

Python Interface

Release 5.5

Luis Saavedra

Nov 20, 2026

CONTENTS

1 Introduction **1**

2 Installation **3**

3 Preliminary **5**

4 *Python GetFEM* interface **7**
- 4.1 Introduction . . . 7
- 4.2 Parallel version . . . 7
- 4.3 Memory Management . . . 8
- 4.4 Documentation . . . 8
- 4.5 *Python GetFEM* organization . . . 8

5 Examples **11**
- 5.1 A step-by-step basic example . . . 11
- 5.2 Another Laplacian with exact solution (source term) . . . 14
- 5.3 Linear and non-linear elasticity . . . 17
- 5.4 Avoiding the model framework . . . 20
- 5.5 Other examples . . . 21

6 How-tos **23**
- 6.1 Import gmsh mesh . . . 23

7 Visualisation of meshes and fields **25**
- 7.1 Quick start . . . 25
- 7.2 How it works . . . 26
- 7.3 Plotting a field: `plot` . . . 26
- 7.4 Saving figures . . . 28
- 7.5 Plotting a mesh: `plot_mesh` . . . 28
- 7.6 Using the extracted data . . . 29

8 API reference **31**
- 8.1 ContStruct . . . 31
- 8.2 CvStruct . . . 33
- 8.3 Eltm . . . 34
- 8.4 Fem . . . 35
- 8.5 GeoTrans . . . 37
- 8.6 GlobalFunction . . . 38
- 8.7 Integ . . . 39
- 8.8 LevelSet . . . 41

8.9 Mesh . . . 42
8.10 MeshFem . . . 51
8.11 MeshIm . . . 57
8.12 MeshImData . . . 59
8.13 MeshLevelSet . . . 60
8.14 MesherObject . . . 61
8.15 Model . . . 62
8.16 MumpsContext . . . 92
8.17 Precond . . . 94
8.18 Slice . . . 95
8.19 Spmat . . . 99
8.20 Module asm . . . 102
8.21 Module compute . . . 107
8.22 Module delete . . . 109
8.23 Module linsolve . . . 110
8.24 Module poly . . . 110
8.25 Module util . . . 111

Index **113**

CHAPTER

ONE

INTRODUCTION

This guide provides a reference about the *Python* interface of *GetFEM*. For a complete reference of *GetFEM*, please report to the specific guides, but you should be able to use the *getfem-interface*'s without any particular knowledge of the *GetFEM* internals, although a basic knowledge about Finite Elements is required. This documentation is however not self contained. You should in particular refer to the user documentation to have a more extensive description of the structures algorithms and concepts used.

Copyright © 2004-2026 *GetFEM* project.

The text of the *GetFEM* website and the documentations are available for modification and reuse under the terms of the GNU Free Documentation License

GetFEM is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version along with the GCC Runtime Library Exception either version 3.1 or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License and GCC Runtime Library Exception for more details. You should have received a copy of the GNU Lesser General Public License along with this program. If not, see https://www.gnu.org/licenses/.

CHAPTER

TWO

INSTALLATION

If installing from sources, use the option *–enable-python* of the *configure* script.

For the parallel version of the interface, see also ud-parallel.

See the download and install page for the installation of *GetFEM* on different plateforms.

CHAPTER

THREE

PRELIMINARY

This is just a short summary of the terms employed in this manual. If you are not familiar with finite elements, this should be useful (but in any case, you should definitively read the dp).

The `mesh` is composed of `convexes`. What we call convexes can be simple line segments, prisms, tetrahedrons, curved triangles, of even something which is not convex (in the geometrical sense). They all have an associated `reference convex`: for segments, this will be the $[0, 1]$ segment, for triangles this will be the canonical triangle $(0, 0) - (0, 1) - (1, 0)$, etc. All convexes of the mesh are constructed from the reference convex through a `geometric transformation`. In simple cases (when the convexes are simplices for example), this transformation will be linear (hence it is easily inverted, which can be a great advantage). In order to define the geometric transformation, one defines `geometrical nodes` on the reference convex. The geometrical transformation maps these nodes to the `mesh nodes`.

On the mesh, one defines a set of basis functions: the `FEM`. A FEM is associated at each convex. The basis functions are also attached to some geometrical points (which can be arbitrarily chosen). These points are similar to the mesh nodes, but **they don't have to be the same** (this only happens on very simple cases, such as a classical P_1 fem on a triangular mesh). The set of all basis functions on the mesh forms the basis of a vector space, on which the PDE will be solved. These basis functions (and their associated geometrical point) are the `degrees of freedom` (contracted to `dof`). The FEM is said to be `Lagrangian` when each of its basis functions is equal to one at its attached geometrical point, and is null at the geometrical points of others basis functions. This is an important property as it is very easy to `interpolate` an arbitrary function on the finite elements space.

The finite elements method involves evaluation of integrals of these basis functions (or product of basis functions etc.) on convexes (and faces of convexes). In simple cases (polynomial basis functions and linear geometrical transformation), one can evaluate analytically these integrals. In other cases, one has to approximate it using `quadrature formulas`. Hence, at each convex is attached an `integration method` along with the FEM. If you have to use an approximate integration method, always choose carefully its order (i.e. highest degree of the polynomials who are exactly integrated with the method): the degree of the FEM, of the polynomial degree of the geometrical transformation, and the nature of the elementary matrix have to be taken into account. If you are unsure about the appropriate degree, always prefer a high order integration method (which will slow down the assembly) to a low order one which will produce a useless linear-system.

The process of construction of a global linear system from integrals of basis functions on each convex is the `assembly`.

A mesh, with a set of FEM attached to its convexes is called a `mesh_fem` object in *GetFEM*.

A mesh, with a set of integration methods attached to its convexes is called a `mesh_im` object in *GetFEM*.

A *mesh_fem* can be used to approximate scalar fields (heat, pression, ...), or vector fields (displacement, electric field, ...). A *mesh_im* will be used to perform numerical integrations on these fields. Most of the finite elements implemented in *GetFEM* are scalar (however, TR_0 and edges elements are also available).

Of course, these scalar FEMs can be used to approximate each component of a vector field. This is done by setting the $Qdim$ of the *mesh_fem* to the dimension of the vector field (i.e. $Qdim = 1$ ℝightarrow scalar field, $Qdim = 2$ ℝightarrow 2D vector field etc.).

When solving a PDE, one often has to use more than one FEM. The most important one will be of course the one on which is defined the solution of the PDE. But most PDEs involve various coefficients, for example:

$$\nabla \cdot (\lambda(x)\nabla u) = f(x).$$

Hence one has to define an FEM for the main unknown u, but also for the data $\lambda(x)$ and $f(x)$ if they are not constant. In order to interpolate easily these coefficients in their finite element space, one often choose a Lagrangian FEM.

The convexes, mesh nodes, and dof are all numbered. We sometimes refer to the number associated to a convex as its `convex id` (contracted to `cvid`). Mesh node numbers are also called `point id` (contracted to `pid`). Faces of convexes do not have a global numbering, but only a local number in each convex. Hence functions which need or return a list of faces will always use a two-rows matrix, the first one containing convex ids, and the second one containing local face number.

While the dof are always numbered consecutively, **this is not always the case for point ids and convex ids**, especially if you have removed points or convexes from the mesh. To ensure that they form a continuous sequence (starting from 1), you have to call:

```
>>> m.set('optimize structure')
```

CHAPTER

FOUR

PYTHON GETFEM INTERFACE

4.1 Introduction

GetFEM provides an interface to the *Python* scripting language. *Python* is a nice, cross-platform, and free language. With the addition of the numpy package, python provides a subset of Matlab functionalities (i.e. dense arrays). The VTK toolkit may provide visualization tools via its python interface (or via MayaVi), and data files for OpenDX may be exported. In this guide, nevertheless, to visualize the results, we will export to Gmsh post-processing format. The sparse matrix routines are provided by the getfem interface.

The python interface is available via a python module getfem.py. In order to use the interface you have to load it with:

```
import getfem
m = getfem.Mesh('cartesian', range(0, 3), range(0,3))
```

or:

```
from getfem import *
m = Mesh('cartesian', range(0, 3), range(0,3))
```

If the getfem.py (and the internal _getfem.so) module is not installed in a standard location for python, you may have to set the `PYTHONPATH` environment variable to its location. For example with:

```
import sys
sys.path.append('.../getfem/getfem++/interface/src/python/')
```

4.2 Parallel version

The python interface is the only one for the moment to interface the mpi based parallel version of Getfem. See ud-parallel.

4.3 Memory Management

A nice advantage over the Matlab interface is that you do not have to explicitly delete objects that are not used any more, this is done automagically. You can however inspect the content of the getfem workspace with the function `getfem.memstats()`.

4.4 Documentation

The *getfem* module is largely documented. This documentation has been extracted into the *API reference*. The getfem-matlab user guide may also be used, as 95% of its content translates quite directly into python (with the exception of the plotting functions, which are specific to matlab).

4.5 *Python GetFEM* organization

The general organization of the python-interface is the following:

- Each class from the matlab interface has a corresponding class in the python interface: the gfMesh class becomes the getfem.Mesh class in python, the gfSlice becomes the getfem.Slice etc.
- Each get and set method of the matlab interface has been translated into a method of the corresponding class in the python interface. For example:

```
gf_mesh_get(m, 'outer faces');
gf_mesh_get(m, 'pts');
```

  becomes:

```
m.outer_faces();
m.pts();
```

  Some methods have been renamed when there was ambiguity, for example `gf_mesh_set(m, 'pts', P)` is `m.set_pts(P)`.

- The other getfem-matlab function have a very simple mapping to their python equivalent:

gf_compute(mf,U,'foo',...)	getfem.compute_foo(mf,U) or getfem.compute('foo',...)
gf_asm('foobar',...)	getfem.asm_foobar(...) or getfem.asm('foobar',...)
gf_linsolve('gmres',...)	getfem.linsolve_gmres(...) or getfem.linsolve('gmres',...)

class CvStruct(*self*, **args*)

Descriptor for a convex structure objects, stores formal information convex structures (nb. of points, nb. of faces which are themselves convex structures)

class GeoTrans(*self*, **args*)

Descriptor for geometric transformations objects (defines the shape/position of the convexes).

class Mesh(*self*, **args*)

Descriptor for mesh structure (nodes, convexes, geometric transformations for each convex).

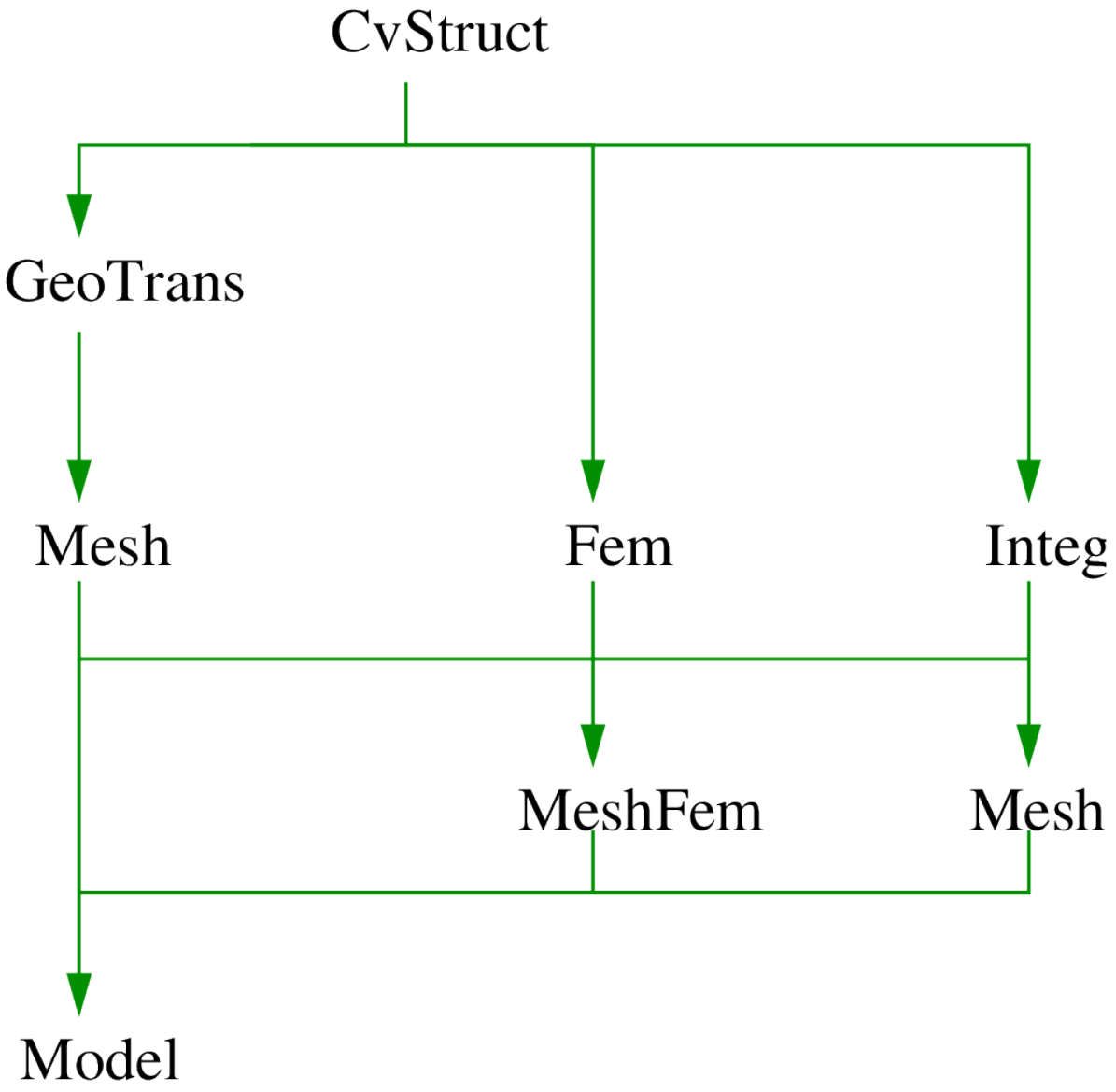


Fig. 1: python-getfem interface main objects hierarchy.

