN RF2UTS UTS2UP

— package UTSODETL UTSodetools —

```
)abbrev package UTSODETL UTSodetools
++ Author: Manuel Bronstein
++ Date Created: 31 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spad{RUTSodetools} provides tools to interface with the series
++ ODE solver when presented with linear ODEs.

UTSodetools(F, UP, L, UTS): Exports == Implementation where
  F  : Ring
  UP : UnivariatePolynomialCategory F
  L  : LinearOrdinaryDifferentialOperatorCategory UP
  UTS: UnivariateTaylorSeriesCategory F

  Exports ==> with
      UP2UTS:   UP -> UTS
          ++ UP2UTS(p) converts \spad{p} to a Taylor series.
      UTS2UP:   (UTS, NonNegativeInteger) -> UP
          ++ UTS2UP(s, n) converts the first \spad{n} terms of \spad{s}
          ++ to a univariate polynomial.
```

```
      LODO2FUN: L -> (List UTS -> UTS)
          ++ LODO2FUN(op) returns the function to pass to the series ODE
          ++ solver in order to solve \spad{op y = 0}.
      if F has IntegralDomain then
          RF2UTS: Fraction UP -> UTS
              ++ RF2UTS(f) converts \spad{f} to a Taylor series.

  Implementation ==> add
      fun: (Vector UTS, List UTS) -> UTS

      UP2UTS p ==
        q := p(monomial(1, 1) + center(0)::UP)
        +/[monomial(coefficient(q, i), i)$UTS for i in 0..degree q]

      UTS2UP(s, n) ==
        xmc     := monomial(1, 1)$UP - center(0)::UP
        xmcn:UP := 1
        ans:UP  := 0
        for i in 0..n repeat
            ans  := ans + coefficient(s, i) * xmcn
            xmcn := xmc * xmcn
        ans

      LODO2FUN op ==
          a := recip(UP2UTS(- leadingCoefficient op))::UTS
          n := (degree(op) - 1)::NonNegativeInteger
          v := [a * UP2UTS coefficient(op, i) for i in 0..n]$Vector(UTS)
          r := (l1: List UTS): UTS +-> fun(v, l1)
          r

      fun(v, l) ==
          ans:UTS := 0
          for b in l for i in 1.. repeat ans := ans + v.i * b
          ans

      if F has IntegralDomain then
          RF2UTS f == UP2UTS(numer f) * recip(UP2UTS denom f)::UTS
```

———-

— UTSODETL.dotabb —

```
"UTSODETL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSODETL"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTSODETL" -> "UTSCAT"
```

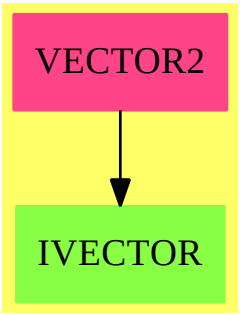
———-

Chapter 23

Chapter V

23.1 package VECTOR2 VectorFunctions2

23.1.1 VectorFunctions2 (VECTOR2)



Exports:
map reduce scan

— package VECTOR2 VectorFunctions2 —

```
)abbrev package VECTOR2 VectorFunctions2
++ Author:
++ Date Created:
++ Date Last Updated:
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ References:
++ Description:
++ This package provides operations which all take as arguments
++ vectors of elements of some type \spad{A} and functions from \spad{A} to
++ another of type B. The operations all iterate over their vector argument
++ and either return a value of type B or a vector over B.

VectorFunctions2(A, B): Exports == Implementation where
  A, B: Type

  VA ==> Vector A
  VB ==> Vector B
  O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)
  UB ==> Union(B,"failed")

  Exports ==> with
    scan   : ((A, B) -> B, VA, B) -> VB
      ++ scan(func,vec,ident) creates a new vector whose elements are
      ++ the result of applying reduce to the binary function func,
      ++ increasing initial subsequences of the vector vec,
      ++ and the element ident.
    reduce : ((A, B) -> B, VA, B) -> B
      ++ reduce(func,vec,ident) combines the elements in vec using the
      ++ binary function func. Argument ident is returned if vec is empty.
    map    : (A -> B, VA) -> VB
      ++ map(f, v) applies the function f to every element of the vector v
      ++ producing a new vector containing the values.
    map : (A -> UB, VA) -> Union(VB,"failed")
      ++ map(f, v) applies the function f to every element of the vector v
      ++ producing a new vector containing the values or \spad{"failed"}.

  Implementation ==> add
    scan(f, v, b)   == scan(f, v, b)$O2
    reduce(f, v, b) == reduce(f, v, b)$O2
    map(f:(A->B), v:VA):VB == map(f, v)$O2

    map(f:(A -> UB), a:VA):Union(VB,"failed") ==
     res : List B  := []
     for u in entries(a) repeat
       r := f u
       r = "failed" => return "failed"
       res := [r::B,:res]
     vector reverse! res
```

———-

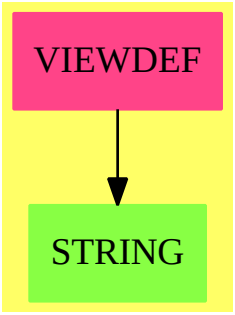
— VECTOR2.dotabb —

```
"VECTOR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VECTOR2"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"VECTOR2" -> "IVECTOR"
```

———-

23.2 package VIEWDEF ViewDefaultsPackage

23.2.1 ViewDefaultsPackage (VIEWDEF)



Exports:

axesColorDefault	lineColorDefault	pointColorDefault
pointSizeDefault	tubePointsDefault	tubeRadiusDefault
unitsColorDefault	var1StepsDefault	var2StepsDefault
unitsColorDefault	viewDefaults	viewPosDefault
viewSizeDefault	viewWriteAvailable	viewWriteDefault

— package VIEWDEF ViewDefaultsPackage —

```
)abbrev package VIEWDEF ViewDefaultsPackage
++ Author: Jim Wen
++ Date Created: 15 January 1990
++ Date Last Updated:
++ Basic Operations: pointColorDefault, lineColorDefault, axesColorDefault,
++ unitsColorDefault, pointSizeDefault, viewPosDefault, viewSizeDefault,
++ viewDefaults, viewWriteDefault, viewWriteAvailable, var1StepsDefault,
++ var2StepsDefault, tubePointsDefault, tubeRadiusDefault
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ ViewportDefaultsPackage describes default and user definable
```

```
++ values for graphics

ViewDefaultsPackage():Exports == Implementation where
  I       ==> Integer
  C       ==> Color
  PAL     ==> Palette
  L       ==> List
  S       ==> String
  E       ==> Expression
  PI      ==> PositiveInteger
  NNI     ==> NonNegativeInteger
  SF      ==> DoubleFloat
  B       ==> Boolean

  writeAvailable ==>  (["PIXMAP","BITMAP","POSTSCRIPT","IMAGE"]::L S)
    -- need not worry about case of letters

  Exports ==> with
    pointColorDefault   :  ()                          -> PAL
      ++ pointColorDefault() returns the default color of points in a 2D
      ++ viewport.
    pointColorDefault   :  PAL                         -> PAL
      ++ pointColorDefault(p) sets the default color of points in a 2D viewport
      ++ to the palette p.
    lineColorDefault    :  ()                          -> PAL
      ++ lineColorDefault() returns the default color of lines connecting
      ++ points in a 2D viewport.
    lineColorDefault    :  PAL                         -> PAL
      ++ lineColorDefault(p) sets the default color of lines connecting points
      ++ in a 2D viewport to the palette p.
    axesColorDefault    :  ()                          -> PAL
      ++ axesColorDefault() returns the default color of the axes in a
      ++ 2D viewport.
    axesColorDefault    :  PAL                         -> PAL
      ++ axesColorDefault(p) sets the default color of the axes in a 2D
      ++ viewport to the palette p.
    unitsColorDefault   :  ()                          -> PAL
      ++ unitsColorDefault() returns the default color of the unit ticks in
      ++ a 2D viewport.
    unitsColorDefault   :  PAL                         -> PAL
      ++ unitsColorDefault(p) sets the default color of the unit ticks in
      ++ a 2D viewport to the palette p.
    pointSizeDefault    :  ()                          -> PI
      ++ pointSizeDefault() returns the default size of the points in
      ++ a 2D viewport.
    pointSizeDefault    :  PI                          -> PI
      ++ pointSizeDefault(i) sets the default size of the points in a 2D
      ++ viewport to i.
    viewPosDefault      :  () -> L NNI
      ++ viewPosDefault() returns the default x and y position of a
```

```
    ++ viewport window unless overriden explicityly, newly created
    ++ viewports will have this X and Y coordinate.
  viewPosDefault      :  L NNI -> L NNI
    ++ viewPosDefault([x,y]) sets the default x and y position of a
    ++ viewport window unless overriden explicityly, newly created
    ++ viewports will have th X and Y coordinates x, y.
  viewSizeDefault : () -> L PI
    ++ viewSizeDefault() returns the default viewport width and height.
  viewSizeDefault : L PI -> L PI
    ++ viewSizeDefault([w,h]) sets the default viewport width to w and height
    ++ to h.
  viewDefaults :  ()                          -> Void
    ++ viewDefaults() resets all the default graphics settings.
  viewWriteDefault  :  ()                           -> L S
    ++ viewWriteDefault() returns the list of things to write in a viewport
    ++ data file; a viewalone file is always generated.
  viewWriteDefault  :  L S                          -> L S
    ++ viewWriteDefault(l) sets the default list of things to write in a
    ++ viewport data file to the strings in l; a viewalone file is always
    ++ genereated.
  viewWriteAvailable : ()                   -> L S
    ++ viewWriteAvailable() returns a list of available methods for writing,
    ++ such as BITMAP, POSTSCRIPT, etc.
  var1StepsDefault       : () -> PI
    ++ var1StepsDefault() is the current setting for the number of steps to
    ++ take when creating a 3D mesh in the direction of the first defined
    ++ free variable (a free variable is considered defined when its
    ++ range is specified (e.g. x=0..10)).
  var2StepsDefault       : () -> PI
    ++ var2StepsDefault() is the current setting for the number of steps to
    ++ take when creating a 3D mesh in the direction of the first defined
    ++ free variable (a free variable is considered defined when its
    ++ range is specified (e.g. x=0..10)).
  var1StepsDefault       : PI -> PI
    ++ var1StepsDefault(i) sets the number of steps to take when creating a
    ++ 3D mesh in the direction of the first defined free variable to i
    ++ (a free variable is considered defined when its range is specified
    ++ (e.g. x=0..10)).
  var2StepsDefault       : PI -> PI
    ++ var2StepsDefault(i) sets the number of steps to take when creating a
    ++ 3D mesh in the direction of the first defined free variable to i
    ++ (a free variable is considered defined when its range is specified
    ++ (e.g. x=0..10)).
  tubePointsDefault  : PI -> PI
    ++ tubePointsDefault(i) sets the number of points to use when creating
    ++ the circle to be used in creating a 3D tube plot to i.
  tubePointsDefault  : () -> PI
    ++ tubePointsDefault() returns the number of points to be used when
    ++ creating the circle to be used in creating a 3D tube plot.
  tubeRadiusDefault  : Float -> SF   -- current tube.spad asks for SF
```

```
      ++ tubeRadiusDefault(r) sets the default radius for a 3D tube plot to r.
    tubeRadiusDefault  : () -> SF
      ++ tubeRadiusDefault() returns the radius used for a 3D tube plot.

  Implementation ==> add

    import Color()
    import Palette()
    --import StringManipulations()

    defaultPointColor : Reference(PAL)  := ref bright red()
    defaultLineColor  : Reference(PAL)  := ref pastel green() --bright blue()
    defaultAxesColor  : Reference(PAL)  := ref dim red()
    defaultUnitsColor : Reference(PAL)  := ref dim yellow()
    defaultPointSize  : Reference(PI)   := ref(3::PI)
    defaultXPos       : Reference(NNI)  := ref(0::NNI)
    defaultYPos       : Reference(NNI)  := ref(0::NNI)
    defaultWidth      : Reference(PI)   := ref(400::PI)
    defaultHeight     : Reference(PI)   := ref(400::PI)
    defaultThingsToWrite : Reference(L S) := ref([]::L S)
    defaultVar1Steps  : Reference(PI)   := ref(27::PI)
    defaultVar2Steps  : Reference(PI)   := ref(27::PI)
    defaultTubePoints : Reference(PI)   := ref(6::PI)
    defaultTubeRadius : Reference(SF)   := ref(convert(0.5)@SF)
    defaultClosed     : Reference(B)    := ref(false)

--%Viewport window dimensions specifications
    viewPosDefault   == [defaultXPos(),defaultYPos()]
    viewPosDefault l ==
      #l < 2 => error "viewPosDefault expects a list with two elements"
      [defaultXPos() := first l,defaultYPos() := last l]

    viewSizeDefault   == [defaultWidth(),defaultHeight()]
    viewSizeDefault l ==
      #l < 2 => error "viewSizeDefault expects a list with two elements"
      [defaultWidth() := first l,defaultHeight() := last l]

    viewDefaults ==
      defaultPointColor : Reference(PAL)  := ref bright red()
      defaultLineColor  : Reference(PAL)  := ref pastel green() --bright blue()
      defaultAxesColor  : Reference(PAL)  := ref dim red()
      defaultUnitsColor : Reference(PAL)  := ref dim yellow()
      defaultPointSize  : Reference(PI)   := ref(3::PI)
      defaultXPos       : Reference(NNI)  := ref(0::NNI)
      defaultYPos       : Reference(NNI)  := ref(0::NNI)
      defaultWidth      : Reference(PI)   := ref(400::PI)
      defaultHeight     : Reference(PI)   := ref(427::PI)

--%2D graphical output specifications
    pointColorDefault   == defaultPointColor()
```

```
    pointColorDefault p == defaultPointColor() := p

    lineColorDefault   == defaultLineColor()
    lineColorDefault p == defaultLineColor() := p

    axesColorDefault   == defaultAxesColor()
    axesColorDefault p == defaultAxesColor() := p

    unitsColorDefault   == defaultUnitsColor()
    unitsColorDefault p == defaultUnitsColor() := p

    pointSizeDefault   == defaultPointSize()
    pointSizeDefault x == defaultPointSize() := x


--%3D specific stuff
    var1StepsDefault   == defaultVar1Steps()
    var1StepsDefault i == defaultVar1Steps() := i

    var2StepsDefault   == defaultVar2Steps()
    var2StepsDefault i == defaultVar2Steps() := i

    tubePointsDefault   == defaultTubePoints()
    tubePointsDefault i == defaultTubePoints() := i

    tubeRadiusDefault   == defaultTubeRadius()
    tubeRadiusDefault f == defaultTubeRadius() := convert(f)@SF

--%File output stuff
    viewWriteAvailable == writeAvailable

    viewWriteDefault == defaultThingsToWrite()

    viewWriteDefault listOfThings ==
      thingsToWrite : L S := []
      for aTypeOfFile in listOfThings repeat
        if (writeTypeInt := position(upperCase aTypeOfFile,viewWriteAvailable())) < 0 then
          sayBrightly(["  > ",concat(aTypeOfFile,
                       " is not a valid file type for writing a viewport")])$Lisp
        else
          thingsToWrite := append(thingsToWrite,[aTypeOfFile])
      defaultThingsToWrite() := thingsToWrite
```

———-

— VIEWDEF.dotabb —

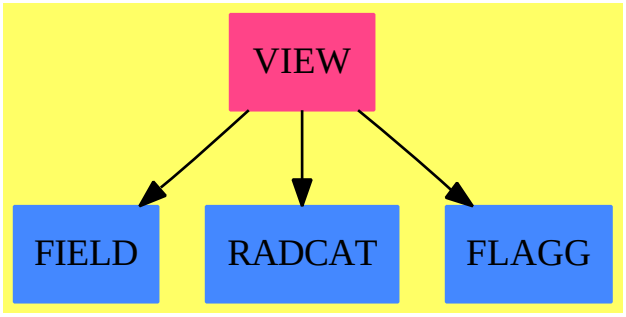
```
"VIEWDEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEWDEF"]
```

```
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VIEWDEF" -> "STRING"
```

———-

23.3 package VIEW ViewportPackage

23.3.1 ViewportPackage (VIEW)



Exports:
coerce drawCurves graphCurves

— package VIEW ViewportPackage —

```
)abbrev package VIEW ViewportPackage
++ Author: Jim Wen
++ Date Created: 30 April 1989
++ Date Last Updated: 15 June 1990
++ Basic Operations: graphCurves, drawCurves, coerce
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ ViewportPackage provides functions for creating GraphImages
++ and TwoDimensionalViewports from lists of lists of points.

ViewportPackage():Exports == Implementation where
  DROP    ==> DrawOption
  GRIMAGE ==> GraphImage
  L       ==> List
  P       ==> Point DoubleFloat
  PAL     ==> Palette
```

```
  PI      ==> PositiveInteger
  VIEW2D  ==> TwoDimensionalViewport

  Exports ==> with

    graphCurves : (L L P,PAL,PAL,PI,L DROP) -> GRIMAGE
      ++ graphCurves([[p0],[p1],...,[pn]],ptColor,lineColor,ptSize,[options])
      ++ creates a \spadtype{GraphImage} from the list of lists of points, p0
      ++ throught pn, using the options specified in the list \spad{options}.
      ++ The graph point color is specified by \spad{ptColor}, the graph line
      ++ color is specified by \spad{lineColor}, and the size of the points is
      ++ specified by \spad{ptSize}.
    graphCurves : L L P -> GRIMAGE
      ++ graphCurves([[p0],[p1],...,[pn]]) creates a \spadtype{GraphImage} from
      ++ the list of lists of points indicated by p0 through pn.
    graphCurves : (L L P,L DROP) -> GRIMAGE
      ++ graphCurves([[p0],[p1],...,[pn]],[options]) creates a
      ++ \spadtype{GraphImage} from the list of lists of points, p0 throught pn,
      ++ using the options specified in the list \spad{options}.
    drawCurves : (L L P,PAL,PAL,PI,L DROP) -> VIEW2D
      ++ drawCurves([[p0],[p1],...,[pn]],ptColor,lineColor,ptSize,[options])
      ++ creates a \spadtype{TwoDimensionalViewport} from the list of lists of
      ++ points, p0 throught pn, using the options specified in the list
      ++ \spad{options}. The point color is specified by \spad{ptColor}, the
      ++ line color is specified by \spad{lineColor}, and the point size is
      ++ specified by \spad{ptSize}.
    drawCurves : (L L P,L DROP) -> VIEW2D
      ++ drawCurves([[p0],[p1],...,[pn]],[options]) creates a
      ++ \spadtype{TwoDimensionalViewport} from the list of lists of points,
      ++ p0 throught pn, using the options specified in the list \spad{options};
    coerce : GRIMAGE -> VIEW2D
      ++ coerce(gi) converts the indicated \spadtype{GraphImage}, gi, into the
      ++ \spadtype{TwoDimensionalViewport} form.

  Implementation ==> add

    import ViewDefaultsPackage
    import DrawOptionFunctions0

--% Functions that return GraphImages

    graphCurves(listOfListsOfPoints) ==
      graphCurves(listOfListsOfPoints, pointColorDefault(),_
                  lineColorDefault(), pointSizeDefault(),nil())

    graphCurves(listOfListsOfPoints,optionsList) ==
      graphCurves(listOfListsOfPoints, pointColorDefault(),_
                  lineColorDefault(), pointSizeDefault(),optionsList)

    graphCurves(listOfListsOfPoints,ptColor,lineColor,ptSize,optionsList) ==
```

```
      len := #listOfListsOfPoints
      listOfPointColors : L PAL := [ptColor for i in 1..len]
      listOfLineColors  : L PAL := [lineColor for i in 1..len]
      listOfPointSizes  : L PI  := [ptSize  for i in 1..len]
      makeGraphImage(listOfListsOfPoints,listOfPointColors, _
                          listOfLineColors,listOfPointSizes,optionsList)

--% Functions that return Two Dimensional Viewports

    drawCurves(listOfListsOfPoints,optionsList) ==
      drawCurves(listOfListsOfPoints,pointColorDefault(),_
                 lineColorDefault(),pointSizeDefault(),optionsList)

    drawCurves(ptLists:L L P,ptColor:PAL,lColor:PAL,ptSize:PI,optList:L DROP) ==
      v := viewport2D()
      options(v,optList)
      g :=  graphCurves(ptLists,ptColor,lColor,ptSize,optList)
      putGraph(v,g,1)
      makeViewport2D v

--% Coercions

    coerce(graf:GRIMAGE):VIEW2D ==
      if (key graf = 0) then makeGraphImage graf
      v := viewport2D()
      title(v,"VIEW2D")
--      dimensions(v,viewPosDefault().1,viewPosDefault().2,viewSizeDefault().1,viewSizeDefault()
      putGraph(v,graf,1::PI)
      makeViewport2D v
```

———-

— VIEW.dotabb —

```
"VIEW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEW"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"VIEW" -> "FIELD"
"VIEW" -> "RADCAT"
"VIEW" -> "FLAGG"
```

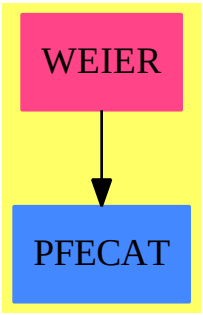
———-

Chapter 24

Chapter W

24.1 package WEIER WeierstrassPreparation

24.1.1 WeierstrassPreparation (WEIER)



Exports:

cfirst	clikeUniv	crest	qqq
sts2stst	weierstrass		

— package WEIER WeierstrassPreparation —

```
)abbrev package WEIER WeierstrassPreparation
++ Author:William H. Burge
++ Date Created:Sept 1988
++ Date Last Updated:Feb 15 1992
++ Basic Operations:
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords:
```

```
++ Examples:
++ References:
++ Description:
++ This package implements the Weierstrass preparation
++ theorem f or multivariate power series.
++ weierstrass(v,p) where v is a variable, and p is a
++ TaylorSeries(R) in which the terms
++ of lowest degree s must include c*v**s where c is a constant,s>0,
++ is a list of TaylorSeries coefficients A[i] of the equivalent polynomial
++ A = A[0] + A[1]*v + A[2]*v**2 + ... + A[s-1]*v**(s-1) + v**s
++ such that p=A*B , B being a TaylorSeries of minimum degree 0

WeierstrassPreparation(R): Defn == Impl where
    R : Field
    VarSet==>Symbol
    SMP ==> Polynomial R
    PS  ==> InnerTaylorSeries SMP
    NNI ==> NonNegativeInteger
    ST  ==> Stream
    StS ==> Stream SMP
    STPS==>StreamTaylorSeriesOperations
    STTAYLOR==>StreamTaylorSeriesOperations
    SUP==> SparseUnivariatePolynomial(SMP)
    ST2==>StreamFunctions2
    SMPS==>  TaylorSeries(R)
    L==>List
    null ==> empty?
    likeUniv ==> univariate
    coef ==> coefficient$SUP
    nil ==> empty


    Defn ==>  with

        crest:(NNI->( StS-> StS))
          ++\spad{crest n} is used internally.
        cfirst:(NNI->( StS-> StS))
          ++\spad{cfirst n} is used internally.
        sts2stst:(VarSet,StS)->ST StS
          ++\spad{sts2stst(v,s)} is used internally.
        clikeUniv:VarSet->(SMP->SUP)
          ++\spad{clikeUniv(v)} is used internally.
        weierstrass:(VarSet,SMPS)->L SMPS
          ++\spad{weierstrass(v,ts)} where v is a variable and ts is
          ++ a TaylorSeries, impements the Weierstrass Preparation
          ++ Theorem. The result is a list of TaylorSeries that
          ++ are the coefficients of the equivalent series.
        qqq:(NNI,SMPS,ST SMPS)->((ST SMPS)->ST SMPS)
          ++\spad{qqq(n,s,st)} is used internally.
```

```
Impl ==>  add
    import TaylorSeries(R)
    import StreamTaylorSeriesOperations SMP
    import StreamTaylorSeriesOperations SMPS


    map1==>map$(ST2(SMP,SUP))
    map2==>map$(ST2(StS,SMP))
    map3==>map$(ST2(StS,StS))
    transback:ST SMPS->L SMPS
    transback smps==
        if null smps
        then nil()$(L SMPS)
        else
          if null first (smps:(ST StS))
          then nil()$(L SMPS)
          else
            cons(map2(first,smps:ST StS):SMPS,
               transback(map3(rest,smps:ST StS):(ST SMPS)))$(L SMPS)


    clikeUniv(var)==p +-> likeUniv(p,var)
    mind:(NNI,StS)->NNI
    mind(n, sts)==
       if null sts
       then error "no mindegree"
       else if first sts=0
            then mind(n+1,rest sts)
            else n
    mindegree (sts:StS):NNI== mind(0,sts)


    streamlikeUniv:(SUP,NNI)->StS
    streamlikeUniv(p:SUP,n:NNI): StS ==
      if n=0
      then cons(coef (p,0),nil()$StS)
      else cons(coef (p,n),streamlikeUniv(p,(n-1):NNI))

    transpose:ST StS->ST StS
    transpose(s:ST StS)==delay(
       if null s
       then nil()$(ST StS)
       else cons(map2(first,s),transpose(map3(rest,rst s))))

    zp==>map$StreamFunctions3(SUP,NNI,StS)

    sts2stst(var, sts)==
       zp((x,y) +-> streamlikeUniv(x,y),
         map1(clikeUniv var, sts),(integers 0):(ST NNI))
```

```
        tp:(VarSet,StS)->ST StS
        tp(v,sts)==transpose sts2stst(v,sts)
        map4==>map$(ST2 (StS,StS))
        maptake:(NNI,ST StS)->ST SMPS
        maptake(n,p)== map4(cfirst n,p) pretend ST SMPS
        mapdrop:(NNI,ST StS)->ST SMPS
        mapdrop(n,p)== map4(crest n,p) pretend ST SMPS
        YSS==>Y$ParadoxicalCombinatorsForStreams(SMPS)
        weier:(VarSet,StS)->ST SMPS
        weier(v,sts)==
             a:=mindegree sts
             if a=0
             then error "has constant term"
             else
               p:=tp(v,sts) pretend (ST SMPS)
               b:StS:=rest(((first p pretend StS)),a::NNI)
               c:=retractIfCan first b
               c case "failed"=>_
 error "the coefficient of the lowest degree of the variable should _
 be a constant"
               e:=recip b
               f:= if e case "failed"
                   then error "no reciprocal"
                   else e::StS
               q:=(YSS qqq(a,f:SMPS,rest p))
               maptake(a,(p*q) pretend ST StS)

        cfirst n == s +-> first(s,n)$StS
        crest n  == s +-> rest(s,n)$StS
        qq:(NNI,SMPS,ST SMPS,ST SMPS)->ST SMPS
        qq(a,e,p,c)==
            cons(e,(-e)*mapdrop(a,(p*c)pretend(ST StS)))
        qqq(a,e,p)==  s +-> qq(a,e,p,s)
        wei:(VarSet,SMPS)->ST SMPS
        wei(v:VarSet,s:SMPS)==weier(v,s:StS)
        weierstrass(v,smps)== transback wei (v,smps)
```

———-

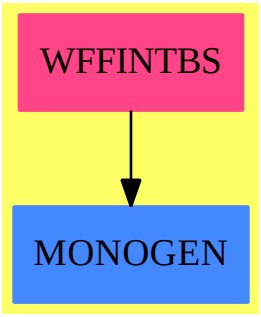
— WEIER.dotabb —

```
"WEIER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=WEIER"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"WEIER" -> "PFECAT"
```

———-

24.2 package WFFINTBS WildFunctionFieldIntegralBasis

24.2.1 WildFunctionFieldIntegralBasis (WFFINTBS)



Exports:
integralBasis localIntegralBasis

— package WFFINTBS WildFunctionFieldIntegralBasis —

```
)abbrev package WFFINTBS WildFunctionFieldIntegralBasis
++ Authors: Victor Miller, Clifton Williamson
++ Date Created: 24 July 1991
++ Date Last Updated: 20 September 1994
++ Basic Operations: integralBasis, localIntegralBasis
++ Related Domains: IntegralBasisTools(R,UP,F),
++   TriangularMatrixOperations(R,Vector R,Vector R,Matrix R)
++ Also See: FunctionFieldIntegralBasis, NumberFieldIntegralBasis
++ AMS Classifications:
++ Keywords: function field, integral basis
++ Examples:
++ References:
++ Description:
++ In this package K is a finite field, R is a ring of univariate
++ polynomials over K, and F is a framed algebra over R.  The package
++ provides a function to compute the integral closure of R in the quotient
++ field of F as well as a function to compute a "local integral basis"
++ at a specific prime.

WildFunctionFieldIntegralBasis(K,R,UP,F): Exports == Implementation where
  K  : FiniteFieldCategory
  --K  : Join(Field,Finite)
  R  : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R
  F  : FramedAlgebra(R,UP)
```

```
I       ==> Integer
Mat     ==> Matrix R
NNI     ==> NonNegativeInteger
SAE     ==> SimpleAlgebraicExtension
RResult ==> Record(basis: Mat, basisDen: R, basisInv:Mat)
IResult ==> Record(basis: Mat, basisDen: R, basisInv:Mat,discr: R)
MATSTOR ==> StorageEfficientMatrixOperations

Exports ==> with
  integralBasis : () -> RResult
    ++ \spad{integralBasis()} returns a record
    ++ \spad{[basis,basisDen,basisInv]} containing information regarding
    ++ the integral closure of R in the quotient field of F, where
    ++ F is a framed algebra with R-module basis \spad{w1,w2,...,wn}.
    ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of \spad{basis} contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
    ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
    ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.
  localIntegralBasis : R -> RResult
    ++ \spad{integralBasis(p)} returns a record
    ++ \spad{[basis,basisDen,basisInv]} containing information regarding
    ++ the local integral closure of R at the prime \spad{p} in the quotient
    ++ field of F, where F is a framed algebra with R-module basis
    ++ \spad{w1,w2,...,wn}.
    ++ If \spad{basis} is the matrix \spad{(aij, i = 1..n, j = 1..n)}, then
    ++ the \spad{i}th element of the local integral basis is
    ++ \spad{vi = (1/basisDen) * sum(aij * wj, j = 1..n)}, i.e. the
    ++ \spad{i}th row of \spad{basis} contains the coordinates of the
    ++ \spad{i}th basis vector.  Similarly, the \spad{i}th row of the
    ++ matrix \spad{basisInv} contains the coordinates of \spad{wi} with
    ++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
    ++ matrix \spad{(bij, i = 1..n, j = 1..n)}, then
    ++ \spad{wi = sum(bij * vj, j = 1..n)}.

Implementation ==> add
  import IntegralBasisTools(R, UP, F)
  import ModularHermitianRowReduction(R)
  import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)
  import DistinctDegreeFactorize(K,R)

  listSquaredFactors: R -> List R
  listSquaredFactors px ==
    -- returns a list of the factors of px which occur with
    -- exponent > 1
```

```
    ans : List R := empty()
    factored := factor(px)$DistinctDegreeFactorize(K,R)
    for f in factors(factored) repeat
      if f.exponent > 1 then ans := concat(f.factor,ans)
    ans

  iLocalIntegralBasis: (Vector F,Vector F,Matrix R,Matrix R,R,R) -> IResult
  iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime) ==
    n := rank()$F; standardBasis := basis()$F
    -- 'standardBasis' is the basis for F as a FramedAlgebra;
    -- usually this is [1,y,y**2,...,y**(n-1)]
    p2 := prime * prime; sae := SAE(K,R,prime)
    p := characteristic()$F; q := size()$sae
    lp := leastPower(q,n)
    rb := scalarMatrix(n,1); rbinv := scalarMatrix(n,1)
    -- rb    = basis matrix of current order
    -- rbinv = inverse basis matrix of current order
    -- these are wrt the orginal basis for F
    rbden : R := 1; index : R := 1; oldIndex : R := 1
    -- rbden = denominator for current basis matrix
    -- index = index of original order in current order
    repeat
      -- pows = [(w1 * rbden) ** q,...,(wn * rbden) ** q], where
      -- bas = [w1,...,wn] is 'rbden' times the basis for the order B = 'rb'
      for i in 1..n repeat
        bi : F := 0
        for j in 1..n repeat
          bi := bi + qelt(rb,i,j) * qelt(standardBasis,j)
        qsetelt_!(bas,i,bi)
        qsetelt_!(pows,i,bi ** p)
      coor0 := transpose coordinates(pows,bas)
      denPow := rbden ** ((p - 1) :: NNI)
      (coMat0 := coor0 exquo denPow) case "failed" =>
        error "can't happen"
      -- the jth column of coMat contains the coordinates of (wj/rbden)**q
      -- with respect to the basis [w1/rbden,...,wn/rbden]
      coMat := coMat0 :: Matrix R
      -- the ith column of 'pPows' contains the coordinates of the pth power
      -- of the ith basis element for B/prime.B over 'sae' = R/prime.R
      pPows := map(reduce,coMat)$MatrixCategoryFunctions2(R,Vector R,
                  Vector R,Matrix R,sae,Vector sae,Vector sae,Matrix sae)
      -- 'frob' will eventually be the Frobenius matrix for B/prime.B over
      -- 'sae' = R/prime.R; at each stage of the loop the ith column will
      -- contain the coordinates of p^k-th powers of the ith basis element
      frob := copy pPows; tmpMat : Matrix sae := new(n,n,0)
      for r in 2..leastPower(p,q) repeat
        for i in 1..n repeat for j in 1..n repeat
          qsetelt_!(tmpMat,i,j,qelt(frob,i,j) ** p)
        times_!(frob,pPows,tmpMat)$MATSTOR(sae)
      frobPow := frob ** lp
```

```
    -- compute the p-radical
    ns := nullSpace frobPow
    for i in 1..n repeat for j in 1..n repeat qsetelt_!(tfm,i,j,0)
    for vec in ns for i in 1.. repeat
      for j in 1..n repeat
        qsetelt_!(tfm,i,j,lift qelt(vec,j))
    id := squareTop rowEchelon(tfm,prime)
    -- id = basis matrix of the p-radical
    idinv := UpTriBddDenomInv(id, prime)
    -- id * idinv = prime * identity
    -- no need to check for inseparability in this case
    rbinv := idealiser(id * rb, rbinv * idinv, prime * rbden)
    index := diagonalProduct rbinv
    rb := rowEchelon LowTriBddDenomInv(rbinv,rbden * prime)
    if divideIfCan_!(rb,matrixOut,prime,n) = 1
      then rb := matrixOut
      else rbden := rbden * prime
    rbinv := UpTriBddDenomInv(rb,rbden)
    indexChange := index quo oldIndex
    oldIndex := index
    disc := disc quo (indexChange * indexChange)
    (not sizeLess?(1,indexChange)) or ((disc exquo p2) case "failed") =>
      return [rb, rbden, rbinv, disc]

integralBasis() ==
  traceMat := traceMatrix()$F; n := rank()$F
  disc := determinant traceMat        -- discriminant of current order
  zero? disc => error "integralBasis: polynomial must be separable"
  singList := listSquaredFactors disc -- singularities of relative Spec
  runningRb := scalarMatrix(n,1); runningRbinv := scalarMatrix(n,1)
  -- runningRb    = basis matrix of current order
  -- runningRbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  runningRbden : R := 1
  -- runningRbden = denominator for current basis matrix
  empty? singList => [runningRb, runningRbden, runningRbinv]
  bas : Vector F := new(n,0); pows : Vector F := new(n,0)
  -- storage for basis elements and their powers
  tfm : Matrix R := new(n,n,0)
  -- 'tfm' will contain the coordinates of a lifting of the kernel
  -- of a power of Frobenius
  matrixOut : Matrix R := new(n,n,0)
  for prime in singList repeat
    lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)
    rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
    disc := lb.discr
    -- update 'running integral basis' if newly computed
    -- local integral basis is non-trivial
    if sizeLess?(1,rbden) then
      mat := vertConcat(rbden * runningRb,runningRbden * rb)
```

```
        runningRbden := runningRbden * rbden
        runningRb := squareTop rowEchelon(mat,runningRbden)
        runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
      [runningRb, runningRbden, runningRbinv]

    localIntegralBasis prime ==
      traceMat := traceMatrix()$F; n := rank()$F
      disc := determinant traceMat        -- discriminant of current order
      zero? disc => error "localIntegralBasis: polynomial must be separable"
      (disc exquo (prime * prime)) case "failed" =>
        [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
      bas : Vector F := new(n,0); pows : Vector F := new(n,0)
      -- storage for basis elements and their powers
      tfm : Matrix R := new(n,n,0)
      -- 'tfm' will contain the coordinates of a lifting of the kernel
      -- of a power of Frobenius
      matrixOut : Matrix R := new(n,n,0)
      lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)
      [lb.basis, lb.basisDen, lb.basisInv]
```

———-

— **WFFINTBS.dotabb** —

```
"WFFINTBS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=WFFINTBS"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"WFFINTBS" -> "MONOGEN"
```

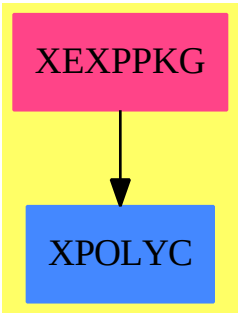
———-

Chapter 25

Chapter X

25.1 package XEXPPKG XExponentialPackage

25.1.1 XExponentialPackage (XEXPPKG)



Exports:
exp Hausdorff log

— **XEXPPKG.dotabb** —

```
"XEXPPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=XEXPPKG"]
```

———-

— **package XEXPPKG XExponentialPackage** —

```
)abbrev package XEXPPKG XExponentialPackage
++ Author: Michel Petitot (petitot@lifl.fr).
```

```
++ Date Created: 91
++ Date Last Updated: 7 Juillet 92
++ Fix History: compilation v 2.1 le 13 dec 98
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ This package provides computations of logarithms and exponentials
++ for polynomials in non-commutative variables.

XExponentialPackage(R, VarSet, XPOLY): Public == Private where
    RN     ==> Fraction Integer
    NNI    ==> NonNegativeInteger
    I      ==> Integer
    R      : Join(Ring, Module RN)
    -- R      : Field
    VarSet : OrderedSet
    XPOLY  : XPolynomialsCat(VarSet, R)

    Public == with
       exp:  (XPOLY, NNI) -> XPOLY
         ++ \axiom{exp(p, n)} returns the exponential of \axiom{p}
         ++ truncated at order \axiom{n}.
       log:  (XPOLY, NNI) -> XPOLY
         ++ \axiom{log(p, n)} returns the logarithm of \axiom{p}
         ++ truncated at order \axiom{n}.
       Hausdorff: (XPOLY, XPOLY, NNI) -> XPOLY
         ++ \axiom{Hausdorff(a,b,n)} returns log(exp(a)*exp(b))
         ++ truncated at order \axiom{n}.

    Private == add

        log (p,n) ==
           p1 : XPOLY := p - 1
           not quasiRegular? p1 =>
             error "constant term <> 1, impossible log"
           s : XPOLY := 0       -- resultat
           k : I := n :: I
           for i in 1 .. n repeat
              k1 :RN := 1/k
              k2 : R := k1 * 1$R
              s := trunc( trunc(p1,i) * (k2 :: XPOLY - s) , i)
              k := k - 1
           s

        exp (p,n) ==
           not quasiRegular? p =>
```

```
            error "constant term <> 0, exp impossible"
         p = 0 => 1
         s : XPOLY := 1$XPOLY       -- resultat
         k : I := n :: I
         for i in 1 .. n repeat
            k1 :RN := 1/k
            k2 : R := k1 * 1$R
            s := trunc( 1 +$XPOLY k2 * trunc(p,i) * s , i)
            k := k - 1
         s

      Hausdorff(p,q,n) ==
         p1: XPOLY := exp(p,n)
         q1: XPOLY := exp(q,n)
         log(p1*q1, n)
```

———-

— XEXPPKG.dotabb —

```
"XEXPPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=XEXPPKG"]
"XPOLYC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=XPOLYC"]
"XEXPPKG" -> "XPOLYC"
```

———-

Chapter 26

Chapter Y

Chapter 27

Chapter Z

27.1 package ZDSOLVE ZeroDimensionalSolvePackage

Based on triangular decompositions and the **RealClosure** constructor, the pacakge **ZeroDimensionalSolvePackage** provides operations for computing symbolically the real or complex roots of polynomial systems with finitely many solutions.