class Fem(*self*, *fem_name*)

Descriptor for FEM (Finite Element Method) objects (one per convex, can be PK, QK, HERMITE, etc...).

class Integ(*self*, **args*)

Descriptor for Integration Method objects (exact, quadrature formulaldots). Although not linked directly to GeoTrans, an integration method is usually specific to a given convex structure.

class MeshFem(*self*, **args*)

Descriptor for object linked to a mesh, where each convex has been assigned an FEM.

class MeshIm(*self*, **args*)

Descriptor for object linked to a mesh, where each convex has been assigned an integration method.

class Model(*self*, **args*)

Descriptor for *model* object, holds the global data, variables and description of a model. Evolution of *model state* and *model brick* object for 4.0 version of *GetFEM*.

CHAPTER

FIVE

EXAMPLES

5.1 A step-by-step basic example

This example shows the basic usage of getfem, on the über-canonical problem above all others: solving the Laplacian, $-\Delta u = f$ on a square, with the Dirichlet condition $u = g(x)$ on the domain boundary. You can find the **py-file** of this example under the name **demo_step_by_step.py** in the directory `interface/tests/python/` of the *GetFEM* distribution.

The first step is to **create a Mesh object**. It is possible to create simple structured meshes or unstructured meshes for simple geometries (see `getfem.Mesh('generate', mesher_object mo, scalar h)`) or to rely on an external mesher (see `getfem.Mesh('import', string FORMAT, string FILENAME)`), or use very simple meshes. For this example, we just consider a regular meshindex{cartesian mesh} whose nodes are $\{x_{i=0..10,j=0..10} = (i/10, j/10)\}$

```
# import basic modules
import numpy as np
import getfem as gf

# creation of a simple cartesian mesh
m = gf.Mesh('cartesian', np.arange(0,1.1,0.1), np.arange(0,1.1,0.1))
```

The next step is to **create a MeshFem object**. This one links a mesh with a set of FEM

```
# create a MeshFem of for a field of dimension 1 (i.e. a scalar field)
mf = gf.MeshFem(m, 1)
# assign the Q2 fem to all convexes of the MeshFem
mf.set_fem(gf.Fem('FEM_QK(2,2)'))
```

The first instruction builds a new MeshFem object, the second argument specifies that this object will be used to interpolate scalar fields (since the unknown u is a scalar field). The second instruction assigns the Q^2 FEM to every convex (each basis function is a polynomial of degree 4, remember that $P^k\mathbb{R}\mathrm{ightarrow}$ polynomials of degree k, while $Q^k\mathbb{R}\mathrm{ightarrow}$ polynomials of degree $2k$). As Q^2 is a polynomial FEM, you can view the expression of its basis functions on the reference convex:

```
# view the expression of its basis functions on the reference convex
print gf.Fem('FEM_QK(2,2)').poly_str()
```

Now, in order to perform numerical integrations on `mf`, we need to **build a MeshIm object**

```
# an exact integration will be used
mim = gf.MeshIm(m, gf.Integ('IM_EXACT_PARALLELEPIPED(2)'))
```

The integration method will be used to compute the various integrals on each element: here we choose to perform exact computations (no `quadrature formula`), which is possible since the geometric transformation of these convexes from the reference convex is linear (this is true for all simplices, and this is also true for the parallelepipeds of our regular mesh, but it is not true for general quadrangles), and the chosen FEM is polynomial. Hence it is possible to analytically integrate every basis function/product of basis functions/gradients/etc. There are many alternative FEM methods and integration methods (see ud).

Note however that in the general case, approximate integration methods are a better choice than exact integration methods.

Now we have to **find the** <boundary> **of the domain**, in order to set a Dirichlet condition. A mesh object has the ability to store some sets of convexes and convex faces. These sets (called <regions>) are accessed via an integer *#id*

```
# detect the border of the mesh
border = m.outer_faces()
# mark it as boundary #42
m.set_region(42, border)
```

Here we find the faces of the convexes which are on the boundary of the mesh (i.e. the faces which are not shared by two convexes).

The array `border` has two rows, on the first row is a convex number, on the second row is a face number (which is local to the convex, there is no global numbering of faces). Then this set of faces is assigned to the region number 42.

At this point, we just have to describe the model and run the solver to get the solution! The "`model`" is created with the Model constructor. A model is basically an object which build a global linear system (tangent matrix for non-linear problems) and its associated right hand side. Typical modifications are insertion of the stiffness matrix for the problem considered (linear elasticity, laplacian, etc), handling of a set of constraints, Dirichlet condition, addition of a source term to the right hand side etc. The global tangent matrix and its right hand side are stored in the "`model`" structure.

Let us build a problem with an easy solution: $u = x(x-1) - y(y-1)$, then we have $-\Delta u = 0$ (the FEM won't be able to catch the exact solution since we use a Q^2 method).

We start with an empty real model

```
# empty real model
md = gf.Model('real')
```

(a model is either `'real'` or `'complex'`). And we declare that `u` is an unknown of the system on the finite element method *mf* by

```
# declare that "u" is an unknown of the system
# on the finite element method `mf`
md.add_fem_variable('u', mf)
```

Now, we add a *generic elliptic* brick, which handles $-\nabla \cdot (A : \nabla u) = \ldots$ problems, where A can be a scalar field, a matrix field, or an order 4 tensor field. By default, $A = 1$. We add it on our main variable `u` with

```
# add generic elliptic brick on "u"
md.add_Laplacian_brick(mim, 'u');
```

Next we add a Dirichlet condition on the domain boundary

```
# add Dirichlet condition
g = mf.eval('x*(x-1) - y*(y-1)')
md.add_initialized_fem_data('DirichletData', mf, g)
md.add_Dirichlet_condition_with_multipliers(mim, 'u', mf, 42, 'DirichletData')
```

The two first lines defines a data of the model which represents the value of the Dirichlet condition. The third one add a Dirichlet condition to the variable `u` on the boundary number `42`. The dirichlet condition is imposed with lagrange multipliers. Another possibility is to use a penalization. A MeshFem argument is also required, as the Dirichlet condition $u = g$ is imposed in a weak form $\int_\Gamma u(x)v(x) = \int_\Gamma g(x)v(x)\ \forall v$ where v is taken in the space of multipliers given by here by `mf`.

Remark:

the polynomial expression was interpolated on `mf`. It is possible only if `mf` is of Lagrange type. In this first example we use the same MeshFem for the unknown and for the data such as `g`, but in the general case, `mf` won't be Lagrangian and another (Lagrangian) MeshFem will be used for the description of Dirichlet conditions, source terms etc.

A source term can be added with (uncommented) the following lines

```
# add source term
#f = mf.eval('0')
#md.add_initialized_fem_data('VolumicData', mf, f)
#md.add_source_term_brick(mim, 'u', 'VolumicData')
```

It only remains now to launch the solver. The linear system is assembled and solve with the instruction

```
# solve the linear system
md.solve()
```

The model now contains the solution (as well as other things, such as the linear system which was solved). It is extracted

```
# extracted solution
u = md.variable('u')
```

Then export solution

```
# export computed solution
mf.export_to_pos('u.pos',u,'Computed solution')
```

and view with `gmsh u.pos`, see figure *Computed solution*.

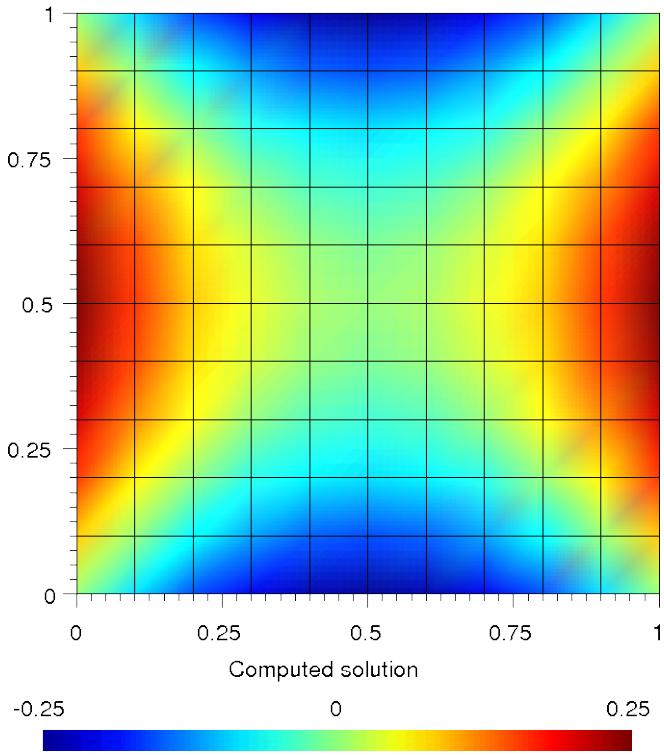


Fig. 1: Computed solution

5.2 Another Laplacian with exact solution (source term)

This example shows the basic usage of getfem, on the canonical problem: solving the Laplacian, $-\Delta u = f$ on a square, with the Dirichlet condition $u = g(x)$ on the domain boundary Γ_D and the Neumann condition $\frac{\partial u}{\partial \eta} = h(x)$ on the domain boundary Γ_N. You can find the **py-file** of this example under the name **demo_laplacian.py** in the directory `interface/tests/python/` of the *GetFEM* distribution.

We create Mesh, MeshFem, MeshIm object and find the boundary of the domain in the same way as the previous example

```
import numpy as np

# import basic modules
import getfem as gf

# boundary names
top   = 101 # Dirichlet boundary
down  = 102 # Neumann boundary
left  = 103 # Dirichlet boundary
right = 104 # Neumann boundary

# parameters
NX = 40                                # Mesh parameter
Dirichlet_with_multipliers = True;  # Dirichlet condition with multipliers or␣
↪penalization
dirichlet_coefficient = 1e10;       # Penalization coefficient

```

(continues on next page)

(continued from previous page)

```
# mesh creation
m = gf.Mesh('regular_simplices', np.arange(0,1+1./NX,1./NX), np.arange(0,1+1./
↪NX,1./NX))

# create a MeshFem for u and rhs fields of dimension 1 (i.e. a scalar field)
mfu   = gf.MeshFem(m, 1)
mfrhs = gf.MeshFem(m, 1)
# assign the P2 fem to all convexes of the both MeshFem
mfu.set_fem(gf.Fem('FEM_PK(2,2)'))
mfrhs.set_fem(gf.Fem('FEM_PK(2,2)'))

# an exact integration will be used
mim = gf.MeshIm(m, gf.Integ('IM_TRIANGLE(4)'))

# boundary selection
flst   = m.outer_faces()
fnor   = m.normal_of_faces(flst)
ttop   = abs(fnor[1,:]-1) < 1e-14
tdown  = abs(fnor[1,:]+1) < 1e-14
tleft  = abs(fnor[0,:]+1) < 1e-14
tright = abs(fnor[0,:]-1) < 1e-14
ftop   = np.compress(ttop, flst, axis=1)
fdown  = np.compress(tdown, flst, axis=1)
fleft  = np.compress(tleft, flst, axis=1)
fright = np.compress(tright, flst, axis=1)

# mark it as boundary
m.set_region(top, ftop)
m.set_region(down, fdown)
m.set_region(left, fleft)
```

then, we interpolate the exact solution and source terms

```

# interpolate the exact solution (assuming mfu is a Lagrange fem)
g = mfu.eval('y*(y-1)*x*(x-1)+x*x*x*x*x*x')

# interpolate the source terms (assuming mfrhs is a Lagrange fem)
f = mfrhs.eval('-(2*(x*x+y*y)-2*x-2*y+20*x*x*x)')
```

and we bricked the problem as in the previous example

```

# model
md = gf.Model('real')

# add variable and data to model
md.add_fem_variable('u', mfu)                # main unknown
md.add_initialized_fem_data('f', mfrhs, f) # volumic source term
md.add_initialized_fem_data('g', mfrhs, g) # Dirichlet condition
```