— **ZeroDimensionalSolvePackage.input** —

```
)set break resume
)spool ZeroDimensionalSolvePackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 28
R := Integer
--R
--R
--R   (1)  Integer
--R                                                          Type: Domain
--E 1

--S 2 of 28
ls : List Symbol := [x,y,z,t]
--R
--R
--R   (2)  [x,y,z,t]
--R                                                   Type: List(Symbol)
--E 2

--S 3 of 28
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
--R
```

```
--R
--R   (3)  [x,y,z,t,%A]
--R                                                        Type: List(Symbol)
--E 3

--S 4 of 28
pack := ZDSOLVE(R,ls,ls2)
--R
--R
--R   (4)  ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])
--R                                                              Type: Domain
--E 4

--S 5 of 28
p1 := x**2*y*z + x*y**2*z + x*y*z**2 + x*y*z + x*y + x*z + y*z
--R
--R
--R            2       2     2
--R   (5)  x y z  + (x y  + (x  + x + 1)y + x)z + x y
--R                                                  Type: Polynomial(Integer)
--E 5

--S 6 of 28
p2 := x**2*y**2*z + x*y**2*z**2 + x**2*y*z + x*y*z + y*z + x + z
--R
--R
--R          2 2     2 2     2
--R   (6)  x y z  + (x y  + (x  + x + 1)y + 1)z + x
--R                                                  Type: Polynomial(Integer)
--E 6

--S 7 of 28
p3 := x**2*y**2*z**2 + x**2*y**2*z + x*y**2*z + x*y*z + x*z + z + 1
--R
--R
--R         2 2 2      2      2
--R   (7)  x y z  + ((x  + x)y  + x y + x + 1)z + 1
--R                                                  Type: Polynomial(Integer)
--E 7

--S 8 of 28
lp := [p1, p2, p3]
--R
--R
--R   (8)
--R         2       2     2
--R   [x y z  + (x y  + (x  + x + 1)y + x)z + x y,
--R       2 2     2 2     2
--R    x y z  + (x y  + (x  + x + 1)y + 1)z + x,
--R     2 2 2      2      2
```

```
--R    x y z  + ((x  + x)y  + x y + x + 1)z + 1]
--R                                                Type: List(Polynomial(Integer))
--E 8

--S 9 of 28
triangSolve(lp)$pack
--R
--R
--R   (9)
--R   [
--R     {
--R          20     19      18      17       16      15       14       13       12
--R         z   - 6z   - 41z   + 71z   + 106z   + 92z   + 197z   + 145z   + 257z
--R       +
--R             11       10       9       8       7       6      5       4      3
--R         278z   + 201z   + 278z  + 257z  + 145z  + 197z  + 92z  + 106z  + 71z
--R       +
--R              2
--R         - 41z  - 6z + 1
--R       ,
--R
--R                       19            18             17             16
--R             14745844z   + 50357474z   - 130948857z   - 185261586z
--R           +
--R                           15             14             13             12
--R             - 180077775z   - 338007307z   - 275379623z   - 453190404z
--R           +
--R                           11             10             9             8
--R             - 474597456z   - 366147695z   - 481433567z  - 430613166z
--R           +
--R                           7             6             5             4
--R             - 261878358z  - 326073537z  - 163008796z  - 177213227z
--R           +
--R                           3            2
--R             - 104356755z  + 65241699z  + 9237732z - 1567348
--R        *
--R           y
--R       +
--R                  19           18            17            16            15
--R         1917314z   + 6508991z   - 16973165z   - 24000259z   - 23349192z
--R       +
--R                     14            13            12            11            10
--R         - 43786426z   - 35696474z   - 58724172z   - 61480792z   - 47452440z
--R       +
--R                     9            8            7            6            5
--R         - 62378085z  - 55776527z  - 33940618z  - 42233406z  - 21122875z
--R       +
--R                     4            3           2
--R         - 22958177z  - 13504569z  + 8448317z  + 1195888z - 202934
--R       ,
```

```
--R         3       2       3    2              2             2
--R      ((z  - 2z)y  + (- z  - z  - 2z - 1)y - z  - z + 1)x + z  - 1}
--R     ]
--R                                Type: List(RegularChain(Integer,[x,y,z,t]))
--E 9

--S 10 of 28
univariateSolve(lp)$pack
--R
--R
--R   (10)
--R   [
--R     [
--R       complexRoots =
--R            12      11      10     9     8      7      6      5     4     3
--R          ?   - 12?   + 24?   + 4?  - 9?  + 27?  - 21?  + 27?  - 9?  + 4?
--R        +
--R             2
--R          24?  - 12? + 1
--R       ,
--R
--R       coordinates =
--R         [
--R                        11        10         9        8        7         6
--R            63x + 62%A   - 721%A   + 1220%A  + 705%A  - 285%A  + 1512%A
--R          +
--R                   5         4       3        2
--R            - 735%A  + 1401%A  - 21%A  + 215%A  + 1577%A - 142
--R          ,
--R
--R                        11        10         9        8        7         6
--R            63y - 75%A   + 890%A   - 1682%A  - 516%A  + 588%A  - 1953%A
--R          +
--R                  5         4        3        2
--R            1323%A  - 1815%A  + 426%A  - 243%A  - 1801%A + 679
--R          ,
--R         z - %A]
--R       ]
--R     ,
--R
--R                     6    5    4    3    2
--R     [complexRoots= ?  + ?  + ?  + ?  + ?  + ? + 1,
--R                          5       3
--R      coordinates= [x - %A ,y - %A ,z - %A]]
--R     ,
--R                    2
--R    [complexRoots= ?  + 5? + 1,coordinates= [x - 1,y - 1,z - %A]]]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynom
--E 10
```

```
--S 11 of 28
lr := realSolve(lp)$pack
--R
--R
--R   (11)
--R   [
--R     [%B1,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B1   - ------- %B1   - ------- %B1   + -------- %B1
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B1   + -------- %B1   + ------ %B1   + --------- %B1
--R          548457           182819           3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B1   + --------- %B1   + --------- %B1  + --------- %B1
--R          182819           1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B1  + ------- %B1  + ------ %B1  + ------- %B1
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         --------- %B1  + -------- %B1  - ---- %B1 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B1   + ------ %B1   + ------- %B1   - ------- %B1
--R           705159         705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B1   - -------- %B1   - -------- %B1   - ------ %B1
--R            705159           705159           705159           18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B1   - -------- %B1   - -------- %B1  - ------- %B1
--R            100737           235053           235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B1  - -------- %B1  - -------- %B1  - -------- %B1
--R            6027           705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B1  - -------- %B1  - ------- %B1 + ------
--R            26117          705159          705159       705159
--R       ]
--R     ,
```

```
--R
--R     [%B2,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B2   - ------- %B2   - ------- %B2   + -------- %B2
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B2   + -------- %B2   + ------ %B2   + ---------- %B2
--R          548457          182819           3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B2   + ---------- %B2   + ---------- %B2  + ---------- %B2
--R          182819          1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B2  + ------- %B2  + ------ %B2  + ------- %B2
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         --------- %B2  + -------- %B2  - ---- %B2 - --------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B2   + ------ %B2   + ------- %B2   - ------- %B2
--R           705159         705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B2   - -------- %B2   - -------- %B2   - ------ %B2
--R            705159          705159          705159          18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B2   - -------- %B2   - -------- %B2  - ------- %B2
--R            100737          235053          235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B2  - -------- %B2  - -------- %B2  - -------- %B2
--R            6027          705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B2  - -------- %B2  - ------- %B2 + ------
--R            26117         705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B3,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B3   - ------- %B3   - ------- %B3   + -------- %B3
```

```
--R         1645371           548457           182819           1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B3   + -------- %B3   + ------ %B3   + ---------- %B3
--R          548457           182819           3731            1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B3   + ---------- %B3   + ---------- %B3  + ---------- %B3
--R          182819           1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         ---------- %B3  + ------- %B3  + ------ %B3  + ------- %B3
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         ---------- %B3  + -------- %B3  - ---- %B3 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B3   + ------ %B3   + ------- %B3   - ------- %B3
--R           705159          705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B3   - -------- %B3   - -------- %B3   - ------ %B3
--R            705159           705159           705159           18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B3   - -------- %B3   - -------- %B3  - ------- %B3
--R            100737           235053           235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B3  - -------- %B3  - -------- %B3  - -------- %B3
--R            6027           705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B3  - -------- %B3  - ------- %B3 + ------
--R            26117          705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B4,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B4   - ------- %B4   - ------- %B4   + -------- %B4
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B4   + -------- %B4   + ------ %B4   + ---------- %B4
--R          548457           182819           3731            1645371
```

```
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B4   + --------- %B4   + --------- %B4  + --------- %B4
--R          182819           1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B4  + ------- %B4  + ------ %B4  + ------- %B4
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         --------- %B4  + -------- %B4  - ---- %B4 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B4   + ------ %B4   + ------- %B4   - ------- %B4
--R           705159         705159          705159          235053
--R       +
--R            13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B4   - -------- %B4   - -------- %B4   - ------ %B4
--R             705159           705159           705159           18081
--R       +
--R            5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B4   - -------- %B4   - -------- %B4  - ------- %B4
--R             100737          235053           235053          100737
--R       +
--R            409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B4  - -------- %B4  - -------- %B4  - -------- %B4
--R             6027          705159          705159          705159
--R       +
--R            801511    3   17206178    2   4406102       377534
--R         - ------ %B4  - -------- %B4  - ------- %B4 + ------
--R             26117         705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B5,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B5   - ------- %B5   - ------- %B5   + -------- %B5
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B5   + -------- %B5   + ------ %B5   + --------- %B5
--R          548457           182819           3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B5   + --------- %B5   + --------- %B5  + --------- %B5
--R          182819           1645371           1645371           548457
--R       +
```

```
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B5  + ------- %B5  + ------ %B5  + ------- %B5
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         --------- %B5  + -------- %B5  - ---- %B5 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B5   + ------ %B5   + ------- %B5   - ------- %B5
--R           705159         705159          705159          235053
--R       +
--R            13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B5   - -------- %B5   - -------- %B5   - ------ %B5
--R           705159           705159          705159           18081
--R       +
--R            5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B5   - -------- %B5   - -------- %B5  - ------- %B5
--R           100737          235053          235053          100737
--R       +
--R            409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B5  - -------- %B5  - -------- %B5  - -------- %B5
--R           6027          705159          705159          705159
--R       +
--R            801511    3   17206178    2   4406102       377534
--R         - ------ %B5  - -------- %B5  - ------- %B5 + ------
--R           26117         705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B6,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B6   - ------- %B6   - ------- %B6   + -------- %B6
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B6   + -------- %B6   + ------ %B6   + ---------- %B6
--R          548457           182819          3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B6   + ---------- %B6   + ---------- %B6  + ---------- %B6
--R          182819           1645371          1645371          548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B6  + ------- %B6  + ------ %B6  + ------- %B6
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
```

```
--R         --------- %B6  + -------- %B6  - ---- %B6 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B6   + ------ %B6   + ------- %B6   - ------- %B6
--R           705159         705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B6   - -------- %B6   - -------- %B6   - ------ %B6
--R            705159           705159           705159           18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B6   - -------- %B6   - -------- %B6  - ------- %B6
--R            100737          235053           235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B6  - -------- %B6  - -------- %B6  - -------- %B6
--R            6027          705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B6  - -------- %B6  - ------- %B6 + ------
--R            26117         705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B7,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B7   - ------- %B7   - ------- %B7   + -------- %B7
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B7   + -------- %B7   + ------ %B7   + --------- %B7
--R          548457           182819           3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B7   + --------- %B7   + --------- %B7  + --------- %B7
--R          182819           1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         --------- %B7  + ------- %B7  + ------ %B7  + ------- %B7
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         --------- %B7  + -------- %B7  - ---- %B7 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
```

```
--R         - ------ %B7   + ------ %B7   + ------- %B7   - ------- %B7
--R           705159         705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B7   - -------- %B7   - -------- %B7   - ------ %B7
--R            705159           705159           705159           18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B7   - -------- %B7   - -------- %B7  - ------- %B7
--R            100737          235053           235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B7  - -------- %B7  - -------- %B7  - -------- %B7
--R            6027          705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B7  - -------- %B7  - ------- %B7 + ------
--R            26117         705159          705159       705159
--R       ]
--R     ,
--R
--R     [%B8,
--R
--R         1184459    19   2335702    18   5460230    17   79900378    16
--R         ------- %B8   - ------- %B8   - ------- %B8   + -------- %B8
--R         1645371          548457          182819          1645371
--R       +
--R         43953929    15   13420192    14   553986    13   193381378    12
--R         -------- %B8   + -------- %B8   + ------ %B8   + ---------- %B8
--R          548457           182819           3731           1645371
--R       +
--R         35978916    11   358660781    10   271667666    9   118784873    8
--R         -------- %B8   + ---------- %B8   + ---------- %B8  + ---------- %B8
--R          182819           1645371           1645371           548457
--R       +
--R         337505020    7   1389370    6   688291    5   3378002    4
--R         ---------- %B8  + ------- %B8  + ------ %B8  + ------- %B8
--R          1645371          11193          4459          42189
--R       +
--R         140671876    3   32325724    2   8270       9741532
--R         ---------- %B8  + -------- %B8  - ---- %B8 - -------
--R          1645371          548457          343       1645371
--R       ,
--R
--R            91729    19   487915    18   4114333    17   1276987    16
--R         - ------ %B8   + ------ %B8   + ------- %B8   - ------- %B8
--R           705159         705159          705159          235053
--R       +
--R           13243117    15   16292173    14   26536060    13   722714    12
--R         - -------- %B8   - -------- %B8   - -------- %B8   - ------ %B8
```

```
--R            705159           705159           705159           18081
--R       +
--R           5382578    11   15449995    10   14279770    9   6603890    8
--R         - ------- %B8   - -------- %B8   - -------- %B8  - ------- %B8
--R            100737          235053           235053          100737
--R       +
--R           409930    7   37340389    6   34893715    5   26686318    4
--R         - ------ %B8  - -------- %B8  - -------- %B8  - -------- %B8
--R            6027          705159          705159          705159
--R       +
--R           801511    3   17206178    2   4406102       377534
--R         - ------ %B8  - -------- %B8  - ------- %B8 + ------
--R            26117         705159          705159       705159
--R       ]
--R     ]
--R                             Type: List(List(RealClosure(Fraction(Integer))))
--E 11

--S 12 of 28
# lr
--R
--R
--R   (12)  8
--R                                                        Type: PositiveInteger
--E 12

--S 13 of 28
[ [approximate(r,1/1000000) for r in point] for point in lr]
--R
--R
--R   (13)
--R   [
--R        10048059
--R     [- --------,
--R         2097152
--R
--R        450305731698538794352439791383896641459673197621176821933588120838551 6_
--R         314058924567176091423629695777403099833360761048898228916578137094309_
--R         838597331137202584846939132376157019506760357601165917454986815382098_
--R         789094851523420392811293126141329856546977145464661495487825919941188_
--R         447041722440491921567263542158028061437758844364634410045253024786561_
--R         923163288214175
--R      /
--R        450305728302524548851651180698582663508310069375732046528055470686564 4_
--R         949577509916867201889438090408354817931718593862797624551518983570793_
--R         048774424291488708829840324189200301436123314860200821443733790755311_
--R         243632919864895421704228949571290016119498807957023663865443069392027_
--R         148979688266712323356043491523434068924275280417338574817381189277066_
--R         143312396681216
--R       ,
```

```
--R
--R        21062607688234750738947986804860165962496071486906855387636837150206 39_
--R         68085864965079005588950564689330944709709993780218732909532589878524 7_
--R         24902071750498366048207515661873872451468533306001120296463516638135 1_
--R         54325598220025030528398108683711061484230702609121129792987689628568 1_
--R         83047905476005638076266490561846205530604781619178201158870378913898 8_
--R         1895
--R      /
--R        21062606094984641924721138048164741753419629532964341024139031423687 57_
--R         96768527388858559097596521177886218987288195394364024629735706195981 2_
--R         32610365979902512686325867656720234210687703171018424748418142328892 1_
--R         83768123706270847029570621848592886740077193782849920092376059331416 8_
--R         90100066637389634759811822855673103707202647449677622838376299392328 0_
--R         0768
--R       ]
--R     ,
--R
--R        2563013
--R     [- -------,
--R        2097152
--R
--R       -
--R           26113461767919277896986176932377577192382599630635417819227523304 40_
--R            18989966807292833849076862359320744212592598673381593224350480929 4_
--R            83752303023733723680666816744617300172727135331157124289 7
--R         /
--R           11652254005052225305839819160045891437572266102768589900087901348 19_
--R            91494092241375398397139401952343332040813992815318882949575545516 3_
--R            96341761930839597754479714023146923426903492193805559398 4
--R       ,
--R
--R        35725945502759172210965887296157882729985170546756032395781981410060 34_
--R         09173528282659062190230446696394197103892330452627332931637375745006 1_
--R         97898922861109769970872504662353 73
--R      /
--R        10395482693455989368770712448340260558008145511201705922005223665917 59_
--R         40965948644233914102945295026517998996010481187582253020534650513158 1_
--R         24390172472891738650147029663088 64
--R       ]
--R     ,
--R
--R        1715967
--R     [- -------,
--R        2097152
--R
--R       -
--R           42130935337843035210848395179770823903772615039695862248289984366 06_
--R            03065607635937456481377349837660312126782256580143620693951995146 5_
--R            18222580524697287410022543952491
--R         /
```

```
--R           944181414418537445864969203434922405243659747096625366393064196079 5_
--R            805882585493199840191699917659443264824641135187383583888147867340_
--R            193078576058203641958568223047 68
--R       ,
--R
--R         76358333471126442225156254244108312253474756690085893388341621725019 04_
--R          994376346730876809042845208919919925302105720971453918982731389072591_
--R          4035
--R      /
--R         26241887640860971997842976104780666339342304678958516022785809785037 84_
--R          549205788499019640602266966026891580103543567625039018629887141284916_
--R          75648
--R       ]
--R     ,
--R
--R          437701
--R     [- -------,
--R         2097152
--R
--R         16831069086383495883221723326542259135629863131819510314527501614414 97_
--R          473455328150721364868355579646781603507777199075077835213366484533654_
--R          91383623741304759
--R      /
--R         16831068680952133890017099827059136389630776687312261111677851880049 07_
--R          425226298680325887810962614140298597366984264887998908377068799998454_
--R          23381649008099328
--R       ,
--R
--R         49615501098350101864226810134221087359587148010037606397079680966469 12_
--R          826708472834443117239172191042492134509663124111 33
--R      /
--R         49615498727577383155091920782102090298528971186110971262363840408293 76_
--R          592619143131702548674647927183634921604824422154 24
--R       ]
--R     ,
--R
--R       222801
--R     [-------,
--R      2097152
--R
--R       -
--R           899488488040242826510759512197069142713604569254197827557300186521 3_
--R            759921588137716696126349101655220195142994932299137183241705867672_
--R            383477
--R         /
--R           116788999866502637217776510069188858270896996022993476969083575245 7_
--R            077779416435209473767866507769405888942764587718542434255625992456_
--R            372224
--R       ,
--R
```

```
--R       -
--R           23897048881331568783208015443738083956127715092084910198474529918 85_
--R            50954651952546783901661359399969388664003628357055232115503787129 1_
--R            458703265
--R         /
--R           53554872736450963260904032866899319059882254446854114332215938336 81_
--R            19295756283367146865429034074699365628592559911760212044618344314 5_
--R            479421952
--R       ]
--R     ,
--R
--R       765693
--R     [-------,
--R      2097152
--R
--R        85589692198167162678732447611781980887246989586166701402137657543220 02_
--R         30325168578611867833084020332883765433952341870491774951834077251289 9_
--R         000391009630373148561
--R      /
--R        29414424455330107909764284113763934998155802159458569179064525354957 23_
--R         01385681894170233022877989014129623672113815423199723891732215671196 5_
--R         2444639331719460159488
--R       ,
--R
--R       -
--R           20576182305825721012476503248602425611113025815435888088439236627 67_
--R            54938224165936271229077761280019292142057440894808519374368858276 2_
--R            2246433251878894899015
--R         /
--R           26715982033257355380979523535014502205763137598908350970917225206 42_
--R            71019877190266718394890628986371475967836029248394920461647153777 7_
--R            775324180661095366656
--R       ]
--R     ,
--R
--R      5743879
--R     [-------,
--R      2097152
--R
--R        10762888169689068479555463947735702081714567249426186140236631235747 68_
--R         96085043426397139807254659277266215883344979769861745539788756290007 2_
--R         98476800060834355318980169340872720504761255988923275756383052868895 3_
--R         53542180948277105891754260289006094194962087408300785836666945350176 6_
--R         24841488732463225
--R      /
--R        31317689570803179466484619400235520441903766134585849862285496319161 96_
--R         60161621978176561553253229474652964827643058381089407937456646075782 3_
--R         14688858119555602920851521883888320031865840746939942606326058982861 2_
--R         30923159666912970798648131985157194292723034062293402392348670304206 8_
--R         1530440845099008
```

```
--R       ,
--R
--R       -
--R           211328669918575091836412047556545843787017248986548599438982813533 5_
--R            264444665284557526492734931691731407872701432935503473348172076098_
--R            720545849008780077564160534317894688366119529739980502944162668550_
--R            098127961950496210221942878089359674925850594427768502251789758706_
--R            752831632503615
--R         /
--R           162761558493798758024290662434710458088914446616845971804315383940 8_
--R            372525533309808070363699585502216011211087103263609551026027769414_
--R            087391148126221168139781682587438075322591466131939975457200522349_
--R            838568964285634448018562038272378787354460106106141518010935617205_
--R            1706396253618176
--R       ]
--R     ,
--R
--R      19739877
--R     [--------,
--R       2097152
--R
--R       -
--R           299724993683270330379901580486152094921504038750070717770128576672 0_
--R            192530579422478953566024359860143101547801638082771611160372212874_
--R            847778035809872843149225484238365858013629341705321702582333350918_
--R            009601789937023985935304900460493389873837030853410347089908880814_
--R            853981132018464582458800615394770741699487295875960210750215891948_
--R            814476854871031530931295467332190133702671098200902282300510751860_
--R            718592845703027780739779652581386276223928699610680972802367 5
--R         /
--R           230843327485227859072891008119181102390650414132143264612393679487 3_
--R            933319270608960702138193417647898360620229519176632937631786851455_
--R            014766027206259022252505551741823688896883806636602574431760472240_
--R            292093196729475160247268834121141893318848728661844434927287285112_
--R            897080767552864895056585864033178565910387065006112801516403522741_
--R            037360990556054476949527059227070809593049491257519554708879259595_
--R            529299201108585608125566354854294715540316759795426563813539 84
--R       ,
--R
--R       -
--R           512818926354822848909627639786894008060093841066308045940796633584 5_
--R            009264109490520459825316250084723010047035024497436523038925818959_
--R            289312931584701353927621435434398674263047293909122850133851990696_
--R            490231566094371994333795070782624011727587749989296611277318372294_
--R            624207116537910436554574146082884701305543912620419354885410735940_
--R            157775896602822364575864611831512943973974715166920465061850603762_
--R            875162561958470524125872828391391946429139 55
--R         /
--R           228828193977843933053120879318129047118363109245536899038639082424 3_
--R            509463644236249773080647438987739144921607794682653851741189091711_
```

```
--R            74186814511497833728419182249767586835872948664473085662255268720 9_
--R            20372441180048140570283719831064229127567619577461444381599671350 2_
--R            62939174978359004147086012775237299648862774267248762248006326880 8_
--R            88932489185084249493434733760307593998026820848290485967817775144 4_
--R            65749979827872616963053217673201717237252096
--R       ]
--R     ]
--R                                          Type: List(List(Fraction(Integer)))
--E 13

--S 14 of 28
lpr := positiveSolve(lp)$pack
--R
--R
--R   (14)  []
--R                              Type: List(List(RealClosure(Fraction(Integer))))
--E 14

--S 15 of 28
f0 := x**3 + y + z + t- 1
--R
--R
--R                  3
--R   (15)  z + y + x  + t - 1
--R                                                  Type: Polynomial(Integer)
--E 15

--S 16 of 28
f1 := x + y**3 + z + t -1
--R
--R
--R              3
--R   (16)  z + y  + x + t - 1
--R                                                  Type: Polynomial(Integer)
--E 16

--S 17 of 28
f2 := x + y + z**3 + t-1
--R
--R
--R          3
--R   (17)  z  + y + x + t - 1
--R                                                  Type: Polynomial(Integer)
--E 17

--S 18 of 28
f3 := x + y + z + t**3 -1
--R
--R
--R                      3
```

```
--R   (18)  z + y + x + t  - 1
--R                                                    Type: Polynomial(Integer)
--E 18

--S 19 of 28
lf := [f0, f1, f2, f3]
--R
--R
--R   (19)
--R             3              3              3                              3
--R   [z + y + x  + t - 1,z + y  + x + t - 1,z  + y + x + t - 1,z + y + x + t  - 1]
--R                                              Type: List(Polynomial(Integer))
--E 19

--S 20 of 28
lts := triangSolve(lf)$pack
--R
--R
--R   (20)
--R   [
--R       2           3        3
--R     {t  + t + 1, z  - z - t  + t,
--R
--R                 3      2      2      3           6     3            3      2
--R         (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
--R       +
--R            6     3          9     6     3
--R         (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
--R       ,
--R      x + y + z}
--R     ,
--R
--R       16     13     10     7      4      2
--R     {t   - 6t   + 9t   + 4t  + 15t  - 54t  + 27,
--R
--R                      15            14             13            12            11
--R              4907232t   + 40893984t   - 115013088t   + 22805712t   + 36330336t
--R           +
--R                        10             9             8             7
--R              162959040t   - 159859440t  - 156802608t  + 117168768t
--R           +
--R                       6             5             4             3
--R              126282384t  - 129351600t  + 306646992t  + 475302816t
--R           +
--R                      2
--R              - 1006837776t  - 237269088t + 480716208
--R         *
--R            z
--R       +
--R             54       51        48      46         45        43          42
```

```
--R         48t   - 912t   + 8232t   - 72t   - 46848t   + 1152t   + 186324t
--R       +
--R                40          39         38          37           36          35
--R         - 3780t   - 543144t   - 3168t   - 21384t   + 1175251t   + 41184t
--R       +
--R                34           33           32           31           30
--R         278003t   - 1843242t   - 301815t   - 1440726t   + 1912012t
--R       +
--R                29           28           27           26            25
--R         1442826t   + 4696262t   - 922481t   - 4816188t   - 10583524t
--R       +
--R                 24            23            22           21            20
--R         - 208751t   + 11472138t   + 16762859t   - 857663t   - 19328175t
--R       +
--R                  19           18            17            16           15
--R         - 18270421t   + 4914903t   + 22483044t   + 12926517t   - 8605511t
--R       +
--R                  14           13           12           11          10
--R         - 17455518t   - 5014597t   + 8108814t   + 8465535t   + 190542t
--R       +
--R                 9          8          7          6          5          4
--R         - 4305624t  - 2226123t  + 661905t  + 1169775t  + 226260t  - 209952t
--R       +
--R                3
--R         - 141183t  + 27216t
--R       ,
--R
--R                3      2      2      3           6     3           3      2
--R         (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
--R       +
--R            6     3          9     6     3
--R         (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
--R       ,
--R                  3
--R      x + y + z + t  - 1}
--R     ,
--R              2                       2                     2
--R    {t,z - 1,y  - 1,x + y}, {t - 1,z,y  - 1,x + y}, {t - 1,z  - 1,z y + 1,x},
--R
--R       16     13     10     7      4      2
--R     {t   - 6t   + 9t   + 4t  + 15t  - 54t  + 27,
--R
--R                    29            28             27           26             25
--R             4907232t   + 40893984t   - 115013088t   - 1730448t   - 168139584t
--R           +
--R                        24             23             22              21
--R             738024480t   - 195372288t   + 315849456t   - 2567279232t
--R           +
--R                        20              19              18              17
--R             937147968t   + 1026357696t   + 4780488240t   - 2893767696t
```

```
--R           +
--R                          16              15              14              13
--R             - 5617160352t   - 3427651728t   + 5001100848t   + 8720098416t
--R           +
--R                         12             11              10             9
--R             2331732960t   - 499046544t   - 16243306272t   - 9748123200t
--R           +
--R                         8              7              6              5
--R             3927244320t  + 25257280896t  + 10348032096t  - 17128672128t
--R           +
--R                           4             3              2
--R             - 14755488768t  + 544086720t  + 10848188736t  + 1423614528t
--R           +
--R             - 2884297248
--R        *
--R           z
--R       +
--R              68        65         62       60          59        57          56
--R         - 48t   + 1152t   - 13560t   + 360t   + 103656t   - 7560t   - 572820t
--R       +
--R               54           53        52          51           50         49
--R         71316t   + 2414556t   + 2736t   - 402876t   - 7985131t   - 49248t
--R       +
--R                48            47          46          45            44
--R         1431133t   + 20977409t   + 521487t   - 2697635t   - 43763654t
--R       +
--R                  43           42            41            40            39
--R         - 3756573t   - 2093410t   + 71546495t   + 19699032t   + 35025028t
--R       +
--R                   38            37             36            35             34
--R         - 89623786t   - 77798760t   - 138654191t   + 87596128t   + 235642497t
--R       +
--R                  33            32             31             30             29
--R         349607642t   - 93299834t   - 551563167t   - 630995176t   + 186818962t
--R       +
--R                  28             27             26              25
--R         995427468t   + 828416204t   - 393919231t   - 1076617485t
--R       +
--R                    24             23              22              21
--R         - 1609479791t   + 595738126t   + 1198787136t   + 4342832069t
--R       +
--R                    20              19              18              17
--R         - 2075938757t   - 4390835799t   - 4822843033t   + 6932747678t
--R       +
--R                   16              15              14              13
--R         6172196808t   + 1141517740t   - 4981677585t   - 9819815280t
--R       +
--R                    12             11              10              9
--R         - 7404299976t   - 157295760t   + 29124027630t   + 14856038208t
--R       +
```

```
--R                      8              7             6              5
--R        - 16184101410t  - 26935440354t  - 3574164258t  + 10271338974t
--R      +
--R                   4             3             2
--R        11191425264t  + 6869861262t  - 9780477840t  - 3586674168t + 2884297248
--R      ,
--R
--R           3      3      2      6      3           9     6    3
--R        (3z  + (6t  - 6)z  + (6t  - 12t  + 3)z + 2t  - 6t  + t  + 3t)y
--R      +
--R           3      3      6      3      2      9      6      3          12     9
--R        (3t  - 3)z  + (6t  - 12t  + 6)z  + (4t  - 12t  + 11t  - 3)z + t   - 4t
--R      +
--R          6     3
--R        5t  - 2t
--R      ,
--R                  3
--R     x + y + z + t  - 1}
--R    ,
--R           2
--R   {t - 1,z  - 1,y,x + z},
--R
--R      8    7    6     5     4     3      2
--R    {t  + t  + t  - 2t  - 2t  - 2t  + 19t  + 19t - 8,
--R
--R                     7           6           5            4           3
--R            2395770t  + 3934440t  - 3902067t  - 10084164t  - 1010448t
--R          +
--R                      2
--R            32386932t  + 22413225t - 10432368
--R       *
--R          z
--R      +
--R                 7           6           5           4            3
--R        - 463519t  + 3586833t  + 9494955t  - 8539305t  - 33283098t
--R      +
--R                2
--R        35479377t  + 46263256t - 17419896
--R      ,
--R
--R              4      3      3       6      3      2          3
--R            3z  + (9t  - 9)z  + (12t  - 24t  + 9)z  + (- 152t  + 219t - 67)z
--R          +
--R                 6      4      3
--R            - 41t  + 57t  + 25t  - 57t + 16
--R       *
--R          y
--R      +
--R           3      4      6      3      3          3              2
--R        (3t  - 3)z  + (9t  - 18t  + 9)z  + (- 181t  + 270t - 89)z
```

```
--R       +
--R               6       4      3                     7      6      4       3
--R         (- 92t  + 135t  + 49t  - 135t + 43)z + 27t  - 27t  - 54t  + 396t
--R       +
--R         - 486t + 144
--R       ,
--R                    3
--R      x + y + z + t  - 1}
--R     ,
--R            3
--R    {t,z - t  + 1,y - 1,x - 1}, {t - 1,z,y,x}, {t,z - 1,y,x}, {t,z,y - 1,x},
--R    {t,z,y,x - 1}]
--R                                    Type: List(RegularChain(Integer,[x,y,z,t]))
--E 20

--S 21 of 28
univariateSolve(lf)$pack
--R
--R
--R   (21)
--R   [[complexRoots= ?,coordinates= [x - 1,y - 1,z + 1,t - %A]],
--R    [complexRoots= ?,coordinates= [x,y - 1,z,t - %A]],
--R    [complexRoots= ? - 1,coordinates= [x,y,z,t - %A]],
--R    [complexRoots= ?,coordinates= [x - 1,y,z,t - %A]],
--R    [complexRoots= ?,coordinates= [x,y,z - 1,t - %A]],
--R    [complexRoots= ? - 2,coordinates= [x - 1,y + 1,z,t - 1]],
--R    [complexRoots= ?,coordinates= [x + 1,y - 1,z,t - 1]],
--R    [complexRoots= ? - 1,coordinates= [x - 1,y + 1,z - 1,t]],
--R    [complexRoots= ? + 1,coordinates= [x + 1,y - 1,z - 1,t]],
--R
--R                      6     3     2
--R     [complexRoots= ?  - 2?  + 3?  - 3,
--R                           3                 3
--R      coordinates= [2x + %A  + %A - 1,2y + %A  + %A - 1,z - %A,t - %A]]
--R     ,
--R
--R                      5     3     2
--R     [complexRoots= ?  + 3?  - 2?  + 3? - 3,
--R                                    3
--R      coordinates= [x - %A,y - %A,z + %A  + 2%A - 1,t - %A]]
--R     ,
--R
--R                      4    3     2
--R     [complexRoots= ?  - ?  - 2?  + 3,
--R                           3                3                 3
--R      coordinates= [x + %A  - %A - 1,y + %A  - %A - 1,z - %A  + 2%A + 1,t - %A]]
--R     ,
--R    [complexRoots= ? + 1,coordinates= [x - 1,y - 1,z,t - %A]],
--R
--R                      6     3     2
```

```
--R     [complexRoots= ?  + 2?  + 3?  - 3,
--R                           3                          3
--R      coordinates= [2x - %A  - %A - 1,y + %A,2z - %A  - %A - 1,t + %A]]
--R     ,
--R
--R                     6      4      3      2
--R     [complexRoots= ?  + 12?  + 20?  - 45?  - 42? - 953,
--R
--R       coordinates =
--R                        5       4       3        2
--R         [12609x + 23%A  + 49%A  - 46%A  + 362%A  - 5015%A - 8239,
--R                        5       4       3        2
--R          25218y + 23%A  + 49%A  - 46%A  + 362%A  + 7594%A - 8239,
--R                        5       4       3        2
--R          25218z + 23%A  + 49%A  - 46%A  + 362%A  + 7594%A - 8239,
--R                        5       4       3        2
--R          12609t + 23%A  + 49%A  - 46%A  + 362%A  - 5015%A - 8239]
--R       ]
--R     ,
--R
--R                     5      3      2
--R     [complexRoots= ?  + 12?  - 16?  + 48? - 96,
--R                           3
--R      coordinates= [8x + %A  + 8%A - 8,2y - %A,2z - %A,2t - %A]]
--R     ,
--R
--R                     5    4     3     2
--R     [complexRoots= ?  + ?  - 5?  - 3?  + 9? + 3,
--R
--R       coordinates =
--R                 3                    3                    3
--R         [2x - %A  + 2%A - 1, 2y + %A  - 4%A + 1, 2z - %A  + 2%A - 1,
--R                 3
--R          2t - %A  + 2%A - 1]
--R       ]
--R     ,
--R
--R                     4     3     2
--R     [complexRoots= ?  - 3?  + 4?  - 6? + 13,
--R
--R       coordinates =
--R                  3      2                   3      2
--R         [9x - 2%A  + 4%A  - %A + 2, 9y + %A  - 2%A  + 5%A - 1,
--R                 3      2                   3      2
--R          9z + %A  - 2%A  + 5%A - 1, 9t + %A  - 2%A  - 4%A - 1]
--R       ]
--R     ,
--R
--R                     4      2
--R     [complexRoots= ?  - 11?  + 37,
```

```
--R
--R       coordinates =
--R                 2           2                 2           2
--R           [3x - %A  + 7,6y + %A  + 3%A - 7,3z - %A  + 7,6t + %A  - 3%A - 7]
--R       ]
--R     ,
--R    [complexRoots= ? + 1,coordinates= [x - 1,y,z - 1,t + 1]],
--R    [complexRoots= ? + 2,coordinates= [x,y - 1,z - 1,t + 1]],
--R    [complexRoots= ? - 2,coordinates= [x,y - 1,z + 1,t - 1]],
--R    [complexRoots= ?,coordinates= [x,y + 1,z - 1,t - 1]],
--R    [complexRoots= ? - 2,coordinates= [x - 1,y,z + 1,t - 1]],
--R    [complexRoots= ?,coordinates= [x + 1,y,z - 1,t - 1]],
--R
--R                     4     3      2
--R     [complexRoots= ?  + 5?  + 16?  + 30? + 57,
--R
--R       coordinates =
--R                     3       2                          3       2
--R         [151x + 15%A  + 54%A  + 104%A + 93, 151y - 10%A  - 36%A  - 19%A - 62,
--R                    3       2                         3       2
--R          151z - 5%A  - 18%A  - 85%A - 31, 151t - 5%A  - 18%A  - 85%A - 31]
--R       ]
--R     ,
--R
--R                     4    3     2
--R     [complexRoots= ?  - ?  - 2?  + 3,
--R                      3                  3                       3
--R      coordinates= [x - %A  + 2%A + 1,y + %A  - %A - 1,z - %A,t + %A  - %A - 1]]
--R     ,
--R
--R                     4     3     2
--R     [complexRoots= ?  + 2?  - 8?  + 48,
--R
--R       coordinates =
--R                 3                           3                  3
--R         [8x - %A  + 4%A - 8,2y + %A,8z + %A  - 8%A + 8,8t - %A  + 4%A - 8]
--R       ]
--R     ,
--R
--R                     5    4     3     2
--R     [complexRoots= ?  + ?  - 2?  - 4?  + 5? + 8,
--R                        3           3           3
--R      coordinates= [3x + %A  - 1,3y + %A  - 1,3z + %A  - 1,t - %A]]
--R     ,
--R                   3
--R    [complexRoots= ?  + 3? - 1,coordinates= [x - %A,y - %A,z - %A,t - %A]]]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynom
--E 21

--S 22 of 28
```

```
ts := lts.1
--R
--R
--R   (22)
--R     2          3        3
--R   {t  + t + 1, z  - z - t  + t,
--R
--R              3      2      2      3           6     3            3      2
--R       (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
--R     +
--R          6     3          9     6     3
--R       (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
--R     ,
--R    x + y + z}
--R                                        Type: RegularChain(Integer,[x,y,z,t])
--E 22

--S 23 of 28
univariateSolve(ts)$pack
--R
--R
--R   (23)
--R   [
--R                      4     3      2
--R     [complexRoots= ?  + 5?  + 16?  + 30? + 57,
--R
--R       coordinates =
--R                      3       2                          3       2
--R         [151x + 15%A  + 54%A  + 104%A + 93, 151y - 10%A  - 36%A  - 19%A - 62,
--R                     3       2                         3       2
--R          151z - 5%A  - 18%A  - 85%A - 31, 151t - 5%A  - 18%A  - 85%A - 31]
--R       ]
--R     ,
--R
--R                      4    3     2
--R     [complexRoots= ?  - ?  - 2?  + 3,
--R                           3                  3                       3
--R      coordinates= [x - %A  + 2%A + 1,y + %A  - %A - 1,z - %A,t + %A  - %A - 1]]
--R     ,
--R
--R                      4     3     2
--R     [complexRoots= ?  + 2?  - 8?  + 48,
--R
--R       coordinates =
--R                 3                           3                  3
--R         [8x - %A  + 4%A - 8,2y + %A,8z + %A  - 8%A + 8,8t - %A  + 4%A - 8]
--R       ]
--R     ]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer)
--E 23
```

```
--S 24 of 28
realSolve(ts)$pack
--R
--R
--R   (24)  []
--R                                Type: List(List(RealClosure(Fraction(Integer))))
--E 24

--S 25 of 28
lr2 := realSolve(lf)$pack
--R
--R
--R   (25)
--R   [[0,- 1,1,1], [0,0,1,0], [1,0,0,0], [0,0,0,1], [0,1,0,0], [1,0,%B37,- %B37],
--R    [1,0,%B38,- %B38], [0,1,%B35,- %B35], [0,1,%B36,- %B36], [- 1,0,1,1],
--R
--R     [%B32,
--R
--R          1     15    2     14    1     13    4     12   11     11    4     10
--R         -- %B32   + -- %B32   + -- %B32   - -- %B32   - -- %B32   - -- %B32
--R         27          27          27          27          27          27
--R       +
--R          1     9   14     8    1     7   2     6   1     5   2     4       3
--R         -- %B32  + -- %B32  + -- %B32  + - %B32  + - %B32  + - %B32  + %B32
--R         27          27          27          9          3          9
--R       +
--R         4     2
--R         - %B32  - %B32 - 2
--R         3
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B32   - -- %B32   - -- %B32   + -- %B32   + -- %B32   + -- %B32
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B32  - -- %B32  - -- %B32  - - %B32  - - %B32  - - %B32  - %B32
--R           54          27          54          9          6          9
--R       +
--R           2     2   1        3
--R         - - %B32  + - %B32 + -
--R           3          2        2
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B32   - -- %B32   - -- %B32   + -- %B32   + -- %B32   + -- %B32
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
```

```
--R         - -- %B32  - -- %B32  - -- %B32  - - %B32  - - %B32  - - %B32  - %B32
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B32  + - %B32 + -
--R           3         2        2
--R       ]
--R     ,
--R
--R     [%B33,
--R
--R          1     15    2     14    1     13    4     12   11     11    4     10
--R         -- %B33   + -- %B33   + -- %B33   - -- %B33   - -- %B33   - -- %B33
--R         27          27          27          27          27          27
--R       +
--R          1     9   14     8    1     7   2     6   1     5   2     4       3
--R         -- %B33  + -- %B33  + -- %B33  + - %B33  + - %B33  + - %B33  + %B33
--R         27         27         27         9         3         9
--R       +
--R         4     2
--R         - %B33  - %B33 - 2
--R         3
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B33   - -- %B33   - -- %B33   + -- %B33   + -- %B33   + -- %B33
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B33  - -- %B33  - -- %B33  - - %B33  - - %B33  - - %B33  - %B33
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B33  + - %B33 + -
--R           3         2        2
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B33   - -- %B33   - -- %B33   + -- %B33   + -- %B33   + -- %B33
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B33  - -- %B33  - -- %B33  - - %B33  - - %B33  - - %B33  - %B33
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B33  + - %B33 + -
--R           3         2        2
--R       ]
--R     ,
```

```
--R
--R     [%B34,
--R
--R          1     15    2     14    1     13    4     12   11     11    4     10
--R        -- %B34   + -- %B34   + -- %B34   - -- %B34   - -- %B34   - -- %B34
--R        27          27          27          27          27          27
--R       +
--R          1     9   14     8    1     7   2     6   1     5   2     4       3
--R        -- %B34  + -- %B34  + -- %B34  + - %B34  + - %B34  + - %B34  + %B34
--R        27         27         27         9         3         9
--R       +
--R        4     2
--R        - %B34  - %B34 - 2
--R        3
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R        - -- %B34   - -- %B34   - -- %B34   + -- %B34   + -- %B34   + -- %B34
--R          54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R        - -- %B34  - -- %B34  - -- %B34  - - %B34  - - %B34  - - %B34  - %B34
--R          54         27         54         9         6         9
--R       +
--R          2     2   1        3
--R        - - %B34  + - %B34 + -
--R          3         2        2
--R       ,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R        - -- %B34   - -- %B34   - -- %B34   + -- %B34   + -- %B34   + -- %B34
--R          54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R        - -- %B34  - -- %B34  - -- %B34  - - %B34  - - %B34  - - %B34  - %B34
--R          54         27         54         9         6         9
--R       +
--R          2     2   1        3
--R        - - %B34  + - %B34 + -
--R          3         2        2
--R       ]
--R     ,
--R    [- 1,1,0,1], [- 1,1,1,0],
--R
--R     [%B23,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R        - -- %B23   - -- %B23   - -- %B23   + -- %B23   + -- %B23   + -- %B23
--R          54          27          54          27          54          27
--R       +
```

```
--R           1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B23  - -- %B23  - -- %B23  - - %B23  - - %B23  - - %B23  - %B23
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B23  + - %B23 + -
--R           3         2        2
--R       ,
--R      %B30,
--R
--R                  1     15    1     14    1     13    2     12   11     11
--R         - %B30 + -- %B23   + -- %B23   + -- %B23   - -- %B23   - -- %B23
--R                  54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B23   + -- %B23  + -- %B23  + -- %B23  + - %B23  + - %B23
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B23  + - %B23  - - %B23 - -
--R         9         3         2        2
--R       ]
--R     ,
--R
--R     [%B23,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B23   - -- %B23   - -- %B23   + -- %B23   + -- %B23   + -- %B23
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B23  - -- %B23  - -- %B23  - - %B23  - - %B23  - - %B23  - %B23
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B23  + - %B23 + -
--R           3         2        2
--R       ,
--R      %B31,
--R
--R                  1     15    1     14    1     13    2     12   11     11
--R         - %B31 + -- %B23   + -- %B23   + -- %B23   - -- %B23   - -- %B23
--R                  54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B23   + -- %B23  + -- %B23  + -- %B23  + - %B23  + - %B23
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B23  + - %B23  - - %B23 - -
```

```
--R         9          3          2         2
--R       ]
--R     ,
--R
--R     [%B24,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B24   - -- %B24   - -- %B24   + -- %B24   + -- %B24   + -- %B24
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B24  - -- %B24  - -- %B24  - - %B24  - - %B24  - - %B24  - %B24
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B24  + - %B24 + -
--R           3         2        2
--R       ,
--R      %B28,
--R
--R                  1     15    1     14    1     13    2     12   11     11
--R         - %B28 + -- %B24   + -- %B24   + -- %B24   - -- %B24   - -- %B24
--R                  54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B24   + -- %B24  + -- %B24  + -- %B24  + - %B24  + - %B24
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B24  + - %B24  - - %B24 - -
--R         9         3         2        2
--R       ]
--R     ,
--R
--R     [%B24,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B24   - -- %B24   - -- %B24   + -- %B24   + -- %B24   + -- %B24
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B24  - -- %B24  - -- %B24  - - %B24  - - %B24  - - %B24  - %B24
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B24  + - %B24 + -
--R           3         2        2
--R       ,
--R      %B29,
--R
```

```
--R                  1     15    1     14    1     13    2     12   11     11
--R         - %B29 + -- %B24   + -- %B24   + -- %B24   - -- %B24   - -- %B24
--R                 54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B24   + -- %B24  + -- %B24  + -- %B24  + - %B24  + - %B24
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B24  + - %B24  - - %B24 - -
--R         9         3         2        2
--R       ]
--R     ,
--R
--R     [%B25,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B25   - -- %B25   - -- %B25   + -- %B25   + -- %B25   + -- %B25
--R           54          27          54          27          54          27
--R       +
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B25  - -- %B25  - -- %B25  - - %B25  - - %B25  - - %B25  - %B25
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B25  + - %B25 + -
--R           3         2        2
--R       ,
--R      %B26,
--R
--R                  1     15    1     14    1     13    2     12   11     11
--R         - %B26 + -- %B25   + -- %B25   + -- %B25   - -- %B25   - -- %B25
--R                 54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B25   + -- %B25  + -- %B25  + -- %B25  + - %B25  + - %B25
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B25  + - %B25  - - %B25 - -
--R         9         3         2        2
--R       ]
--R     ,
--R
--R     [%B25,
--R
--R            1     15    1     14    1     13    2     12   11     11    2     10
--R         - -- %B25   - -- %B25   - -- %B25   + -- %B25   + -- %B25   + -- %B25
--R           54          27          54          27          54          27
--R       +
```

```
--R            1     9    7     8    1     7   1     6   1     5   1     4       3
--R         - -- %B25  - -- %B25  - -- %B25  - - %B25  - - %B25  - - %B25  - %B25
--R           54         27         54         9         6         9
--R       +
--R           2     2   1        3
--R         - - %B25  + - %B25 + -
--R           3         2        2
--R       ,
--R      %B27,
--R
--R                   1     15    1     14    1     13    2     12   11     11
--R         - %B27 + -- %B25   + -- %B25   + -- %B25   - -- %B25   - -- %B25
--R                  54          27          54          27          54
--R       +
--R            2     10    1     9    7     8    1     7   1     6   1     5
--R         - -- %B25   + -- %B25  + -- %B25  + -- %B25  + - %B25  + - %B25
--R           27          54         27         54         9         6
--R       +
--R         1     4   2     2   1        1
--R         - %B25  + - %B25  - - %B25 - -
--R         9         3         2        2
--R       ]
--R     ,
--R    [1,%B21,- %B21,0], [1,%B22,- %B22,0], [1,%B19,0,- %B19], [1,%B20,0,- %B20],
--R            1     3   1   1     3   1   1     3   1
--R    [%B17,- - %B17  + -,- - %B17  + -,- - %B17  + -],
--R            3         3   3         3   3         3
--R            1     3   1   1     3   1   1     3   1
--R    [%B18,- - %B18  + -,- - %B18  + -,- - %B18  + -]]
--R            3         3   3         3   3         3
--R                              Type: List(List(RealClosure(Fraction(Integer))))
--E 25

--S 26 of 28
#lr2
--R
--R
--R   (26)  27
--R                                                    Type: PositiveInteger
--E 26

--S 27 of 28
lpr2 := positiveSolve(lf)$pack
--R
--R
--R                  1     3   1   1     3   1   1     3   1
--R   (27)  [[%B40,- - %B40  + -,- - %B40  + -,- - %B40  + -]]
--R                  3         3   3         3   3         3
--R                               Type: List(List(RealClosure(Fraction(Integer))))
--E 27
```

```
--S 28 of 28
[approximate(r,1/10**21)::Float for r in lpr2.1]
--R
--R
--R   (28)
--R   [0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291,
--R    0.3221853546 2608559291]
--R                                                        Type: List(Float)
--E 28
)spool
)lisp (bye)
```

———-

— ZeroDimensionalSolvePackage.help —

```
=====================================================================
ZeroDimensionalSolvePackage examples
=====================================================================

The ZeroDimensionalSolvePackage package constructor provides
operations for computing symbolically the complex or real roots of
zero-dimensional algebraic systems.

The package provides no multiplicity information (i.e. some returned
roots may be double or higher) but only distinct roots are returned.

Complex roots are given by means of univariate representations of
irreducible regular chains.  These representations are computed by the
univariateSolve operation (by calling the
InternalRationalUnivariateRepresentationPackage package constructor
which does the job).

Real roots are given by means of tuples of coordinates lying in the
RealClosure of the coefficient ring.  They are computed by the
realSolve and positiveSolve operations.  The former computes all the
solutions of the input system with real coordinates whereas the later
concentrate on the solutions with (strictly) positive coordinates.  In
both cases, the computations are performed by the RealClosure constructor.

Both computations of complex roots and real roots rely on triangular
decompositions.  These decompositions can be computed in two different
ways.  First, by a applying the zeroSetSplit operation from the REGSET
domain constructor.  In that case, no Groebner bases are computed.
This strategy is used by default.  Secondly, by applying the zeroSetSplit
from LEXTRIPK.  To use this later strategy with the operations
univariateSolve, realSolve and positiveSolve one just needs to use an
extra boolean argument.
```

```
Note that the way of understanding triangular decompositions is
detailed in the example of the RegularTriangularSet constructor.

The ZeroDimensionalSolvePackage constructor takes three arguments.The
first one R is the coefficient ring; it must belong to the categories
OrderedRing, EuclideanDomain, CharacteristicZero and RealConstant. This
means essentially that R is Integer or Fraction(Integer).  The second
argument ls is the list of variables involved in the systems to solve.
The third one MUST BE concat(ls,s) where s is an additional symbol used
for the univariate representations. The abbreviation for
ZeroDimensionalSolvePackage is ZDSOLVE.

We illustrate now how to use the constructor ZDSOLVE by two examples:
the Arnborg and Lazard system and the L-3 system (Aubry and Moreno
Maza).  Note that the use of this package is also demonstrated in the
example of the LexTriangularPackage constructor.

Define the coefficient ring.

  R := Integer
    Integer
                        Type: Domain

Define the lists of variables:

  ls : List Symbol := [x,y,z,t]
    [x,y,z,t]
                        Type: List Symbol

and:

  ls2 : List Symbol := [x,y,z,t,new()$Symbol]
    [x,y,z,t,%A]
                        Type: List Symbol

Call the package:

  pack := ZDSOLVE(R,ls,ls2)
    ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])
                        Type: Domain

Define a polynomial system (Arnborg-Lazard)

  p1 := x**2*y*z + x*y**2*z + x*y*z**2 + x*y*z + x*y + x*z + y*z
        2       2     2
    x y z  + (x y  + (x  + x + 1)y + x)z + x y
                        Type: Polynomial Integer

  p2 := x**2*y**2*z + x*y**2*z**2 + x**2*y*z + x*y*z + y*z + x + z
```

```
      2 2     2 2     2
   x y z  + (x y  + (x  + x + 1)y + 1)z + x
                      Type: Polynomial Integer

  p3 := x**2*y**2*z**2 + x**2*y**2*z + x*y**2*z + x*y*z + x*z + z + 1
     2 2 2      2      2
    x y z  + ((x  + x)y  + x y + x + 1)z + 1
                      Type: Polynomial Integer

  lp := [p1, p2, p3]
        2       2     2
   [x y z  + (x y  + (x  + x + 1)y + x)z + x y,
      2 2     2 2     2
    x y z  + (x y  + (x  + x + 1)y + 1)z + x,
     2 2 2      2      2
    x y z  + ((x  + x)y  + x y + x + 1)z + 1]
                      Type: List Polynomial Integer
```

Note that these polynomials do not involve the variable t; we will use it in the second example.

First compute a decomposition into regular chains (i.e. regular triangular sets).

```
  triangSolve(lp)$pack
   [
    {
         20     19      18      17       16      15       14       13       12
        z   - 6z   - 41z   + 71z   + 106z   + 92z   + 197z   + 145z   + 257z
      +
           11       10       9       8       7       6      5       4      3
        278z   + 201z   + 278z  + 257z  + 145z  + 197z  + 92z  + 106z  + 71z
      +
            2
        - 41z  - 6z + 1
      ,
                    19            18             17             16
          14745844z   + 50357474z   - 130948857z   - 185261586z
        +
                    15             14             13             12
          - 180077775z   - 338007307z   - 275379623z   - 453190404z
        +
                    11             10             9             8
          - 474597456z   - 366147695z   - 481433567z  - 430613166z
        +
                    7             6             5             4
          - 261878358z  - 326073537z  - 163008796z  - 177213227z
        +
                    3           2
          - 104356755z  + 65241699z  + 9237732z - 1567348
```

```
       *
          y
       +
                 19           18            17            16            15
         1917314z   + 6508991z   - 16973165z   - 24000259z   - 23349192z
       +
                 14            13            12            11            10
         - 43786426z   - 35696474z   - 58724172z   - 61480792z   - 47452440z
       +
                 9            8            7            6            5
         - 62378085z  - 55776527z  - 33940618z  - 42233406z  - 21122875z
       +
                 4            3           2
         - 22958177z  - 13504569z  + 8448317z  + 1195888z - 202934
       ,
         3       2       3    2               2              2
       ((z  - 2z)y  + (- z  - z  - 2z - 1)y - z  - z + 1)x + z  - 1}
      ]
                      Type: List RegularChain(Integer,[x,y,z,t])
```

We can see easily from this decomposition (consisting of a single regular chain) that the input system has 20 complex roots.

Then we compute a univariate representation of this regular chain.

```
  univariateSolve(lp)$pack
   [
    [
      complexRoots =
           12      11      10     9     8      7      6      5     4     3
          ?   - 12?   + 24?   + 4?  - 9?  + 27?  - 21?  + 27?  - 9?  + 4?
        +
             2
          24?  - 12? + 1
      ,
       coordinates =
        [
                      11         10         9         8         7         6
          63x + 62%A   - 721%A   + 1220%A  + 705%A  - 285%A  + 1512%A
        +
                   5         4        3         2
          - 735%A  + 1401%A  - 21%A  + 215%A  + 1577%A - 142
        ,
                     11         10         9         8         7         6
          63y - 75%A   + 890%A   - 1682%A  - 516%A  + 588%A  - 1953%A
        +
                  5         4         3         2
          1323%A  - 1815%A  + 426%A  - 243%A  - 1801%A + 679
        ,
         z - %A]
```

```
      ]
    ,
                    6    5    4    3    2
    [complexRoots= ?  + ?  + ?  + ?  + ?  + ? + 1,
                         5       3
     coordinates= [x - %A ,y - %A ,z - %A]]
    ,
                  2
   [complexRoots= ?  + 5? + 1,coordinates= [x - 1,y - 1,z - %A]]]
           Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
                             coordinates: List Polynomial Integer)
```

We see that the zeros of our regular chain are split into three components. This is due to the use of univariate polynomial factorization.

Each of these components consist of two parts. The first one is an irreducible univariate polynomial p(?) which defines a simple algebraic extension of the field of fractions of R. The second one consists of multivariate polynomials pol1(x,%A), pol2(y,%A) and pol3(z,%A). Each of these polynomials involve two variables: one is an indeterminate x, y or z of the input system lp and the other is %A which represents any root of p(?). Recall that this %A is the last element of the third parameter of ZDSOLVE. Thus any complex root ? of p(?) leads to a solution of the input system lp by replacing %A by this ? in pol1(x,%A), pol2(y,%A) and pol3(z,%A). Note that the polynomials pol1(x,%A), pol2(y,%A) and pol3(z,%A) have degree one w.r.t. x, y or z respectively. This is always the case for all univariate representations. Hence the operation univariateSolve replaces a system of multivariate polynomials by a list of univariate polynomials, what justifies its name. Another example of univariate representations illustrates the LexTriangularPackage package constructor.

We now compute the solutions with real coordinates:

```
  lr := realSolve(lp)$pack
   [
    [%B1,
        1184459    19   2335702    18   5460230    17   79900378    16
        ------- %B1   - ------- %B1   - ------- %B1   + -------- %B1
        1645371          548457          182819          1645371
      +
        43953929    15   13420192    14   553986    13   193381378    12
        -------- %B1   + -------- %B1   + ------ %B1   + --------- %B1
         548457           182819           3731           1645371
      +
        35978916    11   358660781    10   271667666    9   118784873    8
        -------- %B1   + --------- %B1   + --------- %B1  + --------- %B1
         182819           1645371           1645371           548457
      +
        337505020    7   1389370    6   688291    5   3378002    4
```

```
    --------- %B1  + ------- %B1  + ------ %B1  + ------- %B1
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B1  + -------- %B1  - ---- %B1 - -------
     1645371          548457          343       1645371
  ,
      91729    19   487915    18   4114333    17   1276987    16
    - ------ %B1   + ------ %B1   + ------- %B1   - ------- %B1
      705159         705159          705159          235053
  +
      13243117    15   16292173    14   26536060    13   722714    12
    - -------- %B1   - -------- %B1   - -------- %B1   - ------ %B1
       705159           705159           705159           18081
  +
      5382578    11   15449995    10   14279770    9   6603890    8
    - ------- %B1   - -------- %B1   - -------- %B1  - ------- %B1
       100737          235053           235053          100737
  +
      409930    7   37340389    6   34893715    5   26686318    4
    - ------ %B1  - -------- %B1  - -------- %B1  - -------- %B1
       6027          705159          705159          705159
  +
      801511    3   17206178    2   4406102       377534
    - ------ %B1  - -------- %B1  - ------- %B1 + ------
       26117         705159          705159       705159
  ]
,
 [%B2,
    1184459    19   2335702    18   5460230    17   79900378    16
    ------- %B2   - ------- %B2   - ------- %B2   + -------- %B2
    1645371          548457          182819          1645371
  +
    43953929    15   13420192    14   553986    13   193381378    12
    -------- %B2   + -------- %B2   + ------ %B2   + ---------- %B2
     548457           182819           3731           1645371
  +
    35978916    11   358660781    10   271667666    9   118784873    8
    -------- %B2   + ---------- %B2   + ---------- %B2  + ---------- %B2
     182819           1645371           1645371           548457
  +
    337505020    7   1389370    6   688291    5   3378002    4
    --------- %B2  + ------- %B2  + ------ %B2  + ------- %B2
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B2  + -------- %B2  - ---- %B2 - -------
     1645371          548457          343       1645371
  ,
      91729    19   487915    18   4114333    17   1276987    16
```

```
    - ------ %B2   + ------ %B2   + ------- %B2   - ------- %B2
      705159         705159          705159          235053
  +
      13243117    15   16292173    14   26536060    13   722714    12
    - -------- %B2   - -------- %B2   - -------- %B2   - ------ %B2
       705159           705159           705159           18081
  +
      5382578    11   15449995    10   14279770    9   6603890    8
    - ------- %B2   - -------- %B2   - -------- %B2  - ------- %B2
       100737           235053           235053          100737
  +
      409930    7   37340389    6   34893715    5   26686318    4
    - ------ %B2  - -------- %B2  - -------- %B2  - -------- %B2
       6027          705159          705159          705159
  +
      801511    3   17206178    2   4406102       377534
    - ------ %B2  - -------- %B2  - ------- %B2 + ------
       26117         705159          705159       705159
  ]
,
  [%B3,
    1184459    19   2335702    18   5460230    17   79900378    16
    ------- %B3   - ------- %B3   - ------- %B3   + -------- %B3
    1645371          548457          182819          1645371
  +
    43953929    15   13420192    14   553986    13   193381378    12
    -------- %B3   + -------- %B3   + ------ %B3   + --------- %B3
     548457           182819           3731           1645371
  +
    35978916    11   358660781    10   271667666    9   118784873    8
    -------- %B3   + --------- %B3   + --------- %B3  + --------- %B3
     182819           1645371          1645371          548457
  +
    337505020    7   1389370    6   688291    5   3378002    4
    --------- %B3  + ------- %B3  + ------ %B3  + ------- %B3
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B3  + -------- %B3  - ---- %B3 - -------
     1645371          548457          343       1645371
  ,
       91729    19   487915    18   4114333    17   1276987    16
    - ------ %B3   + ------ %B3   + ------- %B3   - ------- %B3
      705159         705159          705159          235053
  +
      13243117    15   16292173    14   26536060    13   722714    12
    - -------- %B3   - -------- %B3   - -------- %B3   - ------ %B3
       705159           705159           705159           18081
  +
      5382578    11   15449995    10   14279770    9   6603890    8
```

```
    - ------- %B3   - -------- %B3   - -------- %B3  - ------- %B3
       100737          235053           235053          100737
  +
       409930    7   37340389    6   34893715    5   26686318    4
    - ------ %B3  - -------- %B3  - -------- %B3  - -------- %B3
       6027          705159          705159          705159
  +
       801511    3   17206178    2   4406102       377534
    - ------ %B3  - -------- %B3  - ------- %B3 + ------
       26117         705159          705159       705159
  ]
,
  [%B4,
     1184459    19   2335702    18   5460230    17   79900378    16
     ------- %B4   - ------- %B4   - ------- %B4   + -------- %B4
     1645371          548457          182819          1645371
   +
     43953929    15   13420192    14   553986    13   193381378    12
     -------- %B4   + -------- %B4   + ------ %B4   + --------- %B4
      548457           182819           3731           1645371
   +
     35978916    11   358660781    10   271667666    9   118784873    8
     -------- %B4   + --------- %B4   + --------- %B4  + --------- %B4
      182819           1645371           1645371           548457
   +
     337505020    7   1389370    6   688291    5   3378002    4
     --------- %B4  + ------- %B4  + ------ %B4  + ------- %B4
      1645371          11193          4459          42189
   +
     140671876    3   32325724    2   8270       9741532
     --------- %B4  + -------- %B4  - ---- %B4 - -------
      1645371          548457          343       1645371
  ,
       91729    19   487915    18   4114333    17   1276987    16
     - ------ %B4   + ------ %B4   + ------- %B4   - ------- %B4
       705159         705159          705159          235053
   +
       13243117    15   16292173    14   26536060    13   722714    12
     - -------- %B4   - -------- %B4   - -------- %B4   - ------ %B4
        705159           705159           705159           18081
   +
       5382578    11   15449995    10   14279770    9   6603890    8
     - ------- %B4   - -------- %B4   - -------- %B4  - ------- %B4
       100737          235053           235053          100737
   +
       409930    7   37340389    6   34893715    5   26686318    4
     - ------ %B4  - -------- %B4  - -------- %B4  - -------- %B4
       6027          705159          705159          705159
   +
       801511    3   17206178    2   4406102       377534
```

```
   - ------ %B4  - -------- %B4  - ------- %B4 + ------
      26117          705159          705159       705159
 ]
,
 [%B5,
   1184459    19   2335702    18   5460230    17   79900378    16
   ------- %B5   - ------- %B5   - ------- %B5   + -------- %B5
   1645371          548457          182819          1645371
 +
   43953929    15   13420192    14   553986    13   193381378    12
   -------- %B5   + -------- %B5   + ------ %B5   + --------- %B5
    548457           182819           3731           1645371
 +
   35978916    11   358660781    10   271667666    9   118784873    8
   -------- %B5   + --------- %B5   + --------- %B5  + --------- %B5
    182819           1645371           1645371           548457
 +
   337505020    7   1389370    6   688291    5   3378002    4
   --------- %B5  + ------- %B5  + ------ %B5  + ------- %B5
    1645371          11193          4459          42189
 +
   140671876    3   32325724    2   8270       9741532
   --------- %B5  + -------- %B5  - ---- %B5 - -------
    1645371          548457          343       1645371
 ,
      91729    19   487915    18   4114333    17   1276987    16
   - ------ %B5   + ------ %B5   + ------- %B5   - ------- %B5
     705159          705159          705159          235053
 +
     13243117    15   16292173    14   26536060    13   722714    12
   - -------- %B5   - -------- %B5   - -------- %B5   - ------ %B5
      705159           705159           705159           18081
 +
     5382578    11   15449995    10   14279770    9   6603890    8
   - ------- %B5   - -------- %B5   - -------- %B5  - ------- %B5
      100737           235053           235053          100737
 +
     409930    7   37340389    6   34893715    5   26686318    4
   - ------ %B5  - -------- %B5  - -------- %B5  - -------- %B5
      6027           705159          705159          705159
 +
     801511    3   17206178    2   4406102       377534
   - ------ %B5  - -------- %B5  - ------- %B5 + ------
      26117          705159          705159       705159
 ]
,
 [%B6,
   1184459    19   2335702    18   5460230    17   79900378    16
   ------- %B6   - ------- %B6   - ------- %B6   + -------- %B6
   1645371          548457          182819          1645371
```

```
  +
    43953929    15   13420192    14   553986    13   193381378    12
    -------- %B6   + -------- %B6   + ------ %B6   + --------- %B6
     548457           182819           3731           1645371
  +
    35978916    11   358660781    10   271667666    9   118784873    8
    -------- %B6   + --------- %B6   + --------- %B6  + --------- %B6
     182819           1645371           1645371           548457
  +
    337505020    7   1389370    6   688291    5   3378002    4
    --------- %B6  + ------- %B6  + ------ %B6  + ------- %B6
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B6  + -------- %B6  - ---- %B6 - -------
     1645371          548457          343       1645371
  ,
       91729    19   487915    18   4114333    17   1276987    16
    - ------ %B6   + ------ %B6   + ------- %B6   - ------- %B6
      705159         705159          705159          235053
  +
      13243117    15   16292173    14   26536060    13   722714    12
    - -------- %B6   - -------- %B6   - -------- %B6   - ------ %B6
       705159           705159           705159           18081
  +
      5382578    11   15449995    10   14279770    9   6603890    8
    - ------- %B6   - -------- %B6   - -------- %B6  - ------- %B6
       100737           235053           235053          100737
  +
      409930    7   37340389    6   34893715    5   26686318    4
    - ------ %B6  - -------- %B6  - -------- %B6  - -------- %B6
       6027           705159          705159          705159
  +
      801511    3   17206178    2   4406102       377534
    - ------ %B6  - -------- %B6  - ------- %B6 + ------
       26117          705159          705159       705159
  ]
,
  [%B7,
    1184459    19   2335702    18   5460230    17   79900378    16
    ------- %B7   - ------- %B7   - ------- %B7   + -------- %B7
    1645371          548457          182819          1645371
  +
    43953929    15   13420192    14   553986    13   193381378    12
    -------- %B7   + -------- %B7   + ------ %B7   + --------- %B7
     548457           182819           3731           1645371
  +
    35978916    11   358660781    10   271667666    9   118784873    8
    -------- %B7   + --------- %B7   + --------- %B7  + --------- %B7
     182819           1645371           1645371           548457
```

```
  +
    337505020    7   1389370    6   688291    5   3378002    4
    --------- %B7  + ------- %B7  + ------ %B7  + ------- %B7
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B7  + -------- %B7  - ---- %B7 - -------
     1645371          548457          343       1645371
  ,
       91729    19   487915    18   4114333    17   1276987    16
    - ------ %B7   + ------ %B7   + ------- %B7   - ------- %B7
      705159         705159          705159          235053
  +
      13243117    15   16292173    14   26536060    13   722714    12
    - -------- %B7   - -------- %B7   - -------- %B7   - ------ %B7
       705159           705159           705159           18081
  +
      5382578    11   15449995    10   14279770    9   6603890    8
    - ------- %B7   - -------- %B7   - -------- %B7  - ------- %B7
       100737          235053           235053          100737
  +
      409930    7   37340389    6   34893715    5   26686318    4
    - ------ %B7  - -------- %B7  - -------- %B7  - -------- %B7
       6027          705159          705159          705159
  +
      801511    3   17206178    2   4406102       377534
    - ------ %B7  - -------- %B7  - ------- %B7 + ------
       26117         705159          705159       705159
  ]
,
  [%B8,
    1184459    19   2335702    18   5460230    17   79900378    16
    ------- %B8   - ------- %B8   - ------- %B8   + -------- %B8
    1645371          548457          182819          1645371
  +
    43953929    15   13420192    14   553986    13   193381378    12
    -------- %B8   + -------- %B8   + ------ %B8   + --------- %B8
     548457           182819           3731           1645371
  +
    35978916    11   358660781    10   271667666    9   118784873    8
    -------- %B8   + --------- %B8   + --------- %B8  + --------- %B8
     182819           1645371           1645371           548457
  +
    337505020    7   1389370    6   688291    5   3378002    4
    --------- %B8  + ------- %B8  + ------ %B8  + ------- %B8
     1645371          11193          4459          42189
  +
    140671876    3   32325724    2   8270       9741532
    --------- %B8  + -------- %B8  - ---- %B8 - -------
     1645371          548457          343       1645371
```

```
      ,
           91729     19   487915     18   4114333     17   1276987     16
        - ------ %B8   + ------ %B8   + ------- %B8   - ------- %B8
          705159          705159           705159           235053
      +
          13243117     15   16292173     14   26536060     13   722714     12
        - -------- %B8   - -------- %B8   - -------- %B8   - ------ %B8
           705159           705159           705159           18081
      +
          5382578     11   15449995     10   14279770     9   6603890     8
        - ------- %B8   - -------- %B8   - -------- %B8  - ------- %B8
           100737           235053           235053           100737
      +
          409930     7   37340389     6   34893715     5   26686318     4
        - ------ %B8  - -------- %B8  - -------- %B8  - -------- %B8
           6027           705159           705159           705159
      +
          801511     3   17206178     2   4406102       377534
        - ------ %B8  - -------- %B8  - ------- %B8 + ------
           26117          705159           705159       705159
      ]
    ]
                    Type: List List RealClosure Fraction Integer
```

The number of real solutions for the input system is:

```
  # lr
    8
                    Type: PositiveInteger
```

Each of these real solutions is given by a list of elements in RealClosure(R). In these 8 lists, the first element is a value of z, the second of y and the last of x. This is logical since by setting the list of variables of the package to [x,y,z,t] we mean that the elimination ordering on the variables is t < z < y < x. Note that each system treated by the ZDSOLVE package constructor needs only to be zero-dimensional w.r.t. the variables involved in the system it-self and not necessarily w.r.t. all the variables used to define the package.

We can approximate these real numbers as follows. This computation takes between 30 sec. and 5 min, depending on your machine.

```
  [ [approximate(r,1/1000000) for r in point] for point in lr]
   [
        10048059
     [- --------,
         2097152

        450305731698538794352439791383896641459673197621176821933588120838551 6_
         314058924567176091423629695777403099833360761048898228916578137094309_
```

```
    838597331137202584846939132376157019506760357601165917454986815382098_
    789094851523420392811293126141329856546977145464661495487825919941188_
    447041722440491921567263542158028061437758844364634410045253024786561_
    923163288214175
   /
    450305728302524548851651180698582663508310069375732046528055470686564 4_
    949577509916867201889438090408354817931718593862797624551518983570793_
    048774424291488708829840324189200301436123314860200821443733790755311_
    243632919864895421704228949571290016119498807957023663865443069392027_
    148979688266712323356043491523434068924275280417338574817381189277066_
    143312396681216
   ,

    210626076882347507389479868048601659624960714869068553876368371502063 9_
    680858649650790055889505646893309447097099937802187329095325898785247_
    249020717504983660482075156618738724514685333060011202964635166381351_
    543255982200250305283981086837110614842307026091211297929876896285681_
    830479054760056380762664905618462055306047816191782011588703789138988_
    1895
   /
    210626060949846419247211380481647417534196295329643410241390314236875 7_
    967685273888585590975965211778862189872881953943640246297357061959812_
    326103659799025126863258676567202342106877031710184247484181423288921_
    837681237062708470295706218485928867400771937828499200923760593314168_
    901000666373896347598118228556731037072026474496776228383762993923280_
    0768
   ]
  ,

    2563013
  [- -------,
    2097152

   -
       261134617679192778969861769323775771923825996306354178192275233044 0_
        189899668072928338490768623593207442125925986733815932243504809294_
        837523030237337236806668167446173001727271353311571242897
     /
       116522540050522253058398191600458914375722661027685899000879013481 9_
        914940922413753983971394019523433320408139928153188829495755455163_
        963417619308395977544797140231469234269034921938055593984
   ,

    357259455027591722109658872961578827299851705467560323957819814100603 4_
    091735282826590621902304466963941971038923304526273329316373757450061_
    978989228611097699708725046623537 3
   /
    103954826934559893687707124483402605580081455112017059220052236659175 9_
    409659486442339141029452950265179989960104811875822530205346505131581_
    243901724728917386501470296630886 4
```

```
      ]
    ,

      1715967
    [- -------,
      2097152

     -
        421309353378430352108483951797708239037726150396958622482899843660 6_
         030656076359374564813773498376603121267822565801436206939519951465_
         182225805246972874100225439524 91
       /
        944181414418537445864969203434922405243659747096625366393064196079 5_
         805882585493199840191699917659443264824641135187383583888147867340_
         193078576058203641958568223047 68

     ,

      763583334711264422251562542441083122534747566900858933883416217250190 4_
       994376346730876809042845208919919925302105720971453918982731389072591_
       4035
    /
      262418876408609719978429761047806663393423046789585160227858097850378 4_
       549205788499019640602266966026891580103543567625039018629887141284916_
       75648
     ]
    ,

       437701
    [- -------,
      2097152

      168310690863834958832217233265422591356298631318195103145275016144149 7_
       473455328150721364868355579646781603507777199075077835213366484533654_
       91383623741304759
    /
      168310686809521338900170998270591363896307766873122611116778518800490 7_
       425226298680325887810962614140298597366984264887998908377068799998454_
       23381649008099328
     ,

      496155010983501018642268101342210873595871480100376063970796809664691 2_
       826708472834443117239172191042492134509663124111 33
    /
      496154987275773831550919207821020902985289711861109712623638404082937 6_
       592619143131702548674647927183634921604824422154 24
     ]
    ,

     222801
    [-------,
```

```
 2097152

  -
      899488488040242826510759512197069142713604569254197827557300186521 3_
       759921588137716696126349101655220195142994932299137183241705867672_
       383477
    /
      116788999866502637217776510069188858270896996022993476969083575245 7_
       077779416435209473767866507769405888942764587718542434255625992456_
       372224
  ,

  -
      238970488813315687832080154437380839561277150920849101984745299188 5_
       509546519525467839016613593999693886640036283570552321155037871291_
       458703265
    /
      535548727364509632609040328668993190598822544468541143322159383368 1_
       192957562833671468654290340746993656285925599117602120446183443145_
       479421952
  ]
,

  765693
[-------,
 2097152

   855896921981671626787324476117819808872469895861667014021376575432200 2_
    303251685786118678330840203328837654339523418704917749518340772512899_
    000391009630373148561
 /
   294144244553301079097642841137639349981558021594585691790645253549572 3_
    013856818941702330228779890141296236721138154231997238917322156711965_
    2444639331719460159488
  ,

  -
      205761823058257210124765032486024256111130258154358808843923662767_
       549382241659362712290777612800192921420574408948085193743688582762_
       224643325187889489901 5
    /
      267159820332573553809795235350145022057631375989083509709172252064 2_
       710198771902667183948906289863714759678360292483949204616471537777_
       775324180661095366656
  ]
,

 5743879
[-------,
 2097152
```

```
    1076288816968906847955546394773570208171456724942618614023663123574768_
     9608504342639713980725465927726621588334497976986174553978875629000 72_
     9847680006083435531898016934087272050476125598892327575638305286889 53_
     5354218094827710589175426028900609419496208740830078583666694535017 66_
     24841488732463225
   /
    3131768957080317946648461940023552044190376613458584986228549631916196_
     6016162197817656155325322947465296482764305838108940793745664607578 23_
     1468885811955560292085152188388832003186584074693994260632605898286 12_
     3092315966691297079864813198515719429272303406229340239234867030420 68_
     1530440845099008
   ,

   -
       211328669918575091836412047556545843787017248986548599438982813533 5_
        264444665284557526492734931691731407872701432935503473348172076098_
        720545849008780077564160534317894688366119529739980502944162668550_
        098127961950496210221942878089359674925850594427768502251789758706_
        752831632503615
     /
       162761558493798758024290662434710458088914446616845971804315383940 8_
        372525533309808070363699585502216011211087103263609551026027769414_
        087391148126221168139781682587438075322591466131939975457200522349_
        838568964285634448018562038272378787354460106106141518010935617205_
        1706396253618176
   ]
 ,

  19739877
 [--------,
   2097152

   -
       299724993683270330379901580486152094921504038750070717770128576672 0_
        192530579422478953566024359860143101547801638082771611160372212874_
        847778035809872843149225484238365858013629341705321702582333350918_
        009601789937023985935304900460493389873837030853410347089908880814_
        853981132018464582458800615394770741699487295875960210750215891948_
        814476854871031530931295467332190133702671098200902282300510751860_
        718592845703027780739779652581386276223928699610680972802367 5
     /
       230843327485227859072891008119181102390650414132143264612393679487 3_
        933319270608960702138193417647898360620229519176632937631786851455_
        014766027206259022252505551741823688896883806636602574431760472240_
        292093196729475160247268834121141893318848728661844434927287285112_
        897080767552864895056585864033178565910387065006112801516403522741_
        037360990556054476949527059227070809593049491257519554708879259595_
        529299201108585608125566354854294715540316759795426563813539 84
   ,
```

```
      -
          5128189263548228489096276397868940080600938410663080459407966335845_
           0092641094905204598253162500847230100470350244974365230389258189 59_
           2893129315847013539276214354343986742630472939091228501338519906 96_
           4902315660943719943337950707826240117275877499892966112773183722 94_
           6242071165379104365545741460828847013055439126204193548854107359 40_
           1577758966028223645758646118315129439739747151669204650618506037 62_
           8751625619584705241258728283913919464291395 5
         /
          2288281939778439330531208793181290471183631092455368990386390824243_
           5094636442362497730806474389877391449216077946826538517411890917 11_
           7418681451149783372841918224976758683587294866447308566225526872 09_
           2037244118004814057028371983106422912756761957746144438159967135 02_
           6293917497835900414708601277523729964886277426724876224800632688 08_
           8893248918508424949343473376030759399802682084829048596781777514 44_
           6574997982787261696305321767320171723725209 6
       ]
     ]
                    Type: List List Fraction Integer
```

We can also concentrate on the solutions with real (strictly) positive coordinates:

```
  lpr := positiveSolve(lp)$pack
    []
                    Type: List List RealClosure Fraction Integer
```

Thus we have checked that the input system has no solution with strictly positive coordinates.

Let us define another polynomial system (L-3).

```
  f0 := x**3 + y + z + t- 1
             3
    z + y + x  + t - 1
                     Type: Polynomial Integer

  f1 := x + y**3 + z + t -1
         3
    z + y  + x + t - 1
                     Type: Polynomial Integer

  f2 := x + y + z**3 + t-1
     3
    z  + y + x + t - 1
                     Type: Polynomial Integer

  f3 := x + y + z + t**3 -1
                  3
```

```
    z + y + x + t  - 1
                    Type: Polynomial Integer

  lf := [f0, f1, f2, f3]
           3             3             3                                3
  [z + y + x  + t - 1,z + y  + x + t - 1,z  + y + x + t - 1,z + y + x + t  - 1]
                    Type: List Polynomial Integer
```

First compute a decomposition into regular chains (i.e. regular triangular sets).

```
  lts := triangSolve(lf)$pack
  [
      2           3        3
    {t  + t + 1, z  - z - t  + t,
                3      2      2      3           6     3            3      2
       (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
      +
           6     3          9     6     3
       (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
      ,
     x + y + z}
    ,
      16     13     10     7      4      2
    {t   - 6t   + 9t   + 4t  + 15t  - 54t  + 27,
                   15            14             13            12            11
           4907232t   + 40893984t   - 115013088t   + 22805712t   + 36330336t
         +
                     10             9             8             7
           162959040t   - 159859440t  - 156802608t  + 117168768t
         +
                     6             5             4             3
           126282384t  - 129351600t  + 306646992t  + 475302816t
         +
                       2
           - 1006837776t  - 237269088t + 480716208
       *
         z
      +
           54       51        48      46         45        43          42
        48t   - 912t   + 8232t   - 72t   - 46848t   + 1152t   + 186324t
      +
             40          39        38         37           36         35
        - 3780t   - 543144t   - 3168t   - 21384t   + 1175251t   + 41184t
      +
             34           33          32           31           30
        278003t   - 1843242t   - 301815t   - 1440726t   + 1912012t
      +
              29           28          27           26            25
        1442826t   + 4696262t   - 922481t   - 4816188t   - 10583524t
```

```
   +
             24             23             22           21             20
     - 208751t   + 11472138t   + 16762859t   - 857663t   - 19328175t
   +
               19           18             17            16           15
     - 18270421t   + 4914903t   + 22483044t   + 12926517t   - 8605511t
   +
               14           13           12           11          10
     - 17455518t   - 5014597t   + 8108814t   + 8465535t   + 190542t
   +
              9           8          7           6          5          4
     - 4305624t  - 2226123t  + 661905t  + 1169775t  + 226260t  - 209952t
   +
             3
     - 141183t  + 27216t
   ,
            3      2      2      3           6     3            3      2
     (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
   +
        6     3          9     6     3
     (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
   ,
              3
    x + y + z + t  - 1}
  ,
          2                       2                     2
 {t,z - 1,y  - 1,x + y}, {t - 1,z,y  - 1,x + y}, {t - 1,z  - 1,z y + 1,x},
    16     13     10     7      4      2
  {t   - 6t   + 9t   + 4t  + 15t  - 54t  + 27,
                  29            28             27           26             25
          4907232t   + 40893984t   - 115013088t   - 1730448t   - 168139584t
        +
                    24             23             22              21
          738024480t   - 195372288t   + 315849456t   - 2567279232t
        +
                   20              19              18              17
          937147968t   + 1026357696t   + 4780488240t   - 2893767696t
        +
                     16              15              14              13
          - 5617160352t   - 3427651728t   + 5001100848t   + 8720098416t
        +
                   12             11              10             9
          2331732960t   - 499046544t   - 16243306272t   - 9748123200t
        +
                   8              7              6              5
          3927244320t  + 25257280896t  + 10348032096t  - 17128672128t
        +
                    4             3              2
          - 14755488768t  + 544086720t  + 10848188736t  + 1423614528t
        +
```

```
      - 2884297248
 *
    z
+
      68        65         62       60          59        57          56
 - 48t   + 1152t   - 13560t   + 360t   + 103656t   - 7560t   - 572820t
+
       54          53        52          51          50         49
  71316t   + 2414556t   + 2736t   - 402876t   - 7985131t   - 49248t
+
         48           47          46          45           44
  1431133t   + 20977409t   + 521487t   - 2697635t   - 43763654t
+
           43          42           41           40           39
  - 3756573t   - 2093410t   + 71546495t   + 19699032t   + 35025028t
+
            38           37            36           35            34
  - 89623786t   - 77798760t   - 138654191t   + 87596128t   + 235642497t
+
           33           32            31           30            29
  349607642t   - 93299834t   - 551563167t   - 630995176t   + 186818962t
+
           28           27            26           25
  995427468t   + 828416204t   - 393919231t   - 1076617485t
+
              24           23            22            21
  - 1609479791t   + 595738126t   + 1198787136t   + 4342832069t
+
              20            19            18            17
  - 2075938757t   - 4390835799t   - 4822843033t   + 6932747678t
+
             16            15            14            13
  6172196808t   + 1141517740t   - 4981677585t   - 9819815280t
+
              12            11             10             9
  - 7404299976t   - 157295760t   + 29124027630t   + 14856038208t
+
               8             7             6             5
  - 16184101410t  - 26935440354t  - 3574164258t  + 10271338974t
+
             4             3             2
  11191425264t  + 6869861262t  - 9780477840t  - 3586674168t + 2884297248
,
     3      3      2      6      3           9     6    3
  (3z  + (6t  - 6)z  + (6t  - 12t  + 3)z + 2t  - 6t  + t  + 3t)y
+
     3      3      6      3      2      9      6      3          12     9
  (3t  - 3)z  + (6t  - 12t  + 6)z  + (4t  - 12t  + 11t  - 3)z + t   - 4t
+
    6     3
```

```
        5t  - 2t
      ,
                 3
     x + y + z + t  - 1}
    ,
           2
   {t - 1,z  - 1,y,x + z},
      8    7    6     5     4     3      2
    {t  + t  + t  - 2t  - 2t  - 2t  + 19t  + 19t - 8,
                   7           6           5            4           3
           2395770t  + 3934440t  - 3902067t  - 10084164t  - 1010448t
         +
                    2
           32386932t  + 22413225t - 10432368
       *
          z
      +
                7           6           5           4            3
        - 463519t  + 3586833t  + 9494955t  - 8539305t  - 33283098t
      +
                2
        35479377t  + 46263256t - 17419896
      ,
              4      3      3       6      3      2          3
           3z  + (9t  - 9)z  + (12t  - 24t  + 9)z  + (- 152t  + 219t - 67)z
         +
              6      4      3
           - 41t  + 57t  + 25t  - 57t + 16
       *
          y
      +
           3      4      6      3      3          3              2
        (3t  - 3)z  + (9t  - 18t  + 9)z  + (- 181t  + 270t - 89)z
      +
             6       4      3                     7      6      4       3
        (- 92t  + 135t  + 49t  - 135t + 43)z + 27t  - 27t  - 54t  + 396t
      +
        - 486t + 144
      ,
                 3
     x + y + z + t  - 1}
    ,
           3
   {t,z - t  + 1,y - 1,x - 1}, {t - 1,z,y,x}, {t,z - 1,y,x}, {t,z,y - 1,x},
   {t,z,y,x - 1}]
                    Type: List RegularChain(Integer,[x,y,z,t])
```

Then we compute a univariate representation.

```
  univariateSolve(lf)$pack
```

```
[[complexRoots= ?,coordinates= [x - 1,y - 1,z + 1,t - %A]],
 [complexRoots= ?,coordinates= [x,y - 1,z,t - %A]],
 [complexRoots= ? - 1,coordinates= [x,y,z,t - %A]],
 [complexRoots= ?,coordinates= [x - 1,y,z,t - %A]],
 [complexRoots= ?,coordinates= [x,y,z - 1,t - %A]],
 [complexRoots= ? - 2,coordinates= [x - 1,y + 1,z,t - 1]],
 [complexRoots= ?,coordinates= [x + 1,y - 1,z,t - 1]],
 [complexRoots= ? - 1,coordinates= [x - 1,y + 1,z - 1,t]],
 [complexRoots= ? + 1,coordinates= [x + 1,y - 1,z - 1,t]],

                6     3     2
  [complexRoots= ?  - 2?  + 3?  - 3,
                    3                 3
   coordinates= [2x + %A  + %A - 1,2y + %A  + %A - 1,z - %A,t - %A]]
  ,

                5     3     2
  [complexRoots= ?  + 3?  - 2?  + 3? - 3,
                                  3
   coordinates= [x - %A,y - %A,z + %A  + 2%A - 1,t - %A]]
  ,

                4    3     2
  [complexRoots= ?  - ?  - 2?  + 3,
                   3               3               3
   coordinates= [x + %A  - %A - 1,y + %A  - %A - 1,z - %A  + 2%A + 1,t - %A]]
  ,
 [complexRoots= ? + 1,coordinates= [x - 1,y - 1,z,t - %A]],

                6     3     2
  [complexRoots= ?  + 2?  + 3?  - 3,
                    3                       3
   coordinates= [2x - %A  - %A - 1,y + %A,2z - %A  - %A - 1,t + %A]]
  ,

                6      4      3      2
  [complexRoots= ?  + 12?  + 20?  - 45?  - 42? - 953,

    coordinates =
                   5       4       3        2
      [12609x + 23%A  + 49%A  - 46%A  + 362%A  - 5015%A - 8239,
                   5       4       3        2
       25218y + 23%A  + 49%A  - 46%A  + 362%A  + 7594%A - 8239,
                   5       4       3        2
       25218z + 23%A  + 49%A  - 46%A  + 362%A  + 7594%A - 8239,
                   5       4       3        2
       12609t + 23%A  + 49%A  - 46%A  + 362%A  - 5015%A - 8239]
    ]
  ,
```

```
                5      3      2
[complexRoots= ?  + 12?  - 16?  + 48? - 96,
                     3
 coordinates= [8x + %A  + 8%A - 8,2y - %A,2z - %A,2t - %A]]
,

                5    4     3     2
[complexRoots= ?  + ?  - 5?  - 3?  + 9? + 3,

  coordinates =
            3                    3                    3
    [2x - %A  + 2%A - 1, 2y + %A  - 4%A + 1, 2z - %A  + 2%A - 1,
            3
     2t - %A  + 2%A - 1]
  ]
,

                4     3     2
[complexRoots= ?  - 3?  + 4?  - 6? + 13,

  coordinates =
             3      2                   3      2
    [9x - 2%A  + 4%A  - %A + 2, 9y + %A  - 2%A  + 5%A - 1,
             3      2                   3      2
     9z + %A  - 2%A  + 5%A - 1, 9t + %A  - 2%A  - 4%A - 1]
  ]
,

                4      2
[complexRoots= ?  - 11?  + 37,

  coordinates =
            2            2                   2            2
    [3x - %A  + 7,6y + %A  + 3%A - 7,3z - %A  + 7,6t + %A  - 3%A - 7]
  ]
,
[complexRoots= ? + 1,coordinates= [x - 1,y,z - 1,t + 1]],
[complexRoots= ? + 2,coordinates= [x,y - 1,z - 1,t + 1]],
[complexRoots= ? - 2,coordinates= [x,y - 1,z + 1,t - 1]],
[complexRoots= ?,coordinates= [x,y + 1,z - 1,t - 1]],
[complexRoots= ? - 2,coordinates= [x - 1,y,z + 1,t - 1]],
[complexRoots= ?,coordinates= [x + 1,y,z - 1,t - 1]],

                4     3      2
 [complexRoots= ?  + 5?  + 16?  + 30? + 57,

  coordinates =
                3       2                          3       2
    [151x + 15%A  + 54%A  + 104%A + 93, 151y - 10%A  - 36%A  - 19%A - 62,
              3       2                          3       2
```

```
        151z - 5%A  - 18%A  - 85%A - 31, 151t - 5%A  - 18%A  - 85%A - 31]
     ]
   ,

                  4    3     2
   [complexRoots= ?  - ?  - 2?  + 3,
                        3                 3                        3
    coordinates= [x - %A  + 2%A + 1,y + %A  - %A - 1,z - %A,t + %A  - %A - 1]]
   ,

                  4     3     2
   [complexRoots= ?  + 2?  - 8?  + 48,

     coordinates =
                3                            3                   3
       [8x - %A  + 4%A - 8,2y + %A,8z + %A  - 8%A + 8,8t - %A  + 4%A - 8]
     ]
   ,

                  5    4     3     2
   [complexRoots= ?  + ?  - 2?  - 4?  + 5? + 8,
                       3            3           3
    coordinates= [3x + %A  - 1,3y + %A  - 1,3z + %A  - 1,t - %A]]
   ,
                 3
  [complexRoots= ?  + 3? - 1,coordinates= [x - %A,y - %A,z - %A,t - %A]]]
        Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
                          coordinates: List Polynomial Integer)
```

Note that this computation is made from the input system lf.

However it is possible to reuse a pre-computed regular chain as follows:

```
  ts := lts.1
     2          3        3
   {t  + t + 1, z  - z - t  + t,

               3      2      2      3           6     3            3      2
      (3z + 3t  - 3)y  + (3z  + (6t  - 6)z + 3t  - 6t  + 3)y + (3t  - 3)z
    +
         6     3          9     6     3
      (3t  - 6t  + 3)z + t  - 3t  + 5t  - 3t
     ,
    x + y + z}
                      Type: RegularChain(Integer,[x,y,z,t])

  univariateSolve(ts)$pack
  [
                  4     3      2
    [complexRoots= ?  + 5?  + 16?  + 30? + 57,
```

```
      coordinates =
                    3       2                          3       2
        [151x + 15%A  + 54%A  + 104%A + 93, 151y - 10%A  - 36%A  - 19%A - 62,
                    3       2                          3       2
         151z - 5%A  - 18%A  - 85%A - 31, 151t - 5%A  - 18%A  - 85%A - 31]
      ]
    ,
                    4    3     2
    [complexRoots= ?  - ?  - 2?  + 3,
                         3                 3                        3
     coordinates= [x - %A  + 2%A + 1,y + %A  - %A - 1,z - %A,t + %A  - %A - 1]]
    ,
                    4     3     2
    [complexRoots= ?  + 2?  - 8?  + 48,
      coordinates =
                3                          3                  3
        [8x - %A  + 4%A - 8,2y + %A,8z + %A  - 8%A + 8,8t - %A  + 4%A - 8]
      ]
    ]
         Type: List Record(complexRoots: SparseUnivariatePolynomial Integer,
                           coordinates: List Polynomial Integer)

  realSolve(ts)$pack
    []
                        Type: List List RealClosure Fraction Integer
```

We compute now the full set of points with real coordinates:

```
  lr2 := realSolve(lf)$pack
  [[0,- 1,1,1], [0,0,1,0], [1,0,0,0], [0,0,0,1], [0,1,0,0], [1,0,%B37,- %B37],
   [1,0,%B38,- %B38], [0,1,%B35,- %B35], [0,1,%B36,- %B36], [- 1,0,1,1],
    [%B32,
         1     15    2     14    1     13    4     12   11     11    4     10
        -- %B32   + -- %B32   + -- %B32   - -- %B32   - -- %B32   - -- %B32
        27           27           27           27           27           27
      +
         1     9   14     8    1     7   2     6   1     5   2     4       3
        -- %B32  + -- %B32  + -- %B32  + - %B32  + - %B32  + - %B32  + %B32
        27          27          27          9          3          9
      +
        4     2
        - %B32  - %B32 - 2
        3
      ,
           1     15    1     14    1     13    2     12   11     11    2     10
        - -- %B32   - -- %B32   - -- %B32   + -- %B32   + -- %B32   + -- %B32
          54           27           54           27           54           27
      +
           1     9    7     8    1     7   1     6   1     5   1     4       3
        - -- %B32  - -- %B32  - -- %B32  - - %B32  - - %B32  - - %B32  - %B32
```

```
    54         27         54         9         6         9
  +
    2     2   1        3
  - - %B32  + - %B32 + -
    3         2        2
  ,
     1     15    1     14    1     13    2     12   11     11    2     10
  - -- %B32   - -- %B32   - -- %B32   + -- %B32   + -- %B32   + -- %B32
    54          27          54          27          54          27
  +
     1     9    7     8    1     7   1     6   1     5   1     4       3
  - -- %B32  - -- %B32  - -- %B32  - - %B32  - - %B32  - - %B32  - %B32
    54         27         54         9         6         9
  +
    2     2   1        3
  - - %B32  + - %B32 + -
    3         2        2
  ]
,
[%B33,
    1     15    2     14    1     13    4     12   11     11    4     10
  -- %B33   + -- %B33   + -- %B33   - -- %B33   - -- %B33   - -- %B33
  27          27          27          27          27          27
  +
    1     9   14     8    1     7   2     6   1     5   2     4       3
  -- %B33  + -- %B33  + -- %B33  + - %B33  + - %B33  + - %B33  + %B33
  27         27         27         9         3         9
  +
  4     2
  - %B33  - %B33 - 2
  3
  ,
     1     15    1     14    1     13    2     12   11     11    2     10
  - -- %B33   - -- %B33   - -- %B33   + -- %B33   + -- %B33   + -- %B33
    54          27          54          27          54          27
  +
     1     9    7     8    1     7   1     6   1     5   1     4       3
  - -- %B33  - -- %B33  - -- %B33  - - %B33  - - %B33  - - %B33  - %B33
    54         27         54         9         6         9
  +
    2     2   1        3
  - - %B33  + - %B33 + -
    3         2        2
  ,
     1     15    1     14    1     13    2     12   11     11    2     10
  - -- %B33   - -- %B33   - -- %B33   + -- %B33   + -- %B33   + -- %B33
    54          27          54          27          54          27
  +
     1     9    7     8    1     7   1     6   1     5   1     4       3
  - -- %B33  - -- %B33  - -- %B33  - - %B33  - - %B33  - - %B33  - %B33
```

```
      54          27          54          9          6          9
  +
      2     2   1         3
    - - %B33  + - %B33 + -
      3          2         2
  ]
 ,
 [%B34,
      1     15    2     14    1     13    4     12   11     11    4     10
    -- %B34   + -- %B34   + -- %B34   - -- %B34   - -- %B34   - -- %B34
    27          27          27          27          27          27
  +
      1     9   14     8    1     7   2     6   1     5   2     4       3
    -- %B34  + -- %B34  + -- %B34  + - %B34  + - %B34  + - %B34  + %B34
    27          27          27          9          3          9
  +
    4     2
    - %B34  - %B34 - 2
    3
  ,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B34   - -- %B34   - -- %B34   + -- %B34   + -- %B34   + -- %B34
      54          27          54          27          54          27
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B34  - -- %B34  - -- %B34  - - %B34  - - %B34  - - %B34  - %B34
      54          27          54          9          6          9
  +
      2     2   1         3
    - - %B34  + - %B34 + -
      3          2         2
  ,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B34   - -- %B34   - -- %B34   + -- %B34   + -- %B34   + -- %B34
      54          27          54          27          54          27
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B34  - -- %B34  - -- %B34  - - %B34  - - %B34  - - %B34  - %B34
      54          27          54          9          6          9
  +
      2     2   1         3
    - - %B34  + - %B34 + -
      3          2         2
  ]
 ,
[- 1,1,0,1], [- 1,1,1,0],
 [%B23,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B23   - -- %B23   - -- %B23   + -- %B23   + -- %B23   + -- %B23
      54          27          54          27          54          27
```

```
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B23  - -- %B23  - -- %B23  - - %B23  - - %B23  - - %B23  - %B23
      54          27          54          9          6          9
  +
      2     2   1         3
    - - %B23  + - %B23 + -
      3         2         2
 ,
%B30,
              1     15    1     14    1     13    2     12   11     11
    - %B30 + -- %B23   + -- %B23   + -- %B23   - -- %B23   - -- %B23
             54          27          54          27          54
  +
       2     10    1     9    7     8    1     7   1     6   1     5
    - -- %B23   + -- %B23  + -- %B23  + -- %B23  + - %B23  + - %B23
      27          54          27          54          9          6
  +
    1     4   2     2   1         1
    - %B23  + - %B23  - - %B23 - -
    9         3         2         2
  ]
,
[%B23,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B23   - -- %B23   - -- %B23   + -- %B23   + -- %B23   + -- %B23
      54          27          54          27          54          27
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B23  - -- %B23  - -- %B23  - - %B23  - - %B23  - - %B23  - %B23
      54          27          54          9          6          9
  +
      2     2   1         3
    - - %B23  + - %B23 + -
      3         2         2
 ,
%B31,
              1     15    1     14    1     13    2     12   11     11
    - %B31 + -- %B23   + -- %B23   + -- %B23   - -- %B23   - -- %B23
             54          27          54          27          54
  +
       2     10    1     9    7     8    1     7   1     6   1     5
    - -- %B23   + -- %B23  + -- %B23  + -- %B23  + - %B23  + - %B23
      27          54          27          54          9          6
  +
    1     4   2     2   1         1
    - %B23  + - %B23  - - %B23 - -
    9         3         2         2
  ]
,
```

```
[%B24,
       1     15    1     14    1     13    2     12   11     11    2     10
   - -- %B24   - -- %B24   - -- %B24   + -- %B24   + -- %B24   + -- %B24
     54           27           54           27           54           27
 +
      1     9    7     8    1     7   1     6   1     5   1     4       3
   - -- %B24  - -- %B24  - -- %B24  - - %B24  - - %B24  - - %B24  - %B24
     54          27          54          9          6          9
 +
     2     2   1         3
   - - %B24  + - %B24 + -
     3         2         2
 ,
%B28,
              1     15    1     14    1     13    2     12   11     11
   - %B28 + -- %B24   + -- %B24   + -- %B24   - -- %B24   - -- %B24
             54           27           54           27           54
 +
      2     10    1     9    7     8    1     7   1     6   1     5
   - -- %B24   + -- %B24  + -- %B24  + -- %B24  + - %B24  + - %B24
     27           54          27          54          9          6
 +
   1     4   2     2   1         1
   - %B24  + - %B24  - - %B24 - -
   9         3         2         2
 ]
,
[%B24,
       1     15    1     14    1     13    2     12   11     11    2     10
   - -- %B24   - -- %B24   - -- %B24   + -- %B24   + -- %B24   + -- %B24
     54           27           54           27           54           27
 +
      1     9    7     8    1     7   1     6   1     5   1     4       3
   - -- %B24  - -- %B24  - -- %B24  - - %B24  - - %B24  - - %B24  - %B24
     54          27          54          9          6          9
 +
     2     2   1         3
   - - %B24  + - %B24 + -
     3         2         2
 ,
%B29,
              1     15    1     14    1     13    2     12   11     11
   - %B29 + -- %B24   + -- %B24   + -- %B24   - -- %B24   - -- %B24
             54           27           54           27           54
 +
      2     10    1     9    7     8    1     7   1     6   1     5
   - -- %B24   + -- %B24  + -- %B24  + -- %B24  + - %B24  + - %B24
     27           54          27          54          9          6
 +
   1     4   2     2   1         1
```

```
    - %B24  + - %B24  - - %B24 - -
    9         3         2        2
  ]
,
[%B25,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B25   - -- %B25   - -- %B25   + -- %B25   + -- %B25   + -- %B25
      54          27          54          27          54          27
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B25  - -- %B25  - -- %B25  - - %B25  - - %B25  - - %B25  - %B25
      54         27         54         9         6         9
  +
      2     2   1        3
    - - %B25  + - %B25 + -
      3         2        2
 ,
 %B26,
              1     15    1     14    1     13    2     12   11     11
    - %B26 + -- %B25   + -- %B25   + -- %B25   - -- %B25   - -- %B25
             54          27          54          27          54
  +
       2     10    1     9    7     8    1     7   1     6   1     5
    - -- %B25   + -- %B25  + -- %B25  + -- %B25  + - %B25  + - %B25
      27          54         27         54         9         6
  +
    1     4   2     2   1        1
    - %B25  + - %B25  - - %B25 - -
    9         3         2        2
  ]
,
[%B25,
       1     15    1     14    1     13    2     12   11     11    2     10
    - -- %B25   - -- %B25   - -- %B25   + -- %B25   + -- %B25   + -- %B25
      54          27          54          27          54          27
  +
       1     9    7     8    1     7   1     6   1     5   1     4       3
    - -- %B25  - -- %B25  - -- %B25  - - %B25  - - %B25  - - %B25  - %B25
      54         27         54         9         6         9
  +
      2     2   1        3
    - - %B25  + - %B25 + -
      3         2        2
 ,
 %B27,
              1     15    1     14    1     13    2     12   11     11
    - %B27 + -- %B25   + -- %B25   + -- %B25   - -- %B25   - -- %B25
             54          27          54          27          54
  +
       2     10    1     9    7     8    1     7   1     6   1     5
```

```
        - -- %B25   + -- %B25  + -- %B25  + -- %B25  + - %B25  + - %B25
          27           54          27          54          9          6
      +
        1     4   2     2   1        1
        - %B25  + - %B25  - - %B25 - -
        9         3         2        2
      ]
    ,
   [1,%B21,- %B21,0], [1,%B22,- %B22,0], [1,%B19,0,- %B19], [1,%B20,0,- %B20],
          1     3   1   1     3   1   1     3   1
   [%B17,- - %B17  + -,- - %B17  + -,- - %B17  + -],
          3         3   3         3   3         3
          1     3   1   1     3   1   1     3   1
   [%B18,- - %B18  + -,- - %B18  + -,- - %B18  + -]]
          3         3   3         3   3         3
                      Type: List List RealClosure Fraction Integer
```

The number of real solutions for the input system is:

```
  #lr2
    27
                              Type: PositiveInteger
```

Another example of computation of real solutions illustrates the LexTriangularPackage package constructor.