(continues on next page)

(continued from previous page)

```
md.add_initialized_fem_data('h', mfrhs, h) # Neumann condition

# bricked the problem
md.add_Laplacian_brick(mim, 'u')                                    # laplacian term␣
↪on u
md.add_source_term_brick(mim, 'u', 'f')                             # volumic source␣
↪term
md.add_normal_source_term_brick(mim, 'u', 'h', down)                # Neumann␣
↪condition
md.add_normal_source_term_brick(mim, 'u', 'h', left)                # Neumann␣
↪condition

# Dirichlet condition on the top
if (Dirichlet_with_multipliers):
  md.add_Dirichlet_condition_with_multipliers(mim, 'u', mfu, top, 'g')
else:
  md.add_Dirichlet_condition_with_penalization(mim, 'u', dirichlet_
↪coefficient, top, 'g')

# Dirichlet condition on the right
if (Dirichlet_with_multipliers):
  md.add_Dirichlet_condition_with_multipliers(mim, 'u', mfu, right, 'g')
else:
```

the only change is the add of *source term* bricks. Finally the solution of the problem is extracted and exported

```

# assembly of the linear system and solve.
md.solve()

# main unknown
u  = md.variable('u')
L2error = gf.compute(mfu, u-g, 'L2 norm', mim)
H1error = gf.compute(mfu, u-g, 'H1 norm', mim)

if (H1error > 1e-3):
    print 'Error in L2 norm : ', L2error
    print 'Error in H1 norm : ', H1error
    print 'Error too large !'

# export data
mfu.export_to_pos('sol.pos', g,'Exact solution',
```

view with `gmsh sol.pos`:

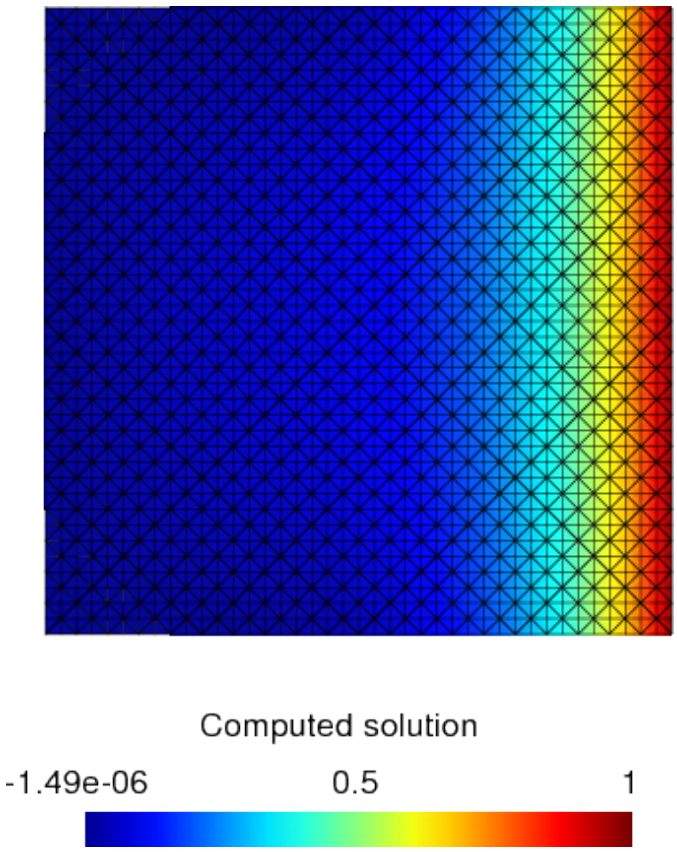


Fig. 2: Differences

5.3 Linear and non-linear elasticity

This example uses a mesh that was generated with GiD. The object is meshed with quadratic tetrahedrons. You can find the **py-file** of this example under the name `demo_tripod.py` in the directory `interface/tests/python/` of the *GetFEM* distribution.

```
import numpy as np

import getfem as gf

with_graphics=True
try:
    import getfem.getfem_plot as gfp
except ImportError:
    try:  # in the build tree, getfem is the module getfem.py, not the package
        import getfem_plot as gfp
    except ImportError:
        gfp = None
if gfp is None or not gfp.available_backends():
    print("\n** getfem_plot with pyvista or matplotlib not available --graphical output disabled **\n")
    with_graphics=False


```

(continues on next page)

(continued from previous page)

```
m=gf.Mesh('import','gid','../meshes/tripod.GiD.msh')
print('done!')
mfu=gf.MeshFem(m,3) # displacement
mfp=gf.MeshFem(m,1) # pressure
mfd=gf.MeshFem(m,1) # data
mim=gf.MeshIm(m, gf.Integ('IM_TETRAHEDRON(5)'))
degree = 2
linear = False
incompressible = False # ensure that degree > 1 when incompressible is on..

mfu.set_fem(gf.Fem('FEM_PK(3,%d)' % (degree,)))
mfd.set_fem(gf.Fem('FEM_PK(3,0)'))
mfp.set_fem(gf.Fem('FEM_PK_DISCONTINUOUS(3,0)'))

print('nbcvs=%d, nbpts=%d, qdim=%d, fem = %s, nbdof=%d' % \
      (m.nbcvs(), m.nbpts(), mfu.qdim(), mfu.fem()[0].char(), mfu.nbdof()))

P=m.pts()
print('test', P[1,:])
ctop=(abs(P[1,:] - 13) < 1e-6)
cbot=(abs(P[1,:] + 10) < 1e-6)
pidtop=np.compress(ctop, list(range(0, m.nbpts())))
pidbot=np.compress(cbot, list(range(0, m.nbpts())))

ftop=m.faces_from_pid(pidtop)
fbot=m.faces_from_pid(pidbot)
NEUMANN_BOUNDARY = 1
DIRICHLET_BOUNDARY = 2

m.set_region(NEUMANN_BOUNDARY,ftop)
m.set_region(DIRICHLET_BOUNDARY,fbot)

E=1e3
Nu=0.3
Lambda = E*Nu/((1+Nu)*(1-2*Nu))
Mu =E/(2*(1+Nu))


md = gf.Model('real')
md.add_fem_variable('u', mfu)
if linear:
    md.add_initialized_data('cmu', Mu)
    md.add_initialized_data('clambda', Lambda)
    md.add_isotropic_linearized_elasticity_brick(mim, 'u', 'clambda', 'cmu')
    if incompressible:
        md.add_fem_variable('p', mfp)
        md.add_linear_incompressibility_brick(mim, 'u', 'p')
else:
    md.add_initialized_data('params', [Lambda, Mu]);
```

(continues on next page)

(continued from previous page)

```
    if incompressible:
        lawname = 'Incompressible Mooney Rivlin';
        md.add_finite_strain_elasticity_brick(mim, lawname, 'u', 'params')
        md.add_fem_variable('p', mfp);
        md.add_finite_strain_incompressibility_brick(mim, 'u', 'p');
    else:
        lawname = 'SaintVenant Kirchhoff';
        md.add_finite_strain_elasticity_brick(mim, lawname, 'u', 'params');


md.add_initialized_data('VolumicData', [0,-1,0]);
md.add_source_term_brick(mim, 'u', 'VolumicData');

# Attach the tripod to the ground
md.add_Dirichlet_condition_with_multipliers(mim, 'u', mfu, 2);

print('running solve...')
md.solve('noisy', 'max iter', 1);
U = md.variable('u');
print('solve done!')


mfdu=gf.MeshFem(m,1)
mfdu.set_fem(gf.Fem('FEM_PK_DISCONTINUOUS(3,1)'))
if linear:
  VM = md.compute_isotropic_linearized_Von_Mises_or_Tresca('u','clambda','cmu
↪', mfdu);
else:
  VM = md.compute_finite_strain_elasticity_Von_Mises(lawname, 'u', 'params',␣
↪mfdu);

# post-processing
sl=gf.Slice(('boundary',), mfu, degree)

print('Von Mises range: ', VM.min(), VM.max())

# export results to VTK
sl.export_to_vtk('tripod.vtk', 'ascii', mfdu,  VM, 'Von Mises Stress', mfu, U,
↪ 'Displacement')
sl.export_to_pos('tripod.pos', mfdu, VM, 'Von Mises Stress', mfu, U,
↪'Displacement')
```

The Von Mises stress is then displayed on the deformed tripod with the module `getfem.getfem_plot` (see *Visualisation of meshes and fields*), in an interactive window when PyVista is installed

```
if with_graphics:
  print("Close the window to continue..")
  gfp.plot(mfdu, VM, name="Von Mises", refine=2, deformation=U, deformation_
↪mf=mfu,
```

(continues on next page)

(continued from previous page)

```
4           deformation_scale='20%', title="Tripod, Von Mises stress",␣
↪show=True)
```

Here is the final figure, displaying the Von Mises stress and displacements norms:

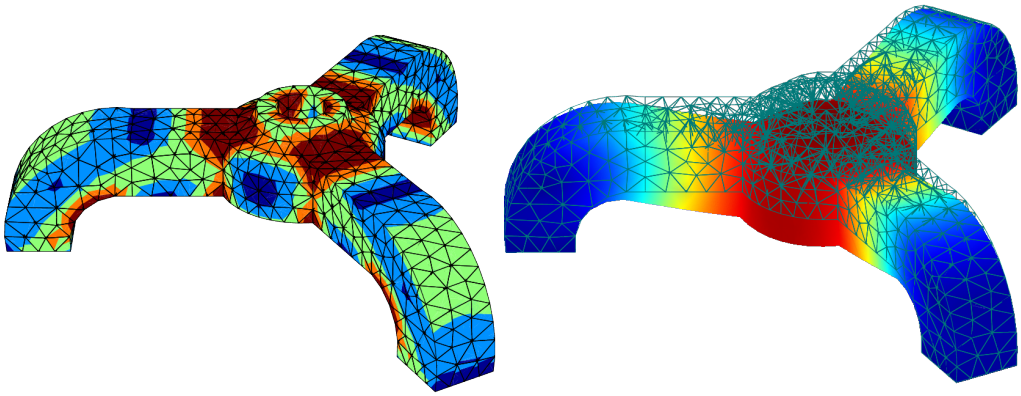


Fig. 3: (a) Tripod Von Mises, (b) Tripod displacements norms.

5.4 Avoiding the model framework

The model bricks are very convenient, as they hide most of the details of the assembly of the final linear systems. However it is also possible to stay at a lower level, and handle the assembly of linear systems, and their resolution, directly in *Python*. For example, the demonstration `demo_tripod_alt.py` is very similar to the `demo_tripod.py` except that the assembly is explicit

```
mfu = gf.MeshFem(m,3) # displacement
mfe = gf.MeshFem(m,1) # for plot von-mises
mfu.set_fem(gf.Fem('FEM_PK(3,%d)' % (degree,)))
m.set_region(DIRICHLET_BOUNDARY,fbot)

# assembly
print "nbd: ",nbd

print "np.repeat([Mu], nbd).shape:",np.repeat([Mu], nbd).shape

# handle Dirichlet condition

print "U0.shape: ",U0.shape

Nt = gf.Spmat('copy',N)
Nt.transpose()
KK = Nt*K*N
FF = Nt*F # FF = Nt*(F-K*U0)
```

(continues on next page)

(continued from previous page)

```
# solve ...
P = gf.Precond('ildlt',KK)
print "UU.shape:",UU.shape
print "U.shape:",U.shape

# post-processing
sl = gf.Slice(('boundary',), mfu, degree)

# compute the Von Mises Stress
DU = gf.compute_gradient(mfu,U,mfe)
VM = np.zeros((DU.shape[2],),'d')
Sigma = DU

for i in range(DU.shape[2]):
  d = np.array(DU[:,:,i])
  E = (d+d.T)*0.5
  Sigma[:,:,i]=E
  VM[i] = np.sum(E**2) - (1./3.)*np.sum(np.diagonal(E))**2
# can be viewed with mayavi -d ./tripod_ev.vtk -f WarpVector -m TensorGlyphs
#print 'mayavi -d ./tripod.vtk -f WarpVector -m BandedSurfaceMap'

# export to Gmsh
sl.export_to_pos('tripod.pos', mfe, VM,'Von Mises Stress', mfu, U,
↪'Displacement')
sl.export_to_pos('tripod_ev.pos', mfu, U, 'Displacement', SigmaSL, 'stress')
```

In *getfem-interface*, the assembly of vectors, and matrices is done via the `gf.asm_*` functions. The Dirichlet condition $h(x)u(x) = r(x)$ is handled in the weak form $\int (h(x)u(x)).v(x) = \int r(x).v(x) \quad \forall v$ (where $h(x)$ is a 3×3 matrix field – here it is constant and equal to the identity). The reduced system `KK UU = FF` is then built via the elimination of Dirichlet constraints from the original system. Note that it might be more efficient (and simpler) to deal with Dirichlet condition via a penalization technique.

5.5 Other examples

- the `demo_refine.py` script shows a simple 2D or 3D bar whose extremity is clamped. An adaptative refinement is used to obtain a better approximation in the area where the stress is singular (the transition between the clamped area and the neumann boundary).
- the `demo_nonlinear_elasticity.py` script shows a 3D bar which is bended and twisted. This is a quasi-static problem as the deformation is applied in many steps. At each step, a non-linear (large deformations) elasticity problem is solved.
- the `demo_stokes_3D_tank.py` script shows a Stokes (viscous fluid) problem in a tank. The `demo_stokes_3D_tank_draw.py` shows how to draw a nice plot of the solution, with mesh slices and stream lines. Note that the `demo_stokes_3D_tank_alt.py` is the old example, which uses the deprecated `gf_solve` function.
- the `demo_bilaplacian.py` script is just an adaption of the *GetFEM* example `tests/bilaplacian.cc`. Solve the bilaplacian (or a Kirchhoff-Love plate model) on a square.

- the `demo_plasticity.py` script is an adaptation of the *GetFEM* example `tests/plasticity.cc`: a 2D or 3D bar is bended in many steps, and the plasticity of the material is taken into account (plastification occurs when the material's Von Mises exceeds a given threshold).
- the `demo_wave2D.py` is a 2D scalar wave equation example (diffraction of a plane wave by a cylinder), with high order geometric transformations and high order FEMs.

CHAPTER

SIX

HOW-TOS

6.1 Import gmsh mesh

If we have in the file *quad.geo* a parameterized mesh, as this:

```
lc = 0.05 ;

Point(1) = {0,0,0,lc};
Point(2) = {1,0,0,lc};
Point(3) = {1,1,0,lc};
Point(4) = {0,1,0,lc};

Line(5) = {1,2};
Line(6) = {2,3};
Line(7) = {3,4};
Line(8) = {4,1};

Line Loop(9) = {5,6,7,8};
Plane Surface(10) = {9};

Physical Line(101) = {7};
Physical Line(102) = {5};
Physical Line(103) = {8};
Physical Line(104) = {6};

Physical Surface(201) = {10};
```

then, when we run:

```
$ gmsh -2 quad.geo -format msh1
```

the file *quad.msh* is created and contains the encoding of the mesh and its regions. We can import that file (*quad.msh*) to getfem:

```
import getfem as gf

m = gf.Mesh('import','gmsh','quad.msh')
print m.regions()
```

with the second command we can see the *regions ids*. When we import the mesh, we might be warned with the following:

```
Level 3 Warning in getfem_import.cc, line 137:
  All regions must have different number!
```

this means that the parametrization of the mesh in *Gmsh .geo file* must assign a **different** number to each region, the problem exists because in *Gmsh* can coexist, for example, "Physical Surface (200)" and "Physical Line (200)", as they are different "types of regions" in *Gmsh*, that which does not occur in *GetFEM* since there is only one "type of region".

CHAPTER

SEVEN

VISUALISATION OF MESHES AND FIELDS

The module `getfem_plot` displays meshes and finite element fields directly from *Python*, in the same spirit as the functions `gf_plot_mesh`, `gf_plot` and `gf_plot_slice` of the Matlab/Octave interface. It relies on two optional libraries, which are only imported when they are used:

- Matplotlib, for 1D and 2D plots (well suited to figures for papers) and small 3D surfaces;
- PyVista (based on VTK), for 3D plots that can be rotated, zoomed and panned with the mouse, also inside Jupyter notebooks, or rendered off-screen to image files.

They can be installed with:

```
$ pip install matplotlib pyvista
```

In 3D, PyVista is used when it is available; otherwise the module falls back on Matplotlib with a warning (Matplotlib is slow for large surfaces and only sorts the faces approximately in depth). The renderer can be chosen explicitly with `backend="matplotlib"` or `backend="pyvista"`.

7.1 Quick start

```
import numpy as np
import getfem as gf
import getfem_plot as gfp

m = gf.Mesh('regular simplices', np.linspace(0, 2, 21), np.linspace(0, 1, 11))
m.set_region(1, m.outer_faces())
mf = gf.MeshFem(m, 1); mf.set_classical_fem(2)
# ... compute a solution U on mf ...

gfp.plot_mesh(m, regions=[1], vertices=True, show=True)
gfp.plot(mf, U, name="u", refine=4, title="Solution", show=True)
```

For a 3D elasticity problem with a displacement `U` on a vector `MeshFem` `mfu`:

```
# norm of the displacement on the deformed surface (interactive window)
gfp.plot(mfu, U, name="u", deformation=U, deformation_scale="20%")

# vertical component on the half y > 0.5, with arrows
gfp.plot(mfu, U, name="u", component=2, quiver=True,
         slice_op=('boundary', ('planar', +1, [0, 0.5, 0], [0, 1, 0])))
```

(continues on next page)

(continued from previous page)

```
# image file, without opening a window
gfp.plot(mfu, U, name="u", off_screen=True, screenshot="u.png")
```

A complete example is given by the test `check_plot.py` in the directory `interface/tests/python/` of the *GetFEM* distribution.

7.2 How it works

A field is not drawn directly: it is first evaluated on a *slice* of the mesh (see the `Slice` object), then the points, simplexes and values of the slice are given to the renderer as *numpy* arrays. This has several consequences:

- In 3D, only the surface is drawn: the default slice operation is `('boundary',)`, which keeps only the triangles of the skin of the domain. Any other slice operation can be passed with `slice_op` (planar cuts, iso-values, balls, intersections, ...).
- High order elements and curved geometric transformations are represented by refining each convex `refine` times (default 3 for fields).
- The points of the slice are not shared between convexes, so discontinuous fields keep their jumps.
- The deformation is applied by building the slice on the deformed mesh, so that the colour, the arrows and the mesh edges all follow it.

7.3 Plotting a field: `plot`

`gfp.plot(mf, U, **options)` draws the field `U` defined on `mf`:

option	meaning
`name`	name of the field, used in the colour bar
`component`	`None` or `'norm'` (norm of a vector field, the default), or the index of a component
`slice_op`, `refine`	slice operation and refinement (see above)
`cvlst`	restrict to a list of convexes or a region
`deformation`	displacement vector, defined on `deformation_mf` (default `mf`)
`deformation_scale`	a number, or `'p%'` for a maximal displacement of p% of the size of the mesh (default `'10%'`)
`mesh_edges`	draw the edges of the mesh (default `True`)
`faces`	draw the coloured surface (`False` to draw only the arrows or the edges)
`quiver`	draw arrows for a vector field; `quiver_count` sets their number
`contours`	draw this number of iso-value levels
`cmap`, `clim`	colour map and range of the colour bar
`title`, `axis_labels`	title and names of the axes
`edge_color`	colour of the mesh edges
`face_color`	colour of the faces when there is no data
`quiver_color`	colour of the arrows
`line_color`	colour of the curves (e.g. streamlines)
`line_width`	width of the curves
`show_axes`, `colorbar`	display the axes / the colour bar

With Matplotlib, `ax=` draws in existing axes and `(fig, ax)` is returned; use `show=True` to open the window. With PyVista, `plotter=` adds to an existing `pyvista.Plotter`, which is returned; `axes_style` is `'grid'` (default, graduated axes), `'bounds'` or `'triad'`, and `notebook`, `off_screen` and `screenshot` are passed to PyVista.

`gfp.plot_slice(sl, mf, U)` does the same for any `Slice` built by the user, or draws the slice itself when no field is given (a slice made of curves, such as streamlines, is drawn as tubes with PyVista).

Several slices can be combined in the same scene by passing the PyVista plotter (or the Matplotlib axes) returned by the first call:

```
pl = gfp.plot_slice(sl, mfp, P, name="p", backend="pyvista", show=False)
gfp.plot_slice(sl, mfu, U, faces=False, quiver=True, colorbar=False,␣
↪plotter=pl)
gfp.plot_slice(gf.Slice('streamlines', mfu, U, H), line_color="white",␣
↪plotter=pl)
pl.show()
```

See `demo_stokes_3D_tank_draw.py` in `interface/tests/python/` for a complete example. `gfp.available_backends()` returns the renderers which are installed.

7.4 Saving figures

In the interactive PyVista window, the key `i` saves the current view (after any rotation or zoom) in the files `getfem_plot_001.png`, `getfem_plot_002.png`, ... of the current directory, without overwriting existing files. This is set by the options `save_key`, `save_filename` (possibly with a counter field `{:03d}`) and `save_scale` (magnification of the image with respect to the window):

```
gfp.plot(mfu, U, name="u", save_key="s", save_filename="tank_{:02d}.pdf")
```

The format is given by the extension: bitmap (`png`, `jpg`, ...) or vector graphics (`svg`, `eps`, `ps`, `pdf`, `tex`). In vector formats, the colour gradients of the surfaces are less faithful than in a bitmap of high resolution (`save_scale=3` for instance).

From a script, `gfp.save_view(pl, "fig.png")` saves the view of a plotter `pl`. To keep the view chosen interactively, show the window with `pl.show(auto_close=False)`: after closing it, `gfp.save_view` saves the last view as a bitmap image, and `pl.camera_position` can be stored to reproduce the same view later off-screen. Images can also be produced without any window with `off_screen=True, screenshot="file.png"`.

The Matplotlib windows have their own button to save the figure, and `fig.savefig("file.pdf")` can be used in scripts.

7.5 Plotting a mesh: `plot_mesh`

`gfp.plot_mesh(m, **options)` draws a mesh (its skin in 3D). `m` can also be a `MeshFem`, which is required for the dof numbers.

option	meaning
`regions`	regions to highlight, each with its own colour and an entry in the legend
`vertices`	display the vertex numbers
`convexes`	display the convex numbers
`dof`	display the basic dof numbers at their nodes (the dofs sharing a node are displayed together)
`quality`	colour each convex with its quality (0 to 1)
`edges`, `faces`	draw the edges / fill the faces
`cvlst`	restrict to these convexes, or to these faces with a two-row array
`deformation`	as for `plot`; the numbers follow the deformed mesh

In 3D, only the numbers lying on the skin are displayed and PyVista hides the ones which are behind the surface.

7.6 Using the extracted data

The functions `gfp.slice_field` and `gfp.slice_mesh` only extract the data and return a `SliceData` object, with the attributes `points` (n x 3 array), `simplexes` (dictionary dimension -> array of point indices), `values` (n or n x q array), `edges` and `name`. It can be used with any other plotting tool, and `gfp.to_pyvista(data)` converts it to a `pyvista.UnstructuredGrid`.

CHAPTER

EIGHT

API REFERENCE

Please remember that this documentation is not self contained. You should in particular refer to the user documentation to have a more extensive description of the structures algorithms and concepts used.

8.1 ContStruct

class ContStruct(**args*)

GetFEM ContStruct object

This object serves for storing parameters and data used in numerical continuation of solution branches of models (for more details about continuation see the GetFEM user documentation).

General constructor for ContStruct objects

- `S = ContStruct(Model md, string dataname_parameter[,string dataname_init, string dataname_final, string dataname_current], scalar sc_fac[, ...])` The variable *dataname_parameter* should parametrise the model given by *md*. If the parameterization is done via a vector datum, *dataname_init* and *dataname_final* should store two given values of this datum determining the parameterization, and *dataname_current* serves for actual values of this datum. *sc_fac* is a scale factor involved in the weighted norm used in the continuation.