We concentrate now on the solutions with real (strictly) positive coordinates:

```
  lpr2 := positiveSolve(lf)$pack
              1     3   1   1     3   1   1     3   1
     [[%B40,- - %B40  + -,- - %B40  + -,- - %B40  + -]]
              3         3   3         3   3         3
                              Type: List List RealClosure Fraction Integer
```

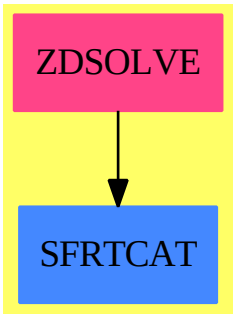
Finally, we approximate the coordinates of this point with 20 exact digits:

```
  [approximate(r,1/10**21)::Float for r in lpr2.1]
   [0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291,
    0.3221853546 2608559291]
                              Type: List Float
```

See Also:
o)show ZeroDimensionalSolvePackage

———————-

27.1.1 ZeroDimensionalSolvePackage (ZDSOLVE)



Exports:
convert positiveSolve realSolve squareFree triangSolve univariateSolve

— package ZDSOLVE ZeroDimensionalSolvePackage —

```
)abbrev package ZDSOLVE ZeroDimensionalSolvePackage
++ Author: Marc Moreno Maza
++ Date Created: 23/01/1999
++ Date Last Updated: 08/02/1999
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++ A package for computing symbolically the complex and real roots of
++ zero-dimensional algebraic systems over the integer or rational
++ numbers. Complex roots are given by means of univariate representations
++ of irreducible regular chains. Real roots are given by means of tuples
++ of coordinates lying in the \spadtype{RealClosure} of the coefficient ring.
++ This constructor takes three arguments. The first one \spad{R} is the
++ coefficient ring. The second one \spad{ls} is the list of variables
++ involved in the systems to solve. The third one must be \spad{concat(ls,s)}
++ where \spad{s} is an additional symbol used for the univariate
++ representations.
++ WARNING. The third argument is not checked.
++ All operations are based on triangular decompositions.
++ The default is to compute these decompositions directly from the input
++ system by using the \spadtype{RegularChain} domain constructor.
++ The lexTriangular algorithm can also be used for computing these
++ decompositions (see \spadtype{LexTriangularPackage} package constructor).
++ For that purpose, the operations univariateSolve, realSolve and
++ positiveSolve admit an optional argument.
```

```
ZeroDimensionalSolvePackage(R,ls,ls2): Exports == Implementation where
  R : Join(OrderedRing,EuclideanDomain,CharacteristicZero,RealConstant)
  ls: List Symbol
  ls2: List Symbol
  V ==> OrderedVariableList(ls)
  N ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  P ==> Polynomial R
  LP ==> List P
  LS ==> List Symbol
  Q ==> NewSparseMultivariatePolynomial(R,V)
  U ==> SparseUnivariatePolynomial(R)
  TS ==> RegularChain(R,ls)
  RUR ==> Record(complexRoots: U, coordinates: LP)
  K ==> Fraction R
  RC ==> RealClosure(K)
  PRC ==> Polynomial RC
  REALSOL ==> List RC
  URC ==> SparseUnivariatePolynomial RC
  V2 ==> OrderedVariableList(ls2)
  Q2 ==> NewSparseMultivariatePolynomial(R,V2)
  E2 ==> IndexedExponents V2
  ST ==> SquareFreeRegularTriangularSet(R,E2,V2,Q2)
  Q2WT ==> Record(val: Q2, tower: ST)
  LQ2WT ==> Record(val: List(Q2), tower: ST)
  WIP ==> Record(reals: List(RC), vars: List(Symbol), pols: List(Q2))
  polsetpack ==> PolynomialSetUtilitiesPackage(R,E2,V2,Q2)
  normpack ==> NormalizationPackage(R,E2,V2,Q2,ST)
  rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E2,V2,Q2,ST)
  quasicomppack ==> SquareFreeQuasiComponentPackage(R,E2,V2,Q2,ST)
  lextripack ==> LexTriangularPackage(R,ls)

  Exports ==  with
     triangSolve: (LP,B,B) -> List RegularChain(R,ls)
       ++ \spad{triangSolve(lp,info?,lextri?)} decomposes the variety
       ++ associated with \axiom{lp} into regular chains.
       ++ Thus a point belongs to this variety iff it is a regular
       ++ zero of a regular set in in the output.
       ++ Note that \axiom{lp} needs to generate a zero-dimensional ideal.
       ++ If \axiom{lp} is not zero-dimensional then the result is only
       ++ a decomposition of its zero-set in the sense of the closure
       ++ (w.r.t. Zarisky topology).
       ++ Moreover, if \spad{info?} is \spad{true} then some information is
       ++ displayed during the computations.
       ++ See zeroSetSplit from RegularTriangularSetCategory(lp,true,info?).
       ++ If \spad{lextri?} is \spad{true} then the lexTriangular algorithm
       ++ is called
       ++ from the \spadtype{LexTriangularPackage} constructor
       ++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
```

```
    ++ Otherwise, the triangular decomposition is computed directly from
    ++ the input
    ++ system by using the zeroSetSplit from RegularChain
  triangSolve: (LP,B) -> List RegularChain(R,ls)
    ++ \spad{triangSolve(lp,info?)} returns the same as
    ++ \spad{triangSolve(lp,false)}
  triangSolve: LP -> List RegularChain(R,ls)
    ++ \spad{triangSolve(lp)} returns the same as
    ++ \spad{triangSolve(lp,false,false)}
  univariateSolve: RegularChain(R,ls) -> _
                              List Record(complexRoots: U, coordinates: LP)
    ++ \spad{univariateSolve(ts)} returns a univariate representation
    ++ of \spad{ts}.
    ++ See rur from RationalUnivariateRepresentationPackage(lp,true).
  univariateSolve: (LP,B,B,B) -> List RUR
    ++ \spad{univariateSolve(lp,info?,check?,lextri?)} returns a univariate
    ++ representation of the variety associated with \spad{lp}.
    ++ Moreover, if \spad{info?} is \spad{true} then some information is
    ++ displayed during the decomposition into regular chains.
    ++ If \spad{check?} is \spad{true} then the result is checked.
    ++ See rur from RationalUnivariateRepresentationPackage(lp,true).
    ++ If \spad{lextri?} is \spad{true} then the lexTriangular
    ++ algorithm is called
    ++ from the \spadtype{LexTriangularPackage} constructor
    ++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
    ++ Otherwise, the triangular decomposition is computed directly
    ++ from the input
    ++ system by using the zeroSetSplit from RegularChain
  univariateSolve: (LP,B,B) -> List RUR
    ++ \spad{univariateSolve(lp,info?,check?)} returns the same as
    ++ \spad{univariateSolve(lp,info?,check?,false)}.
  univariateSolve: (LP,B) -> List RUR
    ++ \spad{univariateSolve(lp,info?)} returns the same as
    ++ \spad{univariateSolve(lp,info?,false,false)}.
  univariateSolve: LP -> List RUR
    ++ \spad{univariateSolve(lp)} returns the same as
    ++ \spad{univariateSolve(lp,false,false,false)}.
  realSolve: RegularChain(R,ls) -> List REALSOL
    ++ \spad{realSolve(ts)} returns the set of the points in the regular
    ++ zero set of \spad{ts} whose coordinates are all real.
    ++ WARNING. For each set of coordinates given by \spad{realSolve(ts)}
    ++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.
  realSolve: (LP,B,B,B) -> List REALSOL
    ++ \spad{realSolve(ts,info?,check?,lextri?)} returns the set of the
    ++ points in the variety associated with \spad{lp} whose coordinates
    ++ are all real.
    ++ Moreover, if \spad{info?} is \spad{true} then some information is
    ++ displayed during decomposition into regular chains.
    ++ If \spad{check?} is \spad{true} then the result is checked.
    ++ If \spad{lextri?} is \spad{true} then the lexTriangular algorithm
```

```
    ++ is called
    ++ from the \spadtype{LexTriangularPackage} constructor
    ++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
    ++ Otherwise, the triangular decomposition is computed directly from
    ++ the input
    ++ system by using the zeroSetSplit from \spadtype{RegularChain}.
    ++ WARNING. For each set of coordinates given by
    ++ \spad{realSolve(ts,info?,check?,lextri?)}
    ++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.
  realSolve: (LP,B,B) -> List REALSOL
    ++ \spad{realSolve(ts,info?,check?)} returns the same as
    ++ \spad{realSolve(ts,info?,check?,false)}.
  realSolve: (LP,B) -> List REALSOL
    ++ \spad{realSolve(ts,info?)} returns the same as
    ++ \spad{realSolve(ts,info?,false,false)}.
  realSolve: LP -> List REALSOL
    ++ \spad{realSolve(lp)} returns the same as
    ++ \spad{realSolve(ts,false,false,false)}
  positiveSolve: RegularChain(R,ls)  -> List REALSOL
    ++ \spad{positiveSolve(ts)} returns the points of the regular
    ++ set of \spad{ts} with (real) strictly positive coordinates.
  positiveSolve: (LP,B,B) -> List REALSOL
    ++ \spad{positiveSolve(lp,info?,lextri?)} returns the set of the points
    ++ in the variety associated with \spad{lp} whose coordinates are
    ++ (real) strictly positive.
    ++ Moreover, if \spad{info?} is \spad{true} then some information is
    ++ displayed during decomposition into regular chains.
    ++ If \spad{lextri?} is \spad{true} then the lexTriangular
    ++ algorithm is called
    ++ from the \spadtype{LexTriangularPackage} constructor
    ++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
    ++ Otherwise, the triangular decomposition is computed directly
    ++ from the input
    ++ system by using the zeroSetSplit from \spadtype{RegularChain}.
    ++ WARNING. For each set of coordinates given by
    ++ \spad{positiveSolve(lp,info?,lextri?)}
    ++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.
  positiveSolve: (LP,B) -> List REALSOL
    ++ \spad{positiveSolve(lp)} returns the same as
    ++ \spad{positiveSolve(lp,info?,false)}.
  positiveSolve: LP -> List REALSOL
    ++ \spad{positiveSolve(lp)} returns the same as
    ++ \spad{positiveSolve(lp,false,false)}.
  squareFree: (TS) -> List ST
    ++ \spad{squareFree(ts)} returns the square-free factorization
    ++ of \spad{ts}. Moreover, each factor is a Lazard triangular set
    ++ and the decomposition
    ++ is a Kalkbrener split of \spad{ts}, which is enough here for
    ++ the matter of solving zero-dimensional algebraic systems.
    ++ WARNING. \spad{ts} is not checked to be zero-dimensional.
```

```
    convert: Q -> Q2
      ++ \spad{convert(q)} converts \spad{q}.
    convert: P -> PRC
      ++ \spad{convert(p)} converts \spad{p}.
    convert: Q2 -> PRC
      ++ \spad{convert(q)} converts \spad{q}.
    convert: U -> URC
      ++ \spad{convert(u)} converts \spad{u}.
    convert: ST -> List Q2
      ++ \spad{convert(st)} returns the members of \spad{st}.

  Implementation == add
    news: Symbol := last(ls2)$(List Symbol)
    newv: V2 := (variable(news)$V2)::V2
    newq: Q2 :=  newv :: Q2

    convert(q:Q):Q2 ==
      ground? q => (ground(q))::Q2
      q2: Q2 := 0
      while not ground?(q) repeat
        v: V := mvar(q)
        d: N := mdeg(q)
        v2: V2 := (variable(convert(v)@Symbol)$V2)::V2
        iq2: Q2 := convert(init(q))@Q2
        lq2: Q2 := (v2 :: Q2)
        lq2 := lq2 ** d
        q2 := iq2 * lq2 + q2
        q := tail(q)
      q2 + (ground(q))::Q2

    squareFree(ts:TS):List(ST) ==
      irred?: Boolean := false
      st: ST := [[newq]$(List Q2)]
      lq: List(Q2) := [convert(p)@Q2 for p in parts(ts)]
      lq := sort(infRittWu?,lq)
      toSee: List LQ2WT := []
      if irred?
        then
          lf := irreducibleFactors([first lq])$polsetpack
          lq := rest lq
          for f in lf repeat
            toSee := cons([cons(f,lq),st]$LQ2WT, toSee)
        else
          toSee := [[lq,st]$LQ2WT]
      toSave: List ST := []
      while not empty? toSee repeat
        lqwt := first toSee; toSee := rest toSee
        lq := lqwt.val; st := lqwt.tower
        empty? lq =>
          toSave := cons(st,toSave)
```

```
    q := first lq; lq := rest lq
    lsfqwt: List Q2WT := squareFreePart(q,st)$ST
    for sfqwt in lsfqwt repeat
      q := sfqwt.val; st := sfqwt.tower
      if not ground? init(q)
        then
          q := normalizedAssociate(q,st)$normpack
      newts := internalAugment(q,st)$ST
      newlq := [remainder(q,newts).polnum for q in lq]
      toSee := cons([newlq,newts]$LQ2WT,toSee)
  toSave


triangSolve(lp: LP, info?: B, lextri?: B): List TS ==
  lq: List(Q) := [convert(p)$Q for p in lp]
  lextri? => zeroSetSplit(lq,false)$lextripack
  zeroSetSplit(lq,true,info?)$TS

triangSolve(lp: LP, info?: B): List TS == triangSolve(lp,info?,false)

triangSolve(lp: LP): List TS == triangSolve(lp,false)

convert(u: U): URC ==
  zero? u => 0
  ground? u => ((ground(u) :: K)::RC)::URC
  uu: URC := 0
  while not ground? u repeat
    uu := monomial((leadingCoefficient(u) :: K):: RC,degree(u)) + uu
    u := reductum u
  uu + ((ground(u) :: K)::RC)::URC

coerceFromRtoRC(r:R): RC ==
  (r::K)::RC

convert(p:P): PRC ==
  map(coerceFromRtoRC,p)$PolynomialFunctions2(R,RC)

convert(q2:Q2): PRC ==
  p: P := coerce(q2)$Q2
  convert(p)@PRC

convert(sts:ST): List Q2 ==
  lq2: List(Q2) := parts(sts)$ST
  lq2 := sort(infRittWu?,lq2)
  rest(lq2)

realSolve(ts: TS): List REALSOL ==
  lsts: List ST := squareFree(ts)
  lr: REALSOL := []
  lv: List Symbol := []
```

```
    toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
    toSave: List REALSOL := []
    while not empty? toSee repeat
      wip := first toSee; toSee := rest toSee
      lr := wip.reals; lv := wip.vars; lq2 := wip.pols
      (empty? lq2) and (not empty? lr) =>
         toSave := cons(reverse(lr),toSave)
      q2 := first lq2; lq2 := rest lq2
      qrc := convert(q2)@PRC
      if not empty? lr
        then
          for r in reverse(lr) for v in reverse(lv) repeat
            qrc := eval(qrc,v,r)
      lv := cons((mainVariable(qrc) :: Symbol),lv)
      urc: URC := univariate(qrc)@URC
      urcRoots := allRootsOf(urc)$RC
      for urcRoot in urcRoots repeat
        toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
    toSave

  realSolve(lp: List(P), info?:Boolean, check?:Boolean, lextri?: Boolean): List REALSOL  ==
    lts: List TS
    lq: List(Q) := [convert(p)$Q for p in lp]
    if lextri?
      then
        lts := zeroSetSplit(lq,false)$lextripack
      else
        lts := zeroSetSplit(lq,true,info?)$TS
    lsts:  List ST := []
    for ts in lts repeat
      lsts := concat(squareFree(ts), lsts)
    lsts := removeSuperfluousQuasiComponents(lsts)$quasicomppack
    lr: REALSOL := []
    lv: List Symbol := []
    toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
    toSave: List REALSOL := []
    while not empty? toSee repeat
      wip := first toSee; toSee := rest toSee
      lr := wip.reals; lv := wip.vars; lq2 := wip.pols
      (empty? lq2) and (not empty? lr) =>
         toSave := cons(reverse(lr),toSave)
      q2 := first lq2; lq2 := rest lq2
      qrc := convert(q2)@PRC
      if not empty? lr
        then
          for r in reverse(lr) for v in reverse(lv) repeat
            qrc := eval(qrc,v,r)
      lv := cons((mainVariable(qrc) :: Symbol),lv)
      urc: URC := univariate(qrc)@URC
      urcRoots := allRootsOf(urc)$RC
```

```
    for urcRoot in urcRoots repeat
      toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
  if check?
    then
      for p in lp repeat
        for realsol in toSave repeat
          prc: PRC := convert(p)@PRC
          for rr in realsol for symb in reverse(ls) repeat
            prc := eval(prc,symb,rr)
          not zero? prc =>
            error "realSolve$ZDSOLVE: bad result"
  toSave

realSolve(lp: List(P), info?:Boolean, check?:Boolean): List REALSOL  ==
  realSolve(lp,info?,check?,false)

realSolve(lp: List(P), info?:Boolean): List REALSOL  ==
  realSolve(lp,info?,false,false)

realSolve(lp: List(P)): List REALSOL  ==
  realSolve(lp,false,false,false)

positiveSolve(ts: TS): List REALSOL ==
  lsts: List ST := squareFree(ts)
  lr: REALSOL := []
  lv: List Symbol := []
  toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
  toSave: List REALSOL := []
  while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    lr := wip.reals; lv := wip.vars; lq2 := wip.pols
    (empty? lq2) and (not empty? lr) =>
       toSave := cons(reverse(lr),toSave)
    q2 := first lq2; lq2 := rest lq2
    qrc := convert(q2)@PRC
    if not empty? lr
      then
        for r in reverse(lr) for v in reverse(lv) repeat
          qrc := eval(qrc,v,r)
    lv := cons((mainVariable(qrc) :: Symbol),lv)
    urc: URC := univariate(qrc)@URC
    urcRoots := allRootsOf(urc)$RC
    for urcRoot in urcRoots repeat
      if positive? urcRoot
        then
          toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
  toSave

positiveSolve(lp: List(P), info?:Boolean, lextri?: Boolean): List REALSOL  ==
  lts: List TS
```

```
    lq: List(Q) := [convert(p)$Q for p in lp]
    if lextri?
      then
        lts := zeroSetSplit(lq,false)$lextripack
      else
        lts := zeroSetSplit(lq,true,info?)$TS
    lsts:  List ST := []
    for ts in lts repeat
      lsts := concat(squareFree(ts), lsts)
    lsts := removeSuperfluousQuasiComponents(lsts)$quasicomppack
    lr: REALSOL := []
    lv: List Symbol := []
    toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
    toSave: List REALSOL := []
    while not empty? toSee repeat
      wip := first toSee; toSee := rest toSee
      lr := wip.reals; lv := wip.vars; lq2 := wip.pols
      (empty? lq2) and (not empty? lr) =>
         toSave := cons(reverse(lr),toSave)
      q2 := first lq2; lq2 := rest lq2
      qrc := convert(q2)@PRC
      if not empty? lr
        then
          for r in reverse(lr) for v in reverse(lv) repeat
            qrc := eval(qrc,v,r)
      lv := cons((mainVariable(qrc) :: Symbol),lv)
      urc: URC := univariate(qrc)@URC
      urcRoots := allRootsOf(urc)$RC
      for urcRoot in urcRoots repeat
        if positive? urcRoot
          then
            toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
    toSave

  positiveSolve(lp: List(P), info?:Boolean): List REALSOL  ==
    positiveSolve(lp, info?, false)

  positiveSolve(lp: List(P)): List REALSOL  ==
    positiveSolve(lp, false, false)

  univariateSolve(ts: TS): List RUR ==
    toSee: List ST := squareFree(ts)
    toSave: List RUR := []
    for st in toSee repeat
      lus: List ST := rur(st,true)$rurpack
      for us in lus repeat
        g: U  := univariate(select(us,newv)::Q2)$Q2
        lc: LP := [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
        toSave := cons([g,lc]$RUR, toSave)
    toSave
```

```
    univariateSolve(lp: List(P), info?:Boolean, check?:Boolean, lextri?: Boolean): List RUR ==
      lts: List TS
      lq: List(Q) := [convert(p)$Q for p in lp]
      if lextri?
        then
          lts := zeroSetSplit(lq,false)$lextripack
        else
          lts := zeroSetSplit(lq,true,info?)$TS
      toSee:  List ST := []
      for ts in lts repeat
        toSee := concat(squareFree(ts), toSee)
      toSee := removeSuperfluousQuasiComponents(toSee)$quasicomppack
      toSave: List RUR := []
      if check?
        then
          lq2: List(Q2) := [convert(p)$Q2 for p in lp]
      for st in toSee repeat
        lus: List ST := rur(st,true)$rurpack
        for us in lus repeat
           if check?
             then
               rems: List(Q2) := [removeZero(q2,us)$ST for q2 in lq2]
               not every?(zero?,rems) =>
                 output(st::OutputForm)$OutputPackage
                 output("Has a bad RUR component:")$OutputPackage
                 output(us::OutputForm)$OutputPackage
                 error "univariateSolve$ZDSOLVE: bad RUR"
           g: U  := univariate(select(us,newv)::Q2)$Q2
           lc: LP := _
             [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
           toSave := cons([g,lc]$RUR, toSave)
      toSave

    univariateSolve(lp: List(P), info?:Boolean, check?:Boolean): List RUR ==
      univariateSolve(lp,info?,check?,false)

    univariateSolve(lp: List(P), info?:Boolean): List RUR ==
      univariateSolve(lp,info?,false,false)

    univariateSolve(lp: List(P)): List RUR ==
      univariateSolve(lp,false,false,false)
```

———-

— **ZDSOLVE.dotabb** —

```
"ZDSOLVE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ZDSOLVE"]
```

```
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"ZDSOLVE" -> "SFRTCAT"
```

———-

Chapter 28

Chunk collections

— algebra —

```
\getchunk{package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis}
\getchunk{package AFALGRES AffineAlgebraicSetComputeWithResultant}
\getchunk{package AF AlgebraicFunction}
\getchunk{package INTHERAL AlgebraicHermiteIntegration}
\getchunk{package INTALG AlgebraicIntegrate}
\getchunk{package INTAF AlgebraicIntegration}
\getchunk{package ALGMANIP AlgebraicManipulations}
\getchunk{package ALGMFACT AlgebraicMultFact}
\getchunk{package ALGPKG AlgebraPackage}
\getchunk{package ALGFACT AlgFactor}
\getchunk{package INTPACK AnnaNumericalIntegrationPackage}
\getchunk{package OPTPACK AnnaNumericalOptimizationPackage}
\getchunk{package ODEPACK AnnaOrdinaryDifferentialEquationPackage}
\getchunk{package PDEPACK AnnaPartialDifferentialEquationPackage}
\getchunk{package ANY1 AnyFunctions1}
\getchunk{package APPRULE ApplyRules}
\getchunk{package APPLYORE ApplyUnivariateSkewPolynomial}
\getchunk{package ASSOCEQ AssociatedEquations}
\getchunk{package PMPRED AttachPredicates}
\getchunk{package AXSERV AxiomServer}

\getchunk{package BALFACT BalancedFactorisation}
\getchunk{package BOP1 BasicOperatorFunctions1}
\getchunk{package BEZOUT BezoutMatrix}
\getchunk{package BLUPPACK BlowUpPackage}
\getchunk{package BOUNDZRO BoundIntegerRoots}
\getchunk{package BRILL BrillhartTests}

\getchunk{package CARTEN2 CartesianTensorFunctions2}
```

```
\getchunk{package CHVAR ChangeOfVariable}
\getchunk{package CPIMA CharacteristicPolynomialInMonogenicalAlgebra}
\getchunk{package CHARPOL CharacteristicPolynomialPackage}
\getchunk{package IBACHIN ChineseRemainderToolsForIntegralBases}
\getchunk{package CVMP CoerceVectorMatrixPackage}
\getchunk{package COMBF CombinatorialFunction}
\getchunk{package CDEN CommonDenominator}
\getchunk{package COMMONOP CommonOperators}
\getchunk{package COMMUPC CommuteUnivariatePolynomialCategory}
\getchunk{package COMPFACT ComplexFactorization}
\getchunk{package COMPLEX2 ComplexFunctions2}
\getchunk{package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation}
\getchunk{package COMPLPAT ComplexPattern}
\getchunk{package CPMATCH ComplexPatternMatch}
\getchunk{package CRFP ComplexRootFindingPackage}
\getchunk{package CMPLXRT ComplexRootPackage}
\getchunk{package CTRIGMNP ComplexTrigonometricManipulations}
\getchunk{package ODECONST ConstantLODE}
\getchunk{package COORDSYS CoordinateSystems}
\getchunk{package CRAPACK CRApackage}
\getchunk{package CYCLES CycleIndicators}
\getchunk{package CSTTOOLS CyclicStreamTools}
\getchunk{package CYCLOTOM CyclotomicPolynomialPackage}

\getchunk{package DFINTTLS DefiniteIntegrationTools}
\getchunk{package DEGRED DegreeReductionPackage}
\getchunk{package DTP DesingTreePackage}
\getchunk{package DIOSP DiophantineSolutionPackage}
\getchunk{package DIRPROD2 DirectProductFunctions2}
\getchunk{package DLP DiscreteLogarithmPackage}
\getchunk{package DISPLAY DisplayPackage}
\getchunk{package DDFACT DistinctDegreeFactorize}
\getchunk{package DFSFUN DoubleFloatSpecialFunctions}
\getchunk{package DBLRESP DoubleResultantPackage}
\getchunk{package DRAWCX DrawComplex}
\getchunk{package DRAWHACK DrawNumericHack}
\getchunk{package DROPT0 DrawOptionFunctions0}
\getchunk{package DROPT1 DrawOptionFunctions1}
\getchunk{package D01AGNT d01AgentsPackage}
\getchunk{package D01WGTS d01WeightsPackage}
\getchunk{package D02AGNT d02AgentsPackage}
\getchunk{package D03AGNT d03AgentsPackage}

\getchunk{package EP EigenPackage}
\getchunk{package EF ElementaryFunction}
\getchunk{package DEFINTEF ElementaryFunctionDefiniteIntegration}
\getchunk{package LODEEF ElementaryFunctionLODESolver}
\getchunk{package ODEEF ElementaryFunctionODESolver}
\getchunk{package SIGNEF ElementaryFunctionSign}
\getchunk{package EFSTRUC ElementaryFunctionStructurePackage}
```

```
\getchunk{package EFULS ElementaryFunctionsUnivariateLaurentSeries}
\getchunk{package EFUPXS ElementaryFunctionsUnivariatePuiseuxSeries}
\getchunk{package INTEF ElementaryIntegration}
\getchunk{package RDEEF ElementaryRischDE}
\getchunk{package RDEEFS ElementaryRischDESystem}
\getchunk{package ELFUTS EllipticFunctionsUnivariateTaylorSeries}
\getchunk{package EQ2 EquationFunctions2}
\getchunk{package ERROR ErrorFunctions}
\getchunk{package GBEUCLID EuclideanGroebnerBasisPackage}
\getchunk{package EVALCYC EvaluateCycleIndicators}
\getchunk{package ESCONT ExpertSystemContinuityPackage}
\getchunk{package ESCONT1 ExpertSystemContinuityPackage1}
\getchunk{package ESTOOLS ExpertSystemToolsPackage}
\getchunk{package ESTOOLS1 ExpertSystemToolsPackage1}
\getchunk{package ESTOOLS2 ExpertSystemToolsPackage2}
\getchunk{package EXPR2 ExpressionFunctions2}
\getchunk{package EXPRSOL ExpressionSolve}
\getchunk{package ES1 ExpressionSpaceFunctions1}
\getchunk{package ES2 ExpressionSpaceFunctions2}
\getchunk{package EXPRODE ExpressionSpaceODESolver}
\getchunk{package OMEXPR ExpressionToOpenMath}
\getchunk{package EXPR2UPS ExpressionToUnivariatePowerSeries}
\getchunk{package EXPRTUBE ExpressionTubePlot}
\getchunk{package EXP3D Export3D}
\getchunk{package E04AGNT e04AgentsPackage}

\getchunk{package FACTFUNC FactoredFunctions}
\getchunk{package FR2 FactoredFunctions2}
\getchunk{package FRUTIL FactoredFunctionUtilities}
\getchunk{package FACUTIL FactoringUtilities}
\getchunk{package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber}
\getchunk{package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber}
\getchunk{package FGLMICPK FGLMIfCanPackage}
\getchunk{package FORDER FindOrderFinite}
\getchunk{package FAMR2 FiniteAbelianMonoidRingFunctions2}
\getchunk{package FDIV2 FiniteDivisorFunctions2}
\getchunk{package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect}
\getchunk{package FFF FiniteFieldFunctions}
\getchunk{package FFHOM FiniteFieldHomomorphisms}
\getchunk{package FFPOLY FiniteFieldPolynomialPackage}
\getchunk{package FFPOLY2 FiniteFieldPolynomialPackage2}
\getchunk{package FFSLPE FiniteFieldSolveLinearPolynomialEquation}
\getchunk{package FFSQFR FiniteFieldSquareFreeDecomposition}
\getchunk{package FLAGG2 FiniteLinearAggregateFunctions2}
\getchunk{package FLASORT FiniteLinearAggregateSort}
\getchunk{package FSAGG2 FiniteSetAggregateFunctions2}
\getchunk{package FLOATCP FloatingComplexPackage}
\getchunk{package FLOATRP FloatingRealPackage}
\getchunk{package FCPAK1 FortranCodePackage1}
\getchunk{package FOP FortranOutputStackPackage}
```

```
\getchunk{package FORT FortranPackage}
\getchunk{package FRIDEAL2 FractionalIdealFunctions2}
\getchunk{package FFFG FractionFreeFastGaussian}
\getchunk{package FFFGF FractionFreeFastGaussianFractions}
\getchunk{package FRAC2 FractionFunctions2}
\getchunk{package FRNAAF2 FramedNonAssociativeAlgebraFunctions2}
\getchunk{package FSPECF FunctionalSpecialFunction}
\getchunk{package FFCAT2 FunctionFieldCategoryFunctions2}
\getchunk{package FFINTBAS FunctionFieldIntegralBasis}
\getchunk{package PMASSFS FunctionSpaceAssertions}
\getchunk{package PMPREDFS FunctionSpaceAttachPredicates}
\getchunk{package FSCINT FunctionSpaceComplexIntegration}
\getchunk{package FS2 FunctionSpaceFunctions2}
\getchunk{package FSINT FunctionSpaceIntegration}
\getchunk{package FSPRMELT FunctionSpacePrimitiveElement}
\getchunk{package FSRED FunctionSpaceReduce}
\getchunk{package SUMFS FunctionSpaceSum}
\getchunk{package FS2EXPXP FunctionSpaceToExponentialExpansion}
\getchunk{package FS2UPS FunctionSpaceToUnivariatePowerSeries}
\getchunk{package FSUPFACT FunctionSpaceUnivariatePolynomialFactor}

\getchunk{package GALFACTU GaloisGroupFactorizationUtilities}
\getchunk{package GALFACT GaloisGroupFactorizer}
\getchunk{package GALPOLYU GaloisGroupPolynomialUtilities}
\getchunk{package GALUTIL GaloisGroupUtilities}
\getchunk{package GAUSSFAC GaussianFactorizationPackage}
\getchunk{package GHENSEL GeneralHenselPackage}
\getchunk{package GENMFACT GeneralizedMultivariateFactorize}
\getchunk{package GPAFF GeneralPackageForAlgebraicFunctionField}
\getchunk{package GENPGCD GeneralPolynomialGcdPackage}
\getchunk{package GENUPS GenerateUnivariatePowerSeries}
\getchunk{package GENEEZ GenExEuclid}
\getchunk{package GENUFACT GenUFactorize}
\getchunk{package INTG0 GenusZeroIntegration}
\getchunk{package GOSPER GosperSummationMethod}
\getchunk{package GRDEF GraphicsDefaults}
\getchunk{package GRAY GrayCode}
\getchunk{package GBF GroebnerFactorizationPackage}
\getchunk{package GBINTERN GroebnerInternalPackage}
\getchunk{package GB GroebnerPackage}
\getchunk{package GROEBSOL GroebnerSolve}
\getchunk{package GUESS Guess}
\getchunk{package GUESSAN GuessAlgebraicNumber}
\getchunk{package GUESSF GuessFinite}
\getchunk{package GUESSF1 GuessFiniteFunctions}
\getchunk{package GUESSINT GuessInteger}
\getchunk{package GUESSP GuessPolynomial}
\getchunk{package GUESSUP GuessUnivariatePolynomial}

\getchunk{package HB HallBasis}
```

```
\getchunk{package HEUGCD HeuGcd}

\getchunk{package IDECOMP IdealDecompositionPackage}
\getchunk{package INCRMAPS IncrementingMaps}
\getchunk{package INFPROD0 InfiniteProductCharacteristicZero}
\getchunk{package INPRODFF InfiniteProductFiniteField}
\getchunk{package INPRODPF InfiniteProductPrimeField}
\getchunk{package ITFUN2 InfiniteTupleFunctions2}
\getchunk{package ITFUN3 InfiniteTupleFunctions3}
\getchunk{package INFINITY Infinity}
\getchunk{package IALGFACT InnerAlgFactor}
\getchunk{package ICDEN InnerCommonDenominator}
\getchunk{package IMATLIN InnerMatrixLinearAlgebraFunctions}
\getchunk{package IMATQF InnerMatrixQuotientFieldFunctions}
\getchunk{package INMODGCD InnerModularGcd}
\getchunk{package INNMFACT InnerMultFact}
\getchunk{package INBFF InnerNormalBasisFieldFunctions}
\getchunk{package INEP InnerNumericEigenPackage}
\getchunk{package INFSP InnerNumericFloatSolvePackage}
\getchunk{package INPSIGN InnerPolySign}
\getchunk{package ISUMP InnerPolySum}
\getchunk{package ITRIGMNP InnerTrigonometricManipulations}
\getchunk{package INFORM1 InputFormFunctions1}
\getchunk{package INTBIT IntegerBits}
\getchunk{package COMBINAT IntegerCombinatoricFunctions}
\getchunk{package INTFACT IntegerFactorizationPackage}
\getchunk{package ZLINDEP IntegerLinearDependence}
\getchunk{package INTHEORY IntegerNumberTheoryFunctions}
\getchunk{package PRIMES IntegerPrimesPackage}
\getchunk{package INTRET IntegerRetractions}
\getchunk{package IROOT IntegerRoots}
\getchunk{package INTSLPE IntegerSolveLinearPolynomialEquation}
\getchunk{package IBATOOL IntegralBasisTools}
\getchunk{package IBPTOOLS IntegralBasisPolynomialTools}
\getchunk{package IR2 IntegrationResultFunctions2}
\getchunk{package IRRF2F IntegrationResultRFToFunction}
\getchunk{package IR2F IntegrationResultToFunction}
\getchunk{package INTTOOLS IntegrationTools}
\getchunk{package INTERGB InterfaceGroebnerPackage}
\getchunk{package IPRNTPK InternalPrintPackage}
\getchunk{package IRURPK InternalRationalUnivariateRepresentationPackage}
\getchunk{package INTFRSP InterpolateFormsPackage}
\getchunk{package INTDIVP IntersectionDivisorPackage}
\getchunk{package IRREDFFX IrredPolyOverFiniteField}
\getchunk{package IRSN IrrRepSymNatPackage}
\getchunk{package INVLAPLA InverseLaplaceTransform}

\getchunk{package KERNEL2 KernelFunctions2}
\getchunk{package KOVACIC Kovacic}
```

```
\getchunk{package LAPLACE LaplaceTransform}
\getchunk{package LAZM3PK LazardSetSolvingPackage}
\getchunk{package LEADCDET LeadingCoefDetermination}
\getchunk{package LEXTRIPK LexTriangularPackage}
\getchunk{package LINDEP LinearDependence}
\getchunk{package LODOF LinearOrdinaryDifferentialOperatorFactorizer}
\getchunk{package LODOOPS LinearOrdinaryDifferentialOperatorsOps}
\getchunk{package LPEFRAC LinearPolynomialEquationByFractions}
\getchunk{package LSMP LinearSystemMatrixPackage}
\getchunk{package LSMP1 LinearSystemMatrixPackage1}
\getchunk{package LSPP LinearSystemPolynomialPackage}
\getchunk{package LGROBP LinGroebnerPackage}
\getchunk{package LOP LinesOpPack}
\getchunk{package LF LiouvillianFunction}
\getchunk{package LIST2 ListFunctions2}
\getchunk{package LIST3 ListFunctions3}
\getchunk{package LIST2MAP ListToMap}
\getchunk{package LPARSPT LocalParametrizationOfSimplePointPackage}

\getchunk{package MKBCFUNC MakeBinaryCompiledFunction}
\getchunk{package MKFLCFN MakeFloatCompiledFunction}
\getchunk{package MKFUNC MakeFunction}
\getchunk{package MKRECORD MakeRecord}
\getchunk{package MKUCFUNC MakeUnaryCompiledFunction}
\getchunk{package MAPHACK1 MappingPackageInternalHacks1}
\getchunk{package MAPHACK2 MappingPackageInternalHacks2}
\getchunk{package MAPHACK3 MappingPackageInternalHacks3}
\getchunk{package MAPPKG1 MappingPackage1}
\getchunk{package MAPPKG2 MappingPackage2}
\getchunk{package MAPPKG3 MappingPackage3}
\getchunk{package MAPPKG4 MappingPackage4}
\getchunk{package MATCAT2 MatrixCategoryFunctions2}
\getchunk{package MCDEN MatrixCommonDenominator}
\getchunk{package MATLIN MatrixLinearAlgebraFunctions}
\getchunk{package MTHING MergeThing}
\getchunk{package MESH MeshCreationRoutinesForThreeDimensions}
\getchunk{package MDDFACT ModularDistinctDegreeFactorizer}
\getchunk{package MHROWRED ModularHermitianRowReduction}
\getchunk{package MRF2 MonoidRingFunctions2}
\getchunk{package MONOTOOL MonomialExtensionTools}
\getchunk{package MSYSCMD MoreSystemCommands}
\getchunk{package MPCPF MPolyCatPolyFactorizer}
\getchunk{package MPRFF MPolyCatRationalFunctionFactorizer}
\getchunk{package MPC2 MPolyCatFunctions2}
\getchunk{package MPC3 MPolyCatFunctions3}
\getchunk{package MRATFAC MRationalFactorize}
\getchunk{package MFINFACT MultFiniteFactorize}
\getchunk{package MMAP MultipleMap}
\getchunk{package MCALCFN MultiVariableCalculusFunctions}
\getchunk{package MULTFACT MultivariateFactorize}
```

```
\getchunk{package MLIFT MultivariateLifting}
\getchunk{package MULTSQFR MultivariateSquareFree}

\getchunk{package NAGF02 NagEigenPackage}
\getchunk{package NAGE02 NagFittingPackage}
\getchunk{package NAGF04 NagLinearEquationSolvingPackage}
\getchunk{package NAGSP NAGLinkSupportPackage}
\getchunk{package NAGD01 NagIntegrationPackage}
\getchunk{package NAGE01 NagInterpolationPackage}
\getchunk{package NAGF07 NagLapack}
\getchunk{package NAGF01 NagMatrixOperationsPackage}
\getchunk{package NAGE04 NagOptimisationPackage}
\getchunk{package NAGD02 NagOrdinaryDifferentialEquationsPackage}
\getchunk{package NAGD03 NagPartialDifferentialEquationsPackage}
\getchunk{package NAGC02 NagPolynomialRootsPackage}
\getchunk{package NAGC05 NagRootFindingPackage}
\getchunk{package NAGC06 NagSeriesSummationPackage}
\getchunk{package NAGS NagSpecialFunctionsPackage}
\getchunk{package NSUP2 NewSparseUnivariatePolynomialFunctions2}
\getchunk{package NEWTON NewtonInterpolation}
\getchunk{package NPOLYGON NewtonPolygon}
\getchunk{package NCODIV NonCommutativeOperatorDivision}
\getchunk{package NONE1 NoneFunctions1}
\getchunk{package NODE1 NonLinearFirstOrderODESolver}
\getchunk{package NLINSOL NonLinearSolvePackage}
\getchunk{package NORMPK NormalizationPackage}
\getchunk{package NORMMA NormInMonogenicAlgebra}
\getchunk{package NORMRETR NormRetractPackage}
\getchunk{package NPCOEF NPCoef}
\getchunk{package NFINTBAS NumberFieldIntegralBasis}
\getchunk{package NUMFMT NumberFormats}
\getchunk{package NTPOLFN NumberTheoreticPolynomialFunctions}
\getchunk{package NUMERIC Numeric}
\getchunk{package NUMODE NumericalOrdinaryDifferentialEquations}
\getchunk{package NUMQUAD NumericalQuadrature}
\getchunk{package NCEP NumericComplexEigenPackage}
\getchunk{package NCNTFRAC NumericContinuedFraction}
\getchunk{package NREP NumericRealEigenPackage}
\getchunk{package NUMTUBE NumericTubePlot}

\getchunk{package OCTCT2 OctonionCategoryFunctions2}
\getchunk{package ODEINT ODEIntegration}
\getchunk{package ODETOOLS ODETools}
\getchunk{package ARRAY12 OneDimensionalArrayFunctions2}
\getchunk{package ONECOMP2 OnePointCompletionFunctions2}
\getchunk{package OMPKG OpenMathPackage}
\getchunk{package OMSERVER OpenMathServerPackage}
\getchunk{package OPQUERY OperationsQuery}
\getchunk{package ORDCOMP2 OrderedCompletionFunctions2}
\getchunk{package ORDFUNS OrderingFunctions}
```

```
\getchunk{package ORTHPOL OrthogonalPolynomialFunctions}
\getchunk{package OUT OutputPackage}

\getchunk{package PAFF PackageForAlgebraicFunctionField}
\getchunk{package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField}
\getchunk{package PFORP PackageForPoly}
\getchunk{package PADEPAC PadeApproximantPackage}
\getchunk{package PADE PadeApproximants}
\getchunk{package PWFFINTB PAdicWildFunctionFieldIntegralBasis}
\getchunk{package YSTREAM ParadoxicalCombinatorsForStreams}
\getchunk{package PLEQN ParametricLinearEquations}
\getchunk{package PARPC2 ParametricPlaneCurveFunctions2}
\getchunk{package PARSC2 ParametricSpaceCurveFunctions2}
\getchunk{package PARSU2 ParametricSurfaceFunctions2}
\getchunk{package PARAMP ParametrizationPackage}
\getchunk{package PFRPAC PartialFractionPackage}
\getchunk{package PARTPERM PartitionsAndPermutations}
\getchunk{package PATTERN1 PatternFunctions1}
\getchunk{package PATTERN2 PatternFunctions2}
\getchunk{package PATMATCH PatternMatch}
\getchunk{package PMASS PatternMatchAssertions}
\getchunk{package PMFS PatternMatchFunctionSpace}
\getchunk{package PMINS PatternMatchIntegerNumberSystem}
\getchunk{package INTPM PatternMatchIntegration}
\getchunk{package PMKERNEL PatternMatchKernel}
\getchunk{package PMLSAGG PatternMatchListAggregate}
\getchunk{package PMPLCAT PatternMatchPolynomialCategory}
\getchunk{package PMDOWN PatternMatchPushDown}
\getchunk{package PMQFCAT PatternMatchQuotientFieldCategory}
\getchunk{package PATRES2 PatternMatchResultFunctions2}
\getchunk{package PMSYM PatternMatchSymbol}
\getchunk{package PMTOOLS PatternMatchTools}
\getchunk{package PERMAN Permanent}
\getchunk{package PGE PermutationGroupExamples}
\getchunk{package PICOERCE PiCoercions}
\getchunk{package PLOT1 PlotFunctions1}
\getchunk{package PLOTTOOL PlotTools}
\getchunk{package PTFUNC2 PointFunctions2}
\getchunk{package PTPACK PointPackage}
\getchunk{package PFO PointsOfFiniteOrder}
\getchunk{package PFOQ PointsOfFiniteOrderRational}
\getchunk{package PFOTOOLS PointsOfFiniteOrderTools}
\getchunk{package POLTOPOL PolToPol}
\getchunk{package PGROEB PolyGroebner}
\getchunk{package PAN2EXPR PolynomialAN2Expression}
\getchunk{package POLYLIFT PolynomialCategoryLifting}
\getchunk{package POLYCATQ PolynomialCategoryQuotientFunctions}
\getchunk{package PCOMP PolynomialComposition}
\getchunk{package PDECOMP PolynomialDecomposition}
\getchunk{package PFBR PolynomialFactorizationByRecursion}
```

```
\getchunk{package PFBRU PolynomialFactorizationByRecursionUnivariate}
\getchunk{package POLY2 PolynomialFunctions2}
\getchunk{package PGCD PolynomialGcdPackage}
\getchunk{package PINTERP PolynomialInterpolation}
\getchunk{package PINTERPA PolynomialInterpolationAlgorithms}
\getchunk{package PNTHEORY PolynomialNumberTheoryFunctions}
\getchunk{package PLPKCRV PolynomialPackageForCurve}
\getchunk{package POLYROOT PolynomialRoots}
\getchunk{package PSETPK PolynomialSetUtilitiesPackage}
\getchunk{package SOLVEFOR PolynomialSolveByFormulas}
\getchunk{package PSQFR PolynomialSquareFree}
\getchunk{package POLY2UP PolynomialToUnivariatePolynomial}
\getchunk{package LIMITPS PowerSeriesLimitPackage}
\getchunk{package PREASSOC PrecomputedAssociatedEquations}
\getchunk{package PRIMARR2 PrimitiveArrayFunctions2}
\getchunk{package PRIMELT PrimitiveElement}
\getchunk{package ODEPRIM PrimitiveRatDE}
\getchunk{package ODEPRRIC PrimitiveRatRicDE}
\getchunk{package PRINT PrintPackage}
\getchunk{package PRJALGPK ProjectiveAlgebraicSetPackage}
\getchunk{package PSEUDLIN PseudoLinearNormalForm}
\getchunk{package PRS PseudoRemainderSequence}
\getchunk{package INTPAF PureAlgebraicIntegration}
\getchunk{package ODEPAL PureAlgebraicLODE}
\getchunk{package PUSHVAR PushVariables}

\getchunk{package QALGSET2 QuasiAlgebraicSet2}
\getchunk{package QCMPACK QuasiComponentPackage}
\getchunk{package QFCAT2 QuotientFieldCategoryFunctions2}

\getchunk{package REP RadicalEigenPackage}
\getchunk{package SOLVERAD RadicalSolvePackage}
\getchunk{package RADUTIL RadixUtilities}
\getchunk{package RDIST RandomDistributions}
\getchunk{package RFDIST RandomFloatDistributions}
\getchunk{package RIDIST RandomIntegerDistributions}
\getchunk{package RANDSRC RandomNumberSource}
\getchunk{package RATFACT RationalFactorize}
\getchunk{package RF RationalFunction}
\getchunk{package DEFINTRF RationalFunctionDefiniteIntegration}
\getchunk{package RFFACT RationalFunctionFactor}
\getchunk{package RFFACTOR RationalFunctionFactorizer}
\getchunk{package INTRF RationalFunctionIntegration}
\getchunk{package LIMITRF RationalFunctionLimitPackage}
\getchunk{package SIGNRF RationalFunctionSign}
\getchunk{package SUMRF RationalFunctionSum}
\getchunk{package INTRAT RationalIntegration}
\getchunk{package RINTERP RationalInterpolation}
\getchunk{package ODERAT RationalLODE}
\getchunk{package RATRET RationalRetractions}
```

```
\getchunk{package ODERTRIC RationalRicDE}
\getchunk{package RURPK RationalUnivariateRepresentationPackage}
\getchunk{package POLUTIL RealPolynomialUtilitiesPackage}
\getchunk{package REALSOLV RealSolvePackage}
\getchunk{package REAL0 RealZeroPackage}
\getchunk{package REAL0Q RealZeroPackageQ}
\getchunk{package RMCAT2 RectangularMatrixCategoryFunctions2}
\getchunk{package RECOP RecurrenceOperator}
\getchunk{package RDIV ReducedDivisor}
\getchunk{package ODERED ReduceLODE}
\getchunk{package REDORDER ReductionOfOrder}
\getchunk{package RSDCMPK RegularSetDecompositionPackage}
\getchunk{package RSETGCD RegularTriangularSetGcdPackage}
\getchunk{package REPDB RepeatedDoubling}
\getchunk{package REPSQ RepeatedSquaring}
\getchunk{package REP1 RepresentationPackage1}
\getchunk{package REP2 RepresentationPackage2}
\getchunk{package RESLATC ResolveLatticeCompletion}
\getchunk{package RETSOL RetractSolvePackage}
\getchunk{package RFP RootsFindingPackage}

\getchunk{package SAERFFC SAERationalFunctionAlgFactor}
\getchunk{package FORMULA1 ScriptFormulaFormat1}
\getchunk{package SEGBIND2 SegmentBindingFunctions2}
\getchunk{package SEG2 SegmentFunctions2}
\getchunk{package SAEFACT SimpleAlgebraicExtensionAlgFactor}
\getchunk{package SIMPAN SimplifyAlgebraicNumberConvertPackage}
\getchunk{package SMITH SmithNormalForm}
\getchunk{package SCACHE SortedCache}
\getchunk{package SORTPAK SortPackage}
\getchunk{package SUP2 SparseUnivariatePolynomialFunctions2}
\getchunk{package SPECOUT SpecialOutputPackage}
\getchunk{package SFQCMPK SquareFreeQuasiComponentPackage}
\getchunk{package SRDCMPK SquareFreeRegularSetDecompositionPackage}
\getchunk{package SFRGCD SquareFreeRegularTriangularSetGcdPackage}
\getchunk{package MATSTOR StorageEfficientMatrixOperations}
\getchunk{package STREAM1 StreamFunctions1}
\getchunk{package STREAM2 StreamFunctions2}
\getchunk{package STREAM3 StreamFunctions3}
\getchunk{package STINPROD StreamInfiniteProduct}
\getchunk{package STTAYLOR StreamTaylorSeriesOperations}
\getchunk{package STNSR StreamTensor}
\getchunk{package STTF StreamTranscendentalFunctions}
\getchunk{package STTFNC StreamTranscendentalFunctionsNonCommutative}
\getchunk{package SCPKG StructuralConstantsPackage}
\getchunk{package SHP SturmHabichtPackage}
\getchunk{package SUBRESP SubResultantPackage}
\getchunk{package SUPFRACF SupFractionFactorizer}
\getchunk{package ODESYS SystemODESolver}
\getchunk{package SYSSOLP SystemSolvePackage}
```

```
\getchunk{package SGCF SymmetricGroupCombinatoricFunctions}
\getchunk{package SYMFUNC SymmetricFunctions}

\getchunk{package TABLBUMP TableauxBumpers}
\getchunk{package TBCMPPK TabulatedComputationPackage}
\getchunk{package TANEXP TangentExpansions}
\getchunk{package UTSSOL TaylorSolve}
\getchunk{package TEMUTL TemplateUtilities}
\getchunk{package TEX1 TexFormat1}
\getchunk{package TOOLSIGN ToolsForSign}
\getchunk{package DRAW TopLevelDrawFunctions}
\getchunk{package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves}
\getchunk{package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions}
\getchunk{package DRAWPT TopLevelDrawFunctionsForPoints}
\getchunk{package TOPSP TopLevelThreeSpace}
\getchunk{package INTHERTR TranscendentalHermiteIntegration}
\getchunk{package INTTR TranscendentalIntegration}
\getchunk{package TRMANIP TranscendentalManipulations}
\getchunk{package RDETR TranscendentalRischDE}
\getchunk{package RDETRS TranscendentalRischDESystem}
\getchunk{package SOLVETRA TransSolvePackage}
\getchunk{package SOLVESER TransSolvePackageService}
\getchunk{package TRIMAT TriangularMatrixOperations}
\getchunk{package TRIGMNIP TrigonometricManipulations}
\getchunk{package TUBETOOL TubePlotTools}
\getchunk{package CLIP TwoDimensionalPlotClipping}
\getchunk{package TWOFACT TwoFactorize}

\getchunk{package UNIFACT UnivariateFactorize}
\getchunk{package UFPS1 UnivariateFormalPowerSeriesFunctions}
\getchunk{package ULS2 UnivariateLaurentSeriesFunctions2}
\getchunk{package UPOLYC2 UnivariatePolynomialCategoryFunctions2}
\getchunk{package UPCDEN UnivariatePolynomialCommonDenominator}
\getchunk{package UPDECOMP UnivariatePolynomialDecompositionPackage}
\getchunk{package UPDIVP UnivariatePolynomialDivisionPackage}
\getchunk{package UP2 UnivariatePolynomialFunctions2}
\getchunk{package UPMP UnivariatePolynomialMultiplicationPackage}
\getchunk{package UPSQFREE UnivariatePolynomialSquareFree}
\getchunk{package UPXS2 UnivariatePuiseuxSeriesFunctions2}
\getchunk{package OREPCTO UnivariateSkewPolynomialCategoryOps}
\getchunk{package UTS2 UnivariateTaylorSeriesFunctions2}
\getchunk{package UTSODE UnivariateTaylorSeriesODESolver}
\getchunk{package UNISEG2 UniversalSegmentFunctions2}
\getchunk{package UDPO UserDefinedPartialOrdering}
\getchunk{package UDVO UserDefinedVariableOrdering}
\getchunk{package UTSODETL UTSodetools}

\getchunk{package VECTOR2 VectorFunctions2}
\getchunk{package VIEWDEF ViewDefaultsPackage}
\getchunk{package VIEW ViewportPackage}
```

```
\getchunk{package WEIER WeierstrassPreparation}
\getchunk{package WFFINTBS WildFunctionFieldIntegralBasis}

\getchunk{package XEXPPKG XExponentialPackage}

\getchunk{package ZDSOLVE ZeroDimensionalSolvePackage}
```