  Additional options:

  - **'lsolver', string SOLVER_NAME** name of the solver to be used for the incorporated linear systems (the default value is 'auto', which lets getfem choose itself); possible values are 'superlu', 'mumps' (if supported), 'cg/ildlt', 'gmres/ilu' and 'gmres/ilut';
  - **'h_init', scalar HIN** initial step size (the default value is 1e-2);
  - **'h_max', scalar HMAX** maximum step size (the default value is 1e-1);
  - **'h_min', scalar HMIN** minimum step size (the default value is 1e-5);
  - **'h_inc', scalar HINC** factor for enlarging the step size (the default value is 1.3);
  - **'h_dec', scalar HDEC** factor for diminishing the step size (the default value is 0.5);
  - **'max_iter', int MIT** maximum number of iterations allowed in the correction (the default value is 10);
  - **'thr_iter', int TIT** threshold number of iterations of the correction for enlarging the step size (the default value is 4);

- **'max_res', scalar RES** target residual value of a new point on the solution curve (the default value is 1e-6);
- **'max_diff', scalar DIFF** determines a convergence criterion for two consecutive points (the default value is 1e-6);
- **'min_cos', scalar MCOS** minimal value of the cosine of the angle between tangents to the solution curve at an old point and a new one (the default value is 0.9);
- **'max_res_solve', scalar RES_SOLVE** target residual value for the linear systems to be solved (the default value is 1e-8);
- **'singularities', int SING** activates tools for detection and treatment of singular points (1 for limit points, 2 for bifurcation points and points requiring special branching techniques);
- **'non-smooth'** determines that some special methods for non-smooth problems can be used;
- **'delta_max', scalar DMAX** maximum size of division for evaluating the test function on the convex combination of two augmented Jacobians that belong to different smooth pieces (the default value is 0.005);
- **'delta_min', scalar DMIN** minimum size of division for evaluating the test function on the convex combination (the default value is 0.00012);
- **'thr_var', scalar TVAR** threshold variation for refining the division (the default value is 0.02);
- **'nb_dir', int NDIR** total number of the linear combinations of one couple of reference vectors when searching for new tangent predictions during location of new one-sided branches (the default value is 40);
- **'nb_span', int NSPAN** total number of the couples of the reference vectors forming the linear combinations (the default value is 1);
- **'noisy' or 'very_noisy'** determines how detailed information has to be displayed during the continuation process (residual values etc.).

`Moore_Penrose_continuation`(*solution*, *parameter*, *tangent_sol*, *tangent_par*, *h*)

Compute one step of the Moore-Penrose continuation: Take the point given by *solution* and *parameter*, the tangent given by *tangent_sol* and *tangent_par*, and the step size *h*. Return a new point on the solution curve, the corresponding tangent, a step size for the next step and optionally the current step size. If the returned step size equals zero, the continuation has failed. Optionally, return the type of any detected singular point. NOTE: The new point need not to be saved in the model in the end!

`bifurcation_test_function`()

Return the last value of the bifurcation test function and eventually the whole calculated graph when passing between different sub-domains of differentiability.

`char`()

Output a (unique) string representation of the ContStruct.

This can be used for performing comparisons between two different ContStruct objects. This function is to be completed.

`compute_tangent`(*solution*, *parameter*, *tangent_sol*, *tangent_par*)

Compute and return an updated tangent.

display()
Display a short summary for a ContStruct object.

init_Moore_Penrose_continuation(*solution*, *parameter*, *init_dir*)
Initialise the Moore-Penrose continuation: Return a unit tangent to the solution curve at the point given by *solution* and *parameter*, and an initial step size for the continuation. Orientation of the computed tangent with respect to the parameter is determined by the sign of *init_dir*.

init_step_size()
Return an initial step size for continuation.

max_step_size()
Return the maximum step size for continuation.

min_step_size()
Return the minimum step size for continuation.

non_smooth_bifurcation_test(*solution1*, *parameter1*, *tangent_sol1*, *tangent_par1*, *solution2*, *parameter2*, *tangent_sol2*, *tangent_par2*)
Test for a non-smooth bifurcation point between the point given by *solution1* and *parameter1* with the tangent given by *tangent_sol1* and *tangent_par1* and the point given by *solution2* and *parameter2* with the tangent given by *tangent_sol2* and *tangent_par2*.

non_smooth_branching(*solution*, *parameter*, *tangent_sol*, *tangent_par*)
Approximate a non-smooth point close to the point given by *solution* and *parameter* and locate one-sided smooth solution branches emanating from there. Save the approximation of the non-smooth point as a singular point into the ContStruct object together with the array of the tangents to the located solution branches that direct away from the non-smooth point. It is supposed that the point given by *solution* and *parameter* is a point on a smooth solution branch within the distance equal to the minimum step size from the end point of this branch, and the corresponding tangent given by *tangent_sol* and *tangent_par* is directed towards the end point.

sing_data()
Return a singular point (*X*, *gamma*) stored in the ContStruct object and a couple of arrays (*T_X*, *T_gamma*) of tangents to all located solution branches that emanate from there.

step_size_decrement()
Return the decrement ratio of the step size for continuation.

step_size_increment()
Return the increment ratio of the step size for continuation.

8.2 CvStruct

class CvStruct(**args*)
GetFEM CvStruct object

General constructor for CvStruct objects

basic_structure()
Get the simplest convex structure.

For example, the 'basic structure' of the 6-node triangle, is the canonical 3-noded triangle.

char()
Output a string description of the CvStruct.

dim()
Get the dimension of the convex structure.

display()
displays a short summary for a CvStruct object.

face(*F*)
Return the convex structure of the face *F*.

facepts(*F*)
Return the list of point indices for the face *F*.

nbpts()
Get the number of points of the convex structure.

8.3 Eltm

class Eltm(**args*)
GetFEM Eltm object

This object represents a type of elementary matrix. In order to obtain a numerical value of these matrices, see MeshIm.eltm().

If you have very particular assembling needs, or if you just want to check the content of an elementary matrix, this function might be useful. But the generic assembly abilities of gf_asm(...) should suit most needs.

General constructor for Eltm objects

- `E = Eltm('base', Fem FEM)` return a descriptor for the integration of shape functions on elements, using the Fem *FEM*.
- `E = Eltm('grad', Fem FEM)` return a descriptor for the integration of the gradient of shape functions on elements, using the Fem *FEM*.
- `E = Eltm('hessian', Fem FEM)` return a descriptor for the integration of the hessian of shape functions on elements, using the Fem *FEM*.
- `E = Eltm('normal')` return a descriptor for the unit normal of convex faces.
- `E = Eltm('grad_geotrans')` return a descriptor to the gradient matrix of the geometric transformation.
- `E = Eltm('grad_geotrans_inv')` return a descriptor to the inverse of the gradient matrix of the geometric transformation (this is rarely used).
- `E = Eltm('product', Eltm A, Eltm B)` return a descriptor for the integration of the tensorial product of elementary matrices *A* and *B*.

8.4 Fem

class Fem(**args*)

GetFEM Fem object

This object represents a finite element method on a reference element.

General constructor for Fem objects

- `F = Fem('interpolated_fem', MeshFem mf_source, MeshIm mim_target, [ivec blocked_dofs[, bool caching]])` Build a special Fem which is interpolated from another MeshFem.

  Using this special finite element, it is possible to interpolate a given MeshFem *mf_source* on another mesh, given the integration method *mim_target* that will be used on this mesh.

  Note that this finite element may be quite slow or consume much memory depending if caching is used or not. By default *caching* is True

- `F = Fem('projected_fem', MeshFem mf_source, MeshIm mim_target, int rg_source, int rg_target[, ivec blocked_dofs[, bool caching]])` Build a special Fem which is interpolated from another MeshFem.

  Using this special finite element, it is possible to interpolate a given MeshFem *mf_source* on another mesh, given the integration method *mim_target* that will be used on this mesh.

  Note that this finite element may be quite slow or consume much memory depending if caching is used or not. By default *caching* is True

- `F = Fem(string fem_name)` The *fem_name* should contain a description of the finite element method. Please refer to the GetFEM manual (especially the description of finite element and integration methods) for a complete reference. Here is a list of some of them:
  - FEM_PK(n,k) : classical Lagrange element Pk on a simplex of dimension *n*.
  - FEM_PK_DISCONTINUOUS(n,k[,alpha]) : discontinuous Lagrange element Pk on a simplex of dimension *n*.
  - FEM_QK(n,k) : classical Lagrange element Qk on quadrangles, hexahedrons etc.
  - FEM_QK_DISCONTINUOUS(n,k[,alpha]) : discontinuous Lagrange element Qk on quadrangles, hexahedrons etc.
  - FEM_Q2_INCOMPLETE(n) : incomplete Q2 elements with 8 and 20 dof (serendipity Quad 8 and Hexa 20 elements).
  - FEM_PK_PRISM(n,k) : classical Lagrange element Pk on a prism of dimension *n*.
  - FEM_PK_PRISM_DISCONTINUOUS(n,k[,alpha]) : classical discontinuous Lagrange element Pk on a prism.
  - FEM_PK_WITH_CUBIC_BUBBLE(n,k) : classical Lagrange element Pk on a simplex with an additional volumic bubble function.
  - FEM_P1_NONCONFORMING : non-conforming P1 method on a triangle.
  - FEM_P1_BUBBLE_FACE(n) : P1 method on a simplex with an additional bubble function on face 0.
  - FEM_P1_BUBBLE_FACE_LAG : P1 method on a simplex with an additional lagrange dof on face 0.

- FEM_PK_HIERARCHICAL(n,k) : PK element with a hierarchical basis.
- FEM_QK_HIERARCHICAL(n,k) : QK element with a hierarchical basis.
- FEM_PK_PRISM_HIERARCHICAL(n,k) : PK element on a prism with a hierarchical basis.
- FEM_STRUCTURED_COMPOSITE(Fem f,k) : Composite Fem f on a grid with k divisions.
- FEM_PK_HIERARCHICAL_COMPOSITE(n,k,s) : Pk composite element on a grid with s subdivisions and with a hierarchical basis.
- FEM_PK_FULL_HIERARCHICAL_COMPOSITE(n,k,s) : Pk composite element with s subdivisions and a hierarchical basis on both degree and subdivision.
- FEM_PRODUCT(A,B) : tensorial product of two polynomial elements.
- FEM_HERMITE(n) : Hermite element P3 on a simplex of dimension $n = 1, 2, 3$.
- FEM_ARGYRIS : Argyris element P5 on the triangle.
- FEM_HCT_TRIANGLE : Hsieh-Clough-Tocher element on the triangle (composite P3 element which is C1), should be used with IM_HCT_COMPOSITE() integration method.
- FEM_QUADC1_COMPOSITE : Quadrilateral element, composite P3 element and C1 (16 dof).
- FEM_REDUCED_QUADC1_COMPOSITE : Quadrilateral element, composite P3 element and C1 (12 dof).
- FEM_RT0(n) : Raviart-Thomas element of order 0 on a simplex of dimension n.
- FEM_NEDELEC(n) : Nedelec edge element of order 0 on a simplex of dimension n.

Of course, you have to ensure that the selected fem is compatible with the geometric transformation: a Pk fem has no meaning on a quadrangle.

base_value(p)
Evaluate all basis functions of the FEM at point p.

p is supposed to be in the reference convex!

char()
Ouput a (unique) string representation of the Fem.

This can be used to perform comparisons between two different Fem objects.

dim()
Return the dimension (dimension of the reference convex) of the Fem.

display()
displays a short summary for a Fem object.

estimated_degree()
Return an estimation of the polynomial degree of the Fem.

This is an estimation for fem which are not polynomials.

grad_base_value(p)
Evaluate the gradient of all base functions of the Fem at point p.

p is supposed to be in the reference convex!

hess_base_value(*p*)
Evaluate the Hessian of all base functions of the Fem at point *p*.

p is supposed to be in the reference convex!

index_of_global_dof(*cv*)
Return the index of global dof for special fems such as interpolated fem.

is_equivalent()
Return 0 if the Fem is not equivalent.

Equivalent Fem are evaluated on the reference convex. This is the case of most classical Fem's.

is_lagrange()
Return 0 if the Fem is not of Lagrange type.

is_polynomial()
Return 0 if the basis functions are not polynomials.

nbdof(*cv=None*)
Return the number of dof for the Fem.

Some specific Fem (for example 'interpolated_fem') may require a convex number *cv* to give their result. In most of the case, you can omit this convex number.

poly_str()
Return the polynomial expressions of its basis functions in the reference convex. The result is expressed as a tuple of strings. Of course this will fail on non-polynomial Fem's.

pts(*cv=None*)
Get the location of the dof on the reference element.

Some specific Fem may require a convex number *cv* to give their result (for example 'interpolated_fem'). In most of the case, you can omit this convex number.

target_dim()
Return the dimension of the target space.

The target space dimension is usually 1, except for vector Fem.

8.5 GeoTrans

class GeoTrans(**args*)
GetFEM GeoTrans object

The geometric transformation must be used when you are building a custom mesh convex by convex (see the add_convex() function of Mesh): it also defines the kind of convex (triangle, hexahedron, prism, etc..)

General constructor for GeoTrans objects

- `GT = GeoTrans(string name)` The name argument contains the specification of the geometric transformation as a string, which may be:
  - GT_PK(n,k) : Transformation on simplexes, dim *n*, degree *k*.

- GT_QK(n,k) : Transformation on parallelepipeds, dim *n*, degree *k*.
- GT_PRISM(n,k) : Transformation on prisms, dim *n*, degree *k*.
- GT_PRODUCT(A,B) : Tensorial product of two transformations.
- GT_LINEAR_PRODUCT(GeoTrans gt1,GeoTrans gt2) : Linear tensorial product of two transformations

char()
Output a (unique) string representation of the GeoTrans.

This can be used to perform comparisons between two different GeoTrans objects.

dim()
Get the dimension of the GeoTrans.

This is the dimension of the source space, i.e. the dimension of the reference convex.

display()
displays a short summary for a GeoTrans object.

is_linear()
Return 0 if the GeoTrans is not linear.

nbpts()
Return the number of points of the GeoTrans.

normals()
Get the normals for each face of the reference convex of the GeoTrans.

The normals are stored in the columns of the output matrix.

pts()
Return the reference convex points of the GeoTrans.

The points are stored in the columns of the output matrix.

transform(*G*, *Pr*)
Apply the GeoTrans to a set of points.

G is the set of vertices of the real convex, *Pr* is the set of points (in the reference convex) that are to be transformed. The corresponding set of points in the real convex is returned.

8.6 GlobalFunction

class GlobalFunction(**args*)
GetFEM GlobalFunction object

Global function object is represented by three functions:

- The function *val*.
- The function gradient *grad*.
- The function Hessian *hess*.

this type of function is used as local and global enrichment function. The global function Hessian is an optional parameter (only for fourth order derivative problems).

General constructor for GlobalFunction objects

- `GF = GlobalFunction('cutoff', int fn, scalar r, scalar r1, scalar r0)` Create a cutoff global function.
- `GF = GlobalFunction('crack', int fn)` Create a near-tip asymptotic global function for modelling cracks.
- `GF = GlobalFunction('parser', string val[, string grad[, string hess]])` Create a global function from strings *val*, *grad* and *hess*. This function could be improved by using the derivation of the generic assembly language ... to be done.
- `GF = GlobalFunction('product', GlobalFunction F, GlobalFunction G)` Create a product of two global functions.
- `GF = GlobalFunction('add', GlobalFunction gf1, GlobalFunction gf2)` Create a add of two global functions.

char()
Output a (unique) string representation of the GlobalFunction.

This can be used to perform comparisons between two different GlobalFunction objects. This function is to be completed.

display()
displays a short summary for a GlobalFunction object.

grad(*PTs*)
Return *grad* function evaluation in *PTs* (column points).

On return, each column of *GRADs* is of the form [Gx,Gy].

hess(*PTs*)
Return *hess* function evaluation in *PTs* (column points).

On return, each column of *HESSs* is of the form [Hxx,Hxy,Hyx,Hyy].

val(*PTs*)
Return *val* function evaluation in *PTs* (column points).

8.7 Integ

class Integ(**args*)
GetFEM Integ object

General object for obtaining handles to various integrations methods on convexes (used when the elementary matrices are built).

General constructor for Integ objects

- `I = Integ(string method)` Here is a list of some integration methods defined in GetFEM (see the description of finite element and integration methods for a complete reference):
  - IM_EXACT_SIMPLEX(n) : Exact integration on simplices (works only with linear geometric transformations and PK Fem's).
  - IM_PRODUCT(A,B) : Product of two integration methods.
  - IM_EXACT_PARALLELEPIPED(n) : Exact integration on parallelepipeds.
  - IM_EXACT_PRISM(n) : Exact integration on prisms.

- IM_GAUSS1D(k) : Gauss method on the segment, order *k=1,3,...,99*.
- IM_NC(n,k) : Newton-Cotes approximative integration on simplexes, order *k*.
- IM_NC_PARALLELEPIPED(n,k) : Product of Newton-Cotes integration on parallelepipeds.
- IM_NC_PRISM(n,k) : Product of Newton-Cotes integration on prisms.
- IM_GAUSS_PARALLELEPIPED(n,k) : Product of Gauss1D integration on parallelepipeds.
- IM_TRIANGLE(k) : Gauss methods on triangles *k=1,3,5,6,7,8,9,10,13,17,19*.
- IM_QUAD(k) : Gauss methods on quadrilaterons *k=2,3,5, ...,17*. Note that IM_GAUSS_PARALLELEPIPED should be prefered for QK Fem's.
- IM_TETRAHEDRON(k) : Gauss methods on tetrahedrons *k=1,2,3,5,6 or 8*.
- IM_SIMPLEX4D(3) : Gauss method on a 4-dimensional simplex.
- IM_STRUCTURED_COMPOSITE(im,k) : Composite method on a grid with *k* divisions.
- IM_HCT_COMPOSITE(im) : Composite integration suited to the HCT composite finite element.

Example:

- I = Integ('IM_PRODUCT(IM_GAUSS1D(5),IM_GAUSS1D(5))')

is the same as:

- I = Integ('IM_GAUSS_PARALLELEPIPED(2,5)')

Note that 'exact integration' should be avoided in general, since they only apply to linear geometric transformations, are quite slow, and subject to numerical stability problems for high degree Fem's.

char()
Ouput a (unique) string representation of the integration method.

This can be used to comparisons between two different Integ objects.

coeffs()
Returns the coefficients associated to each integration point.

Only for approximate methods, this has no meaning for exact integration methods!

dim()
Return the dimension of the reference convex of the method.

display()
displays a short summary for a Integ object.

face_coeffs(*F*)
Returns the coefficients associated to each integration of a face.

Only for approximate methods, this has no meaning for exact integration methods!

face_pts(*F*)
Return the list of integration points for a face.

Only for approximate methods, this has no meaning for exact integration methods!

is_exact()
Return 0 if the integration is an approximate one.

nbpts()
Return the total number of integration points.

Count the points for the volume integration, and points for surface integration on each face of the reference convex.

Only for approximate methods, this has no meaning for exact integration methods!

pts()
Return the list of integration points

Only for approximate methods, this has no meaning for exact integration methods!

8.8 LevelSet

class LevelSet(**args*)
GetFEM LevelSet object

The level-set object is represented by a primary level-set and optionally a secondary level-set used to represent fractures (if p(x) is the primary level-set function and s(x) is the secondary level-set, the crack is defined by $p(x) = 0$ and $s(x) \leq 0$: the role of the secondary is to determine the crack front/tip).

note:

All tools listed below need the package qhull installed on your system. This package is widely available. It computes convex hull and delaunay triangulations in arbitrary dimension.

General constructor for LevelSet objects

- `LS = LevelSet(Mesh m, int d[, string 'ws'| string f1[, string f2 | string 'ws']])` Create a LevelSet object on a Mesh represented by a primary function (and optional secondary function, both) defined on a lagrange MeshFem of degree *d*.

  If *ws* (with secondary) is set; this levelset is represented by a primary function and a secondary function. If *f1* is set; the primary function is defined by that expression (with the syntax of the high generic assembly language). If *f2* is set; this levelset is represented by a primary function and a secondary function defined by these expressions.

char()
Output a (unique) string representation of the LevelSet.

This can be used to perform comparisons between two different LevelSet objects. This function is to be completed.

degree()
Return the degree of lagrange representation.

display()
displays a short summary for a LevelSet.

memsize()
Return the amount of memory (in bytes) used by the level-set.

mf()
Return a reference on the MeshFem object.

set_values(**args*)
Synopsis: LevelSet.set_values(self, {mat v1|string func_1}[, mat v2|string func_2])

Set values of the vector of dof for the level-set functions.

Set the primary function with the vector of dof *v1* (or the expression *func_1*) and the secondary function (if any) with the vector of dof *v2* (or the expression *func_2*)

simplify(*eps=0.01*)
Simplify dof of level-set optionally with the parameter *eps*.

values(*nls*)
Return the vector of dof for *nls* function.

If *nls* is 0, the method return the vector of dof for the primary level-set function. If *nls* is 1, the method return the vector of dof for the secondary level-set function (if any).

8.9 Mesh

class Mesh(**args*)
GetFEM Mesh object

This object is able to store any element in any dimension even if you mix elements with different dimensions.

General constructor for Mesh objects

- `M = Mesh('empty', int dim)` Create a new empty mesh.
- `M = Mesh('cartesian', vec X[, vec Y[, vec Z,..]])` Build quickly a regular mesh of quadrangles, cubes, etc.
- `M = Mesh('pyramidal', vec X[, vec Y[, vec Z,..]])` Build quickly a regular mesh of pyramids, etc.
- `M = Mesh('cartesian Q1', vec X, vec Y[, vec Z,..])` Build quickly a regular mesh of quadrangles, cubes, etc. with Q1 elements.
- `M = Mesh('triangles grid', vec X, vec Y)` Build quickly a regular mesh of triangles.

  This is a very limited and somehow deprecated function (See also `Mesh('ptND')`, `Mesh('regular simplices')` and `Mesh('cartesian')`).
- `M = Mesh('regular simplices', vec X[, vec Y[, vec Z,...]][,'degree', int k][,'noised'])` Mesh a n-dimensional parallelepiped with simplices (triangles, tetrahedrons etc) .

  The optional degree may be used to build meshes with non linear geometric transformations.
- `M = Mesh('curved', Mesh m, vec F)` Build a curved (n+1)-dimensions mesh from a n-dimensions mesh *m*.

  The points of the new mesh have one additional coordinate, given by the vector *F*. This can be used to obtain meshes for shells. *m* may be a MeshFem object, in that case its linked mesh will be used.

- `M = Mesh('prismatic', Mesh m, int nl[, int degree])` Extrude a prismatic Mesh *M* from a Mesh *m*.

  In the additional dimension there are *nl* layers of elements distributed from `0` to `1`. If the optional parameter *degree* is provided with a value greater than the default value of `1`, a non-linear transformation of corresponding degree is considered in the extrusion direction.

- `M = Mesh('pt2D', mat P, imat T[, int n])` Build a mesh from a 2D triangulation.

  Each column of *P* contains a point coordinate, and each column of *T* contains the point indices of a triangle. *n* is optional and is a zone number. If *n* is specified then only the zone number *n* is converted (in that case, *T* is expected to have 4 rows, the fourth containing these zone numbers).

- `M = Mesh('ptND', mat P, imat T)` Build a mesh from a n-dimensional "triangulation".

  Similar function to 'pt2D', for building simplexes meshes from a triangulation given in *T*, and a list of points given in *P*. The dimension of the mesh will be the number of rows of *P*, and the dimension of the simplexes will be the number of rows of *T*.

- `M = Mesh('load', string filename)` Load a mesh from a GetFEM ascii mesh file.

  See also `Mesh.save(string filename)`.

- `M = Mesh('from string', string s)` Load a mesh from a string description.

  For example, a string returned by `Mesh.char()`.

- `M = Mesh('import', string format, string filename)` Import a mesh from the file *filename*.

  format may be:

  - 'gmsh' for a mesh created with *Gmsh*
  - 'gmsh_with_lower_dim_elt' for a mesh created with *Gmsh* and including elements of lower dimension than the mesh
  - 'gid' for a mesh created with *GiD*
  - 'cdb' for a mesh created with *ANSYS*
  - 'am_fmt' for a mesh created with *EMC2*
  - 'structured' for a structured cartesian mesh. In this case *filename* is a string describing the mesh e.g. "GT='GT_QK(2,2)';ORG=[0,0];SIZES=[1,1];NSUBDIV=[5,10]"
  - 'structured_ball' for a structured mesh of a circular disc or a sphere. In this case *filename* is a string describing the mesh "GT='GT_QK(2,2)';ORG=[0,0];SIZES=[4];NSUBDIV=[3,4];SYMMETRIES=1" The number of symmetries divides a circle into a half (1) or quarter of circle (2). Similarly a sphere can be divided with up to 3 symmetry planes.
  - 'structured_ball_shell' for a structured mesh of a ball shell (a ring in 2D), e.g. "GT='GT_QK(2,2)';ORG=[0,0];SIZES=[4,1];NSUBDIV=[3,2]" with SIZES the outer radius and the thickness and NSUBDIV the circumferential and radial subdivisions (SYMMETRIES as for 'structured_ball'). With an ORG of dimension dim(GT)+1, the sphere (or circle) itself is meshed with elements of dimension dim(GT), for shells for instance, e.g. "GT='GT_QK(2,2)';ORG=[0,0,0];SIZES=[4];NSUBDIV=[3];SYMMETRIES=2"

- `M = Mesh('clone', Mesh m2)` Create a copy of a mesh.
- `M = Mesh('generate', MesherObject mo, scalar h[, int K = 1[, mat vertices]])` Call the experimental mesher of GetFEM on the geometry represented by *mo*. Please control the conformity of the produced mesh. You can help the mesher by adding a priori vertices in the array *vertices* which should be of size `n x m` where `n` n is the dimension of the mesh and `m` the number of points. *h* is approximate diameter of the elements. *K* is the degree of the mesh (>1 for curved boundaries). The mesher will try to optimize the quality of the elements. This operation may be time consuming. Note that if the mesh generation fails, because of some random procedure used, it can be run again since it will not give necessarily the same result due to random procedures used. The messages send to the console by the mesh generation can be deactivated using *gf_util('trace level', 2)*. More information can be obtained by *gf_util('trace level', 4)*. See `MesherObject` to manipulate geometric primitives in order to describe the geometry.

add_convex(*GT*, *PTS*)
Add a new convex into the mesh.