———-

Chapter 29

Index

Index

-?
    STTAYLOR, 3138
?**?
    AF, 13
    COMBF, 175
    EFULS, 451
    EFUPXS, 459
    INBFF, 1021
    MAPPKG1, 1341
    MATSTOR, 3126
    STTF, 3150
    STTFNC, 3160
?*?
    INBFF, 1021
    MAPPKG3, 1362
    MAPPKG4, 1369
    PRS, 2864
    STTAYLOR, 3138
    TUBETOOL, 3308
?+?
    MAPPKG4, 1369
    STTAYLOR, 3138
    TUBETOOL, 3308
?-?
    MAPPKG4, 1369
    STTAYLOR, 3138
    TUBETOOL, 3308
?/?
    INBFF, 1021
    MAPPKG4, 1369
    STTAYLOR, 3138

abelianGroup
    PGE, 2718
abs
    FSPECF, 699
accuracyIF
    D02AGNT, 375
acos
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
acosh
    EF, 403
    EFULS, 451
    EFUPXS, 459
acoshIfCan
    EFULS, 451
    EFUPXS, 459
acosIfCan
    EFULS, 451
    EFUPXS, 459
acot
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
acoth
    EF, 403
    EFULS, 451
    EFUPXS, 459
acothIfCan
    EFULS, 451
    EFUPXS, 459
acotIfCan
    EFULS, 451
    EFUPXS, 459
acsc
    EF, 403
    EFULS, 451
    EFUPXS, 459

STTF, 3150
STTFNC, 3160
acsch
EF, 403
EFULS, 451
EFUPXS, 459
acschIfCan
EFULS, 451
EFUPXS, 459
acscIfCan
EFULS, 451
EFUPXS, 459
aCubic
SOLVEFOR, 2819
adaptive
DROPT0, 357
GRDEF, 854
addBadValue
PATTERN1, 2675
addiag
STTAYLOR, 3138
adjoint
MATLIN, 1375
adjunctionDivisor
DTP, 268
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
AF, 13
?**?, 13
belong?, 13
definingPolynomial, 13
droot, 13
inrootof, 13
iroot, 13
minPoly, 13
operator, 13
rootOf, 13
AFALGGRO, 4
affineAlgSet, 4
affineRationalPoints, 4
affineSingularPoints, 4
AFALGRES, 9
affineAlgSet, 9
affineAlgSetLocal, 9
affineRationalPoints, 9
affineSingularPoints, 9
allPairsAmong, 9
polyRing2UPUP, 9
AffineAlgebraicSetComputeWithGroebnerBasis, 4
AffineAlgebraicSetComputeWithResultant, 9
affineAlgSet
AFALGGRO, 4
AFALGRES, 9
affineAlgSetLocal
AFALGRES, 9
affineRationalPoints
AFALGGRO, 4
AFALGRES, 9
affineSingularPoints
AFALGGRO, 4
AFALGRES, 9
airyAi
DFSFUN, 309
FSPECF, 699
airyBi
DFSFUN, 309
FSPECF, 699
algDsolve
ODEPAL, 2892
algebraicDecompose
RSDCMPK, 3029
SRDCMPK, 3110
AlgebraicFunction, 13
AlgebraicHermiteIntegration, 19
AlgebraicIntegrate, 21
AlgebraicIntegration, 28
AlgebraicManipulations, 30
AlgebraicMultFact, 35
algebraicSet
PRJALGPK, 2733
algebraicSort
QCMPACK, 2900
SFQCMPK, 3101
AlgebraPackage, 37
ALGFACT, 47
doublyTransitive?, 47
factor, 47
split, 47
AlgFactor, 47
algint

INTAF, 28
algintegrate
INTALG, 21
ALGMANIP, 30
ratDenom, 30
ratPoly, 30
rootKerSimp, 30
rootPower, 30
rootProduct, 30
rootSimp, 30
rootSplit, 30
ALGMFACT, 35
factor, 35
ALGPKG, 37
basis, 37
basisOfCenter, 37
basisOfCentroid, 37
basisOfCommutingElements, 37
basisOfLeftAnnihilator, 37
basisOfLeftNucleus, 37
basisOfLeftNucloid, 37
basisOfMiddleNucleus, 37
basisOfNucleus, 37
basisOfRightAnnihilator, 37
basisOfRightNucleus, 37
basisOfRightNucloid, 37
biRank, 37
doubleRank, 37
leftRank, 37
radicalOfLeftTraceForm, 37
rightRank, 37
weakBiRank, 37
aLinear
SOLVEFOR, 2819
allPairsAmong
AFALGRES, 9
alternating
CYCLES, 249
alternatingGroup
PGE, 2718
anfactor
FSUPFACT, 758
AnnaNumericalIntegrationPackage, 50
AnnaNumericalOptimizationPackage, 61
AnnaOrdinaryDifferentialEquationPackage, 69
AnnaPartialDifferentialEquationPackage, 78
anticoord
LGROBP, 1285
antisymmetricTensors
REP1, 3046
ANY1, 84
coerce, 84
retract, 84
retractable?, 84
retractIfCan, 84
AnyFunctions1, 84
API, 90
ApplicationProgramInterface, 90
apply
APPLYORE, 95
OREPCTO, 3349
APPLYORE, 95
apply, 95
ApplyRules, 92
applyRules
APPRULE, 92
applyTransform
BLUPPACK, 146
ApplyUnivariateSkewPolynomial, 95
approxNthRoot
IROOT, 1099
approxSqrt
IROOT, 1099
APPRULE, 92
applyRules, 92
localUnquote, 92
aQuadratic
SOLVEFOR, 2819
aQuartic
SOLVEFOR, 2819
areEquivalent?
REP2, 3053
arg1
MAPHACK2, 1330
arg2
MAPHACK2, 1330
aromberg
NUMQUAD, 2563
ARRAY12, 2591
map, 2591
reduce, 2591
scan, 2591

asec
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
asech
    EF, 403
    EFULS, 451
    EFUPXS, 459
asechIfCan
    EFULS, 451
    EFUPXS, 459
asecIfCan
    EFULS, 451
    EFUPXS, 459
asimpson
    NUMQUAD, 2563
asin
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
asinh
    EF, 403
    EFULS, 451
    EFUPXS, 459
asinhIfCan
    EFULS, 451
    EFUPXS, 459
asinIfCan
    EFULS, 451
    EFUPXS, 459
aspFilename
    NAGSP, 1739
assert
    PMASS, 2681
    PMASSFS, 712
ASSOCEQ, 96
    associatedEquations, 96
    associatedSystem, 96
    axServer, 100
    getDatabase, 100
    uncouplingMatrices, 96
AssociatedEquations, 96
associatedEquations
    ASSOCEQ, 96
associatedSystem
    ASSOCEQ, 96
atan
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
atanh
    EF, 403
    EFULS, 451
    EFUPXS, 459
atanhIfCan
    EFULS, 451
    EFUPXS, 459
atanIfCan
    EFULS, 451
    EFUPXS, 459
atrapezoidal
    NUMQUAD, 2563
att2Result
    ESTOOLS, 534
AttachPredicates, 99
axesColorDefault
    VIEWDEF, 3367
AxiomServer, 100
AXSERV, 100
    multiServ, 100
axServer
    ASSOCEQ, 100

B1solve
    PLEQN, 2648
badNum
    PFOTOOLS, 2749
badValues
    PATTERN1, 2675
BalancedFactorisation, 117
balancedFactorisation
    BALFACT, 117
BALFACT, 117
    balancedFactorisation, 117
bandedHessian
    MCALCFN, 1426

bandedJacobian
MCALCFN, 1426
baseRDE
RDETR, 3273
baseRDEsys
RDETRS, 3277
BasicMethod
INTFACT, 1057
BasicOperatorFunctions1, 119
basis
ALGPKG, 37
INBFF, 1021
basisOfCenter
ALGPKG, 37
basisOfCentroid
ALGPKG, 37
basisOfCommutingElements
ALGPKG, 37
basisOfInterpolateForms
INTFRSP, 1131
basisOfInterpolateFormsForFact
INTFRSP, 1131
basisOfLeftAnnihilator
ALGPKG, 37
basisOfLeftNucleus
ALGPKG, 37
basisOfLeftNucloid
ALGPKG, 37
basisOfMiddleNucleus
ALGPKG, 37
basisOfNucleus
ALGPKG, 37
basisOfRightAnnihilator
ALGPKG, 37
basisOfRightNucleus
ALGPKG, 37
basisOfRightNucloid
ALGPKG, 37
bat
TABLBUMP, 3207
bat1
TABLBUMP, 3207
beauzamyBound
GALFACTU, 763
belong?
AF, 13
COMBF, 175
EF, 403
FSPECF, 699
LF, 1296
bernoulli
INTHEORY, 1082
PNTHEORY, 2793
bernoulliB
NTPOLFN, 2544
besselI
DFSFUN, 309
FSPECF, 699
besselJ
DFSFUN, 309
FSPECF, 699
besselK
DFSFUN, 309
FSPECF, 699
besselY
DFSFUN, 309
FSPECF, 699
Beta
DFSFUN, 309
FSPECF, 699
RFDIST, 2939
BEZIER, 139
cubicBezier, 139
linearBezier, 139
quadraticBezier, 139
Bezier, 139
BEZOUT, 141
bezoutDiscriminant, 141
bezoutMatrix, 141
bezoutResultant, 141
brillhartIrreducible?, 154
brillhartTrials, 154
sylvesterMatrix, 141
bezoutDiscriminant
BEZOUT, 141
BezoutMatrix, 141
bezoutMatrix
BEZOUT, 141
bezoutResultant
BEZOUT, 141
binaryFunction
MKBCFUNC, 1315

binomial
COMBF, 175
COMBINAT, 1053
RIDIST, 2941
bipolar
COORDSYS, 224
bipolarCylindrical
COORDSYS, 224
biRank
ALGPKG, 37
biringToPolyRing
BLUPPACK, 146
bitCoef
INTBIT, 1049
bitLength
INTBIT, 1049
bitTruth
INTBIT, 1049
bivariate?
PSETPK, 2802
bivariatePolynomials
PSETPK, 2802
bivariateSLPEBR
PFBR, 2768
blowUp
DTP, 268
BlowUpPackage, 146
blowUpWithExcpDiv
DTP, 268
BLUPPACK, 146
applyTransform, 146
biringToPolyRing, 146
newtonPolySlope, 146
polyRingToBlUpRing, 146
quadTransform, 146
stepBlowUp, 146
bombieriNorm
GALFACTU, 763
BOP1, 119
constantOperator, 119
constantOpIfCan, 119
derivative, 119
evaluate, 119
BoundIntegerRoots, 151
boundOfCauchy
POLUTIL, 2993
BOUNDZRO, 151
integerBound, 152
branchIfCan
QCMPACK, 2900
SFQCMPK, 3101
bright
DISPLAY, 285
BRILL, 154
noLinearFactor?, 154
brillhartIrreducible?
BEZOUT, 154
BrillhartTests, 154
brillhartTrials
BEZOUT, 154
bringDown
FSRED, 726
bsolve
PLEQN, 2648
btwFact
GALFACT, 767
BumInSepFFE
UPSQFREE, 3345
bumprow
TABLBUMP, 3207
bumptab
TABLBUMP, 3207
bumptab1
TABLBUMP, 3207

c02aff
NAGC02, 2267
c02agf
NAGC02, 2267
c05adf
NAGC05, 2286
c05nbf
NAGC05, 2286
c05pbf
NAGC05, 2286
c06eaf
NAGC06, 2333
c06ebf
NAGC06, 2333
c06ecf
NAGC06, 2333
c06ekf

NAGC06, 2333
c06fpf
NAGC06, 2333
c06fqf
NAGC06, 2333
c06frf
NAGC06, 2333
c06fuf
NAGC06, 2333
c06gbf
NAGC06, 2333
c06gcf
NAGC06, 2333
c06gqf
NAGC06, 2333
c06gsf
NAGC06, 2333
cache
SCACHE, 3092
calcRanges
PLOTTOOL, 2729
cap
CYCLES, 249
CARTEN2, 157
map, 157
reshape, 157
cartesian
COORDSYS, 224
CartesianTensorFunctions2, 157
CDEN, 187
clearDenominator, 187
center
DISPLAY, 285
central?
D03AGNT, 381
certainlySubVariety?
PSETPK, 2802
cfirst
WEIER, 3375
chainSubResultants
PRS, 2864
changeBase
PSEUDLIN, 2860
changeName
D01AGNT, 363
changeNameToObjf
E04AGNT, 577
ChangeOfVariable, 159
changeVar
ODEPRRIC, 2853
characteristicPolynomial
CHARPOL, 164
CPIMA, 162
EP, 385
NCEP, 2575
NREP, 2579
CharacteristicPolynomialInMonogenicalAlgebra, 162
CharacteristicPolynomialPackage, 164
CHARPOL, 164
characteristicPolynomial, 164
charpol
INEP, 1028
chebyshevT
ORTHPOL, 2603
PNTHEORY, 2793
chebyshevU
ORTHPOL, 2603
PNTHEORY, 2793
check
MULTSQFR, 1436
checkForZero
DFINTTLS, 259
checkPrecision
NAGSP, 1739
checkRur
IRURPK, 1125
chineseRemainder
CRAPACK, 228
IBACHIN, 165
INTHEORY, 1082
ChineseRemainderToolsForIntegralBases, 165
chiSquare
RFDIST, 2939
chiSquare1
RFDIST, 2939
choosemon
LGROBP, 1285
CHVAR, 159
chvar, 159
eval, 159
goodPoint, 159

mkIntegral, 159
radPoly, 159
rootPoly, 159
chvar
CHVAR, 159
Ci
LF, 1296
CINTSLPE, 199
solveLinearPolynomialEquation, 199
classNumber
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
clearCache
SCACHE, 3092
clearDenominator
CDEN, 187
ICDEN, 996
MCDEN, 1373
UPCDEN, 3334
clearFortranOutputStack
FOP, 676
clikeUniv
WEIER, 3375
CLIP, 3311
clip, 3311
clipParametric, 3311
clipWithRanges, 3311
clip
CLIP, 3311
clipBoolean
DROPT0, 357
clipParametric
CLIP, 3311
clipPointsDefault
GRDEF, 854
clipWithRanges
CLIP, 3311
CMPLXRT, 216
complexZeros, 216
cn
ELFUTS, 485
coefChoose
MULTSQFR, 1436
coerce
ANY1, 84
CVMP, 170
DRAWHACK, 356
FFHOM, 622
FORMULA1, 3080
MAPPKG1, 1341
NONE1, 2514
PAN2EXPR, 2759
PICOERCE, 2726
RESLATC, 3070
RF, 2947
STTAYLOR, 3138
TEX1, 3221
VIEW, 3372
coerceP
CVMP, 170
CoerceVectorMatrixPackage, 170
coleman
SGCF, 3194
color
PTPACK, 2738
COMBF, 175
?**?, 175
belong?, 175
binomial, 175
commonDenominator, 187
factorial, 175
factorials, 175
iibinom, 175
iidprod, 175
iidsum, 175
iifact, 175
iiperm, 175
iipow, 175
ipow, 175
operator, 175
permutation, 175
product, 175
splitDenominator, 187
summation, 175
COMBINAT, 1053
binomial, 1053
factorial, 1053
multinomial, 1053
partition, 1053
permutation, 1053
stirling1, 1053

stirling2, 1053
CombinatorialFunction, 175
combineFeatureCompatibility
D02AGNT, 375
commaSeparate
D01AGNT, 363
CommonDenominator, 187
commonDenominator
COMBF, 187
ICDEN, 996
MCDEN, 1373
UPCDEN, 3334
COMMONOP, 189
operator, 189
CommonOperators, 189
COMMUPC, 194
swap, 194
CommuteUnivariatePolynomialCategory, 194
comp
MAPHACK3, 1331
companionBlocks
PSEUDLIN, 2860
compBound
GENEEZ, 834
compdegd
MULTSQFR, 1436
COMPFACT, 195
factor, 196
compiledFunction
MKBCFUNC, 1315
MKUCFUNC, 1327
complete
CYCLES, 249
completeEchelonBasis
REP2, 3053
completeEval
FACUTIL, 593
completeHensel
GHENSEL, 794
completeHermite
SMITH, 3087
completeSmith
SMITH, 3087
COMPLEX2, 198
map, 198
complexEigenvalues
NCEP, 2575
complexEigenvectors
NCEP, 2575
complexElementary
CTRIGMNP, 218
TRIGMNIP, 3304
complexExpand
IR2F, 1115
IRRF2F, 1113
ComplexFactorization, 195
complexForm
CTRIGMNP, 218
TRIGMNIP, 3304
ComplexFunctions2, 198
ComplexIntegerSolveLinearPolynomialEquation, 199
complexIntegrate
FSCINT, 716
IRRF2F, 1113
complexLimit
LIMITPS, 2830
LIMITRF, 2957
complexNormalize
CTRIGMNP, 218
TRIGMNIP, 3304
complexNumeric
NUMERIC, 2546
complexNumericIfCan
NUMERIC, 2546
ComplexPattern, 201
ComplexPatternMatch, 202
ComplexRootFindingPackage, 204
ComplexRootPackage, 216
complexRoots
FLOATCP, 667
complexSolve
FLOATCP, 667
ComplexTrigonometricManipulations, 218
complexZeros
CMPLXRT, 216
CRFP, 204
COMPLPAT, 201
convert, 201
compose
PCOMP, 2765
STTAYLOR, 3138

computeBasis
LGROBP, 1285
computeCycleEntry
CSTTOOLS, 254
computeCycleLength
CSTTOOLS, 254
computeInt
DFINTTLS, 259
concat
ESTOOLS, 534
STREAM1, 3130
conical
COORDSYS, 224
conjugate
PARTPERM, 2672
conjugates
PARTPERM, 2672
consnewpol
MULTSQFR, 1436
const
MAPPKG2, 1352
constant
MAPPKG2, 1352
PFORP, 2628
PMASS, 2681
PMASSFS, 712
constantCoefficientRicDE
ODEPRRIC, 2853
constantIfCan
KERNEL2, 1155
constantKernel
KERNEL2, 1155
constantLeft
MAPPKG3, 1362
ConstantLODE, 221
constantOperator
BOP1, 119
constantOpIfCan
BOP1, 119
constantRight
MAPPKG3, 1362
constantToUnaryFunction
EXPRTUBE, 569
constDsolve
ODECONST, 221
content
HEUGCD, 966
continuedFraction
NCNTFRAC, 2577
contract
IDECOMP, 973
contractSolve
SOLVERAD, 2929
convert
COMPLPAT, 201
ESTOOLS, 534
RSDCMPK, 3029
SRDCMPK, 3110
ZDSOLVE, 3454
coord
DROPT0, 357
LGROBP, 1285
coordinates
SCPKG, 3165
CoordinateSystems, 223
COORDSYS, 223
bipolar, 224
bipolarCylindrical, 224
cartesian, 224
conical, 224
cylindrical, 224
elliptic, 224
ellipticCylindrical, 224
oblateSpheroidal, 224
parabolic, 224
parabolicCylindrical, 224
paraboloidal, 224
polar, 224
prolateSpheroidal, 224
spherical, 224
toroidal, 224
copies
DISPLAY, 285
cos
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
cos2sec
TRMANIP, 3264
cosh

EF, 403
EFULS, 451
EFUPXS, 459
cosh2sech
TRMANIP, 3264
coshIfCan
EFULS, 451
EFUPXS, 459
cosIfCan
EFULS, 451
EFUPXS, 459
cosSinInfo
TUBETOOL, 3308
cot
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
cot2tan
TRMANIP, 3264
cot2trig
TRMANIP, 3264
coth
EF, 403
EFULS, 451
EFUPXS, 459
coth2tanh
TRMANIP, 3264
coth2trigh
TRMANIP, 3264
cothIfCan
EFULS, 451
EFUPXS, 459
cotIfCan
EFULS, 451
EFUPXS, 459
countRealRoots
SHP, 3169
countRealRootsMultiple
SHP, 3169
CPIMA, 162
characteristicPolynomial, 162
CPMATCH, 202
patternMatch, 202
CRAPACK, 228
chineseRemainder, 228
modTree, 228
multiEuclideanTree, 228
CRApackage, 228
createGenericMatrix
REP1, 3046
createIrreduciblePoly
FFPOLY, 630
createLowComplexityNormalBasis
FFF, 617
createLowComplexityTable
FFF, 617
createMultiplicationMatrix
FFF, 617
createMultiplicationTable
FFF, 617
createNormalPoly
FFPOLY, 630
createNormalPrimitivePoly
FFPOLY, 630
createPrimitiveNormalPoly
FFPOLY, 630
createPrimitivePoly
FFPOLY, 630
createRandomElement
REP2, 3053
createThreeSpace
TOPSP, 3251
createZechTable
FFF, 617
credPol
GBINTERN, 871
crest
WEIER, 3375
CRFP, 204
complexZeros, 204
divisorCascade, 204
factor, 204
graeffe, 204
norm, 204
pleskenSplit, 204
reciprocalPolynomial, 204
rootRadius, 204
schwerpunkt, 204
setErrorBound, 204
startPolynomial, 204

critB
GBINTERN, 871
critBonD
GBINTERN, 871
critM
GBINTERN, 871
critMonD1
GBINTERN, 871
critMTonD1
GBINTERN, 871
critpOrder
GBINTERN, 871
critT
GBINTERN, 871
cross
TUBETOOL, 3308
crushedSet
PSETPK, 2802
csc
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
csc2sin
TRMANIP, 3264
csch
EF, 403
EFULS, 451
EFUPXS, 459
csch2sinh
TRMANIP, 3264
cschIfCan
EFULS, 451
EFUPXS, 459
cscIfCan
EFULS, 451
EFUPXS, 459
CSTTOOLS, 254
computeCycleEntry, 254
computeCycleLength, 254
cycleElt, 254
CTRIGMNP, 218
complexElementary, 218
complexForm, 218
complexNormalize, 218
imag, 218
real, 218
trigs, 218
cubic
SOLVEFOR, 2819
cubicBezier
BEZIER, 139
cup
CYCLES, 249
curry
MAPPKG2, 1352
curryLeft
MAPPKG3, 1362
curryRight
MAPPKG3, 1362
curveColorPalette
DROPT0, 357
CVMP, 170
coerce, 170
coerceP, 170
cycleElt
CSTTOOLS, 254
CycleIndicators, 249
CYCLES, 249
alternating, 249
cap, 249
complete, 249
cup, 249
cyclic, 249
dihedral, 249
elementary, 249
eval, 249
graphs, 249
powerSum, 249
SFunction, 249
skewSFunction, 249
wreath, 249
cyclic
CYCLES, 249
cyclicGroup
PGE, 2718
CyclicStreamTools, 254
cyclicSubmodule
REP2, 3053
CYCLOTOM, 256
cyclotomic, 256

cyclotomicDecomposition, 256
cyclotomicFactorization, 256
cyclotomic
CYCLOTOM, 256
NTPOLFN, 2544
PNTHEORY, 2793
cyclotomicDecomposition
CYCLOTOM, 256
cyclotomicFactorization
CYCLOTOM, 256
CyclotomicPolynomialPackage, 256
cylindrical
COORDSYS, 224

d01AgentsPackage, 363
D01AGNT, 363
changeName, 363
commaSeparate, 363
df2st, 363
functionIsContinuousAtEndPoints, 363
functionIsOscillatory, 363
gethi, 363
getlo, 363
ldf2lst, 363
problemPoints, 363
rangeIsFinite, 363
sdf2lst, 363
singularitiesOf, 363
d01ajf
NAGD01, 1818
d01akf
NAGD01, 1818
d01alf
NAGD01, 1818
d01amf
NAGD01, 1818
d01anf
NAGD01, 1818
d01apf
NAGD01, 1818
d01aqf
NAGD01, 1818
d01asf
NAGD01, 1818
d01bbf
NAGD01, 1818
d01fcf
NAGD01, 1818
d01gaf
NAGD01, 1818
d01gbf
NAGD01, 1818
d01WeightsPackage, 369
D01WGTS, 369
exprHasAlgebraicWeight, 369
exprHasLogarithmicWeights, 369
exprHasWeightCosWXorSinWX, 369
d02AgentsPackage, 375
D02AGNT, 375
accuracyIF, 375
combineFeatureCompatibility, 375
eval, 375
expenseOfEvaluationIF, 375
intermediateResultsIF, 375
jacobian, 375
sparsityIF, 375
stiffnessAndStabilityFactor, 375
stiffnessAndStabilityOfODEIF, 375
systemSizeIF, 375
d02bbf
NAGD02, 2203
d02bhf
NAGD02, 2203
d02cjf
NAGD02, 2203
d02ejf
NAGD02, 2203
d02gaf
NAGD02, 2203
d02gbf
NAGD02, 2203
d02kef
NAGD02, 2203
d02raf
NAGD02, 2203
d03AgentsPackage, 381
D03AGNT, 381
central?, 381
elliptic?, 381
subscriptedVariables, 381
varList, 381
d03edf

NAGD03, 2249
d03eef
NAGD03, 2249
d03faf
NAGD03, 2249
dAndcExp
INBFF, 1021
DBLRESP, 350
doubleResultant, 350
DDFACT, 288
distdfact, 288
exptMod, 288
factor, 288
factorSquareFree, 288
irreducible?, 288
separateDegrees, 288
separateFactors, 288
trace2PowMod, 288
tracePowMod, 288
ddFact
MDDFACT, 1387
decompose
MONOTOOL, 1400
PDECOMP, 2766
RSDCMPK, 3029
SRDCMPK, 3110
decomposeFunc
SOLVESER, 3299
definingPolynomial
AF, 13
DefiniteIntegrationTools, 259
DEFINTEF, 421
innerint, 421
integrate, 421
DEFINTRF, 2949
integrate, 2949
degOneCoef
PFORP, 2628
DEGRED, 265
expand, 265
reduce, 265
degree
FACUTIL, 593
PFORP, 2628
degreeOfMinimalForm
PFORP, 2628
degreePartition
GALFACT, 767
GALPOLYU, 784
DegreeReductionPackage, 265
degreeSubResultant
PRS, 2864
degreeSubResultantEuclidean
PRS, 2864
denomLODE
ODEPRIM, 2849
denomRicDE
ODEPRRIC, 2853
deriv
STTAYLOR, 3138
derivative
BOP1, 119
desingTree
DTP, 268
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
desingTreeAtPoint
DTP, 268
DesingTreePackage, 268
desingTreeWoFullParam
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
determinant
MATLIN, 1375
df2ef
ESTOOLS, 534
df2fi
ESTOOLS, 534
df2mf
ESTOOLS, 534
df2st
D01AGNT, 363
ESCONT, 527
ESTOOLS, 534
DFINTTLS, 259
checkForZero, 259
computeInt, 259
ignore?, 259
dflist
ESTOOLS, 534

dfRange
ESTOOLS, 534
DFSFUN, 309
airyAi, 309
airyBi, 309
besselI, 309
besselJ, 309
besselK, 309
besselY, 309
Beta, 309
digamma, 309
E1, 309
Ei, 309
Ei1, 309
Ei2, 309
Ei3, 309
Ei4, 309
Ei5, 309
Ei6, 309
En, 309
fresnelC, 309
fresnelS, 309
Gamma, 309
hypergeometric0F1, 309
logGamma, 309
polygamma, 309
diag
MAPPKG2, 1352
diagonalProduct
IBATOOL, 1105
diff
ODEINT, 2587
DiffAction
FFFG, 683
DiffC
FFFG, 683
diffHP
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
digamma
DFSFUN, 309
FSPECF, 699
dihedral
CYCLES, 249
dihedralGroup
PGE, 2718
dilog
LF, 1296
dimensionOfIrreducibleRepresentation
IRSN, 1143
dimensionsOf
NAGSP, 1739
DiophantineSolutionPackage, 277
diophantineSystem
SMITH, 3087
dioSolve
DIOSP, 277
DIOSP, 277
dioSolve, 277
direction
TOOLSIGN, 3222
DirectProductFunctions2, 281
directSum
LODOOPS, 1271
DIRPROD2, 281
map, 281
reduce, 281
scan, 281
DiscreteLogarithmPackage, 283
discriminant
NFINTBAS, 2534
PRS, 2864
discriminantEuclidean
PRS, 2864
DISPLAY, 285
bright, 285
center, 285
copies, 285
newLine, 285
say, 285
sayLength, 285
DisplayPackage, 285
distdfact
DDFACT, 288
distFact
LEADCDET, 1188
DistinctDegreeFactorize, 288
distinguishedCommonRootsOf

RFP, 3074
distinguishedRootsOf
RFP, 3074
divergence
MCALCFN, 1426
divide
PRS, 2864
divideIfCan
UPDIVP, 3339
divisorAtDesingTree
DTP, 268
divisorCascade
CRFP, 204
divisors
INTHEORY, 1082
DLP, 283
shanksDiscLogAlgorithm, 283
dmp2rfi
PLEQN, 2648
dmpToHdmp
POLTOPOL, 2755
dmpToP
POLTOPOL, 2755
dn
ELFUTS, 485
dot
TUBETOOL, 3308
double
REPDB, 3043
doubleDisc
PFOTOOLS, 2749
DoubleFloatSpecialFunctions, 309
doubleRank
ALGPKG, 37
doubleResultant
DBLRESP, 350
DoubleResultantPackage, 350
doublyTransitive?
ALGFACT, 47
DRAW, 3224
draw, 3224
makeObject, 3224
draw
DRAW, 3224
DRAWCFUN, 3235
DRAWCURV, 3231
DRAWPT, 3248
DRAWCFUN, 3235
draw, 3235
makeObject, 3235
recolor, 3235
DrawComplex, 352
drawComplex
DRAWCX, 352
drawComplexVectorField
DRAWCX, 352
DRAWCURV, 3231
draw, 3231
drawCurves
VIEW, 3372
DRAWCX, 352
drawComplex, 352
drawComplexVectorField, 352
setClipValue, 352
setImagSteps, 352
setRealSteps, 352
DRAWHACK, 356
coerce, 356
DrawNumericHack, 356
DrawOptionFunctions0, 357
DrawOptionFunctions1, 361
DRAWPT, 3248
draw, 3248
drawToScale
GRDEF, 854
droot
AF, 13
DROPT0, 357
adaptive, 357
clipBoolean, 357
coord, 357
curveColorPalette, 357
pointColorPalette, 357
ranges, 357
space, 357
style, 357
title, 357
toScale, 357
tubePoints, 357
tubeRadius, 357
units, 357
var1Steps, 357

var2Steps, 357
viewpoint, 357
DROPT1, 361
option, 361
DTP, 268
adjunctionDivisor, 268
blowUp, 268
blowUpWithExcpDiv, 268
desingTree, 268
desingTreeAtPoint, 268
divisorAtDesingTree, 268
fullParamInit, 268
genus, 268
genusNeg, 268
genusTree, 268
genusTreeNeg, 268
inBetweenExcpDiv, 268
initializeParamOfPlaces, 268
initParLocLeaves, 268

e01baf
NAGE01, 1866
e01bef
NAGE01, 1866
e01bff
NAGE01, 1866
e01bgf
NAGE01, 1866
e01bhf
NAGE01, 1866
e01daf
NAGE01, 1866
e01saf
NAGE01, 1866
e01sbf
NAGE01, 1866
e01sef
NAGE01, 1866
e01sff
NAGE01, 1866
e02adf
NAGE02, 1653
e02aef
NAGE02, 1653
e02agf
NAGE02, 1653
e02ahf
NAGE02, 1653
e02ajf
NAGE02, 1653
e02akf
NAGE02, 1653
e02baf
NAGE02, 1653
e02bbf
NAGE02, 1653
e02bcf
NAGE02, 1653
e02bdf
NAGE02, 1653
e02bef
NAGE02, 1653
e02daf
NAGE02, 1653
e02dcf
NAGE02, 1653
e02ddf
NAGE02, 1653
e02def
NAGE02, 1653
e02dff
NAGE02, 1653
e02gaf
NAGE02, 1653
e02zaf
NAGE02, 1653
e04AgentsPackage, 577
E04AGNT, 577
changeNameToObjf, 577
expenseOfEvaluation, 577
finiteBound, 577
linear?, 577
linearMatrix, 577
linearPart, 577
nonLinearPart, 577
optAttributes, 577
quadratic?, 577
simpleBounds?, 577
sortConstraints, 577
splitLinear, 577
sumOfSquares, 577
variables, 577

varList, 577
e04dgf
NAGE04, 2104
e04fdf
NAGE04, 2104
e04gcf
NAGE04, 2104
e04jaf
NAGE04, 2104
e04mbf
NAGE04, 2104
e04naf
NAGE04, 2104
e04ucf
NAGE04, 2104
e04ycf
NAGE04, 2104
E1
DFSFUN, 309
edf2df
ESTOOLS, 534
edf2ef
ESTOOLS, 534
edf2efi
ESTOOLS, 534
edf2fi
ESTOOLS, 534
EF, 403
acos, 403
acosh, 403
acot, 403
acoth, 403
acsc, 403
acsch, 403
asec, 403
asech, 403
asin, 403
asinh, 403
atan, 403
atanh, 403
belong?, 403
cos, 403
cosh, 403
cot, 403
coth, 403
csc, 403
csch, 403
exp, 403
iiacos, 403
iiacosh, 403
iiacot, 403
iiacoth, 403
iiacsc, 403
iiacsch, 403
iiasec, 403
iiasech, 403
iiasin, 403
iiasinh, 403
iiatan, 403
iiatanh, 403
iicos, 403
iicosh, 403
iicot, 403
iicoth, 403
iicsc, 403
iicsch, 403
iiexp, 403
iilog, 403
iisec, 403
iisech, 403
iisin, 403
iisinh, 403
iisqrt2, 403
iisqrt3, 403
iitan, 403
iitanh, 403
localReal?, 403
log, 403
operator, 403
pi, 403
sec, 403
sech, 403
sin, 403
sinh, 403
specialTrigs, 403
tan, 403
tanh, 403
ef2edf
ESTOOLS, 534
EFSTRUC, 442
normalize, 442
realElementary, 442

rischNormalize, 442
rootNormalize, 442
tanQ, 442
validExponential, 442
EFULS, 451
?**?, 451
acos, 451
acosh, 451
acoshIfCan, 451
acosIfCan, 451
acot, 451
acoth, 451
acothIfCan, 451
acotIfCan, 451
acsc, 451
acsch, 451
acschIfCan, 451
acscIfCan, 451
asec, 451
asech, 451
asechIfCan, 451
asecIfCan, 451
asin, 451
asinh, 451
asinhIfCan, 451
asinIfCan, 451
atan, 451
atanh, 451
atanhIfCan, 451
atanIfCan, 451
cos, 451
cosh, 451
coshIfCan, 451
cosIfCan, 451
cot, 451
coth, 451
cothIfCan, 451
cotIfCan, 451
csc, 451
csch, 451
cschIfCan, 451
cscIfCan, 451
exp, 451
expIfCan, 451
log, 451
logIfCan, 451
nthRootIfCan, 451
sec, 451
sech, 451
sechIfCan, 451
secIfCan, 451
sin, 451
sinh, 451
sinhIfCan, 451
sinIfCan, 451
tan, 451
tanh, 451
tanhIfCan, 451
tanIfCan, 451
EFUPXS, 459
?**?, 459
acos, 459
acosh, 459
acoshIfCan, 459
acosIfCan, 459
acot, 459
acoth, 459
acothIfCan, 459
acotIfCan, 459
acsc, 459
acsch, 459
acschIfCan, 459
acscIfCan, 459
asec, 459
asech, 459
asechIfCan, 459
asecIfCan, 459
asin, 459
asinh, 459
asinhIfCan, 459
asinIfCan, 459
atan, 459
atanh, 459
atanhIfCan, 459
atanIfCan, 459
cos, 459
cosh, 459
coshIfCan, 459
cosIfCan, 459
cot, 459
coth, 459
cothIfCan, 459

cotIfCan, 459
csc, 459
csch, 459
cschIfCan, 459
cscIfCan, 459
exp, 459
expIfCan, 459
log, 459
logIfCan, 459
nthRootIfCan, 459
sec, 459
sech, 459
sechIfCan, 459
secIfCan, 459
sin, 459
sinh, 459
sinhIfCan, 459
sinIfCan, 459
tan, 459
tanh, 459
tanhIfCan, 459
tanIfCan, 459
Ei
DFSFUN, 309
LF, 1296
Ei1
DFSFUN, 309
Ei2
DFSFUN, 309
Ei3
DFSFUN, 309
Ei4
DFSFUN, 309
Ei5
DFSFUN, 309
Ei6
DFSFUN, 309
eigenMatrix
REP, 2915
EigenPackage, 385
eigenvalues
EP, 385
eigenvector
EP, 385
eigenvectors
EP, 385
eisensteinIrreducible?
GALFACT, 767
elementary
CYCLES, 249
ElementaryFunction, 403
ElementaryFunctionDefiniteIntegration, 421
ElementaryFunctionLODESolver, 426
ElementaryFunctionODESolver, 432
ElementaryFunctionSign, 438
ElementaryFunctionStructurePackage, 442
ElementaryFunctionsUnivariateLaurentSeries, 451
ElementaryFunctionsUnivariatePuiseuxSeries, 459
ElementaryIntegration, 465
ElementaryRischDE, 474
ElementaryRischDESystem, 482
ELFUTS, 485
cn, 485
dn, 485
sn, 485
sncndn, 485
elliptic
COORDSYS, 224
elliptic?
D03AGNT, 381
ellipticCylindrical
COORDSYS, 224
EllipticFunctionsUnivariateTaylorSeries, 485
En
DFSFUN, 309
enterInCache
SCACHE, 3092
EP, 385
characteristicPolynomial, 385
eigenvalues, 385
eigenvector, 385
eigenvectors, 385
generalizedEigenvector, 385
generalizedEigenvectors, 385
EQ2, 487
map, 487
EquationFunctions2, 487
erf
LF, 1296
ERROR, 488

error, 488
error
ERROR, 488
ErrorFunctions, 488
ES1, 548
map, 549
ES2, 550
map, 550
ESCONT, 526
df2st, 527
functionIsFracPolynomial?, 527
gethi, 527
getlo, 527
ldf2lst, 527
polynomialZeros, 527
problemPoints, 527
sdf2lst, 527
singularitiesOf, 527
zerosOf, 527
ESCONT1, 532
in?, 532
ESTOOLS, 534
att2Result, 534
concat, 534
convert, 534
df2ef, 534
df2fi, 534
df2mf, 534
df2st, 534
dflist, 534
dfRange, 534
edf2df, 534
edf2ef, 534
edf2efi, 534
edf2fi, 534
ef2edf, 534
expenseOfEvaluation, 534
f2df, 534
f2st, 534
fi2df, 534
gethi, 534
getlo, 534
iflist2Result, 534
in?, 534
isQuotient, 534
ldf2lst, 534
ldf2vmf, 534
mat, 534
measure2Result, 534
numberOfOperations, 534
ocf2ocdf, 534
outputMeasure, 534
pdf2df, 534
pdf2ef, 534
sdf2lst, 534
socf2socdf, 534
vedf2vef, 534
ESTOOLS1, 542
neglist, 542
ESTOOLS2, 543
map, 543
euclideanGroebner
GBEUCLID, 513
EuclideanGroebnerBasisPackage, 513
euclideanNormalForm
GBEUCLID, 513
euler
INTHEORY, 1082
PNTHEORY, 2793
eulerE
NTPOLFN, 2544
eulerPhi
INTHEORY, 1082
eval
CHVAR, 159
CYCLES, 249
D02AGNT, 375
EVALCYC, 525
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
PLPKCRV, 2752
RF, 2947
STTAYLOR, 3138
evalADE
RECOP, 3012
EVALCYC, 525
eval, 525
evalIfCan
GPAFF, 801
PAFF, 2611
PAFFFF, 2619

evalRec
RECOP, 3012
evaluate
BOP1, 119
EvaluateCycleIndicators, 525
evenInfiniteProduct
INFPROD0, 983
INPRODFF, 985
INPRODPF, 988
STINPROD, 3136
evenlambert
STTAYLOR, 3138
exp
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
XEXPPKG, 3385
EXP3D, 574
writeObj, 574
expand
DEGRED, 265
IR2F, 1115
IRRF2F, 1113
TRMANIP, 3264
expandLog
TRMANIP, 3264
expandPower
TRMANIP, 3264
expandTrigProducts
TRMANIP, 3264
expenseOfEvaluation
E04AGNT, 577
ESTOOLS, 534
expenseOfEvaluationIF
D02AGNT, 375
ExpertSystemContinuityPackage, 526
ExpertSystemContinuityPackage1, 532
ExpertSystemToolsPackage, 534
ExpertSystemToolsPackage1, 542
ExpertSystemToolsPackage2, 543
expextendedint
INTTR, 3254
expIfCan
EFULS, 451
EFUPXS, 459
expint
ODEINT, 2587
expintegrate
INTTR, 3254
expintfldpoly
INTTR, 3254
explimitedint
INTTR, 3254
explogs2trigs
ITRIGMNP, 1041
exponential
RFDIST, 2939
exponential1
RFDIST, 2939
Export3D, 574
expPot
INBFF, 1021
EXPR2, 544
map, 545
EXPR2UPS, 562
laurent, 562
puiseux, 562
series, 562
taylor, 562
ExpressionFunctions2, 544
ExpressionSolve, 546
ExpressionSpaceFunctions1, 548
ExpressionSpaceFunctions2, 550
ExpressionSpaceODESolver, 551
ExpressionToOpenMath, 556
ExpressionToUnivariatePowerSeries, 562
ExpressionTubePlot, 569
exprHasAlgebraicWeight
D01WGTS, 369
exprHasLogarithmicWeights
D01WGTS, 369
exprHasWeightCosWXorSinWX
D01WGTS, 369
EXPRODE, 551
seriesSolve, 551
EXPRSOL, 546
replaceDiffs, 546
seriesSolve, 546
exprToGenUPS
FS2UPS, 742

exprToUPS
FS2UPS, 742
exprToXXP
FS2EXPXP, 730
EXPRTUBE, 569
constantToUnaryFunction, 569
tubePlot, 569
expt
REPSQ, 3045
exptMod
DDFACT, 288
MDDFACT, 1387
exquo
PRS, 2864
STTAYLOR, 3138
extendedint
INTRAT, 2970
extendedIntegrate
INTRF, 2955
extractIfCan
TBCMPPK, 3211