The convex structure (triangle, prism,...) is given by *GT* (obtained with GeoTrans('...')), and its points are given by the columns of *PTS*. On return, *CVIDs* contains the convex #ids. *PTS* might be a 3-dimensional array in order to insert more than one convex (or a two dimensional array correctly shaped according to Fortran ordering).

add_point(*PTS*)
Insert new points in the mesh and return their #ids.

PTS should be an `nxm` matrix , where `n` is the mesh dimension, and `m` is the number of points that will be added to the mesh. On output, *PIDs* contains the point #ids of these new points.

Remark: if some points are already part of the mesh (with a small tolerance of approximately `1e-8`), they won't be inserted again, and *PIDs* will contain the previously assigned #ids of these points.

adjacent_face(*cvid*, *fid*)
Return convex face of the neighbor element if it exists. If the convex have more than one neighbor relatively to the face `f` (think to bar elements in 3D for instance), return the first face found.

all_faces(*CVIDs=None*)
Return the set of faces of the in CVIDs (in all the mesh if CVIDs is omitted). Note that the face shared by two neighbor elements will be represented twice.

boundaries()
DEPRECATED FUNCTION. Use 'regions' instead.

boundary()
DEPRECATED FUNCTION. Use 'region' instead.

char()
Output a string description of the mesh.

convex_area(*CVIDs=None*)
Return an estimate of the area of each convex.

convex_radius(*CVIDs=None*)
Return an estimate of the radius of each convex.

convexes_in_box(*pmin*, *pmax*)
Return the set of convexes lying entirely within the box defined by the corner points *pmin* and *pmax*.

The output *CVIDs* is a two-rows matrix, the first row lists convex #ids, and the second one lists face numbers (local number in the convex). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

curved_edges(*N*, *CVLST=None*)
[OBSOLETE FUNCTION! will be removed in a future release]

Return E and C. More sophisticated version of Mesh.edges() designed for curved elements. This one will return N (N>=2) points of the (curved) edges. With N==2, this is equivalent to Mesh.edges(). Since the points are no more always part of the mesh, their coordinates are returned instead of points number, in the array E which is a [mesh_dim x 2 x nb_edges] array. If the optional output argument C is specified, it will contain the convex number associated with each edge.

cvid()
Return the list of all convex #id.

Note that their numbering is not supposed to be contiguous from 0 to Mesh.nbcvs()-1, especially if some points have been removed from the mesh. You can use Mesh.optimize_structure() to enforce a contiguous numbering.

cvid_from_pid(*PIDs*, *share=False*)
Return convex #ids related with the point #ids given in *PIDs*.

If *share=False*, search convex whose vertex #ids are in *PIDs*. If *share=True*, search convex #ids that share the point #ids given in *PIDs*. *CVIDs* is a vector (possibly empty).

cvstruct(*CVIDs=None*)
Return an array of the convex structures.

If *CVIDs* is not given, all convexes are considered. Each convex structure is listed once in *S*, and *CV2S* maps the convexes indice in *CVIDs* to the indice of its structure in *S*.

del_convex(*CVIDs*)
Remove one or more convexes from the mesh.

CVIDs should contain the convexes #ids, such as the ones returned by the 'add convex' command.

del_convex_of_dim(*DIMs*)
Remove all convexes of dimension listed in *DIMs*.

For example; `Mesh.del_convex_of_dim([1,2])` remove all line segments, triangles and quadrangles.

del_point(*PIDs*)
Removes one or more points from the mesh.

PIDs should contain the point #ids, such as the one returned by the 'add point' command.

delete_boundary(*rnum*, *CVFIDs*)
DEPRECATED FUNCTION. Use 'delete region' instead.

delete_region(*RIDs*)
Remove the regions whose #ids are listed in *RIDs*.

dim()
Get the dimension of the mesh (2 for a 2D mesh, etc).

display()
displays a short summary for a Mesh object.

edges(*CVLST=None*, **args*)
Synopsis: [E,C] = Mesh.edges(self [, CVLST][, 'merge'])

[OBSOLETE FUNCTION! will be removed in a future release]

Return the list of edges of mesh M for the convexes listed in the row vector CVLST. E is a 2 x nb_edges matrix containing point indices. If CVLST is omitted, then the edges of all convexes are returned. If CVLST has two rows then the first row is supposed to contain convex numbers, and the second face numbers, of which the edges will be returned. If 'merge' is indicated, all common edges of convexes are merged in a single edge. If the optional output argument C is specified, it will contain the convex number associated with each edge.

export_to_dx(*filename*, **args*)
Synopsis: Mesh.export_to_dx(self, string filename, ... ['ascii']['append']['as',string name,['serie',string serie_name]]['edges'])

Exports a mesh to an OpenDX file.

See also MeshFem.export_to_dx(), Slice.export_to_dx().

export_to_pos(*filename*, *name=None*)
Exports a mesh to a POS file .

See also MeshFem.export_to_pos(), Slice.export_to_pos().

export_to_vtk(*filename*, **args*)
Synopsis: Mesh.export_to_vtk(self, string filename, ... ['ascii']['quality'])

Exports a mesh to a VTK file .

If 'quality' is specified, an estimation of the quality of each convex will be written to the file.

See also MeshFem.export_to_vtk(), Slice.export_to_vtk().

export_to_vtu(*filename*, **args*)
Synopsis: Mesh.export_to_vtu(self, string filename, ... ['ascii']['quality'])

Exports a mesh to a VTK(XML) file .

If 'quality' is specified, an estimation of the quality of each convex will be written to the file.

See also MeshFem.export_to_vtu(), Slice.export_to_vtu().

extend_region(*rnum*, *CVFIDs*)
Extends the region identified by the region number *rnum* to include the set of convexes or/and convex faces provided in the matrix *CVFIDs*, see also `Mesh.(set region)`.

faces_from_cvid(*CVIDs=None*, **args*)
Synopsis: CVFIDs = Mesh.faces_from_cvid(self[, ivec CVIDs][, 'merge'])

Return a list of convex faces from a list of convex #id.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). If *CVIDs* is not given, all convexes are considered. The optional argument 'merge' merges faces shared by the convex of *CVIDs*.

faces_from_pid(*PIDs*)
Return the convex faces whose vertex #ids are in *PIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). For a convex face to be returned, EACH of its points have to be listed in *PIDs*.

geotrans(*CVIDs=None*)
Returns an array of the geometric transformations.

See also Mesh.cvstruct().

inner_faces(*CVIDs=None*)
Return the set of faces shared at least by two elements in CVIDs. Each face is represented only once and is arbitrarily chosen between the two neighbor elements.

max_cvid()
Return the maximum #id of all convexes in the mesh (see 'max pid').

max_pid()
Return the maximum #id of all points in the mesh (see 'max cvid').

memsize()
Return the amount of memory (in bytes) used by the mesh.

merge(*m2*, *tol=None*)
Merge with the Mesh *m2*.

Overlapping points, within a tolerance radius *tol*, will not be duplicated. If *m2* is a MeshFem object, its linked mesh will be used.

nbcvs()
Get the number of convexes of the mesh.

nbpts()
Get the number of points of the mesh.

normal_of_face(*cv*, *f*, *nfpt=None*)
Return the normal vector of convex *cv*, face *f* at the *nfpt* point of the face.

If *nfpt* is not specified, then the normal is evaluated at each geometrical node of the face.

normal_of_faces(*CVFIDs*)
Return matrix of (at face centers) the normal vectors of convexes.

CVFIDs is supposed a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex).

optimize_structure(*with_renumbering=None*)
Reset point and convex numbering.

After optimisation, the points (resp. convexes) will be consecutively numbered from `0` to `Mesh.max_pid()-1` (resp. `Mesh.max_cvid()-1`).

orphaned_pid()
Return point #id which are not linked to a convex.

outer_faces(*dim=None*, **args*)
Synopsis: CVFIDs = Mesh.outer_faces(self[, dim][, CVIDs])

Return the set of faces not shared by two elements.

The output *CVFIDs* is a two-rows matrix, the first row lists convex #ids, and the second one lists face numbers (local number in the convex). If *dim* is provided, the function is forced to detect faces of elements that have dimension *dim*, e.g. *dim `=2 will detect edges of surface elements, even if these belong to a 3D mesh. If `CVIDs* is not given, all convexes are considered, and the function basically returns the mesh boundary. If *CVIDs* is given, it returns the boundary of the convex set whose #ids are listed in *CVIDs*.

outer_faces_in_ball(*center*, *radius*, *dim=None*, **args*)

Synopsis: CVFIDs = Mesh.outer_faces_in_ball(self, vec center, scalar radius[, dim][, CVIDs])

Return the set of faces not shared by two convexes and lying within the ball of corresponding *center* and *radius*.

The output *CVFIDs* is a two-rows matrix, the first row lists convex #ids, and the second one lists face numbers (local number in the convex). The argument *dim* works as in outer_faces(). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

outer_faces_in_box(*pmin*, *pmax*, *dim=None*, **args*)

Synopsis: CVFIDs = Mesh.outer_faces_in_box(self, vec pmin, vec pmax[, dim][, CVIDs])

Return the set of faces not shared by two convexes and lying within the box defined by the corner points *pmin* and *pmax*.

The output *CVFIDs* is a two-rows matrix, the first row lists convex #ids, and the second one lists face numbers (local number in the convex). The argument *dim* works as in outer_faces(). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

outer_faces_with_direction(*v*, *angle*, *dim=None*, **args*)

Synopsis: CVFIDs = Mesh.outer_faces_with_direction(self, vec v, scalar angle[, dim][, CVIDs])

Return the set of faces not shared by two convexes and with a mean outward vector lying within an angle *angle* (in radians) from vector *v*.

The output *CVFIDs* is a two-rows matrix, the first row lists convex #ids, and the second one lists face numbers (local number in the convex). The argument *dim* works as in outer_faces(). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

pid()

Return the list of points #id of the mesh.

Note that their numbering is not supposed to be contiguous from 0 to Mesh.nbpts()-1, especially if some points have been removed from the mesh. You can use Mesh.optimize_structure() to enforce a contiguous numbering.

pid_from_coords(*PTS*, *radius=0*)

Return point #id whose coordinates are listed in *PTS*.

PTS is an array containing a list of point coordinates. On return, *PIDs* is a vector containing points #id for each point found in *eps* range, and -1 for those which where not found in the mesh.

pid_from_cvid(*CVIDs=None*)

Return the points attached to each convex of the mesh.

If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs* = *Mesh.max_cvid()*). *IDx* is a vector, length(IDx) = length(CVIDs)+1. *Pid* is a vector containing the concatenated list of #id of points of each convex in *CVIDs*. Each entry of *IDx* is the position of the corresponding convex point list in *Pid*. Hence, for example, the list of #id of points of the second convex is Pid[IDx(2):IDx(3)].

If *CVIDs* contains convex #id which do not exist in the mesh, their point list will be empty.

pid_in_cvids(*CVIDs*)
Return point #id listed in *CVIDs*.

PIDs is a vector containing points #id.

pid_in_faces(*CVFIDs*)
Return point #id listed in *CVFIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers. On return, *PIDs* is a vector containing points #id.

pid_in_regions(*RIDs*)
Return point #id listed in *RIDs*.

PIDs is a vector containing points #id.

pts(*PIDs=None*)
Return the list of point coordinates of the mesh.

Each column of the returned matrix contains the coordinates of one point. If the optional argument *PIDs* was given, only the points whose #id is listed in this vector are returned. Otherwise, the returned matrix will have Mesh.max_pid() columns, which might be greater than Mesh.nbpts() (if some points of the mesh have been destroyed and no call to Mesh.optimize_structure() have been issued). The columns corresponding to deleted points will be filled with NaN. You can use Mesh.pid() to filter such invalid points.

pts_from_cvid(*CVIDs=None*)
Search point listed in *CVID*.

Return *Pts* and *IDx*. If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs* = *Mesh.max_cvid()*). *IDx* is a vector, length(IDx) = length(CVIDs)+1. *Pts* is a vector containing the concatenated list of points of each convex in *CVIDs*. Each entry of *IDx* is the position of the corresponding convex point list in *Pts*. Hence, for example, the list of points of the second convex is Pts[:,IDx[2]:IDx[3]].

If *CVIDs* contains convex #id which do not exist in the mesh, their point list will be empty.

quality(*CVIDs=None*)
Return an estimation of the quality of each convex ($0 \leq Q \leq 1$).

refine(*CVIDs=None*)
Use a Bank strategy for mesh refinement.

If *CVIDs* is not given, the whole mesh is refined. Note that the regions, and the finite element methods and integration methods of the MeshFem and MeshIm objects linked to this mesh will be automagically refined.

region(*RIDs*)
Return the list of convexes/faces on the regions *RIDs*.