F
RFDIST, 2939
f01brf
NAGF01, 1946
f01bsf
NAGF01, 1946
f01maf
NAGF01, 1946
f01mcf
NAGF01, 1946
f01qcf
NAGF01, 1946
f01qdf
NAGF01, 1946
f01qef
NAGF01, 1946
f01rcf
NAGF01, 1946
f01rdf
NAGF01, 1946
f01ref
NAGF01, 1946
f02aaf
NAGF02, 1511
f02abf
NAGF02, 1511
f02adf
NAGF02, 1511
f02aef
NAGF02, 1511
f02aff
NAGF02, 1511
f02agf
NAGF02, 1511
f02ajf
NAGF02, 1511
f02akf
NAGF02, 1511
f02awf
NAGF02, 1511
f02axf
NAGF02, 1511
f02bbf
NAGF02, 1511
f02bjf
NAGF02, 1511
f02fjf
NAGF02, 1511
f02wef
NAGF02, 1511
f02xef
NAGF02, 1511
f04adf
NAGF04, 1731
f04arf
NAGF04, 1731
f04asf
NAGF04, 1731
f04atf
NAGF04, 1731
f04axf
NAGF04, 1731
f04faf
NAGF04, 1731
f04jgf
NAGF04, 1731
f04maf
NAGF04, 1731
f04mbf
NAGF04, 1731

f04mcf
NAGF04, 1731
f04qaf
NAGF04, 1731
f07adf
NAGF07, 1887
f07aef
NAGF07, 1887
f07fdf
NAGF07, 1887
f07fef
NAGF07, 1887
f2df
ESTOOLS, 534
F2EXPRR
GUESSF1, 953
F2FG
ITRIGMNP, 1041
f2st
ESTOOLS, 534
FACTEXT, 596
factor, 596
factorSqFree, 596
FACTFUNC, 585
log, 585
nthRoot, 585
factor
ALGFACT, 47
ALGMFACT, 35
COMPFACT, 196
CRFP, 204
DDFACT, 288
FACTEXT, 596
FACTRN, 601
FFFACTSE, 612
GALFACT, 767
GAUSSFAC, 790
GENMFACT, 798
GENUFACT, 838
IALGFACT, 993
INNMFACT, 1011
INTFACT, 1057
LODOF, 1267
MDDFACT, 1387
MFINFACT, 1414
MPCPF, 1403
MPRFF, 1405
MRATFAC, 1412
MULTFACT, 1430
RATFACT, 2945
RFFACT, 2952
RFFACTOR, 2953
SAEFACT, 3085
SAERFFC, 3079
SUPFRACF, 3180
UNIFACT, 3323
factor1
LODOF, 1267
factorByRecursion
PFBR, 2768
PFBRU, 2775
factorCantorZassenhaus
FFFACTSE, 612
FactoredFunctions, 585
FactoredFunctions2, 589
FactoredFunctionUtilities, 591
factorGroebnerBasis
GBF, 863
factorial
COMBF, 175
COMBINAT, 1053
factorials
COMBF, 175
FactoringUtilities, 593
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumbe
596
FactorisationOverPseudoAlgebraicClosureOfRationalNumber,
601
factorList
IBACHIN, 165
factorOfDegree
GALFACT, 767
GALPOLYU, 784
factorset
PLEQN, 2648
factorSFBRlcUnit
PFBR, 2768
PFBRU, 2775
factorsOfDegree
GALPOLYU, 784
factorSqFree
FACTEXT, 596

FACTRN, 601
factorSquareFree
DDFACT, 288
FFFACTSE, 612
GALFACT, 767
RATFACT, 2945
UNIFACT, 3323
factorSquareFreeByRecursion
PFBR, 2768
PFBRU, 2775
factorUsingMusser
FFFACTSE, 612
factorUsingYun
FFFACTSE, 612
FACTRN, 601
factor, 601
factorSqFree, 601
FACUTIL, 593
completeEval, 593
degree, 593
lowerPolynomial, 593
normalDeriv, 593
raisePolynomial, 593
ran, 593
variables, 593
FAMR2, 608
map, 608
FCPAK1, 673
identitySquareMatrix, 673
zeroMatrix, 673
zeroSquareMatrix, 673
zeroVector, 673
FDIV2, 609
map, 609
ffactor
FSUPFACT, 758
FFCAT2, 708
map, 708
FFF, 617
createLowComplexityNormalBasis, 617
createLowComplexityTable, 617
createMultiplicationMatrix, 617
createMultiplicationTable, 617
createZechTable, 617
sizeMultiplication, 617
FFFACTSE, 611
factor, 612
factorCantorZassenhaus, 612
factorSquareFree, 612
factorUsingMusser, 612
factorUsingYun, 612
irreducible?, 612
FFFG, 683
DiffAction, 683
DiffC, 683
fffg, 683
generalCoefficient, 683
generalInterpolation, 683
interpolate, 683
qShiftAction, 683
qShiftC, 683
ShiftAction, 683
ShiftC, 683
fffg
FFFG, 683
FFFGF, 693
generalInterpolation, 693
FFHOM, 622
coerce, 622
FFINTBAS, 709
integralBasis, 709
localIntegralBasis, 709
FFPOLY, 630
createIrreduciblePoly, 630
createNormalPoly, 630
createNormalPrimitivePoly, 630
createPrimitiveNormalPoly, 630
createPrimitivePoly, 630
leastAffineMultiple, 630
nextIrreduciblePoly, 630
nextNormalPoly, 630
nextNormalPrimitivePoly, 630
nextPrimitiveNormalPoly, 630
nextPrimitivePoly, 630
normal?, 630
numberOfIrreduciblePoly, 630
numberOfNormalPoly, 630
numberOfPrimitivePoly, 630
primitive?, 630
random, 630
reducedQPowers, 630
FFPOLY2, 651

rootOfIrreduciblePoly, 651
FFSLPE, 654
solveLinearPolynomialEquation, 654
FFSQFR, 656
PolK, 657
Yun, 657
FG2F
ITRIGMNP, 1041
FGLMICPK, 604
fglmIfCan, 604
groebner, 604
zeroDimensional?, 604
fglmIfCan
FGLMICPK, 604
LEXTRIPK, 1260
FGLMIfCanPackage, 604
fi2df
ESTOOLS, 534
fibonacci
INTHEORY, 1082
fillPascalTriangle
GALUTIL, 787
FindOrderFinite, 606
findOrderOfDivisor
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
FiniteAbelianMonoidRingFunctions2, 608
finiteBound
E04AGNT, 577
FiniteDivisorFunctions2, 609
FiniteFieldFactorizationWithSizeParseBySideEffects, 611
FiniteFieldFunctions, 617
FiniteFieldHomomorphisms, 622
FiniteFieldPolynomialPackage, 630
FiniteFieldPolynomialPackage2, 651
FiniteFieldSolveLinearPolynomialEquation, 654
FiniteFieldSquareFreeDecomposition, 656
FiniteLinearAggregateFunctions2, 659
FiniteLinearAggregateSort, 662
finiteSeries2LinSys
LISYSER, 1277
finiteSeries2LinSysWOVectorise
LISYSER, 1277
finiteSeries2Vector
LISYSER, 1277
FiniteSetAggregateFunctions2, 665
firstExponent
PFORP, 2628
firstSubsetGray
GRAY, 857
firstUncouplingMatrix
PREASSOC, 2841
fixedDivisor
PNTHEORY, 2793
fixedPoint
MAPPKG1, 1341
fixedPointExquo
UTSODE, 3354
fixPredicate
PMDOWN, 2701
FLAGG2, 659
map, 659
reduce, 659
scan, 659
FLASORT, 662
heapSort, 662
quickSort, 662
shellSort, 662
FLOATCP, 666
complexRoots, 667
complexSolve, 667
FloatingComplexPackage, 666
FloatingRealPackage, 670
FLOATRP, 670
realRoots, 670
solve, 670
FOP, 676
clearFortranOutputStack, 676
popFortranOutputStack, 676
pushFortranOutputStack, 676
showFortranOutputStack, 676
topFortranOutputStack, 676
FORDER, 606
order, 606
FormatArabic
NUMFMT, 2539
FormatRoman
NUMFMT, 2539
FORMULA1, 3080
coerce, 3080

FORT, 679
  linkToFortran, 679
  outputAsFortran, 679
  setLegalFortranSourceExtensions, 679
FortranCodePackage1, 673
fortranCompilerName
  NAGSP, 1739
fortranLinkerArgs
  NAGSP, 1739
FortranOutputStackPackage, 676
FortranPackage, 679
foundZeroes
  RFP, 3074
fprindINFO
  GBINTERN, 871
FR2, 589
  map, 589
FRAC2, 696
  map, 696
FractionalIdealFunctions2, 681
FractionFreeFastGaussian, 683
FractionFreeFastGaussianFractions, 693
FractionFunctions2, 696
FramedNonAssociativeAlgebraFunctions2, 697
fresnelC
  DFSFUN, 309
fresnelS
  DFSFUN, 309
FRIDEAL2, 681
  map, 681
FRNAAF2, 697
  map, 698
Frobenius
  NORMRETR, 2528
froot
  POLYROOT, 2798
FRUTIL, 591
  mergeFactors, 591
  refine, 591
FS2, 718
  map, 718
FS2EXPXP, 730
  exprToXXP, 730
  localAbs, 730
FS2UPS, 741
  exprToGenUPS, 742
  exprToUPS, 742
  localAbs, 742
FSAGG2, 665
  map, 665
  reduce, 665
  scan, 665
FSCINT, 716
  complexIntegrate, 716
  internalIntegrate, 716
  internalIntegrate0, 716
FSINT, 720
  integrate, 720
FSPECF, 699
  abs, 699
  airyAi, 699
  airyBi, 699
  belong?, 699
  besselI, 699
  besselJ, 699
  besselK, 699
  besselY, 699
  Beta, 699
  digamma, 699
  Gamma, 699
  iiabs, 699
  iiAiryAi, 699
  iiAiryBi, 699
  iiBesselI, 699
  iiBesselJ, 699
  iiBesselK, 699
  iiBesselY, 699
  iiBeta, 699
  iidigamma, 699
  iiGamma, 699
  iipolygamma, 699
  operator, 699
  polygamma, 699
FSPRMELT, 723
  primitiveElement, 723
FSRED, 726
  bringDown, 726
  newReduc, 726
FSUPFACT, 757
  anfactor, 758
  ffactor, 758
  qfactor, 758

UP2ifCan, 758
fullDesTree
PAFF, 2611
PAFFFF, 2619
fullInfClsPt
PAFF, 2611
PAFFFF, 2619
fullParamInit
DTP, 268
function
MKFUNC, 1324
FunctionalSpecialFunction, 699
FunctionFieldCategoryFunctions2, 708
FunctionFieldIntegralBasis, 709
functionIsContinuousAtEndPoints
D01AGNT, 363
functionIsFracPolynomial?
ESCONT, 527
functionIsOscillatory
D01AGNT, 363
FunctionSpaceAssertions, 712
FunctionSpaceAttachPredicates, 714
FunctionSpaceComplexIntegration, 716
FunctionSpaceFunctions2, 718
FunctionSpaceIntegration, 720
FunctionSpacePrimitiveElement, 723
FunctionSpaceReduce, 726
FunctionSpaceSum, 728
FunctionSpaceToExponentialExpansion, 730
FunctionSpaceToUnivariatePowerSeries, 741
FunctionSpaceUnivariatePolynomialFactor, 757

GALFACT, 767
btwFact, 767
degreePartition, 767
eisensteinIrreducible?, 767
factor, 767
factorOfDegree, 767
factorSquareFree, 767
henselFact, 767
makeFR, 767
modularFactor, 767
musserTrials, 767
numberOfFactors, 767
stopMusserTrials, 767
tryFunctionalDecomposition, 767
tryFunctionalDecomposition?, 767
useEisensteinCriterion, 767
useEisensteinCriterion?, 767
useSingleFactorBound, 767
useSingleFactorBound?, 767
GALFACTU, 763
beauzamyBound, 763
bombieriNorm, 763
height, 763
infinityNorm, 763
length, 763
norm, 763
quadraticNorm, 763
rootBound, 763
singleFactorBound, 763
GaloisGroupFactorizationUtilities, 763
GaloisGroupFactorizer, 767
GaloisGroupPolynomialUtilities, 784
GaloisGroupUtilities, 787
GALPOLYU, 784
degreePartition, 784
factorOfDegree, 784
factorsOfDegree, 784
monic?, 784
reverse, 784
scaleRoots, 784
shiftRoots, 784
unvectorise, 784
GALUTIL, 787
fillPascalTriangle, 787
pascalTriangle, 787
rangePascalTriangle, 787
safeCeiling, 787
safeFloor, 787
safetyMargin, 787
sizePascalTriangle, 787
Gamma
DFSFUN, 309
FSPECF, 699
GAUSSFAC, 790
factor, 790
prime?, 790
sumSquares, 790
GaussianFactorizationPackage, 790
GB, 907
groebner, 907

normalForm, 907
gbasis
GBINTERN, 871
GBEUCLID, 513
euclideanGroebner, 513
euclideanNormalForm, 513
GBF, 863
factorGroebnerBasis, 863
groebnerFactorize, 863
GBINTERN, 871
credPol, 871
critB, 871
critBonD, 871
critM, 871
critMonD1, 871
critMTonD1, 871
critpOrder, 871
critT, 871
fprindINFO, 871
gbasis, 871
hMonic, 871
lepol, 871
makeCrit, 871
minGbasis, 871
prinb, 871
prindINFO, 871
prinpolINFO, 871
prinshINFO, 871
redPo, 871
redPol, 871
sPol, 871
updatD, 871
updatF, 871
virtualDegree, 871
gcd
HEUGCD, 966
MDDFACT, 1387
PGCD, 2782
PRS, 2864
gcdcofact
HEUGCD, 966
gcdcofactprim
HEUGCD, 966
gcdPolynomial
GENPGCD, 815
gcdprim
HEUGCD, 966
gcdPrimitive
PGCD, 2782
gderiv
STTAYLOR, 3138
GDRAW, 847
GospersMethod, 847
GENEEZ, 833
compBound, 834
reduction, 834
solveid, 834
tablePow, 834
testModulus, 834
generalCoefficient
FFFG, 683
GeneralHenselPackage, 794
generalInfiniteProduct
INFPROD0, 983
INPRODFF, 985
INPRODPF, 988
STINPROD, 3136
generalInterpolation
FFFG, 683
FFFGF, 693
generalizedEigenvector
EP, 385
generalizedEigenvectors
EP, 385
GeneralizedMultivariateFactorize, 798
generalLambert
STTAYLOR, 3138
GeneralPackageForAlgebraicFunctionField, 801
GeneralPolynomialGcdPackage, 815
generalSqFr
TWOFACT, 3317
generalTwoFactor
TWOFACT, 3317
generate
HB, 963
generateIrredPoly
IRREDFFX, 1141
GenerateUnivariatePowerSeries, 829
genericPosition
GROEBSOL, 911
GenExEuclid, 833
GENMFACT, 798

factor, 798
GENPGCD, 815
gcdPolynomial, 815
randomR, 815
GENUFACT, 838
factor, 838
GenUFactorize, 838
GENUPS, 829
laurent, 829
puiseux, 829
series, 829
taylor, 829
genus
DTP, 268
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
genusNeg
DTP, 268
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
genusTree
DTP, 268
genusTreeNeg
DTP, 268
GenusZeroIntegration, 840
geometric
RIDIST, 2941
getDatabase
ASSOCEQ, 100
OPQUERY, 2599
getEq
RECOP, 3012
getGoodPrime
PFOTOOLS, 2749
gethi
D01AGNT, 363
ESCONT, 527
ESTOOLS, 534
getlo
D01AGNT, 363
ESCONT, 527
ESTOOLS, 534
getOp
RECOP, 3012
getOrder
UDPO, 3359
getShiftRec
RECOP, 3012
getVariableOrder
UDVO, 3362
GF2FG
ITRIGMNP, 1041
GHENSEL, 794
completeHensel, 794
HenselLift, 794
reduction, 794
GnuDraw, 847
goodnessOfFit
OPTPACK, 61
goodPoint
CHVAR, 159
goppaCode
PAFF, 2611
PAFFFF, 2619
GOSPER, 849
GospersMethod, 849
GospersMethod
GDRAW, 847
GOSPER, 849
GosperSummationMethod, 849
GPAFF, 801
adjunctionDivisor, 801
classNumber, 801
desingTree, 801
desingTreeWoFullParam, 801
eval, 801
evalIfCan, 801
findOrderOfDivisor, 801
genus, 801
genusNeg, 801
homogenize, 801
interpolateForms, 801
interpolateFormsForFact, 801
intersectionDivisor, 801
lBasis, 801
LPolynomial, 801
numberOfPlacesOfDegree, 801
numberPlacesDegExtDeg, 801
numberRatPlacesExtDeg, 801
parametrize, 801

placesAbove, 801
placesOfDegree, 801
pointDominateBy, 801
printInfo, 801
rationalPlaces, 801
rationalPoints, 801
reset, 801
setCurve, 801
setSingularPoints, 801
singularPoints, 801
theCurve, 801
ZetaFunction, 801
gradient
MCALCFN, 1426
graeffe
CRFP, 204
gramschmidt
REP, 2915
graphCurves
VIEW, 3372
GraphicsDefaults, 854
graphs
CYCLES, 249
GRAY, 857
firstSubsetGray, 857
nextSubsetGray, 857
GrayCode, 857
GRDEF, 854
adaptive, 854
clipPointsDefault, 854
drawToScale, 854
maxPoints, 854
minPoints, 854
screenResolution, 854
groebgen
LGROBP, 1285
groebner
FGLMICPK, 604
GB, 907
INTERGB, 1047
LEXTRIPK, 1260
GroebnerFactorizationPackage, 863
groebnerFactorize
GBF, 863
GroebnerInternalPackage, 871
GroebnerPackage, 907
GroebnerSolve, 911
GROEBSOL, 911
genericPosition, 911
groebSolve, 911
testDim, 911
groebSolve
GROEBSOL, 911
GUESS, 916
diffHP, 916
guess, 916
guessADE, 916
guessAlg, 916
guessBinRat, 916
guessExpRat, 916
guessHolo, 916
guessHP, 916
guessPade, 916
guessPRec, 916
guessRat, 916
guessRec, 916
shiftHP, 916
Guess, 916
guess
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessADE
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessAlg
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
GuessAlgebraicNumber, 951
GUESSAN, 951
diffHP, 951

guess, 951
guessADE, 951
guessAlg, 951
guessBinRat, 951
guessExpRat, 951
guessHolo, 951
guessHP, 951
guessPade, 951
guessPRec, 951
guessRat, 951
guessRec, 951
shiftHP, 951
guessBinRat
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessExpRat
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
GUESSF, 952
diffHP, 952
guess, 952
guessADE, 952
guessAlg, 952
guessBinRat, 952
guessExpRat, 952
guessHolo, 952
guessHP, 952
guessPade, 952
guessPRec, 952
guessRat, 952
guessRec, 952
shiftHP, 952
GUESSF1, 953
F2EXPRR, 953
GuessFinite, 952
GuessFiniteFunctions, 953
guessHolo
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessHP
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
GUESSINT, 954
diffHP, 954
guess, 954
guessADE, 954
guessAlg, 954
guessBinRat, 954
guessExpRat, 954
guessHolo, 954
guessHP, 954
guessPade, 954
guessPRec, 954
guessRat, 954
guessRec, 954
shiftHP, 954
GuessInteger, 954
GUESSP, 955
diffHP, 955
guess, 955
guessADE, 955
guessAlg, 955
guessBinRat, 955
guessExpRat, 955
guessHolo, 955
guessHP, 955
guessPade, 955
guessPRec, 955
guessRat, 955
guessRec, 955
shiftHP, 955
guessPade
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954

GUESSP, 955
GUESSUP, 956
GuessPolynomial, 955
guessPRec
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessRat
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
guessRec
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
GuessUnivariatePolynomial, 956
GUESSUP, 956
diffHP, 956
guess, 956
guessADE, 956
guessAlg, 956
guessBinRat, 956
guessExpRat, 956
guessHolo, 956
guessHP, 956
guessPade, 956
guessPRec, 956
guessRat, 956
guessRec, 956
shiftHP, 956

hadamard
UFPS1, 3330
HallBasis, 963
harmonic
INTHEORY, 1082
hasoln
PLEQN, 2648
hasSolution?
LSMP, 1279
LSMP1, 1281
Hausdorff
XEXPPKG, 3385
HB, 963
generate, 963
inHallBasis?, 963
lfunc, 963
hdmpToDmp
POLTOPOL, 2755
hdmpToP
POLTOPOL, 2755
heapSort
FLASORT, 662
height
GALFACTU, 763
henselFact
GALFACT, 767
UNIFACT, 3323
HenselLift
GHENSEL, 794
hermite
PNTHEORY, 2793
SMITH, 3087
hermiteH
ORTHPOL, 2603
HermiteIntegrate
INTHERAL, 19
INTHERTR, 3252
hessian
MCALCFN, 1426
HEUGCD, 966
content, 966
gcd, 966
gcdcofact, 966
gcdcofactprim, 966
gcdprim, 966
lintgcd, 966
HeuGcd, 966
hMonic
GBINTERN, 871
homogenize
GPAFF, 801
PAFF, 2611

PAFFFF, 2619
PFORP, 2628
htrigs
TRMANIP, 3264
hue
PTPACK, 2738
hypergeometric0F1
DFSFUN, 309

IALGFACT, 993
factor, 993
IBACHIN, 165
chineseRemainder, 165
factorList, 165
listConjugateBases, 165
IBATOOL, 1105
diagonalProduct, 1105
divideIfCan, 1105
idealiser, 1105
idealiserMatrix, 1105
leastPower, 1105
matrixGcd, 1105
moduleSum, 1105
IBPTOOLS, 1108
mapBivariate, 1109
mapMatrixIfCan, 1109
mapUnivariate, 1109
mapUnivariateIfCan, 1109
ICDEN, 996
clearDenominator, 996
commonDenominator, 996
splitDenominator, 996
id
MAPPKG1, 1341
IdealDecompositionPackage, 973
idealiser
IBATOOL, 1105
idealiserMatrix
IBATOOL, 1105
IDECOMP, 973
contract, 973
primaryDecomp, 973
prime?, 973
radical, 973
zeroDimPrimary?, 973
zeroDimPrime?, 973
identitySquareMatrix
FCPAK1, 673
iflist2Result
ESTOOLS, 534
ignore?
DFINTTLS, 259
iiabs
FSPECF, 699
iiacos
EF, 403
iiacosh
EF, 403
iiacot
EF, 403
iiacoth
EF, 403
iiacsc
EF, 403
iiacsch
EF, 403
iiAiryAi
FSPECF, 699
iiAiryBi
FSPECF, 699
iiasec
EF, 403
iiasech
EF, 403
iiasin
EF, 403
iiasinh
EF, 403
iiatan
EF, 403
iiatanh
EF, 403
iiBesselI
FSPECF, 699
iiBesselJ
FSPECF, 699
iiBesselK
FSPECF, 699
iiBesselY
FSPECF, 699
iiBeta
FSPECF, 699

iibinom
COMBF, 175
iicos
EF, 403
iicosh
EF, 403
iicot
EF, 403
iicoth
EF, 403
iicsc
EF, 403
iicsch
EF, 403
iidigamma
FSPECF, 699
iidprod
COMBF, 175
iidsum
COMBF, 175
iiexp
EF, 403
iifact
COMBF, 175
iiGamma
FSPECF, 699
iilog
EF, 403
iiperm
COMBF, 175
iipolygamma
FSPECF, 699
iipow
COMBF, 175
iisec
EF, 403
iisech
EF, 403
iisin
EF, 403
iisinh
EF, 403
iisqrt2
EF, 403
iisqrt3
EF, 403
iitan
EF, 403
iitanh
EF, 403
imag
CTRIGMNP, 218
TRIGMNIP, 3304
images
AF, 13
AFALGGRO, 4
AFALGRES, 9
ALGFACT, 47
ALGMANIP, 30
ALGMFACT, 35
ALGPKG, 37
ANY1, 84
API, 90
APPLYORE, 95
APPRULE, 92
ARRAY12, 2591
ASSOCEQ, 96
AXSERV, 100
BALFACT, 117
BEZIER, 139
BEZOUT, 141
BLUPPACK, 146
BOP1, 119
BOUNDZRO, 151
BRILL, 154
CARTEN2, 157
CDEN, 187
CHARPOL, 164
CHVAR, 159
CINTSLPE, 199
CLIP, 3311
CMPLXRT, 216
COMBF, 175
COMBINAT, 1053
COMMONOP, 189
COMMUPC, 194
COMPFACT, 195
COMPLEX2, 198
COMPLPAT, 201
COORDSYS, 224
CPIMA, 162
CPMATCH, 202

CRAPACK, 228
CRFP, 204
CSTTOOLS, 254
CTRIGMNP, 218
CVMP, 170
CYCLES, 249
CYCLOTOM, 256
D01AGNT, 363
D01WGTS, 369
D02AGNT, 375
D03AGNT, 381
DBLRESP, 350
DDFACT, 288
DEFINTEF, 421
DEFINTRF, 2949
DEGRED, 265
DFINTTLS, 259
DFSFUN, 309
DIOSP, 277
DIRPROD2, 281
DISPLAY, 285
DLP, 283
DRAW, 3224
DRAWCFUN, 3235
DRAWCURV, 3231
DRAWCX, 352
DRAWHACK, 356
DRAWPT, 3248
DROPT0, 357
DROPT1, 361
DTP, 268
E04AGNT, 577
EF, 403
EFSTRUC, 442
EFULS, 451
EFUPXS, 459
ELFUTS, 485
EP, 385
EQ2, 487
ERROR, 488
ES1, 549
ES2, 550
ESCONT, 526
ESCONT1, 532
ESTOOLS, 534
ESTOOLS1, 542
ESTOOLS2, 543
EVALCYC, 525
EXP3D, 574
EXPR2, 544
EXPR2UPS, 562
EXPRODE, 551
EXPRSOL, 546
EXPRTUBE, 569
FACTEXT, 596
FACTFUNC, 585
FACTRN, 601
FACUTIL, 593
FAMR2, 608
FCPAK1, 673
FDIV2, 609
FFCAT2, 708
FFF, 617
FFFACTSE, 611
FFFG, 683
FFFGF, 693
FFHOM, 622
FFINTBAS, 709
FFPOLY, 630
FFPOLY2, 651
FFSLPE, 654
FFSQFR, 656
FGLMICPK, 604
FLAGG2, 659
FLASORT, 662
FLOATCP, 666
FLOATRP, 670
FOP, 676
FORDER, 606
FORMULA1, 3080
FORT, 679
FR2, 589
FRAC2, 696
FRIDEAL2, 681
FRNAAF2, 697
FRUTIL, 591
FS2, 718
FS2EXPXP, 730
FS2UPS, 741
FSAGG2, 665
FSCINT, 716
FSINT, 720

FSPECF, 699
FSPRMELT, 723
FSRED, 726
FSUPFACT, 757
GALFACT, 767
GALFACTU, 763
GALPOLYU, 784
GALUTIL, 787
GAUSSFAC, 790
GB, 907
GBEUCLID, 513
GBF, 863
GBINTERN, 871
GDRAW, 847
GENEEZ, 833
GENMFACT, 798
GENPGCD, 815
GENUFACT, 838
GENUPS, 829
GHENSEL, 794
GOSPER, 849
GPAFF, 801
GRAY, 857
GRDEF, 854
GROEBSOL, 911
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSF1, 953
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
HB, 963
HEUGCD, 966
IALGFACT, 993
IBACHIN, 165
IBATOOL, 1105
IBPTOOLS, 1108
ICDEN, 996
IDECOMP, 973
IMATLIN, 998
IMATQF, 1003
INBFF, 1020
INCRMAPS, 982
INEP, 1028
INFINITY, 992
INFORM1, 1045
INFPROD0, 983
INFSP, 1033
INMODGCD, 1005
INNMFACT, 1011
INPRODFF, 985
INPRODPF, 988
INPSIGN, 1037
INTAF, 28
INTALG, 21
INTBIT, 1049
INTDIVP, 1138
INTEF, 465
INTERGB, 1047
INTFACT, 1056
INTFRSP, 1131
INTG0, 840
INTHEORY, 1082
INTHERAL, 19
INTHERTR, 3252
INTPACK, 50
INTPAF, 2883
INTPM, 2687
INTRAT, 2970
INTRET, 1098
INTRF, 2955
INTSLPE, 1103
INTTOOLS, 1120
INTTR, 3254
INVLAPLA, 1150
IPRNTPK, 1124
IR2, 1111
IR2F, 1115
IROOT, 1099
IRREDFFX, 1141
IRRF2F, 1113
IRSN, 1143
IRURPK, 1125
ISUMP, 1039
ITFUN2, 990
ITFUN3, 991
ITRIGMNP, 1041
KERNEL2, 1155
KOVACIC, 1156
LAPLACE, 1159
LAZM3PK, 1184

LEADCDET, 1188
LEXTRIPK, 1260
LF, 1296
LGROBP, 1285
LIMITPS, 2830
LIMITRF, 2957
LINDEP, 1265
LIST2, 1301
LIST2MAP, 1304
LIST3, 1302
LISYSER, 1277
LODEEF, 426
LODOF, 1267
LODOOPS, 1271
LOP, 1293
LPARSPT, 1307
LPEFRAC, 1274
LSMP, 1279
LSMP1, 1281
LSPP, 1283
MAPHACK1, 1328
MAPHACK2, 1330
MAPHACK3, 1331
MAPPKG1, 1341
MAPPKG2, 1352
MAPPKG3, 1362
MAPPKG4, 1369
MATCAT2, 1371
MATLIN, 1375
MATSTOR, 3126
MCALCFN, 1426
MCDEN, 1373
MDDFACT, 1387
MESH, 1384
MFINFACT, 1414
MHROWRED, 1393
MKBCFUNC, 1315
MKFLCFN, 1317
MKFUNC, 1324
MKRECORD, 1326
MKUCFUNC, 1327
MMAP, 1424
MONOTOOL, 1400
MPC2, 1409
MPC3, 1410
MPCPF, 1403
MPRFF, 1405
MRATFAC, 1412
MRF2, 1398
MSYSCMD, 1402
MTHING, 1382
MULTFACT, 1430
MULTSQFR, 1436
NAGC02, 2267
NAGC05, 2286
NAGC06, 2333
NAGD01, 1818
NAGD02, 2203
NAGD03, 2249
NAGE01, 1866
NAGE02, 1652
NAGE04, 2104
NAGF01, 1946
NAGF02, 1511
NAGF04, 1731
NAGF07, 1887
NAGS, 2486
NAGSP, 1739
NCEP, 2575
NCNTFRAC, 2577
NCODIV, 2511
NEWTON, 2504
NFINTBAS, 2534
NLINSOL, 2519
NODE1, 2515
NONE1, 2514
NORMMA, 2526
NORMPK, 2521
NORMRETR, 2528
NPCOEF, 2530
NPOLYGON, 2506
NREP, 2579
NSUP2, 2502
NTPOLFN, 2544
NUMERIC, 2546
NUMFMT, 2539
NUMODE, 2555
NUMQUAD, 2563
NUMTUBE, 2581
OCTCT2, 2585
ODECONST, 221
ODEEF, 432

ODEINT, 2587
ODEPACK, 70
ODEPAL, 2892
ODEPRIM, 2849
ODEPRRIC, 2853
ODERAT, 2976
ODERED, 3025
ODERTRIC, 2982
ODESYS, 3182
ODETOOLS, 2589
OMEXPR, 556
OMPKG, 2594
OMSERVER, 2597
ONECOMP2, 2593
OPQUERY, 2599
OPTPACK, 61
ORDCOMP2, 2600
ORDFUNS, 2601
OREPCTO, 3349
ORTHPOL, 2603
OUT, 2606
PADE, 2637
PADEPAC, 2635
PAFF, 2611
PAFFFF, 2619
PAN2EXPR, 2759
PARAMP, 2666
PARPC2, 2662
PARSC2, 2663
PARSU2, 2664
PARTPERM, 2672
PATMATCH, 2678
PATRES2, 2705
PATTERN1, 2675
PATTERN2, 2677
PCOMP, 2765
PDECOMP, 2766
PDEPACK, 78
PERMAN, 2714
PFBR, 2768
PFBRU, 2775
PFO, 2740
PFOQ, 2747
PFORP, 2628
PFOTOOLS, 2749
PFRPAC, 2670
PGCD, 2782
PGE, 2718
PGROEB, 2757
PICOERCE, 2726
PINTERP, 2790
PINTERPA, 2792
PLEQN, 2648
PLOT1, 2728
PLOTTOOL, 2729
PLPKCRV, 2752
PMASS, 2681
PMASSFS, 712
PMDOWN, 2701
PMFS, 2682
PMINS, 2684
PMKERNEL, 2694
PMLSAGG, 2697
PMPLCAT, 2698
PMPRED, 99
PMPREDFS, 714
PMQFCAT, 2704
PMSYM, 2707
PMTOOLS, 2708
PNTHEORY, 2793
POLTOPOL, 2754
POLUTIL, 2993
POLY2, 2780
POLY2UP, 2828
POLYCATQ, 2762
POLYLIFT, 2760
POLYROOT, 2798
PREASSOC, 2841
PRIMARR2, 2844
PRIMELT, 2846
PRIMES, 1088
PRINT, 2859
PRJALGPK, 2733
PRS, 2864
PSETPK, 2801
PSEUDLIN, 2860
PSQFR, 2825
PTFUNC2, 2737
PTPACK, 2738
PUSHVAR, 2893
PWFFINTB, 2641
QALGSET2, 2897

QCMPACK, 2900
QFCAT2, 2909
QUATCT2, 2912
RADUTIL, 2936
RANDSRC, 2943
RATFACT, 2945
RATRET, 2981
RDEEF, 474
RDEEFS, 482
RDETR, 3273
RDETRS, 3277
RDIST, 2937
RDIV, 3023
REAL0, 3001
REAL0Q, 3008
REALSOLV, 2999
RECOP, 3012
REDORDER, 3027
REP, 2915
REP1, 3046
REP2, 3053
REPDB, 3043
REPSQ, 3045
RESLATC, 3070
RETSOL, 3071
RF, 2947
RFDIST, 2939
RFFACT, 2952
RFFACTOR, 2953
RFP, 3074
RIDIST, 2941
RINTERP, 2973
RMCAT2, 3010
RSDCMPK, 3029
RSETGCD, 3035
RURPK, 2989
SAEFACT, 3085
SAERFFC, 3079
SCACHE, 3092
SCPKG, 3165
SEG2, 3083
SEGBIND2, 3082
SFQCMPK, 3100
SFRGCD, 3116
SGCF, 3193
SHP, 3169
SIGNEF, 438
SIGNRF, 2960
SIMPAN, 3086
SMITH, 3087
SOLVEFOR, 2819
SOLVERAD, 2929
SOLVESER, 3299
SOLVETRA, 3288
SORTPAK, 3095
SPECOUT, 3098
SRDCMPK, 3110
STINPROD, 3136
STNSR, 3149
STREAM1, 3130
STREAM2, 3132
STREAM3, 3134
STTAYLOR, 3138
STTF, 3150
STTFNC, 3160
SUBRESP, 3177
SUMFS, 728
SUMRF, 2968
SUP2, 3097
SUPFRACF, 3180
SYMFUNC, 3204
SYSSOLP, 3188
TABLBUMP, 3207
TANEXP, 3214
TBCMPPK, 3210
TEMUTL, 3219
TEX1, 3221
TOOLSIGN, 3222
TOPSP, 3251
TRIGMNIP, 3304
TRIMAT, 3302
TRMANIP, 3264
TUBETOOL, 3308
TWOFACT, 3317
UDPO, 3359
UDVO, 3361
UFPS1, 3330
ULS2, 3331
UNIFACT, 3323
UNISEG2, 3358
UP2, 3341
UPCDEN, 3334

UPDECOMP, 3336
UPDIVP, 3339
UPMP, 3342
UPOLYC2, 3333
UPSQFREE, 3345
UPXS2, 3348
UTS2, 3353
UTSODE, 3354
UTSODETL, 3363
UTSSOL, 3216
VECTOR2, 3365
VIEW, 3372
VIEWDEF, 3367
WEIER, 3375
WFFINTBS, 3379
XEXPPKG, 3385
YSTREAM, 2646
ZDSOLVE, 3454
ZLINDEP, 1067
IMATLIN, 998
IMATQF, 1003
inverse, 1003
nullSpace, 1003
rowEchelon, 1003
in?
ESCONT1, 532
ESTOOLS, 534
inBetweenExcpDiv
DTP, 268
INBFF, 1020
?**?, 1021
?*?, 1021
?/?, 1021
basis, 1021
dAndcExp, 1021
expPot, 1021
index, 1021
inv, 1021
lookup, 1021
minimalPolynomial, 1021
norm, 1021
normal?, 1021
normalElement, 1021
pol, 1021
qPot, 1021
random, 1021
repSq, 1021
setFieldInfo, 1021
trace, 1021
xn, 1021
inconsistent?
PLEQN, 2648
increment
INCRMAPS, 982
incrementBy
INCRMAPS, 982
IncrementingMaps, 982
INCRMAPS, 982
increment, 982
incrementBy, 982
index
INBFF, 1021
indiceSubResultant
PRS, 2864
indiceSubResultantEuclidean
PRS, 2864
indicialEquation
ODEPRIM, 2849
indicialEquationAtInfinity
ODERAT, 2976
indicialEquations
ODEPRIM, 2849
INEP, 1028
charpol, 1028
innerEigenvectors, 1028
solve1, 1028
infieldint
INTRAT, 2970
infieldIntegrate
INTRF, 2955
infiniteProduct
INFPROD0, 983
INPRODFF, 985
INPRODPF, 988
STINPROD, 3136
InfiniteProductCharacteristicZero, 983
InfiniteProductFiniteField, 985
InfiniteProductPrimeField, 988
InfiniteTupleFunctions2, 990
InfiniteTupleFunctions3, 991
INFINITY, 992
infinity, 992

minusInfinity, 992
plusInfinity, 992
Infinity, 992
infinity
INFINITY, 992
infinityNorm
GALFACTU, 763
INFORM1, 1045
interpret, 1045
packageCall, 1045
INFPROD0, 983
evenInfiniteProduct, 983
generalInfiniteProduct, 983
infiniteProduct, 983
oddInfiniteProduct, 983
infRittWu?
QCMPACK, 2900
SFQCMPK, 3101
INFSP, 1033
innerSolve, 1033
innerSolve1, 1033
makeEq, 1033
inHallBasis?
HB, 963
initializeParamOfPlaces
DTP, 268
initParLocLeaves
DTP, 268
INMODGCD, 1005
modularGcd, 1005
modularGcdPrimitive, 1005
reduction, 1005
InnerAlgFactor, 993
InnerCommonDenominator, 996
innerEigenvectors
INEP, 1028
innerint
DEFINTEF, 421
InnerMatrixLinearAlgebraFunctions, 998
InnerMatrixQuotientFieldFunctions, 1003
InnerModularGcd, 1005
InnerMultFact, 1011
InnerNormalBasisFieldFunctions, 1020
InnerNumericEigenPackage, 1028
InnerNumericFloatSolvePackage, 1033
InnerPolySign, 1037
InnerPolySum, 1039
innerSolve
INFSP, 1033
innerSolve1
INFSP, 1033
InnerTrigonometricManipulations, 1041
INNMFACT, 1011
factor, 1011
INPRODFF, 985
evenInfiniteProduct, 985
generalInfiniteProduct, 985
infiniteProduct, 985
oddInfiniteProduct, 985
INPRODPF, 988
evenInfiniteProduct, 988
generalInfiniteProduct, 988
infiniteProduct, 988
oddInfiniteProduct, 988
INPSIGN, 1037
signAround, 1038
InputFormFunctions1, 1045
inrootof
AF, 13
int
ODEINT, 2587
STTAYLOR, 3138
INTAF, 28
algint, 28
INTALG, 21
algintegrate, 21
palginfieldint, 21
palgintegrate, 21
INTBIT, 1049
bitCoef, 1049
bitLength, 1049
bitTruth, 1049
intChoose
MULTSQFR, 1436
intcompBasis
LGROBP, 1285
INTDIVP, 1138
intersectionDivisor, 1138
placesOfDegree, 1138
INTEF, 465
lfextendedint, 465
lfextlimint, 465

lfinfieldint, 465
lfintegrate, 465
lflimitedint, 465
integer
INTRET, 1098
integer?
INTRET, 1098
IntegerBits, 1049
integerBound
BOUNDZRO, 152
IntegerCombinatoricFunctions, 1053
IntegerFactorizationPackage, 1056
integerIfCan
INTRET, 1098
IntegerLinearDependence, 1067
IntegerNumberTheoryFunctions, 1082
IntegerPrimesPackage, 1088
IntegerRetractions, 1098
IntegerRoots, 1099
integers
STTAYLOR, 3138
IntegerSolveLinearPolynomialEquation, 1103
integral
LF, 1296
integralBasis
FFINTBAS, 709
NFINTBAS, 2534
PWFFINTB, 2641
WFFINTBS, 3379
IntegralBasisPolynomialTools, 1108
IntegralBasisTools, 1105
integralLastSubResultant
RSETGCD, 3035
integrate
DEFINTEF, 421
DEFINTRF, 2949
FSINT, 720
INTPACK, 50
INTRAT, 2970
IRRF2F, 1113
STTAYLOR, 3138
IntegrationResultFunctions2, 1111
IntegrationResultRFToFunction, 1113
IntegrationResultToFunction, 1115
IntegrationTools, 1120
InterfaceGroebnerPackage, 1047
INTERGB, 1047
groebner, 1047
intermediateResultsIF
D02AGNT, 375
internalDecompose
RSDCMPK, 3029
SRDCMPK, 3110
internalInfRittWu?
QCMPACK, 2900
SFQCMPK, 3101
internalIntegrate
FSCINT, 716
INTRF, 2955
internalIntegrate0
FSCINT, 716
InternalPrintPackage, 1124
InternalRationalUnivariateRepresentationPackage, 1125
internalSubPolSet?
QCMPACK, 2900
SFQCMPK, 3101
internalSubQuasiComponent?
QCMPACK, 2900
SFQCMPK, 3101
interpolate
FFFG, 683
PINTERP, 2790
RINTERP, 2973
interpolateForms
GPAFF, 801
INTFRSP, 1131
PAFF, 2611
PAFFFF, 2619
interpolateFormsForFact
GPAFF, 801
INTFRSP, 1131
PAFF, 2611
PAFFFF, 2619
InterpolateFormsPackage, 1131
interpret
INFORM1, 1045
interpretString
TEMUTL, 3219
interReduce
PSETPK, 2802
intersectionDivisor

GPAFF, 801
INTDIVP, 1138
PAFF, 2611
PAFFFF, 2619
IntersectionDivisorPackage, 1138
INTFACT, 1056
BasicMethod, 1057
factor, 1057
PollardSmallFactor, 1057
squareFree, 1057
INTFRSP, 1131
basisOfInterpolateForms, 1131
basisOfInterpolateFormsForFact, 1131
interpolateForms, 1131
interpolateFormsForFact, 1131
INTG0, 840
lift, 840
multivariate, 840
palgextint0, 840
palgint0, 840
palglimint0, 840
palgLODE0, 840
palgRDE0, 840
univariate, 840
INTHEORY, 1082
bernoulli, 1082
chineseRemainder, 1082
divisors, 1082
euler, 1082
eulerPhi, 1082
fibonacci, 1082
harmonic, 1082
jacobi, 1082
legendre, 1082
moebiusMu, 1082
numberOfDivisors, 1082
sumOfDivisors, 1082
sumOfKthPowerDivisors, 1082
INTHERAL, 19
HermiteIntegrate, 19
INTHERTR, 3252
HermiteIntegrate, 3252
INTPACK, 50
integrate, 50
measure, 50
INTPAF, 2883
palgextint, 2883
palgint, 2883
palglimint, 2883
palgLODE, 2883
palgRDE, 2883
intPatternMatch
INTTOOLS, 1120
INTPM, 2687
pmComplexintegrate, 2687
pmintegrate, 2687
splitConstant, 2687
INTRAT, 2970
extendedint, 2970
infieldint, 2970
integrate, 2970
limitedint, 2970
INTRET, 1098
integer, 1098
integer?, 1098
integerIfCan, 1098
INTRF, 2955
extendedIntegrate, 2955
infieldIntegrate, 2955
internalIntegrate, 2955
limitedIntegrate, 2955
INTSLPE, 1103
solveLinearPolynomialEquation, 1103
INTTOOLS, 1120
intPatternMatch, 1120
kmax, 1120
ksec, 1120
mkPrim, 1120
removeConstantTerm, 1120
union, 1120
vark, 1120
varselect, 1120
INTTR, 3254
expextendedint, 3254
expintegrate, 3254
expintfldpoly, 3254
explimitedint, 3254
monomialIntegrate, 3254
monomialIntPoly, 3254
primextendedint, 3254
primextintfrac, 3254
primintegrate, 3254

primintfldpoly, 3254
primlimintfrac, 3254
primlimitedint, 3254
tanintegrate, 3254
inv
INBFF, 1021
inverse
IMATQF, 1003
MATLIN, 1375
TABLBUMP, 3207
inverseColeman
SGCF, 3194
inverseLaplace
INVLAPLA, 1150
InverseLaplaceTransform, 1150
invertIfCan
MATLIN, 1375
INVLAPLA, 1150
inverseLaplace, 1150
invmultisect
STTAYLOR, 3138
ipow
COMBF, 175
iprint
IPRNTPK, 1124
IPRNTPK, 1124
iprint, 1124
IR2, 1111
map, 1111
IR2F, 1115
complexExpand, 1115
expand, 1115
split, 1115
IROOT, 1099
approxNthRoot, 1099
approxSqrt, 1099
perfectNthPower?, 1099
perfectNthRoot, 1099
perfectSqrt, 1099
perfectSquare?, 1099
iroot
AF, 13
IRREDFFX, 1141
generateIrredPoly, 1141
IrredPolyOverFiniteField, 1141
irreducible?
DDFACT, 288
FFFACTSE, 612
irreducibleFactors
PSETPK, 2802
irreducibleRepresentation
IRSN, 1143
IRRF2F, 1113
complexExpand, 1113
complexIntegrate, 1113
expand, 1113
integrate, 1113
split, 1113
IrrRepSymNatPackage, 1143
IRSN, 1143
dimensionOfIrreducibleRepresentation, 1143
irreducibleRepresentation, 1143
IRURPK, 1125
checkRur, 1125
rur, 1125
Is
PATMATCH, 2678
is?
PATMATCH, 2678
isAbsolutelyIrreducible?
REP2, 3053
isExpt
POLYCATQ, 2762
isPlus
POLYCATQ, 2762
isPower
POLYCATQ, 2762
isQuotient
ESTOOLS, 534
isTimes
POLYCATQ, 2762
ISUMP, 1039
sum, 1039
iter
MAPHACK1, 1328
ITFUN2, 990
map, 990
ITFUN3, 991
map, 991
ITRIGMNP, 1041
explogs2trigs, 1041
F2FG, 1041

FG2F, 1041
GF2FG, 1041
trigs2explogs, 1041

jacobi
INTHEORY, 1082
jacobian
D02AGNT, 375
MCALCFN, 1426
janko2
PGE, 2718

karatsuba
UPMP, 3342
karatsubaOnce
UPMP, 3342
KERNEL2, 1155
constantIfCan, 1155
constantKernel, 1155
KernelFunctions2, 1155
kmax
INTTOOLS, 1120
KOVACIC, 1156
kovacic, 1156
Kovacic, 1156
kovacic
KOVACIC, 1156
KrullNumber
RSDCMPK, 3029
SRDCMPK, 3110
ksec
INTTOOLS, 1120

lagrange
STTAYLOR, 3138
LagrangeInterpolation
PINTERPA, 2792
laguerre
PNTHEORY, 2793
laguerreL
ORTHPOL, 2603
lambert
STTAYLOR, 3138
LAPLACE, 1159
laplace, 1159
laplace
LAPLACE, 1159
LaplaceTransform, 1159
laplacian
MCALCFN, 1426
largest
UDPO, 3359
lastSubResultant
PRS, 2864
lastSubResultantEuclidean
PRS, 2864
laurent
EXPR2UPS, 562
GENUPS, 829
Lazard
PRS, 2864
Lazard2
PRS, 2864
LazardSetSolvingPackage, 1184
LAZM3PK, 1184
normalizeIfCan, 1185
zeroSetSplit, 1185
lazyGintegrate
STTAYLOR, 3138
lazyIntegrate
STTAYLOR, 3138
lazyIrreducibleFactors
PSETPK, 2802
lazyVariations
POLUTIL, 2993
lBasis
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
ldf2lst
D01AGNT, 363
ESCONT, 527
ESTOOLS, 534
ldf2vmf
ESTOOLS, 534
LEADCDET, 1188
distFact, 1188
polCase, 1188
LeadingCoefDetermination, 1188
leadingCoefficientRicDE
ODEPRRIC, 2853
leastAffineMultiple

FFPOLY, 630
leastPower
IBATOOL, 1105
leftDivide
NCODIV, 2511
OREPCTO, 3349
leftExactQuotient
NCODIV, 2511
leftFactor
PDECOMP, 2766
leftFactorIfCan
UPDECOMP, 3336
leftGcd
NCODIV, 2511
leftLcm
NCODIV, 2511
leftQuotient
NCODIV, 2511
leftRank
ALGPKG, 37
leftRemainder
NCODIV, 2511
legendre
INTHEORY, 1082
PNTHEORY, 2793
legendreP
ORTHPOL, 2603
length
GALFACTU, 763
lepol
GBINTERN, 871
less?
UDPO, 3359
lex
TABLBUMP, 3207
lexGroebner
PGROEB, 2757
lexTriangular
LEXTRIPK, 1260
LexTriangularPackage, 1260
LEXTRIPK, 1260
fglmIfCan, 1260
groebner, 1260
lexTriangular, 1260
squareFreeLexTriangular, 1260
zeroDimensional?, 1260
zeroSetSplit, 1260
LF, 1296
belong?, 1296
Ci, 1296
dilog, 1296
Ei, 1296
erf, 1296
integral, 1296
li, 1296
operator, 1296
Si, 1296
lfextendedint
INTEF, 465
lfextlimint
INTEF, 465
lfinfieldint
INTEF, 465
lfintegrate
INTEF, 465
lflimitedint
INTEF, 465
lfunc
HB, 963
LGROBP, 1285
anticoord, 1285
choosemon, 1285
computeBasis, 1285
coord, 1285
groebgen, 1285
intcompBasis, 1285
linGenPos, 1285
minPol, 1285
totolex, 1285
transform, 1285
li
LF, 1296
lift
INTG0, 840
MULTSQFR, 1436
limit
LIMITPS, 2830
LIMITRF, 2957
limitedint
INTRAT, 2970
limitedIntegrate
INTRF, 2955

LIMITPS, 2830
complexLimit, 2830
limit, 2830
LIMITRF, 2957
complexLimit, 2957
limit, 2957
LINDEP, 1265
linearDependence, 1265
linearlyDependent?, 1265
solveLinear, 1265
linear
SOLVEFOR, 2819
linear?
E04AGNT, 577
PSETPK, 2802
linearBezier
BEZIER, 139
LinearDependence, 1265
linearDependence
LINDEP, 1265
linearDependenceOverZ
ZLINDEP, 1067
linearlyDependent?
LINDEP, 1265
linearlyDependentOverZ?
ZLINDEP, 1067
linearMatrix
E04AGNT, 577
LinearOrdinaryDifferentialOperatorFactorizer, 1267
LinearOrdinaryDifferentialOperatorsOps, 1271
linearPart
E04AGNT, 577
LinearPolynomialEquationByFractions, 1274
linearPolynomials
PSETPK, 2802
linears
MDDFACT, 1387
LinearSystemFromPowerSeriesPackage, 1277
LinearSystemMatrixPackage, 1279
LinearSystemMatrixPackage1, 1281
LinearSystemPolynomialPackage, 1283
lineColorDefault
VIEWDEF, 3367
LinesOpPack, 1293
linGenPos
LGROBP, 1285
LinGroebnerPackage, 1285
linkToFortran
FORT, 679
linSolve
LSPP, 1283
lintgcd
HEUGCD, 966
LiouvillianFunction, 1296
LIST2, 1301
map, 1301
reduce, 1301
scan, 1301
LIST2MAP, 1304
match, 1304
LIST3, 1302
map, 1303
listAllMono
PFORP, 2628
listAllMonoExp
PFORP, 2628
listConjugateBases
IBACHIN, 165
listexp
NPCOEF, 2530
ListFunctions2, 1301
ListFunctions3, 1302
ListToMap, 1304
listVariable
PFORP, 2628
listYoungTableaus
SGCF, 3194
LISYSER, 1277
finiteSeries2LinSys, 1277
finiteSeries2LinSysWOVectorise, 1277
finiteSeries2Vector, 1277
localAbs
FS2EXPXP, 730
FS2UPS, 742
localIntegralBasis
FFINTBAS, 709
NFINTBAS, 2534
PWFFINTB, 2641
WFFINTBS, 3379
localize
LPARSPT, 1307

LocalParametrizationOfSimplePointPackage, 1307
localParamOfSimplePt
  LPARSPT, 1307
localReal?
  EF, 403
localUnquote
  APPRULE, 92
LODEEF, 426
  solve, 426
LODO2FUN
  UTSODETL, 3363
LODOF, 1267
  factor, 1267
  factor1, 1267
LODOOPS, 1271
  directSum, 1271
  symmetricPower, 1271
  symmetricProduct, 1271
log
  EF, 403
  EFULS, 451
  EFUPXS, 459
  FACTFUNC, 585
  XEXPPKG, 3385
logGamma
  DFSFUN, 309
logIfCan
  EFULS, 451
  EFUPXS, 459
lookup
  INBFF, 1021
loopPoints
  TUBETOOL, 3308
LOP, 1293
  quotVecSpaceBasis, 1293
  reduceLineOverLine, 1293
  reduceRow, 1293
  reduceRowOnList, 1293
  rowEchWoZeroLines, 1293
  rowEchWoZeroLinesWOVectorise, 1293
lowerPolynomial
  FACUTIL, 593
LowTriBddDenomInv
  TRIMAT, 3302
LPARSPT, 1307
  localize, 1307
  localParamOfSimplePt, 1307
  pointDominateBy, 1307
  pointToPlace, 1307
  printInfo, 1307
LPEFRAC, 1274
  solveLinearPolynomialEquationByFractions, 1274
LPolynomial
  GPAFF, 801
  PAFF, 2611
  PAFFFF, 2619
LSMP, 1279
  hasSolution?, 1279
  particularSolution, 1279
  rank, 1279
  solve, 1279
LSMP1, 1281
  hasSolution?, 1281
  particularSolution, 1281
  rank, 1281
  solve, 1281
LSPP, 1283
  linSolve, 1283

mainVariable
  POLYCATQ, 2762
  RF, 2947
MakeBinaryCompiledFunction, 1315
makeCrit
  GBINTERN, 871
makeEq
  INFSP, 1033
MakeFloatCompiledFunction, 1317
makeFR
  GALFACT, 767
MakeFunction, 1324
makeObject
  DRAW, 3224
  DRAWCFUN, 3235
MakeRecord, 1326
makeRecord
  MKRECORD, 1326
MakeUnaryCompiledFunction, 1327
makeYoungTableau
  SGCF, 3194

makingStats?
    TBCMPPK, 3211
map
    ARRAY12, 2591
    CARTEN2, 157
    COMPLEX2, 198
    DIRPROD2, 281
    EQ2, 487
    ES1, 549
    ES2, 550
    ESTOOLS2, 543
    EXPR2, 545
    FAMR2, 608
    FDIV2, 609
    FFCAT2, 708
    FLAGG2, 659
    FR2, 589
    FRAC2, 696
    FRIDEAL2, 681
    FRNAAF2, 698
    FS2, 718
    FSAGG2, 665
    IR2, 1111
    ITFUN2, 990
    ITFUN3, 991
    LIST2, 1301
    LIST3, 1303
    MATCAT2, 1371
    MMAP, 1424
    MPC2, 1409
    MPC3, 1411
    MRF2, 1398
    NSUP2, 2502
    OCTCT2, 2585
    ONECOMP2, 2593
    ORDCOMP2, 2600
    PARPC2, 2662
    PARSC2, 2663
    PARSU2, 2664
    PATRES2, 2705
    PATTERN2, 2677
    POLY2, 2780
    POLYLIFT, 2760
    PRIMARR2, 2844
    PTFUNC2, 2737
    QFCAT2, 2909
    QUATCT2, 2912
    RMCAT2, 3010
    SEG2, 3083
    SEGBIND2, 3082
    STREAM2, 3132
    STREAM3, 3134
    SUP2, 3097
    ULS2, 3331
    UNISEG2, 3358
    UP2, 3341
    UPOLYC2, 3333
    UPXS2, 3348
    UTS2, 3353
    VECTOR2, 3365
mapBivariate
    IBPTOOLS, 1109
mapdiv
    STTAYLOR, 3138
mapExponents
    PFORP, 2628
MAPHACK1, 1328
    iter, 1328
    recur, 1328
MAPHACK2, 1330
    arg1, 1330
    arg2, 1330
MAPHACK3, 1331
    comp, 1331
mapMatrixIfCan
    IBPTOOLS, 1109
mapmult
    STTAYLOR, 3138
MappingPackage1, 1341
MappingPackage2, 1352
MappingPackage3, 1362
MappingPackage4, 1369
MappingPackageInternalHacks1, 1328
MappingPackageInternalHacks2, 1330
MappingPackageInternalHacks3, 1331
MAPPKG1, 1341
    ?**?, 1341
    coerce, 1341
    fixedPoint, 1341
    id, 1341
    nullary, 1341
    recur, 1341

MAPPKG2, 1352
const, 1352
constant, 1352
curry, 1352
diag, 1352
MAPPKG3, 1362
?*?, 1362
constantLeft, 1362
constantRight, 1362
curryLeft, 1362
curryRight, 1362
twist, 1362
MAPPKG4, 1369
?*?, 1369
?+?, 1369
?-?, 1369
?/?, 1369
mapSolve
SOLVEFOR, 2819
mapUnivariate
IBPTOOLS, 1109
mapUnivariateIfCan
IBPTOOLS, 1109
mat
ESTOOLS, 534
MATCAT2, 1371
map, 1371
reduce, 1371
match
LIST2MAP, 1304
mathieu11
PGE, 2718
mathieu12
PGE, 2718
mathieu22
PGE, 2718
mathieu23
PGE, 2718
mathieu24
PGE, 2718
MATLIN, 1375
adjoint, 1375
determinant, 1375
elColumn2, 1375
elRow1, 1375
elRow2, 1375
fractionFreeGauss, 1375
inverse, 1375
invertIfCan, 1375
minordet, 1375
normalizedDivide, 1375
nullity, 1375
nullSpace, 1375
rank, 1375
rowEchelon, 1375
MatrixCategoryFunctions2, 1371
MatrixCommonDenominator, 1373
matrixGcd
IBATOOL, 1105
MatrixLinearAlgebraFunctions, 1375
MATSTOR, 3126
?**?, 3126
copy, 3126
leftScalarTimes, 3126
minus, 3126
plus, 3126
power, 3126
rightScalarTimes, 3126
times, 3126
maxPoints
GRDEF, 854
maxrank
PLEQN, 2648
maxrow
TABLBUMP, 3207
MCALCFN, 1426
bandedHessian, 1426
bandedJacobian, 1426
divergence, 1426
gradient, 1426
hessian, 1426
jacobian, 1426
laplacian, 1426
MCDEN, 1373
clearDenominator, 1373
commonDenominator, 1373
splitDenominator, 1373
MDDFACT, 1387
ddFact, 1387
exptMod, 1387
factor, 1387
gcd, 1387

linears, 1387
separateFactors, 1387
measure
INTPACK, 50
ODEPACK, 70
OPTPACK, 61
PDEPACK, 78
measure2Result
ESTOOLS, 534
meatAxe
REP2, 3053
mergeDifference
MTHING, 1382
mergeFactors
FRUTIL, 591
MergeThing, 1382
MESH, 1384
meshFun2Var, 1384
meshPar1Var, 1384
meshPar2Var, 1384
ptFunc, 1384
MeshCreationRoutinesForThreeDimensions, 1384
meshFun2Var
MESH, 1384
meshPar1Var
MESH, 1384
meshPar2Var
MESH, 1384
MFINFACT, 1414
factor, 1414
MHROWRED, 1393
normalizedDivide, 1393
rowEch, 1393
rowEchelon, 1393
rowEchelonLocal, 1393
rowEchLocal, 1393
midpoint
REAL0, 3001
midpoints
REAL0, 3001
minGbasis
GBINTERN, 871
minimalForm
PFORP, 2628
PLPKCRV, 2752
minimalPolynomial
INBFF, 1021
minordet
MATLIN, 1375
minPoints
GRDEF, 854
minPol
LGROBP, 1285
minPoly
AF, 13
minrank
PLEQN, 2648
minset
PLEQN, 2648
minusInfinity
INFINITY, 992
mix
PFOTOOLS, 2749
MKBCFUNC, 1315
binaryFunction, 1315
compiledFunction, 1315
MKFLCFN, 1317
MKFUNC, 1324
function, 1324
mkIntegral
CHVAR, 159
mkPrim
INTTOOLS, 1120
MKRECORD, 1326
makeRecord, 1326
MKUCFUNC, 1327
compiledFunction, 1327
unaryFunction, 1327
MMAP, 1424
map, 1424
modTree
CRAPACK, 228
ModularDistinctDegreeFactorizer, 1387
modularFactor
GALFACT, 767
modularGcd
INMODGCD, 1005
modularGcdPrimitive
INMODGCD, 1005
ModularHermitianRowReduction, 1393
moduleSum
IBATOOL, 1105

moebiusMu
INTHEORY, 1082
monic?
GALPOLYU, 784
monicCompleteDecompose
UPDECOMP, 3336
monicDecomposeIfCan
UPDECOMP, 3336
monicLeftDivide
OREPCTO, 3349
monicRightDivide
OREPCTO, 3349
monicRightFactorIfCan
UPDECOMP, 3336
MonoidRingFunctions2, 1398
monom
STTAYLOR, 3138
MonomialExtensionTools, 1400
monomialIntegrate
INTTR, 3254
monomialIntPoly
INTTR, 3254
monomials
PFORP, 2628
monomRDE
RDETR, 3273
monomRDEsys
RDETRS, 3277
MONOTOOL, 1400
decompose, 1400
normalDenom, 1400
split, 1400
splitSquarefree, 1400
more?
UDPO, 3359
moreAlgebraic?
QCMPACK, 2900
SFQCMPK, 3101
MoreSystemCommands, 1402
MPC2, 1409
map, 1409
reshape, 1409
MPC3, 1410
map, 1411
MPCPF, 1403
factor, 1403
MPolyCatFunctions2, 1409
MPolyCatFunctions3, 1410
MPolyCatPolyFactorizer, 1403
MPolyCatRationalFunctionFactorizer, 1405
MPRFF, 1405
factor, 1405
pushdown, 1405
pushdterm, 1405
pushucoef, 1405
pushuconst, 1405
pushup, 1405
totalfract, 1405
mpsode
UTSODE, 3354
mr
TABLBUMP, 3207
MRATFAC, 1412
factor, 1412
MRationalFactorize, 1412
MRF2, 1398
map, 1398
MSYSCMD, 1402
systemCommand, 1402
MTHING, 1382
mergeDifference, 1382
MULTFACT, 1430
factor, 1430
MultFiniteFactorize, 1414
multiEuclideanTree
CRAPACK, 228
multinomial
COMBINAT, 1053
multiple
PMASS, 2681
PMASSFS, 712
MultipleMap, 1424
multiplicity
NPOLYGON, 2506
PLPKCRV, 2752
multisect
STTAYLOR, 3138
multiServ
AXSERV, 100
MultiVariableCalculusFunctions, 1426
multivariate
INTG0, 840

POLYCATQ, 2762
RF, 2947
MultivariateFactorize, 1430
MultivariateSquareFree, 1436
MULTSQFR, 1436
check, 1436
coefChoose, 1436
compdegd, 1436
consnewpol, 1436
intChoose, 1436
lift, 1436
myDegree, 1436
normDeriv2, 1436
nsqfree, 1436
squareFree, 1436
squareFreePrim, 1436
univcase, 1436
musserTrials
GALFACT, 767
myDegree
MULTSQFR, 1436

NAGC02, 2267
c02aff, 2267
c02agf, 2267
NAGC05, 2286
c05adf, 2286
c05nbf, 2286
c05pbf, 2286
NAGC06, 2333
c06eaf, 2333
c06ebf, 2333
c06ecf, 2333
c06ekf, 2333
c06fpf, 2333
c06fqf, 2333
c06frf, 2333
c06fuf, 2333
c06gbf, 2333
c06gcf, 2333
c06gqf, 2333
c06gsf, 2333
NAGD01, 1818
d01ajf, 1818
d01akf, 1818
d01alf, 1818
d01amf, 1818
d01anf, 1818
d01apf, 1818
d01aqf, 1818
d01asf, 1818
d01bbf, 1818
d01fcf, 1818
d01gaf, 1818
d01gbf, 1818
NAGD02, 2203
d02bbf, 2203
d02bhf, 2203
d02cjf, 2203
d02ejf, 2203
d02gaf, 2203
d02gbf, 2203
d02kef, 2203
d02raf, 2203
NAGD03, 2249
d03edf, 2249
d03eef, 2249
d03faf, 2249
NAGE01, 1866
e01baf, 1866
e01bef, 1866
e01bff, 1866
e01bgf, 1866
e01bhf, 1866
e01daf, 1866
e01saf, 1866
e01sbf, 1866
e01sef, 1866
e01sff, 1866
NAGE02, 1652
e02adf, 1653
e02aef, 1653
e02agf, 1653
e02ahf, 1653
e02ajf, 1653
e02akf, 1653
e02baf, 1653
e02bbf, 1653
e02bcf, 1653
e02bdf, 1653
e02bef, 1653
e02daf, 1653

e02dcf, 1653
e02ddf, 1653
e02def, 1653
e02dff, 1653
e02gaf, 1653
e02zaf, 1653
NAGE04, 2104
e04dgf, 2104
e04fdf, 2104
e04gcf, 2104
e04jaf, 2104
e04mbf, 2104
e04naf, 2104
e04ucf, 2104
e04ycf, 2104
NagEigenPackage, 1511
NAGF01, 1946
f01brf, 1946
f01bsf, 1946
f01maf, 1946
f01mcf, 1946
f01qcf, 1946
f01qdf, 1946
f01qef, 1946
f01rcf, 1946
f01rdf, 1946
f01ref, 1946
NAGF02, 1511
f02aaf, 1511
f02abf, 1511
f02adf, 1511
f02aef, 1511
f02aff, 1511
f02agf, 1511
f02ajf, 1511
f02akf, 1511
f02awf, 1511
f02axf, 1511
f02bbf, 1511
f02bjf, 1511
f02fjf, 1511
f02wef, 1511
f02xef, 1511
NAGF04, 1731
f04adf, 1731
f04arf, 1731
f04asf, 1731
f04atf, 1731
f04axf, 1731
f04faf, 1731
f04jgf, 1731
f04maf, 1731
f04mbf, 1731
f04mcf, 1731
f04qaf, 1731
NAGF07, 1887
f07adf, 1887
f07aef, 1887
f07fdf, 1887
f07fef, 1887
NagFittingPackage, 1652
NagIntegrationPackage, 1818
NagInterpolationPackage, 1866
NagLapack, 1887
NagLinearEquationSolvingPackage, 1731
NAGLinkSupportPackage, 1739
NagMatrixOperationsPackage, 1946
NagOptimisationPackage, 2104
NagOrdinaryDifferentialEquationsPackage, 2203
NagPartialDifferentialEquationsPackage, 2249
NagPolynomialRootsPackage, 2267
NagRootFindingPackage, 2286
NAGS, 2486
s01eaf, 2486
s13aaf, 2486
s13acf, 2486
s13adf, 2486
s14aaf, 2486
s14abf, 2486
s14baf, 2486
s15adf, 2486
s15aef, 2486
s17acf, 2486
s17adf, 2486
s17aef, 2486
s17aff, 2486
s17agf, 2486
s17ahf, 2486
s17ajf, 2486
s17akf, 2486
s17dcf, 2486
s17def, 2486

s17dgf, 2486
s17dhf, 2486
s17dlf, 2486
s18acf, 2486
s18adf, 2486
s18aef, 2486
s18aff, 2486
s18dcf, 2486
s18def, 2486
s19aaf, 2486
s19abf, 2486
s19acf, 2486
s19adf, 2486
s20acf, 2486
s20adf, 2486
s21baf, 2486
s21bbf, 2486
s21bcf, 2486
s21bdf, 2486
NagSeriesSummationPackage, 2333
NAGSP, 1739
aspFilename, 1739
checkPrecision, 1739
dimensionsOf, 1739
fortranCompilerName, 1739
fortranLinkerArgs, 1739
restorePrecision, 1739
NagSpecialFunctionsPackage, 2486
NCEP, 2575
characteristicPolynomial, 2575
complexEigenvalues, 2575
complexEigenvectors, 2575
NCNTFRAC, 2577
continuedFraction, 2577
NCODIV, 2511
leftDivide, 2511
leftExactQuotient, 2511
leftGcd, 2511
leftLcm, 2511
leftQuotient, 2511
leftRemainder, 2511
negAndPosEdge
NPOLYGON, 2506
neglist
ESTOOLS1, 542
newLine
DISPLAY, 285
newReduc
FSRED, 726
NewSparseUnivariatePolynomialFunctions2, 2502
NEWTON, 2504
newton, 2504
newton
NEWTON, 2504
NewtonInterpolation, 2504
NewtonPolygon, 2506
newtonPolygon
NPOLYGON, 2506
newtonPolySlope
BLUPPACK, 146
nextColeman
SGCF, 3194
nextIrreduciblePoly
FFPOLY, 630
nextLatticePermutation
SGCF, 3194
nextNormalPoly
FFPOLY, 630
nextNormalPrimitivePoly
FFPOLY, 630
nextPartition
SGCF, 3194
nextPrime
PRIMES, 1088
nextPrimitiveNormalPoly
FFPOLY, 630
nextPrimitivePoly
FFPOLY, 630
nextsousResultant2
PRS, 2864
nextSublist
PLEQN, 2648
nextSubsetGray
GRAY, 857
NFINTBAS, 2534
discriminant, 2534
integralBasis, 2534
localIntegralBasis, 2534
nlde
STTAYLOR, 3138
NLINSOL, 2519
solve, 2519

solveInField, 2519
NODE1, 2515
solve, 2515
noKaratsuba
UPMP, 3342
noLinearFactor?
BRILL, 154
NonCommutativeOperatorDivision, 2511
NONE1, 2514
coerce, 2514
NoneFunctions1, 2514
NonLinearFirstOrderODESolver, 2515
nonLinearPart
E04AGNT, 577
NonLinearSolvePackage, 2519
nonQsign
TOOLSIGN, 3222
norm
CRFP, 204
GALFACTU, 763
INBFF, 1021
NORMMA, 2526
normal
RFDIST, 2939
normal01
RFDIST, 2939
normal?
FFPOLY, 630
INBFF, 1021
normalDenom
MONOTOOL, 1400
normalDeriv
FACUTIL, 593
normalElement
INBFF, 1021
normalForm
GB, 907
PSEUDLIN, 2860
normalise
REP, 2915
NormalizationPackage, 2521
normalize
EFSTRUC, 442
NORMPK, 2522
normalizedAssociate
NORMPK, 2522
normalizedDivide
MATLIN, 1375
MHROWRED, 1393
normalizeIfCan
LAZM3PK, 1185
normDeriv2
MULTSQFR, 1436
normFactors
NORMRETR, 2528
NormInMonogenicAlgebra, 2526
normInvertible?
NORMPK, 2522
NORMMA, 2526
norm, 2526
NORMPK, 2521
normalize, 2522
normalizedAssociate, 2522
normInvertible?, 2522
outputArgs, 2522
recip, 2522
NORMRETR, 2528
Frobenius, 2528
normFactors, 2528
retractIfCan, 2528
NormRetractPackage, 2528
NPCOEF, 2530
listexp, 2530
npcoef, 2530
NPCoef, 2530
npcoef
NPCOEF, 2530
NPOLYGON, 2506
multiplicity, 2506
negAndPosEdge, 2506
newtonPolygon, 2506
slope, 2506
NREP, 2579
characteristicPolynomial, 2579
realEigenvalues, 2579
realEigenvectors, 2579
nsqfree
MULTSQFR, 1436
NSUP2, 2502
map, 2502
nthr
POLYROOT, 2798

nthRoot
FACTFUNC, 585
nthRootIfCan
EFULS, 451
EFUPXS, 459
NTPOLFN, 2544
bernoulliB, 2544
cyclotomic, 2544
eulerE, 2544
nullary
MAPPKG1, 1341
nullity
MATLIN, 1375
nullSpace
IMATQF, 1003
MATLIN, 1375
NumberFieldIntegralBasis, 2534
NumberFormats, 2539
numberOfDivisors
INTHEORY, 1082
numberOfFactors
GALFACT, 767
numberOfImproperPartitions
SGCF, 3194
numberOfIrreduciblePoly
FFPOLY, 630
numberOfNormalPoly
FFPOLY, 630
numberOfOperations
ESTOOLS, 534
numberOfPlacesOfDegree
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
numberOfPrimitivePoly
FFPOLY, 630
numberOfValuesNeeded
RECOP, 3012
numberOfVariables
RSDCMPK, 3029
SRDCMPK, 3110
numberPlacesDegExtDeg
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
numberRatPlacesExtDeg
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
NumberTheoreticPolynomialFunctions, 2544
NUMERIC, 2546
complexNumeric, 2546
complexNumericIfCan, 2546
numeric, 2546
numericIfCan, 2546
Numeric, 2546
numeric
NUMERIC, 2546
NumericalOrdinaryDifferentialEquations, 2555
NumericalQuadrature, 2563
NumericComplexEigenPackage, 2575
NumericContinuedFraction, 2577
numericIfCan
NUMERIC, 2546
NumericRealEigenPackage, 2579
NumericTubePlot, 2581
NUMFMT, 2539
FormatArabic, 2539
FormatRoman, 2539
ScanArabic, 2539
ScanFloatIgnoreSpaces, 2539
ScanFloatIgnoreSpacesIfCan, 2539
ScanRoman, 2539
NUMODE, 2555
rk4, 2555
rk4a, 2555
rk4f, 2555
rk4qc, 2555
NUMQUAD, 2563
aromberg, 2563
asimpson, 2563
atrapezoidal, 2563
romberg, 2563
rombergo, 2563
simpson, 2563
simpsono, 2563
trapezoidal, 2563
trapezoidalo, 2563
NUMTUBE, 2581
tube, 2581

oblateSpheroidal

COORDSYS, 224
ocf2ocdf
ESTOOLS, 534
OCTCT2, 2585
map, 2585
OctonionCategoryFunctions2, 2585
oddInfiniteProduct
INFPROD0, 983
INPRODFF, 985
INPRODPF, 988
STINPROD, 3136
oddintegers
STTAYLOR, 3138
oddlambert
STTAYLOR, 3138
ode
UTSODE, 3354
ode1
UTSODE, 3354
ode2
UTSODE, 3354
ODECONST, 221
constDsolve, 221
ODEEF, 432
solve, 432
ODEINT, 2587
diff, 2587
expint, 2587
int, 2587
ODEIntegration, 2587
ODEPACK, 69
measure, 70
solve, 70
ODEPAL, 2892
algDsolve, 2892
ODEPRIM, 2849
denomLODE, 2849
indicialEquation, 2849
indicialEquations, 2849
splitDenominator, 2849
ODEPRRIC, 2853
changeVar, 2853
constantCoefficientRicDE, 2853
denomRicDE, 2853
leadingCoefficientRicDE, 2853
polyRicDE, 2853
singRicDE, 2853
ODERAT, 2976
indicialEquationAtInfinity, 2976
ratDsolve, 2976
ODERED, 3025
reduceLODE, 3025
ODERTRIC, 2982
polyRicDE, 2982
ricDsolve, 2982
singRicDE, 2982
ODESYS, 3182
solve, 3182
solveInField, 3182
triangulate, 3182
ODETOOLS, 2589
particularSolution, 2589
variationOfParameters, 2589
wronskianMatrix, 2589
ODETools, 2589
OMEXPR, 556
OMwrite, 556
OMlistCDs
OMPKG, 2595
OMlistSymbols
OMPKG, 2595
OMPKG, 2594
OMlistCDs, 2595
OMlistSymbols, 2595
OMread, 2595
OMreadFile, 2595
OMreadStr, 2595
OMsupportsCD?, 2595
OMsupportsSymbol?, 2595
OMunhandledSymbol, 2595
OMread
OMPKG, 2595
OMreadFile
OMPKG, 2595
OMreadStr
OMPKG, 2595
OMreceive
OMSERVER, 2597
OMsend
OMSERVER, 2597
OMserve
OMSERVER, 2597

OMSERVER, 2597
  OMreceive, 2597
  OMsend, 2597
  OMserve, 2597
OMsupportsCD?
  OMPKG, 2595
OMsupportsSymbol?
  OMPKG, 2595
OMunhandledSymbol
  OMPKG, 2595
OMwrite
  OMEXPR, 556
ONECOMP2, 2593
  map, 2593
OneDimensionalArrayFunctions2, 2591
OnePointCompletionFunctions2, 2593
OpenMathPackage, 2594
OpenMathServerPackage, 2597
OperationsQuery, 2599
operator
  AF, 13
  COMBF, 175
  COMMONOP, 189
  EF, 403
  FSPECF, 699
  LF, 1296
OPQUERY, 2599
  getDatabase, 2599
optAttributes
  E04AGNT, 577
optimize
  OPTPACK, 61
option
  DROPT1, 361
optional
  PMASS, 2681
  PMASSFS, 712
OPTPACK, 61
  goodnessOfFit, 61
  measure, 61
  optimize, 61
ORDCOMP2, 2600
  map, 2600
order
  FORDER, 606
  PFO, 2740
  PFOQ, 2747
  RDIV, 3023
OrderedCompletionFunctions2, 2600
OrderingFunctions, 2601
ORDFUNS, 2601
  pureLex, 2601
  reverseLex, 2601
  totalLex, 2601
OREPCTO, 3349
  apply, 3349
  leftDivide, 3349
  monicLeftDivide, 3349
  monicRightDivide, 3349
  rightDivide, 3349
  times, 3349
OrthogonalPolynomialFunctions, 2603
orthonormalBasis
  REP, 2915
ORTHPOL, 2603
  chebyshevT, 2603
  chebyshevU, 2603
  hermiteH, 2603
  laguerreL, 2603
  legendreP, 2603
OUT, 2606
  output, 2606
  outputList, 2606
output
  OUT, 2606
outputArgs
  NORMPK, 2522
outputAsFortran
  FORT, 679
  SPECOUT, 3098
outputAsScript
  SPECOUT, 3098
outputAsTex
  SPECOUT, 3098
outputList
  OUT, 2606
outputMeasure
  ESTOOLS, 534
OutputPackage, 2606
overset?
  PLEQN, 2648

packageCall
INFORM1, 1045
PackageForAlgebraicFunctionField, 2611
PackageForAlgebraicFunctionFieldOverFiniteField, 2619
PackageForPoly, 2628
PADE, 2637
pade, 2637
padecf, 2637
pade
PADE, 2637
PADEPAC, 2635
PadeApproximantPackage, 2635
PadeApproximants, 2637
padecf
PADE, 2637
PADEPAC, 2635
pade, 2635
PAdicWildFunctionFieldIntegralBasis, 2641
PAFF, 2611
adjunctionDivisor, 2611
classNumber, 2611
desingTree, 2611
desingTreeWoFullParam, 2611
eval, 2611
evalIfCan, 2611
findOrderOfDivisor, 2611
fullDesTree, 2611
fullInfClsPt, 2611
genus, 2611
genusNeg, 2611
goppaCode, 2611
homogenize, 2611
interpolateForms, 2611
interpolateFormsForFact, 2611
intersectionDivisor, 2611
lBasis, 2611
LPolynomial, 2611
numberOfPlacesOfDegree, 2611
numberPlacesDegExtDeg, 2611
numberRatPlacesExtDeg, 2611
parametrize, 2611
placesAbove, 2611
placesOfDegree, 2611
pointDominateBy, 2611
projectivePoint, 2611
rationalPlaces, 2611
rationalPoints, 2611
setCurve, 2611
setSingularPoints, 2611
singularPoints, 2611
theCurve, 2611
ZetaFunction, 2611
PAFFFF, 2619
adjunctionDivisor, 2619
classNumber, 2619
desingTree, 2619
desingTreeWoFullParam, 2619
eval, 2619
evalIfCan, 2619
findOrderOfDivisor, 2619
fullDesTree, 2619
fullInfClsPt, 2619
genus, 2619
genusNeg, 2619
goppaCode, 2619
homogenize, 2619
interpolateForms, 2619
interpolateFormsForFact, 2619
intersectionDivisor, 2619
lBasis, 2619
LPolynomial, 2619
numberOfPlacesOfDegree, 2619
numberPlacesDegExtDeg, 2619
numberRatPlacesExtDeg, 2619
parametrize, 2619
placesAbove, 2619
placesOfDegree, 2619
pointDominateBy, 2619
projectivePoint, 2619
rationalPlaces, 2619
rationalPoints, 2619
setCurve, 2619
setSingularPoints, 2619
singularPoints, 2619
theCurve, 2619
translateToOrigin, 2619
ZetaFunction, 2619
palgextint
INTPAF, 2883
palgextint0
INTG0, 840

palginfieldint
INTALG, 21
palgint
INTPAF, 2883
palgint0
INTG0, 840
palgintegrate
INTALG, 21
palglimint
INTPAF, 2883
palglimint0
INTG0, 840
palgLODE
INTPAF, 2883
palgLODE0
INTG0, 840
palgRDE
INTPAF, 2883
palgRDE0
INTG0, 840
PAN2EXPR, 2759
coerce, 2759
parabolic
COORDSYS, 224
parabolicCylindrical
COORDSYS, 224
paraboloidal
COORDSYS, 224
ParadoxicalCombinatorsForStreams, 2646
ParametricLinearEquations, 2648
ParametricPlaneCurveFunctions2, 2662
ParametricSpaceCurveFunctions2, 2663
ParametricSurfaceFunctions2, 2664
ParametrizationPackage, 2666
parametrize
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
PARAMP, 2666
PARAMP, 2666
parametrize, 2666
ParCond
PLEQN, 2648
ParCondList
PLEQN, 2648
PARPC2, 2662
map, 2662
PARSC2, 2663
map, 2663
PARSU2, 2664
map, 2664
partialFraction
PFRPAC, 2670
PartialFractionPackage, 2670
particularSolution
LSMP, 1279
LSMP1, 1281
ODETOOLS, 2589
SOLVEFOR, 2819
partition
COMBINAT, 1053
partitions
PARTPERM, 2672
PartitionsAndPermutations, 2672
PARTPERM, 2672
conjugate, 2672
conjugates, 2672
partitions, 2672
permutations, 2672
sequences, 2672
shuffle, 2672
shufflein, 2672
pascalTriangle
GALUTIL, 787
PATMATCH, 2678
Is, 2678
is?, 2678
PATRES2, 2705
map, 2705
PATTERN1, 2675
addBadValue, 2675
badValues, 2675
predicate, 2675
satisfy?, 2675
suchThat, 2675
PATTERN2, 2677
map, 2677
PatternFunctions1, 2675
PatternFunctions2, 2677
PatternMatch, 2678
patternMatch
CPMATCH, 202

PMDOWN, 2701
PMFS, 2682
PMINS, 2685
PMKERNEL, 2694
PMLSAGG, 2697
PMPLCAT, 2699
PMQFCAT, 2704
PMSYM, 2707
PMTOOLS, 2708
PatternMatchAssertions, 2681
PatternMatchFunctionSpace, 2682
PatternMatchIntegerNumberSystem, 2684
PatternMatchIntegration, 2687
PatternMatchKernel, 2694
PatternMatchListAggregate, 2697
PatternMatchPolynomialCategory, 2698
PatternMatchPushDown, 2701
PatternMatchQuotientFieldCategory, 2704
PatternMatchResultFunctions2, 2705
PatternMatchSymbol, 2707
patternMatchTimes
PMTOOLS, 2708
PatternMatchTools, 2708
PCOMP, 2765
compose, 2765
PDECOMP, 2766
decompose, 2766
leftFactor, 2766
rightFactorCandidate, 2766
PDEPACK, 78
measure, 78
solve, 78
pdf2df
ESTOOLS, 534
pdf2ef
ESTOOLS, 534
perfectNthPower?
IROOT, 1099
perfectNthRoot
IROOT, 1099
perfectSqrt
IROOT, 1099
perfectSquare?
IROOT, 1099
PERMAN, 2714
permanent, 2714
Permanent, 2714
permanent
PERMAN, 2714
permutation
COMBF, 175
COMBINAT, 1053
PermutationGroupExamples, 2718
permutationRepresentation
REP1, 3046
permutations
PARTPERM, 2672
PFBR, 2768
bivariateSLPEBR, 2768
factorByRecursion, 2768
factorSFBRlcUnit, 2768
factorSquareFreeByRecursion, 2768
randomR, 2768
solveLinearPolynomialEquationByRecursion, 2768
PFBRU, 2775
factorByRecursion, 2775
factorSFBRlcUnit, 2775
factorSquareFreeByRecursion, 2775
randomR, 2775
solveLinearPolynomialEquationByRecursion, 2775
PFO, 2740
order, 2740
torsion?, 2740
torsionIfCan, 2740
PFOQ, 2747
order, 2747
torsion?, 2747
torsionIfCan, 2747
PFORP, 2628
constant, 2628
degOneCoef, 2628
degree, 2628
degreeOfMinimalForm, 2628
firstExponent, 2628
homogenize, 2628
listAllMono, 2628
listAllMonoExp, 2628
listVariable, 2628
mapExponents, 2628
minimalForm, 2628

monomials, 2628
replaceVarByOne, 2628
replaceVarByZero, 2628
subs1stVar, 2628
subs2ndVar, 2628
subsInVar, 2628
totalDegree, 2628
translate, 2628
univariate, 2628
PFOTOOLS, 2749
badNum, 2749
doubleDisc, 2749
getGoodPrime, 2749
mix, 2749
polyred, 2749
PFRPAC, 2670
partialFraction, 2670
PGCD, 2782
gcd, 2782
gcdPrimitive, 2782
PGE, 2718
abelianGroup, 2718
alternatingGroup, 2718
cyclicGroup, 2718
dihedralGroup, 2718
janko2, 2718
mathieu11, 2718
mathieu12, 2718
mathieu22, 2718
mathieu23, 2718
mathieu24, 2718
rubiksGroup, 2718
symmetricGroup, 2718
youngGroup, 2718
PGROEB, 2757
lexGroebner, 2757
totalGroebner, 2757
phiCoord
PTPACK, 2738
pi
EF, 403
PICOERCE, 2726
coerce, 2726
PiCoercions, 2726
PINTERP, 2790
interpolate, 2790
PINTERPA, 2792
LagrangeInterpolation, 2792
placesAbove
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
placesOfDegree
GPAFF, 801
INTDIVP, 1138
PAFF, 2611
PAFFFF, 2619
PLEQN, 2648
B1solve, 2648
bsolve, 2648
dmp2rfi, 2648
factorset, 2648
hasoln, 2648
inconsistent?, 2648
maxrank, 2648
minrank, 2648
minset, 2648
nextSublist, 2648
overset?, 2648
ParCond, 2648
ParCondList, 2648
pr2dmp, 2648
psolve, 2648
rdregime, 2648
redmat, 2648
redpps, 2648
regime, 2648
se2rfi, 2648
sqfree, 2648
wrregime, 2648
pleskenSplit
CRFP, 204
plot
PLOT1, 2728
PLOT1, 2728
plot, 2728
plotPolar, 2728
PlotFunctions1, 2728
plotPolar
PLOT1, 2728
PLOTTOOL, 2729
calcRanges, 2729

PlotTools, 2729
PLPKCRV, 2752
  eval, 2752
  minimalForm, 2752
  multiplicity, 2752
  pointInIdeal?, 2752
  translateToOrigin, 2752
plusInfinity
  INFINITY, 992
PMASS, 2681
  assert, 2681
  constant, 2681
  multiple, 2681
  optional, 2681
PMASSFS, 712
  assert, 712
  constant, 712
  multiple, 712
  optional, 712
pmComplexintegrate
  INTPM, 2687
PMDOWN, 2701
  fixPredicate, 2701
  patternMatch, 2701
PMFS, 2682
  patternMatch, 2682
PMINS, 2684
  patternMatch, 2685
pmintegrate
  INTPM, 2687
PMKERNEL, 2694
  patternMatch, 2694
PMLSAGG, 2697
  patternMatch, 2697
PMPLCAT, 2698
  patternMatch, 2699
PMPRED, 99
  suchThat, 99
PMPREDFS, 714
  suchThat, 714
PMQFCAT, 2704
  patternMatch, 2704
PMSYM, 2707
  patternMatch, 2707
PMTOOLS, 2708
  patternMatch, 2708
  patternMatchTimes, 2708
PNTHEORY, 2793
  bernoulli, 2793
  chebyshevT, 2793
  chebyshevU, 2793
  cyclotomic, 2793
  euler, 2793
  fixedDivisor, 2793
  hermite, 2793
  laguerre, 2793
  legendre, 2793
point
  TUBETOOL, 3308
pointColorDefault
  VIEWDEF, 3367
pointColorPalette
  DROPT0, 357
pointDominateBy
  GPAFF, 801
  LPARSPT, 1307
  PAFF, 2611
  PAFFFF, 2619
PointFunctions2, 2737
pointInIdeal?
  PLPKCRV, 2752
PointPackage, 2738
pointSizeDefault
  VIEWDEF, 3367
PointsOfFiniteOrder, 2740
PointsOfFiniteOrderRational, 2747
PointsOfFiniteOrderTools, 2749
pointToPlace
  LPARSPT, 1307
poisson
  RIDIST, 2941
pol
  INBFF, 1021
polar
  COORDSYS, 224
polCase
  LEADCDET, 1188
PolK
  FFSQFR, 657
PollardSmallFactor
  INTFACT, 1057
POLTOPOL, 2754

dmpToHdmp, 2755
dmpToP, 2755
hdmpToDmp, 2755
hdmpToP, 2755
pToDmp, 2755
pToHdmp, 2755
PolToPol, 2754
POLUTIL, 2993
boundOfCauchy, 2993
lazyVariations, 2993
sturmSequence, 2993
sturmVariationsOf, 2993
sylvesterSequence, 2993
POLY2, 2780
map, 2780
POLY2UP, 2828
univariate, 2828
POLYCATQ, 2762
isExpt, 2762
isPlus, 2762
isPower, 2762
isTimes, 2762
mainVariable, 2762
multivariate, 2762
univariate, 2762
variables, 2762
polygamma
DFSFUN, 309
FSPECF, 699
PolyGroebner, 2757
POLYLIFT, 2760
map, 2760
PolynomialAN2Expression, 2759
PolynomialCategoryLifting, 2760
PolynomialCategoryQuotientFunctions, 2762
PolynomialComposition, 2765
PolynomialDecomposition, 2766
PolynomialFactorizationByRecursion, 2768
PolynomialFactorizationByRecursionUnivariate, 2775
PolynomialFunctions2, 2780
PolynomialGcdPackage, 2782
PolynomialInterpolation, 2790
PolynomialInterpolationAlgorithms, 2792
PolynomialNumberTheoryFunctions, 2793
PolynomialPackageForCurve, 2752
PolynomialRoots, 2798
PolynomialSetUtilitiesPackage, 2801
PolynomialSolveByFormulas, 2819
PolynomialSquareFree, 2825
PolynomialToUnivariatePolynomial, 2828
polynomialZeros
ESCONT, 527
polyRDE
RDETR, 3273
polyred
PFOTOOLS, 2749
polyRicDE
ODEPRRIC, 2853
ODERTRIC, 2982
polyRing2UPUP
AFALGRES, 9
polyRingToBlUpRing
BLUPPACK, 146
POLYROOT, 2798
froot, 2798
nthr, 2798
qroot, 2798
rroot, 2798
popFortranOutputStack
FOP, 676
positiveSolve
ZDSOLVE, 3454
possiblyNewVariety?
PSETPK, 2802
power
STTAYLOR, 3138
powern
STTAYLOR, 3138
PowerSeriesLimitPackage, 2830
powerSum
CYCLES, 249
pr2dmp
PLEQN, 2648
PREASSOC, 2841
firstUncouplingMatrix, 2841
PrecomputedAssociatedEquations, 2841
predicate
PATTERN1, 2675
prepareDecompose
QCMPACK, 2900
SFQCMPK, 3101

prepareSubResAlgo
RSETGCD, 3035
prevPrime
PRIMES, 1088
PRIMARR2, 2844
map, 2844
reduce, 2844
scan, 2844
primaryDecomp
IDECOMP, 973
prime?
GAUSSFAC, 790
IDECOMP, 973
PRIMES, 1088
PRIMELT, 2846
primitiveElement, 2846
PRIMES, 1088
nextPrime, 1088
prevPrime, 1088
prime?, 1088
primes, 1088
primes
PRIMES, 1088
primextendedint
INTTR, 3254
primextintfrac
INTTR, 3254
primintegrate
INTTR, 3254
primintfldpoly
INTTR, 3254
primitive?
FFPOLY, 630
PrimitiveArrayFunctions2, 2844
PrimitiveElement, 2846
primitiveElement
FSPRMELT, 723
PRIMELT, 2846
primitivePart
SUBRESP, 3177
PrimitiveRatDE, 2849
PrimitiveRatRicDE, 2853
primlimintfrac
INTTR, 3254
primlimitedint
INTTR, 3254
prinb
GBINTERN, 871
prindINFO
GBINTERN, 871
prinpolINFO
GBINTERN, 871
prinshINFO
GBINTERN, 871
PRINT, 2859
print, 2859
print
PRINT, 2859
printInfo
GPAFF, 801
LPARSPT, 1307
RSDCMPK, 3029
SRDCMPK, 3110
printingInfo?
TBCMPPK, 3211
PrintPackage, 2859
PRJALGPK, 2733
algebraicSet, 2733
rationalPoints, 2733
singularPoints, 2733
singularPointsWithRestriction, 2733
probablyZeroDim?
PSETPK, 2802
problemPoints
D01AGNT, 363
ESCONT, 527
product
COMBF, 175
ProjectiveAlgebraicSetPackage, 2733
projectivePoint
PAFF, 2611
PAFFFF, 2619
prolateSpheroidal
COORDSYS, 224
PRS, 2864
?*?, 2864
chainSubResultants, 2864
degreeSubResultant, 2864
degreeSubResultantEuclidean, 2864
discriminant, 2864
discriminantEuclidean, 2864
divide, 2864

exquo, 2864
gcd, 2864
indiceSubResultant, 2864
indiceSubResultantEuclidean, 2864
lastSubResultant, 2864
lastSubResultantEuclidean, 2864
Lazard, 2864
Lazard2, 2864
nextsousResultant2, 2864
pseudoDivide, 2864
resultant, 2864
resultantEuclidean, 2864
resultantEuclideannaif, 2864
resultantnaif, 2864
resultantReduit, 2864
resultantReduitEuclidean, 2864
schema, 2864
semiDegreeSubResultantEuclidean, 2864
semiDiscriminantEuclidean, 2864
semiIndiceSubResultantEuclidean, 2864
semiLastSubResultantEuclidean, 2864
semiResultantEuclidean1, 2864
semiResultantEuclidean2, 2864
semiResultantEuclideannaif, 2864
semiResultantReduitEuclidean, 2864
semiSubResultantGcdEuclidean1, 2864
semiSubResultantGcdEuclidean2, 2864
subResultantGcd, 2864
subResultantGcdEuclidean, 2864
PSETPK, 2801
bivariate?, 2802
bivariatePolynomials, 2802
certainlySubVariety?, 2802
crushedSet, 2802
interReduce, 2802
irreducibleFactors, 2802
lazyIrreducibleFactors, 2802
linear?, 2802
linearPolynomials, 2802
possiblyNewVariety?, 2802
probablyZeroDim?, 2802
quasiMonicPolynomials, 2802
removeIrreducibleRedundantFactors, 2802
removeRedundantFactors, 2802
removeRedundantFactorsInContents, 2802
removeRedundantFactorsInPols, 2802
removeRoughlyRedundantFactorsInContents, 2802
removeRoughlyRedundantFactorsInPol, 2802
removeRoughlyRedundantFactorsInPols, 2802
removeSquaresIfCan, 2802
rewriteIdealWithQuasiMonicGenerators, 2802
rewriteSetByReducingWithParticularGenerators, 2802
roughBasicSet, 2802
selectAndPolynomials, 2802
selectOrPolynomials, 2802
selectPolynomials, 2802
squareFreeFactors, 2802
univariate?, 2802
univariatePolynomials, 2802
univariatePolynomialsGcds, 2802
unprotectedRemoveRedundantFactors, 2802
PSEUDLIN, 2860
changeBase, 2860
companionBlocks, 2860
normalForm, 2860
pseudoDivide
PRS, 2864
PseudoLinearNormalForm, 2860
PseudoRemainderSequence, 2864
psolve
PLEQN, 2648
PSQFR, 2825
squareFree, 2825
ptFunc
MESH, 1384
PTFUNC2, 2737
map, 2737
pToDmp
POLTOPOL, 2755
pToHdmp
POLTOPOL, 2755
PTPACK, 2738
color, 2738
hue, 2738
phiCoord, 2738
rCoord, 2738
shade, 2738
thetaCoord, 2738

xCoord, 2738
yCoord, 2738
zCoord, 2738
puiseux
EXPR2UPS, 562
GENUPS, 829
PureAlgebraicIntegration, 2883
PureAlgebraicLODE, 2892
pureLex
ORDFUNS, 2601
pushdown
MPRFF, 1405
pushdterm
MPRFF, 1405
pushFortranOutputStack
FOP, 676
pushucoef
MPRFF, 1405
pushuconst
MPRFF, 1405
pushup
MPRFF, 1405
PUSHVAR, 2893
PushVariables, 2893
PWFFINTB, 2641
integralBasis, 2641
localIntegralBasis, 2641
reducedDiscriminant, 2641

QALGSET2, 2897
radicalSimplify, 2897
QCMPACK, 2900
algebraicSort, 2900
branchIfCan, 2900
infRittWu?, 2900
internalInfRittWu?, 2900
internalSubPolSet?, 2900
internalSubQuasiComponent?, 2900
moreAlgebraic?, 2900
prepareDecompose, 2900
removeSuperfluousCases, 2900
removeSuperfluousQuasiComponents, 2900
startTable, 2900
subCase?, 2900
subPolSet?, 2900
subQuasiComponent?, 2900
subTriSet?, 2900
supDimElseRittWu?, 2900
qfactor
FSUPFACT, 758
QFCAT2, 2909
map, 2909
qPot
INBFF, 1021
qqq
WEIER, 3375
qroot
POLYROOT, 2798
qShiftAction
FFFG, 683
qShiftC
FFFG, 683
quadratic
SOLVEFOR, 2819
quadratic?
E04AGNT, 577
quadraticBezier
BEZIER, 139
quadraticNorm
GALFACTU, 763
quadTransform
BLUPPACK, 146
quartic
SOLVEFOR, 2819
QuasiAlgebraicSet2, 2897
QuasiComponentPackage, 2900
quasiMonicPolynomials
PSETPK, 2802
QUATCT2, 2912
map, 2912
QuaternionCategoryFunctions2, 2912
quickSort
FLASORT, 662
QuotientFieldCategoryFunctions2, 2909
quotVecSpaceBasis
LOP, 1293

radical
IDECOMP, 973
RadicalEigenPackage, 2915
radicalEigenvalues
REP, 2915

radicalEigenvectors
REP, 2915
radicalOfLeftTraceForm
ALGPKG, 37
radicalRoots
SOLVERAD, 2929
radicalSimplify
QALGSET2, 2897
radicalSolve
SOLVERAD, 2929
RadicalSolvePackage, 2929
radix
RADUTIL, 2936
RadixUtilities, 2936
radPoly
CHVAR, 159
RADUTIL, 2936
radix, 2936
raisePolynomial
FACUTIL, 593
ran
FACUTIL, 593
randnum
RANDSRC, 2943
random
FFPOLY, 630
INBFF, 1021
RandomDistributions, 2937
RandomFloatDistributions, 2939
RandomIntegerDistributions, 2941
RandomNumberSource, 2943
randomR
GENPGCD, 815
PFBR, 2768
PFBRU, 2775
RANDSRC, 2943
randnum, 2943
reseed, 2943
seed, 2943
size, 2943
rangeIsFinite
D01AGNT, 363
rangePascalTriangle
GALUTIL, 787
ranges
DROPT0, 357
rank
LSMP, 1279
LSMP1, 1281
MATLIN, 1375
ratDenom
ALGMANIP, 30
ratDsolve
ODERAT, 2976
RATFACT, 2945
factor, 2945
factorSquareFree, 2945
rational
RATRET, 2981
rational?
RATRET, 2981
RationalFactorize, 2945
RationalFunction, 2947
RationalFunctionDefiniteIntegration, 2949
RationalFunctionFactor, 2952
RationalFunctionFactorizer, 2953
RationalFunctionIntegration, 2955
RationalFunctionLimitPackage, 2957
RationalFunctionSign, 2960
RationalFunctionSum, 2968
rationalIfCan
RATRET, 2981
RationalIntegration, 2970
RationalInterpolation, 2973
RationalLODE, 2976
rationalPlaces
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
rationalPoints
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
PRJALGPK, 2733
RationalRetractions, 2981
RationalRicDE, 2982
RationalUnivariateRepresentationPackage, 2989
ratPoly
ALGMANIP, 30
RATRET, 2981
rational, 2981
rational?, 2981

rationalIfCan, 2981
rCoord
PTPACK, 2738
RDEEF, 474
rischDE, 474
RDEEFS, 482
rischDEsys, 482
RDETR, 3273
baseRDE, 3273
monomRDE, 3273
polyRDE, 3273
RDETRS, 3277
baseRDEsys, 3277
monomRDEsys, 3277
rdHack1
RDIST, 2937
RDIST, 2937
rdHack1, 2937
uniform, 2937
weighted, 2937
RDIV, 3023
order, 3023
rdregime
PLEQN, 2648
real
CTRIGMNP, 218
TRIGMNIP, 3304
REAL0, 3001
midpoint, 3001
midpoints, 3001
realZeros, 3001
refine, 3001
REAL0Q, 3008
realZeros, 3008
refine, 3008
real?
TRIGMNIP, 3304
realEigenvalues
NREP, 2579
realEigenvectors
NREP, 2579
realElementary
EFSTRUC, 442
RealPolynomialUtilitiesPackage, 2993
realRoots
FLOATRP, 670
REALSOLV, 2999
realSolve, 2999
solve, 2999
realSolve
REALSOLV, 2999
ZDSOLVE, 3454
RealSolvePackage, 2999
RealZeroPackage, 3001
RealZeroPackageQ, 3008
realZeros
REAL0, 3001
REAL0Q, 3008
recip
NORMPK, 2522
STTAYLOR, 3138
reciprocalPolynomial
CRFP, 204
recolor
DRAWCFUN, 3235
RECOP, 3012
evalADE, 3012
evalRec, 3012
getEq, 3012
getOp, 3012
getShiftRec, 3012
numberOfValuesNeeded, 3012
shiftInfoRec, 3012
RectangularMatrixCategoryFunctions2, 3010
recur
MAPHACK1, 1328
MAPPKG1, 1341
RecurrenceOperator, 3012
redmat
PLEQN, 2648
REDORDER, 3027
ReduceOrder, 3027
redPo
GBINTERN, 871
redPol
GBINTERN, 871
redpps
PLEQN, 2648
reduce
ARRAY12, 2591
DEGRED, 265
DIRPROD2, 281

FLAGG2, 659
FSAGG2, 665
LIST2, 1301
MATCAT2, 1371
PRIMARR2, 2844
RMCAT2, 3010
STREAM2, 3132
VECTOR2, 3365
reducedDiscriminant
PWFFINTB, 2641
ReducedDivisor, 3023
reducedQPowers
FFPOLY, 630
reduceLineOverLine
LOP, 1293
ReduceLODE, 3025
reduceLODE
ODERED, 3025
ReduceOrder
REDORDER, 3027
reduceRow
LOP, 1293
reduceRowOnList
LOP, 1293
reduction
GENEEZ, 834
GHENSEL, 794
INMODGCD, 1005
ReductionOfOrder, 3027
refine
FRUTIL, 591
REAL0, 3001
REAL0Q, 3008
regime
PLEQN, 2648
RegularSetDecompositionPackage, 3029
RegularTriangularSetGcdPackage, 3035
removeConstantTerm
INTTOOLS, 1120
removeCoshSq
TRMANIP, 3264
removeCosSq
TRMANIP, 3264
removeIrreducibleRedundantFactors
PSETPK, 2802
removeRedundantFactors
PSETPK, 2802
removeRedundantFactorsInContents
PSETPK, 2802
removeRedundantFactorsInPols
PSETPK, 2802
removeRoughlyRedundantFactorsInContents
PSETPK, 2802
removeRoughlyRedundantFactorsInPol
PSETPK, 2802
removeRoughlyRedundantFactorsInPols
PSETPK, 2802
removeSinhSq
TRMANIP, 3264
removeSinSq
TRMANIP, 3264
removeSquaresIfCan
PSETPK, 2802
removeSuperfluousCases
QCMPACK, 2900
SFQCMPK, 3101
removeSuperfluousQuasiComponents
QCMPACK, 2900
SFQCMPK, 3101
REP, 2915
eigenMatrix, 2915
gramschmidt, 2915
normalise, 2915
orthonormalBasis, 2915
radicalEigenvalues, 2915
radicalEigenvectors, 2915
REP1, 3046
antisymmetricTensors, 3046
createGenericMatrix, 3046
permutationRepresentation, 3046
symmetricTensors, 3046
tensorProduct, 3046
REP2, 3053
areEquivalent?, 3053
completeEchelonBasis, 3053
createRandomElement, 3053
cyclicSubmodule, 3053
isAbsolutelyIrreducible?, 3053
meatAxe, 3053
scanOneDimSubspaces, 3053
split, 3053
standardBasisOfCyclicSubmodule, 3053

REPDB, 3043
  double, 3043
RepeatedDoubling, 3043
RepeatedSquaring, 3045
replaceDiffs
  EXPRSOL, 546
replaceVarByOne
  PFORP, 2628
replaceVarByZero
  PFORP, 2628
RepresentationPackage1, 3046
RepresentationPackage2, 3053
REPSQ, 3045
  expt, 3045
repSq
  INBFF, 1021
reseed
  RANDSRC, 2943
reset
  GPAFF, 801
resetVariableOrder
  UDVO, 3362
reshape
  CARTEN2, 157
  MPC2, 1409
RESLATC, 3070
  coerce, 3070
ResolveLatticeCompletion, 3070
restorePrecision
  NAGSP, 1739
resultant
  PRS, 2864
resultantEuclidean
  PRS, 2864
resultantEuclideannaif
  PRS, 2864
resultantnaif
  PRS, 2864
resultantReduit
  PRS, 2864
resultantReduitEuclidean
  PRS, 2864
retract
  ANY1, 84
retractable?
  ANY1, 84
retractIfCan
  ANY1, 84
  NORMRETR, 2528
RetractSolvePackage, 3071
RETSOL, 3071
  solveRetract, 3071
reverse
  GALPOLYU, 784
reverseLex
  ORDFUNS, 2601
revert
  STTAYLOR, 3138
rewriteIdealWithQuasiMonicGenerators
  PSETPK, 2802
rewriteSetByReducingWithParticularGenerators
  PSETPK, 2802
RF, 2947
  coerce, 2947
  eval, 2947
  mainVariable, 2947
  multivariate, 2947
  univariate, 2947
  variables, 2947
RF2UTS
  UTSODETL, 3363
RFDIST, 2939
  Beta, 2939
  chiSquare, 2939
  chiSquare1, 2939
  exponential, 2939
  exponential1, 2939
  F, 2939
  normal, 2939
  normal01, 2939
  t, 2939
  uniform, 2939
  uniform01, 2939
RFFACT, 2952
  factor, 2952
RFFACTOR, 2953
  factor, 2953
RFP, 3074
  distinguishedCommonRootsOf, 3074
  distinguishedRootsOf, 3074
  foundZeroes, 3074
  setFoundZeroes, 3074

ricDsolve
ODERTRIC, 2982
ridHack1
RIDIST, 2941
RIDIST, 2941
binomial, 2941
geometric, 2941
poisson, 2941
ridHack1, 2941
uniform, 2941
rightDivide
OREPCTO, 3349
rightFactorCandidate
PDECOMP, 2766
rightFactorIfCan
UPDECOMP, 3336
rightRank
ALGPKG, 37
RINTERP, 2973
interpolate, 2973
rischDE
RDEEF, 474
rischDEsys
RDEEFS, 482
rischNormalize
EFSTRUC, 442
rk4
NUMODE, 2555
rk4a
NUMODE, 2555
rk4f
NUMODE, 2555
rk4qc
NUMODE, 2555
RMCAT2, 3010
map, 3010
reduce, 3010
romberg
NUMQUAD, 2563
rombergo
NUMQUAD, 2563
rootBound
GALFACTU, 763
rootKerSimp
ALGMANIP, 30
rootNormalize
EFSTRUC, 442
rootOf
AF, 13
rootOfIrreduciblePoly
FFPOLY2, 651
rootPoly
CHVAR, 159
rootPower
ALGMANIP, 30
rootProduct
ALGMANIP, 30
rootRadius
CRFP, 204
RootsFindingPackage, 3074
rootSimp
ALGMANIP, 30
rootSplit
ALGMANIP, 30
roughBasicSet
PSETPK, 2802
rowEch
MHROWRED, 1393
rowEchelon
IMATQF, 1003
MATLIN, 1375
MHROWRED, 1393
rowEchelonLocal
MHROWRED, 1393
rowEchLocal
MHROWRED, 1393
rowEchWoZeroLines
LOP, 1293
rowEchWoZeroLinesWOVectorise
LOP, 1293
rroot
POLYROOT, 2798
RSDCMPK, 3029
algebraicDecompose, 3029
convert, 3029
decompose, 3029
internalDecompose, 3029
KrullNumber, 3029
numberOfVariables, 3029
printInfo, 3029
transcendentalDecompose, 3029
upDateBranches, 3029

RSETGCD, 3035
integralLastSubResultant, 3035
prepareSubResAlgo, 3035
startTableGcd, 3035
startTableInvSet, 3035
stopTableGcd, 3035
stopTableInvSet, 3035
toseInvertible?, 3035
toseInvertibleSet, 3035
toseLastSubResultant, 3035
toseSquareFreePart, 3035
rubiksGroup
PGE, 2718
rur
IRURPK, 1125
RURPK, 2989
RURPK, 2989
rur, 2989

s01eaf
NAGS, 2486
s13aaf
NAGS, 2486
s13acf
NAGS, 2486
s13adf
NAGS, 2486
s14aaf
NAGS, 2486
s14abf
NAGS, 2486
s14baf
NAGS, 2486
s15adf
NAGS, 2486
s15aef
NAGS, 2486
s17acf
NAGS, 2486
s17adf
NAGS, 2486
s17aef
NAGS, 2486
s17aff
NAGS, 2486
s17agf
NAGS, 2486
s17ahf
NAGS, 2486
s17ajf
NAGS, 2486
s17akf
NAGS, 2486
s17dcf
NAGS, 2486
s17def
NAGS, 2486
s17dgf
NAGS, 2486
s17dhf
NAGS, 2486
s17dlf
NAGS, 2486
s18acf
NAGS, 2486
s18adf
NAGS, 2486
s18aef
NAGS, 2486
s18aff
NAGS, 2486
s18dcf
NAGS, 2486
s18def
NAGS, 2486
s19aaf
NAGS, 2486
s19abf
NAGS, 2486
s19acf
NAGS, 2486
s19adf
NAGS, 2486
s20acf
NAGS, 2486
s20adf
NAGS, 2486
s21baf
NAGS, 2486
s21bbf
NAGS, 2486
s21bcf

NAGS, 2486
s21bdf
NAGS, 2486
SAEFACT, 3085
factor, 3085
SAERationalFunctionAlgFactor, 3079
SAERFFC, 3079
factor, 3079
safeCeiling
GALUTIL, 787
safeFloor
GALUTIL, 787
safetyMargin
GALUTIL, 787
satisfy?
PATTERN1, 2675
say
DISPLAY, 285
sayLength
DISPLAY, 285
SCACHE, 3092
cache, 3092
clearCache, 3092
enterInCache, 3092
scaleRoots
GALPOLYU, 784
scan
ARRAY12, 2591
DIRPROD2, 281
FLAGG2, 659
FSAGG2, 665
LIST2, 1301
PRIMARR2, 2844
STREAM2, 3132
VECTOR2, 3365
ScanArabic
NUMFMT, 2539
ScanFloatIgnoreSpaces
NUMFMT, 2539
ScanFloatIgnoreSpacesIfCan
NUMFMT, 2539
scanOneDimSubspaces
REP2, 3053
ScanRoman
NUMFMT, 2539
schema
PRS, 2864
schwerpunkt
CRFP, 204
SCPKG, 3165
coordinates, 3165
structuralConstants, 3165
screenResolution
GRDEF, 854
ScriptFormulaFormat1, 3080
sdf2lst
D01AGNT, 363
ESCONT, 527
ESTOOLS, 534
se2rfi
PLEQN, 2648
sec
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
sec2cos
TRMANIP, 3264
sech
EF, 403
EFULS, 451
EFUPXS, 459
sech2cosh
TRMANIP, 3264
sechIfCan
EFULS, 451
EFUPXS, 459
secIfCan
EFULS, 451
EFUPXS, 459
seed
RANDSRC, 2943
SEG2, 3083
map, 3083
SEGBIND2, 3082
map, 3082
SegmentBindingFunctions2, 3082
SegmentFunctions2, 3083
selectAndPolynomials
PSETPK, 2802
selectOrPolynomials

PSETPK, 2802
selectPolynomials
PSETPK, 2802
semiDegreeSubResultantEuclidean
PRS, 2864
semiDiscriminantEuclidean
PRS, 2864
semiIndiceSubResultantEuclidean
PRS, 2864
semiLastSubResultantEuclidean
PRS, 2864
semiResultantEuclidean1
PRS, 2864
semiResultantEuclidean2
PRS, 2864
semiResultantEuclideannaif
PRS, 2864
semiResultantReduitEuclidean
PRS, 2864
semiSubResultantGcdEuclidean1
PRS, 2864
semiSubResultantGcdEuclidean2
PRS, 2864
separateDegrees
DDFACT, 288
separateFactors
DDFACT, 288
MDDFACT, 1387
sequences
PARTPERM, 2672
series
EXPR2UPS, 562
GENUPS, 829
seriesSolve
EXPRODE, 551
EXPRSOL, 546
UTSSOL, 3216
setClipValue
DRAWCX, 352
setCurve
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
setErrorBound
CRFP, 204
setFieldInfo
INBFF, 1021
setFoundZeroes
RFP, 3074
setImagSteps
DRAWCX, 352
setLegalFortranSourceExtensions
FORT, 679
setOrder
UDPO, 3359
setRealSteps
DRAWCX, 352
setSingularPoints
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
setVariableOrder
UDVO, 3362
SFQCMPK, 3100
algebraicSort, 3101
branchIfCan, 3101
infRittWu?, 3101
internalInfRittWu?, 3101
internalSubPolSet?, 3101
internalSubQuasiComponent?, 3101
moreAlgebraic?, 3101
prepareDecompose, 3101
removeSuperfluousCases, 3101
removeSuperfluousQuasiComponents, 3101
startTable, 3101
stopTable, 3101
subCase?, 3101
subPolSet?, 3101
subQuasiComponent?, 3101
subTriSet?, 3101
supDimElseRittWu?, 3101
SFRGCD, 3116
startTableGcd, 3116
startTableInvSet, 3116
stopTableGcd, 3116
stopTableInvSet, 3116
stoseIntegralLastSubResultant, 3116
stoseInternalLastSubResultant, 3116
stoseInvertible?, 3116
stoseInvertible?reg, 3116
stoseInvertible?sqfreg, 3116
stoseInvertibleSet, 3116

stoseInvertibleSetreg, 3116
stoseInvertibleSetsqfreg, 3116
stoseLastSubResultant, 3116
stosePrepareSubResAlgo, 3116
stoseSquareFreePart, 3116
SFunction
CYCLES, 249
SGCF, 3193
coleman, 3194
inverseColeman, 3194
listYoungTableaus, 3194
makeYoungTableau, 3194
nextColeman, 3194
nextLatticePermutation, 3194
nextPartition, 3194
numberOfImproperPartitions, 3194
subSet, 3194
unrankImproperPartitions0, 3194
unrankImproperPartitions1, 3194
shade
PTPACK, 2738
shanksDiscLogAlgorithm
DLP, 283
shellSort
FLASORT, 662
ShiftAction
FFFG, 683
ShiftC
FFFG, 683
shiftHP
GUESS, 916
GUESSAN, 951
GUESSF, 952
GUESSINT, 954
GUESSP, 955
GUESSUP, 956
shiftInfoRec
RECOP, 3012
shiftRoots
GALPOLYU, 784
showFortranOutputStack
FOP, 676
SHP, 3169
countRealRoots, 3169
countRealRootsMultiple, 3169
SturmHabicht, 3169
SturmHabichtCoefficients, 3169
SturmHabichtMultiple, 3169
SturmHabichtSequence, 3169
subresultantSequence, 3169
shuffle
PARTPERM, 2672
shufflein
PARTPERM, 2672
Si
LF, 1296
sign
SIGNEF, 438
SIGNRF, 2961
TOOLSIGN, 3222
signAround
INPSIGN, 1038
SIGNEF, 438
sign, 438
SIGNRF, 2960
sign, 2961
SIMPAN, 3086
simplify, 3086
SimpleAlgebraicExtensionAlgFactor, 3085
simpleBounds?
E04AGNT, 577
simplify
SIMPAN, 3086
TRMANIP, 3264
SimplifyAlgebraicNumberConvertPackage, 3086
simplifyExp
TRMANIP, 3264
simplifyLog
TRMANIP, 3264
simpson
NUMQUAD, 2563
simpsono
NUMQUAD, 2563
sin
EF, 403
EFULS, 451
EFUPXS, 459
STTF, 3150
STTFNC, 3160
sin2csc
TRMANIP, 3264
sincos

STTF, 3150
singleFactorBound
GALFACTU, 763
singRicDE
ODEPRRIC, 2853
ODERTRIC, 2982
singularitiesOf
D01AGNT, 363
ESCONT, 527
singularPoints
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
PRJALGPK, 2733
singularPointsWithRestriction
PRJALGPK, 2733
sinh
EF, 403
EFULS, 451
EFUPXS, 459
sinh2csch
TRMANIP, 3264
sinhcosh
STTF, 3150
sinhIfCan
EFULS, 451
EFUPXS, 459
sinIfCan
EFULS, 451
EFUPXS, 459
size
RANDSRC, 2943
sizeMultiplication
FFF, 617
sizePascalTriangle
GALUTIL, 787
skewSFunction
CYCLES, 249
slex
TABLBUMP, 3207
slope
NPOLYGON, 2506
SMITH, 3087
completeHermite, 3087
completeSmith, 3087
diophantineSystem, 3087
hermite, 3087
smith, 3087
smith
SMITH, 3087
SmithNormalForm, 3087
sn
ELFUTS, 485
sncndn
ELFUTS, 485
socf2socdf
ESTOOLS, 534
solve
FLOATRP, 670
LODEEF, 426
LSMP, 1279
LSMP1, 1281
NLINSOL, 2519
NODE1, 2515
ODEEF, 432
ODEPACK, 70
ODESYS, 3182
PDEPACK, 78
REALSOLV, 2999
SOLVEFOR, 2819
SOLVETRA, 3288
SYSSOLP, 3188
solve1
INEP, 1028
SOLVEFOR, 2819
aCubic, 2819
aLinear, 2819
aQuadratic, 2819
aQuartic, 2819
cubic, 2819
linear, 2819
mapSolve, 2819
particularSolution, 2819
quadratic, 2819
quartic, 2819
solve, 2819
solveid
GENEEZ, 834
solveInField
NLINSOL, 2519
ODESYS, 3182
solveLinear

LINDEP, 1265
solveLinearlyOverQ
ZLINDEP, 1067
solveLinearPolynomialEquation
CINTSLPE, 199
FFSLPE, 654
INTSLPE, 1103
solveLinearPolynomialEquationByFractions
LPEFRAC, 1274
solveLinearPolynomialEquationByRecursion
PFBR, 2768
PFBRU, 2775
SOLVERAD, 2929
contractSolve, 2929
radicalRoots, 2929
radicalSolve, 2929
solveRetract
RETSOL, 3071
SOLVESER, 3299
decomposeFunc, 3299
unvectorise, 3299
SOLVETRA, 3288
solve, 3288
sortConstraints
E04AGNT, 577
SortedCache, 3092
SortPackage, 3095
SORTPAK, 3095
bubbleSort, 3095
insertionSort, 3095
space
DROPT0, 357
SparseUnivariatePolynomialFunctions2, 3097
sparsityIF
D02AGNT, 375
SpecialOutputPackage, 3098
specialTrigs
EF, 403
SPECOUT, 3098
outputAsFortran, 3098
outputAsScript, 3098
outputAsTex, 3098
spherical
COORDSYS, 224
split
ALGFACT, 47
IR2F, 1115
IRRF2F, 1113
MONOTOOL, 1400
REP2, 3053
splitConstant
INTPM, 2687
splitDenominator
COMBF, 187
ICDEN, 996
MCDEN, 1373
ODEPRIM, 2849
UPCDEN, 3334
splitLinear
E04AGNT, 577
splitSquarefree
MONOTOOL, 1400
sPol
GBINTERN, 871
sqfree
PLEQN, 2648
squareFree
INTFACT, 1057
MULTSQFR, 1436
PSQFR, 2825
SUPFRACF, 3180
UPSQFREE, 3345
ZDSOLVE, 3454
squareFreeFactors
PSETPK, 2802
squareFreeLexTriangular
LEXTRIPK, 1260
squareFreePart
UPSQFREE, 3345
squareFreePrim
MULTSQFR, 1436
SquareFreeQuasiComponentPackage, 3100
SquareFreeRegularSetDecompositionPackage, 3109
SquareFreeRegularTriangularSetGcdPackage, 3116
SRDCMPK, 3109
algebraicDecompose, 3110
convert, 3110
decompose, 3110
internalDecompose, 3110
KrullNumber, 3110

numberOfVariables, 3110
printInfo, 3110
transcendentalDecompose, 3110
upDateBranches, 3110
standardBasisOfCyclicSubmodule
REP2, 3053
startPolynomial
CRFP, 204
stepBlowUp
BLUPPACK, 146
stFunc1
UTSODE, 3354
stFunc2
UTSODE, 3354
stFuncN
UTSODE, 3354
stiffnessAndStabilityFactor
D02AGNT, 375
stiffnessAndStabilityOfODEIF
D02AGNT, 375
STINPROD, 3136
evenInfiniteProduct, 3136
generalInfiniteProduct, 3136
infiniteProduct, 3136
oddInfiniteProduct, 3136
stirling1
COMBINAT, 1053
stirling2
COMBINAT, 1053
STNSR, 3149
tensorMap, 3149
stopMusserTrials
GALFACT, 767
StorageEfficientMatrixOperations, 3126
stoseIntegralLastSubResultant
SFRGCD, 3116
stoseInternalLastSubResultant
SFRGCD, 3116
stoseInvertible?
SFRGCD, 3116
stoseInvertible?reg
SFRGCD, 3116
stoseInvertible?sqfreg
SFRGCD, 3116
stoseInvertibleSet
SFRGCD, 3116
stoseInvertibleSetreg
SFRGCD, 3116
stoseInvertibleSetsqfreg
SFRGCD, 3116
stoseLastSubResultant
SFRGCD, 3116
stosePrepareSubResAlgo
SFRGCD, 3116
stoseSquareFreePart
SFRGCD, 3116
STREAM1, 3130
concat, 3130
STREAM2, 3132
map, 3132
reduce, 3132
scan, 3132
STREAM3, 3134
map, 3134
StreamFunctions1, 3130
StreamFunctions2, 3132
StreamFunctions3, 3134
StreamInfiniteProduct, 3136
StreamTaylorSeriesOperations, 3138
StreamTensor, 3149
StreamTranscendentalFunctions, 3150
StreamTranscendentalFunctionsNonCommutative, 3160
stripCommentsAndBlanks
TEMUTL, 3219
structuralConstants
SCPKG, 3165
StructuralConstantsPackage, 3165
sts2stst
WEIER, 3375
STTAYLOR, 3138
-?, 3138
?*?, 3138
?+?, 3138
?-?, 3138
?/?, 3138
addiag, 3138
coerce, 3138
compose, 3138
deriv, 3138
eval, 3138
evenlambert, 3138

exquo, 3138
gderiv, 3138
generalLambert, 3138
int, 3138
integers, 3138
integrate, 3138
invmultisect, 3138
lagrange, 3138
lambert, 3138
lazyGintegrate, 3138
lazyIntegrate, 3138
mapdiv, 3138
mapmult, 3138
monom, 3138
multisect, 3138
nlde, 3138
oddintegers, 3138
oddlambert, 3138
power, 3138
powern, 3138
recip, 3138
revert, 3138
STTF, 3150
?**?, 3150
acos, 3150
acot, 3150
acsc, 3150
asec, 3150
asin, 3150
atan, 3150
cos, 3150
cot, 3150
csc, 3150
exp, 3150
sec, 3150
sin, 3150
sincos, 3150
sinhcosh, 3150
tan, 3150
STTFNC, 3160
?**?, 3160
acos, 3160
acot, 3160
acsc, 3160
asec, 3160
asin, 3160
atan, 3160
cos, 3160
cot, 3160
csc, 3160
exp, 3160
sec, 3160
sin, 3160
tan, 3160
SturmHabicht
SHP, 3169
SturmHabichtCoefficients
SHP, 3169
SturmHabichtMultiple
SHP, 3169
SturmHabichtPackage, 3169
SturmHabichtSequence
SHP, 3169
sturmSequence
POLUTIL, 2993
sturmVariationsOf
POLUTIL, 2993
style
DROPT0, 357
subCase?
QCMPACK, 2900
SFQCMPK, 3101
subPolSet?
QCMPACK, 2900
SFQCMPK, 3101
subQuasiComponent?
QCMPACK, 2900
SFQCMPK, 3101
SUBRESP, 3177
primitivePart, 3177
subresultantVector, 3177
subResultantGcd
PRS, 2864
subResultantGcdEuclidean
PRS, 2864
SubResultantPackage, 3177
subresultantSequence
SHP, 3169
subresultantVector
SUBRESP, 3177
subs1stVar
PFORP, 2628

subs2ndVar
PFORP, 2628
subscriptedVariables
D03AGNT, 381
subSet
SGCF, 3194
subsInVar
PFORP, 2628
subTriSet?
QCMPACK, 2900
SFQCMPK, 3101
suchThat
PATTERN1, 2675
PMPRED, 99
PMPREDFS, 714
sum
ISUMP, 1039
SUMFS, 728
SUMRF, 2968
SUMFS, 728
sum, 728
summation
COMBF, 175
sumOfDivisors
INTHEORY, 1082
sumOfKthPowerDivisors
INTHEORY, 1082
sumOfSquares
E04AGNT, 577
SUMRF, 2968
sum, 2968
sumSquares
GAUSSFAC, 790
SUP2, 3097
map, 3097
supDimElseRittWu?
QCMPACK, 2900
SFQCMPK, 3101
SUPFRACF, 3180
factor, 3180
squareFree, 3180
SupFractionFactorizer, 3180
swap
COMMUPC, 194
sylvesterMatrix
BEZOUT, 141
sylvesterSequence
POLUTIL, 2993
SYMFUNC, 3204
symFunc, 3204
symFunc
SYMFUNC, 3204
SymmetricFunctions, 3204
symmetricGroup
PGE, 2718
SymmetricGroupCombinatoricFunctions, 3193
symmetricPower
LODOOPS, 1271
symmetricProduct
LODOOPS, 1271
symmetricTensors
REP1, 3046
SYSSOLP, 3188
solve, 3188
triangularSystems, 3188
systemCommand
MSYSCMD, 1402
SystemODESolver, 3182
systemSizeIF
D02AGNT, 375
SystemSolvePackage, 3188

t
RFDIST, 2939
tab
TABLBUMP, 3207
tab1
TABLBUMP, 3207
TABLBUMP, 3207
bat, 3207
bat1, 3207
bumprow, 3207
bumptab, 3207
bumptab1, 3207
inverse, 3207
lex, 3207
maxrow, 3207
mr, 3207
slex, 3207
tab, 3207
tab1, 3207
untab, 3207

TableauxBumpers, 3207
tablePow
    GENEEZ, 834
TabulatedComputationPackage, 3210
tan
    EF, 403
    EFULS, 451
    EFUPXS, 459
    STTF, 3150
    STTFNC, 3160
tan2cot
    TRMANIP, 3264
tan2trig
    TRMANIP, 3264
tanAn
    TANEXP, 3214
TANEXP, 3214
    tanAn, 3214
    tanNa, 3214
    tanSum, 3214
TangentExpansions, 3214
tanh
    EF, 403
    EFULS, 451
    EFUPXS, 459
tanh2coth
    TRMANIP, 3264
tanh2trigh
    TRMANIP, 3264
tanhIfCan
    EFULS, 451
    EFUPXS, 459
tanIfCan
    EFULS, 451
    EFUPXS, 459
tanintegrate
    INTTR, 3254
tanNa
    TANEXP, 3214
tanQ
    EFSTRUC, 442
tanSum
    TANEXP, 3214
taylor
    EXPR2UPS, 562
    GENUPS, 829
TaylorSolve, 3216
TBCMPPK, 3210
    clearTable, 3211
    extractIfCan, 3211
    initTable, 3211
    insert, 3211
    makingStats?, 3211
    printInfo, 3211
    printingInfo?, 3211
    printStats, 3211
    startStats, 3211
    usingTable?, 3211
TemplateUtilities, 3219
TEMUTL, 3219
    interpretString, 3219
    stripCommentsAndBlanks, 3219
tensorMap
    STNSR, 3149
tensorProduct
    REP1, 3046
testDim
    GROEBSOL, 911
testModulus
    GENEEZ, 834
TEX1, 3221
    coerce, 3221
TexFormat1, 3221
theCurve
    GPAFF, 801
    PAFF, 2611
    PAFFFF, 2619
thetaCoord
    PTPACK, 2738
times
    OREPCTO, 3349
title
    DROPT0, 357
ToolsForSign, 3222
TOOLSIGN, 3222
    direction, 3222
    nonQsign, 3222
    sign, 3222
topFortranOutputStack
    FOP, 676
TopLevelDrawFunctions, 3224
TopLevelDrawFunctionsForAlgebraicCurves, 3231

TopLevelDrawFunctionsForCompiledFunctions, 3235
TopLevelDrawFunctionsForPoints, 3248
TopLevelThreeSpace, 3251
TOPSP, 3251
  createThreeSpace, 3251
toroidal
  COORDSYS, 224
torsion?
  PFO, 2740
  PFOQ, 2747
torsionIfCan
  PFO, 2740
  PFOQ, 2747
toScale
  DROPT0, 357
toseInvertible?
  RSETGCD, 3035
toseInvertibleSet
  RSETGCD, 3035
toseLastSubResultant
  RSETGCD, 3035
toseSquareFreePart
  RSETGCD, 3035
totalDegree
  PFORP, 2628
totalfract
  MPRFF, 1405
totalGroebner
  PGROEB, 2757
totalLex
  ORDFUNS, 2601
totolex
  LGROBP, 1285
trace
  INBFF, 1021
trace2PowMod
  DDFACT, 288
tracePowMod
  DDFACT, 288
transcendentalDecompose
  RSDCMPK, 3029
  SRDCMPK, 3110
TranscendentalHermiteIntegration, 3252
TranscendentalIntegration, 3254
TranscendentalManipulations, 3264
TranscendentalRischDE, 3273
TranscendentalRischDESystem, 3277
transform
  LGROBP, 1285
translate
  PFORP, 2628
translateToOrigin
  PAFFFF, 2619
  PLPKCRV, 2752
TransSolvePackage, 3288
TransSolvePackageService, 3299
trapezoidal
  NUMQUAD, 2563
trapezoidalo
  NUMQUAD, 2563
triangSolve
  ZDSOLVE, 3454
TriangularMatrixOperations, 3302
triangularSystems
  SYSSOLP, 3188
triangulate
  ODESYS, 3182
TRIGMNIP, 3304
  complexElementary, 3304
  complexForm, 3304
  complexNormalize, 3304
  imag, 3304
  real, 3304
  real?, 3304
  trigs, 3304
TrigonometricManipulations, 3304
trigs
  CTRIGMNP, 218
  TRIGMNIP, 3304
trigs2explogs
  ITRIGMNP, 1041
TRIMAT, 3302
  LowTriBddDenomInv, 3302
  UpTriBddDenomInv, 3302
TRMANIP, 3264
  cos2sec, 3264
  cosh2sech, 3264
  cot2tan, 3264
  cot2trig, 3264
  coth2tanh, 3264
  coth2trigh, 3264

csc2sin, 3264
csch2sinh, 3264
expand, 3264
expandLog, 3264
expandPower, 3264
expandTrigProducts, 3264
htrigs, 3264
removeCoshSq, 3264
removeCosSq, 3264
removeSinhSq, 3264
removeSinSq, 3264
sec2cos, 3264
sech2cosh, 3264
simplify, 3264
simplifyExp, 3264
simplifyLog, 3264
sin2csc, 3264
sinh2csch, 3264
tan2cot, 3264
tan2trig, 3264
tanh2coth, 3264
tanh2trigh, 3264
tryFunctionalDecomposition
GALFACT, 767
tryFunctionalDecomposition?
GALFACT, 767
tube
NUMTUBE, 2581
tubePlot
EXPRTUBE, 569
TubePlotTools, 3308
tubePoints
DROPT0, 357
tubePointsDefault
VIEWDEF, 3367
tubeRadius
DROPT0, 357
tubeRadiusDefault
VIEWDEF, 3367
TUBETOOL, 3308
?*?, 3308
?+?, 3308
?-?, 3308
cosSinInfo, 3308
cross, 3308
dot, 3308
loopPoints, 3308
point, 3308
unitVector, 3308
twist
MAPPKG3, 1362
TwoDimensionalPlotClipping, 3311
TWOFACT, 3317
generalSqFr, 3317
generalTwoFactor, 3317
twoFactor, 3317
twoFactor
TWOFACT, 3317
TwoFactorize, 3317

UDPO, 3359
getOrder, 3359
largest, 3359
less?, 3359
more?, 3359
setOrder, 3359
userOrdered?, 3359
UDVO, 3361
getVariableOrder, 3362
resetVariableOrder, 3362
setVariableOrder, 3362
UFPS1, 3330
hadamard, 3330
ULS2, 3331
map, 3331
unaryFunction
MKUCFUNC, 1327
uncouplingMatrices
ASSOCEQ, 96
UNIFACT, 3323
factor, 3323
factorSquareFree, 3323
henselFact, 3323
uniform
RDIST, 2937
RFDIST, 2939
RIDIST, 2941
uniform01
RFDIST, 2939
union
INTTOOLS, 1120
UNISEG2, 3357

map, 3358
units
DROPT0, 357
unitsColorDefault
VIEWDEF, 3367
unitVector
TUBETOOL, 3308
univariate
INTG0, 840
PFORP, 2628
POLY2UP, 2828
POLYCATQ, 2762
RF, 2947
univariate?
PSETPK, 2802
UnivariateFactorize, 3323
UnivariateFormalPowerSeriesFunctions, 3330
UnivariateLaurentSeriesFunctions2, 3331
UnivariatePolynomialCategoryFunctions2, 3333
UnivariatePolynomialCommonDenominator, 3334
UnivariatePolynomialDecompositionPackage, 3336
UnivariatePolynomialDivisionPackage, 3339
UnivariatePolynomialFunctions2, 3341
UnivariatePolynomialMultiplicationPackage, 3342
univariatePolynomials
PSETPK, 2802
univariatePolynomialsGcds
PSETPK, 2802
UnivariatePolynomialSquareFree, 3345
UnivariatePuiseuxSeriesFunctions2, 3348
UnivariateSkewPolynomialCategoryOps, 3349
univariateSolve
ZDSOLVE, 3454
UnivariateTaylorSeriesFunctions2, 3353
UnivariateTaylorSeriesODESolver, 3354
univcase
MULTSQFR, 1436
UniversalSegmentFunctions2, 3357
unprotectedRemoveRedundantFactors
PSETPK, 2802
unrankImproperPartitions0
SGCF, 3194
unrankImproperPartitions1
SGCF, 3194
untab
TABLBUMP, 3207
unvectorise
GALPOLYU, 784
SOLVESER, 3299
UP2, 3341
map, 3341
UP2ifCan
FSUPFACT, 758
UPCDEN, 3334
clearDenominator, 3334
commonDenominator, 3334
splitDenominator, 3334
updatD
GBINTERN, 871
upDateBranches
RSDCMPK, 3029
SRDCMPK, 3110
updatF
GBINTERN, 871
UPDECOMP, 3336
leftFactorIfCan, 3336
monicCompleteDecompose, 3336
monicDecomposeIfCan, 3336
monicRightFactorIfCan, 3336
rightFactorIfCan, 3336
UPDIVP, 3339
divideIfCan, 3339
UPMP, 3342
karatsuba, 3342
karatsubaOnce, 3342
noKaratsuba, 3342
UPOLYC2, 3333
map, 3333
UPSQFREE, 3345
BumInSepFFE, 3345
squareFree, 3345
squareFreePart, 3345
UpTriBddDenomInv
TRIMAT, 3302
UPXS2, 3348
map, 3348
useEisensteinCriterion
GALFACT, 767
useEisensteinCriterion?
GALFACT, 767
UserDefinedPartialOrdering, 3359

UserDefinedVariableOrdering, 3361
userOrdered?
    UDPO, 3359
useSingleFactorBound
    GALFACT, 767
useSingleFactorBound?
    GALFACT, 767
usingTable?
    TBCMPPK, 3211
UTS2, 3353
    map, 3353
UTS2UP
    UTSODETL, 3363
UTSODE, 3354
    fixedPointExquo, 3354
    mpsode, 3354
    ode, 3354
    ode1, 3354
    ode2, 3354
    stFunc1, 3354
    stFunc2, 3354
    stFuncN, 3354
UTSODETL, 3363
    LODO2FUN, 3363
    RF2UTS, 3363
    UTS2UP, 3363
UTSodetools, 3363
UTSSOL, 3216
    seriesSolve, 3216

validExponential
    EFSTRUC, 442
var1Steps
    DROPT0, 357
var1StepsDefault
    VIEWDEF, 3367
var2Steps
    DROPT0, 357
var2StepsDefault
    VIEWDEF, 3367
variables
    E04AGNT, 577
    FACUTIL, 593
    POLYCATQ, 2762
    RF, 2947
variationOfParameters
    ODETOOLS, 2589
vark
    INTTOOLS, 1120
varList
    D03AGNT, 381
    E04AGNT, 577
varselect
    INTTOOLS, 1120
VECTOR2, 3365
    map, 3365
    reduce, 3365
    scan, 3365
VectorFunctions2, 3365
vedf2vef
    ESTOOLS, 534
VIEW, 3372
    coerce, 3372
    drawCurves, 3372
    graphCurves, 3372
VIEWDEF, 3367
    axesColorDefault, 3367
    lineColorDefault, 3367
    pointColorDefault, 3367
    pointSizeDefault, 3367
    tubePointsDefault, 3367
    tubeRadiusDefault, 3367
    unitsColorDefault, 3367
    var1StepsDefault, 3367
    var2StepsDefault, 3367
    viewDefaults, 3367
    viewPosDefault, 3367
    viewSizeDefault, 3367
    viewWriteAvailable, 3367
    viewWriteDefault, 3367
viewDefaults
    VIEWDEF, 3367
ViewDefaultsPackage, 3367
viewpoint
    DROPT0, 357
ViewportPackage, 3372
viewPosDefault
    VIEWDEF, 3367
viewSizeDefault
    VIEWDEF, 3367
viewWriteAvailable
    VIEWDEF, 3367

viewWriteDefault
VIEWDEF, 3367
virtualDegree
GBINTERN, 871

weakBiRank
ALGPKG, 37
WEIER, 3375
cfirst, 3375
clikeUniv, 3375
crest, 3375
qqq, 3375
sts2stst, 3375
weierstrass, 3375
weierstrass
WEIER, 3375
WeierstrassPreparation, 3375
weighted
RDIST, 2937
WFFINTBS, 3379
integralBasis, 3379
localIntegralBasis, 3379
WildFunctionFieldIntegralBasis, 3379
wreath
CYCLES, 249
writeObj
EXP3D, 574
wronskianMatrix
ODETOOLS, 2589
wrregime
PLEQN, 2648

xCoord
PTPACK, 2738
XExponentialPackage, 3385
XEXPPKG, 3385
exp, 3385
Hausdorff, 3385
log, 3385
xn
INBFF, 1021

Y
YSTREAM, 2647
yCoord
PTPACK, 2738
youngGroup
PGE, 2718
YSTREAM, 2646
Y, 2647
Yun
FFSQFR, 657

zCoord
PTPACK, 2738
ZDSOLVE, 3454
convert, 3454
positiveSolve, 3454
realSolve, 3454
squareFree, 3454
triangSolve, 3454
univariateSolve, 3454
zeroDimensional?
FGLMICPK, 604
LEXTRIPK, 1260
ZeroDimensionalSolvePackage, 3454
zeroDimPrimary?
IDECOMP, 973
zeroDimPrime?
IDECOMP, 973
zeroMatrix
FCPAK1, 673
zeroSetSplit
LAZM3PK, 1185
LEXTRIPK, 1260
zerosOf
ESCONT, 527
zeroSquareMatrix
FCPAK1, 673
zeroVector
FCPAK1, 673
ZetaFunction
GPAFF, 801
PAFF, 2611
PAFFFF, 2619
ZLINDEP, 1067
linearDependenceOverZ, 1067
linearlyDependentOverZ?, 1067
solveLinearlyOverQ, 1067