CVFIDs is a two-rows matrix, the first row lists convex #ids, and the second lists face numbers (local number in the convex). (and -1 when the whole convex is in the regions).

region_intersect(*r1*, *r2*)
Replace the region number *r1* with its intersection with region number *r2*.

region_merge(*r1*, *r2*)
Merge region number *r2* into region number *r1*.

region_subtract(*r1*, *r2*)
Replace the region number *r1* with its difference with region number *r2*.

regions()
Return the list of valid regions stored in the mesh.

save(*filename*)
Save the mesh object to an ascii file.

This mesh can be restored with Mesh('load', filename).

set_boundary(*rnum*, *CVFIDs*)
DEPRECATED FUNCTION. Use 'region' instead.

set_pts(*PTS*)
Replace the coordinates of the mesh points with those given in *PTS*.

set_region(*rnum*, *CVFIDs*)
Assigns the region number *rnum* to the set of convexes or/and convex faces provided in the matrix *CVFIDs*.

The first row of *CVFIDs* contains convex #ids, and the second row contains a face number in the convex (or `-1` for the whole convex (regions are usually used to store a list of convex faces, but you may also use them to store a list of convexes).

If a vector is provided (or a one row matrix) the region will represent the corresponding set of convex.

transform(*T*)
Applies the transformation matrix *T* to each point of the mesh.

Note that *T* is not required to be a `NxN` matrix (with `N = Mesh.dim()`). Hence it is possible to transform a 2D mesh into a 3D one (and reciprocally).

translate(*V*)
Translates each point of the mesh with the vector *V*.

triangulated_surface(*Nrefine*, *CVLIST=None*)
[DEPRECATED FUNCTION! will be removed in a future release]

Similar function to Mesh.curved_edges() : split (if necessary, i.e. if the geometric transformation if non-linear) each face into sub-triangles and return their coordinates in T (see also gf_compute('eval on P1 tri mesh'))

8.10 MeshFem

class MeshFem(**args*)

GetFEM MeshFem object

This object represents a finite element method defined on a whole mesh.

General constructor for MeshFem objects

- `MF = MeshFem(Mesh m[, int Qdim1=1[, int Qdim2=1, ...]])` Build a new MeshFem object.

  The *Qdim* parameters specifies the dimension of the field represented by the finite element method. Qdim1 = 1 for a scalar field, Qdim1 = n for a vector field off size n, Qdim1=m, Qdim2=n for a matrix field of size mxn ... Returns the handle of the created object.

- `MF = MeshFem('load', string fname[, Mesh m])` Load a MeshFem from a file.

  If the mesh *m* is not supplied (this kind of file does not store the mesh), then it is read from the file *fname* and its descriptor is returned as the second output argument.

- `MF = MeshFem('from string', string s[, Mesh m])` Create a MeshFem object from its string description.

  See also `MeshFem.char()`

- `MF = MeshFem('clone', MeshFem mf)` Create a copy of a MeshFem.

- `MF = MeshFem('sum', MeshFem mf1, MeshFem mf2[, MeshFem mf3[, ...]])` Create a MeshFem that spans two (or more) MeshFem's.

  All MeshFem must share the same mesh.

  After that, you should not modify the FEM of *mf1*, *mf2* etc.

- `MF = MeshFem('product', MeshFem mf1, MeshFem mf2)` Create a MeshFem that spans all the product of a selection of shape functions of *mf1* by all shape functions of *mf2*. Designed for Xfem enrichment.

  mf1 and *mf2* must share the same mesh.

  After that, you should not modify the FEM of *mf1*, *mf2*.

- `MF = MeshFem('levelset', MeshLevelSet mls, MeshFem mf)` Create a MeshFem that is conformal to implicit surfaces defined in MeshLevelSet.

- `MF = MeshFem('global function', Mesh m, LevelSet ls, (GlobalFunction GF1,...)[, int Qdim_m])` Create a MeshFem whose base functions are global function given by the user in the system of coordinate defined by the iso-values of the two level-set function of *ls*.

- `MF = MeshFem('bspline_uniform', Mesh m, int NX[, int NY[, int NZ]], int order[, string bcX_low[, string bcY_low[, string bcZ_low]][, string bcX_high[, string bcY_high[, string bcZ_high]]]])` Create a MeshFem on mesh *m*, whose base functions are global functions corresponding to bspline basis of order *order*, in an NX x NY x NZ grid (just NX in 1D or NX x NY in 2D) that spans the entire bounding box of *m*. Optionally boundary conditions at the edges of the domain can be defined with *bcX_low*, *bcY_low*, *bcZ_low*, *bcX_high*, *bcY_high*, and *bcZ_high* set to 'free' (default) or 'periodic' or 'symmetry'.

- `MF = MeshFem('partial', MeshFem mf, ivec DOFs[, ivec RCVs])` Build a restricted MeshFem by keeping only a subset of the degrees of freedom of *mf*.

  If *RCVs* is given, no FEM will be put on the convexes listed in *RCVs*.

adapt()
For a MeshFem levelset object only. Adapt the mesh_fem object to a change of the levelset function.

basic_dof_from_cv(*CVids*)
Return the dof of the convexes listed in *CVids*.

WARNING: the degrees of freedom might be returned in ANY order, do not use this function in your assembly routines. Use 'basic dof from cvid' instead, if you want to be able to map a convex number with its associated degrees of freedom.

One can also get the list of basic dof on a set on convex faces, by indicating on the second row of *CVids* the faces numbers (with respect to the convex number on the first row).

basic_dof_from_cvid(*CVids=None*)
Return the degrees of freedom attached to each convex of the mesh.

If *CVids* is omitted, all the convexes will be considered (equivalent to *CVids = 1 ... Mesh.max_cvid()*).

IDx is a vector, *length(IDx) = length(CVids)+1*. *DOFs* is a vector containing the concatenated list of dof of each convex in *CVids*. Each entry of *IDx* is the position of the corresponding convex point list in *DOFs*. Hence, for example, the list of points of the second convex is DOFs[IDx(2):IDx(3)].

If *CVids* contains convex #id which do not exist in the mesh, their point list will be empty.

basic_dof_nodes(*DOFids=None*)
Get location of basic degrees of freedom.

Return the list of interpolation points for the specified dof #IDs in *DOFids* (if *DOFids* is omitted, all basic dof are considered).

basic_dof_on_region(*Rs*)
Return the list of basic dof (before the optional reduction) lying on one of the mesh regions listed in *Rs*.

More precisely, this function returns the basic dof whose support is non-null on one of regions whose #ids are listed in *Rs* (note that for boundary regions, some dof nodes may not lie exactly on the boundary, for example the dof of Pk(n,0) lies on the center of the convex, but the base function in not null on the convex border).

char(*opt=None*)
Output a string description of the MeshFem.

By default, it does not include the description of the linked mesh object, except if *opt* is 'with_mesh'.

convex_index()
Return the list of convexes who have an FEM.

display()
displays a short summary for a MeshFem object.

dof_from_cv(*CVids*)
Deprecated function. Use MeshFem.basic_dof_from_cv() instead.

dof_from_cvid(*CVids=None*)
Deprecated function. Use MeshFem.basic_dof_from_cvid() instead.

dof_from_im(*mim*, *p=None*)
Return a selection of dof who contribute significantly to the mass-matrix that would be computed with *mf* and the integration method *mim*.

p represents the dimension on what the integration method operates (default *p* = *mesh dimension*).

IMPORTANT: you still have to set a valid integration method on the convexes which are not crosses by the levelset!

dof_nodes(*DOFids=None*)
Deprecated function. Use MeshFem.basic_dof_nodes() instead.

dof_on_region(*Rs*)
Return the list of dof (after the optional reduction) lying on one of the mesh regions listed in *Rs*.

More precisely, this function returns the basic dof whose support is non-null on one of regions whose #ids are listed in *Rs* (note that for boundary regions, some dof nodes may not lie exactly on the boundary, for example the dof of Pk(n,0) lies on the center of the convex, but the base function in not null on the convex border).

For a reduced mesh_fem a dof is lying on a region if its potential corresponding shape function is nonzero on this region. The extension matrix is used to make the correspondence between basic and reduced dofs.

dof_partition()
Get the 'dof_partition' array.

Return the array which associates an integer (the partition number) to each convex of the MeshFem. By default, it is an all-zero array. The degrees of freedom of each convex of the MeshFem are connected only to the dof of neighboring convexes which have the same partition number, hence it is possible to create partially discontinuous MeshFem very easily.

eval(*expression*, *gl={}*, *lo={}*)
interpolate an expression on the (lagrangian) MeshFem.

Examples:

```
mf.eval('x*y') # interpolates the function 'x*y'
mf.eval('[x,y]') # interpolates the vector field '[x,y]'

import numpy as np
mf.eval('np.sin(x)',globals(),locals()) # interpolates the function␣
↪sin(x)
```

export_to_dx(*filename*, **args*)
Synopsis: MeshFem.export_to_dx(self,string filename, ...['as', string mesh_name][,'edges']['serie',string serie_name][,'ascii'][,'append'], U, 'name'...)

Export a MeshFem and some fields to an OpenDX file.

This function will fail if the MeshFem mixes different convex types (i.e. quads and triangles), or if OpenDX does not handle a specific element type (i.e. prism connections are not known by OpenDX).

The FEM will be mapped to order 1 Pk (or Qk) FEMs. If you need to represent high-order FEMs or high-order geometric transformations, you should consider Slice.export_to_dx().

export_to_pos(*filename*, *name=None*, **args*)
Synopsis: MeshFem.export_to_pos(self,string filename[, string name][[,MeshFem mf1], mat U1, string nameU1[[,MeshFem mf2], mat U2, string nameU2,...]])

Export a MeshFem and some fields to a pos file.

The FEM and geometric transformations will be mapped to order 1 isoparametric Pk (or Qk) FEMs (as GMSH does not handle higher order elements).

export_to_vtk(*filename*, **args*)
Synopsis: MeshFem.export_to_vtk(self,string filename, ... ['ascii'], U, 'name'...)

Export a MeshFem and some fields to a vtk file.

The FEM and geometric transformations will be mapped to order 1 or 2 isoparametric Pk (or Qk) FEMs (as VTK does not handle higher order elements). If you need to represent high-order FEMs or high-order geometric transformations, you should consider Slice.export_to_vtk().

export_to_vtu(*filename*, **args*)
Synopsis: MeshFem.export_to_vtu(self,string filename, ... ['ascii'], U, 'name'...)

Export a MeshFem and some fields to a vtu file.

The FEM and geometric transformations will be mapped to order 1 or 2 isoparametric Pk (or Qk) FEMs (as VTK(XML) does not handle higher order elements). If you need to represent high-order FEMs or high-order geometric transformations, you should consider Slice.export_to_vtu().

extend_vector(*V*)
Multiply the provided vector V with the reduction matrix of the MeshFem.

extension_matrix()
Return the optional extension matrix.

fem(*CVids=None*)
Return a list of FEM used by the MeshFem.

FEMs is an array of all Fem objects found in the convexes given in *CVids*. If *CV2F* was supplied as an output argument, it contains, for each convex listed in *CVids*, the index of its correspounding FEM in *FEMs*.

Convexes which are not part of the mesh, or convexes which do not have any FEM have their correspounding entry in *CV2F* set to -1.

has_linked_mesh_levelset()
Is a mesh_fem_level_set or not.

interpolate_convex_data(*Ucv*)
Interpolate data given on each convex of the mesh to the MeshFem dof. The MeshFem has to be lagrangian, and should be discontinuous (typically an FEM_PK(N,0) or FEM_QK(N,0) should be used).

The last dimension of the input vector Ucv should have Mesh.max_cvid() elements.

Example of use: MeshFem.interpolate_convex_data(Mesh.quality())

is_equivalent(*CVids=None*)
Test if the MeshFem is equivalent.

See MeshFem.is_lagrangian()

is_lagrangian(*CVids=None*)
Test if the MeshFem is Lagrangian.

Lagrangian means that each base function Phi[i] is such that Phi[i](P[j]) = delta(i,j), where P[j] is the dof location of the jth base function, and delta(i,j) = 1 if i==j, else 0.

If *CVids* is omitted, it returns 1 if all convexes in the mesh are Lagrangian. If *CVids* is used, it returns the convex indices (with respect to *CVids*) which are Lagrangian.

is_polynomial(*CVids=None*)
Test if all base functions are polynomials.

See MeshFem.is_lagrangian()

is_reduced()
Return 1 if the optional reduction matrix is applied to the dofs.

linked_mesh()
Return a reference to the Mesh object linked to *mf*.

linked_mesh_levelset()
if it is a mesh_fem_level_set gives the linked mesh_level_set.

memsize()
Return the amount of memory (in bytes) used by the mesh_fem object.

The result does not take into account the linked mesh object.

mesh()
Return a reference to the Mesh object linked to *mf*. (identical to Mesh.linked_mesh())

nb_basic_dof()
Return the number of basic degrees of freedom (dof) of the MeshFem.

nbdof()
Return the number of degrees of freedom (dof) of the MeshFem.

non_conformal_basic_dof(*CVids=None*)
Return partially linked degrees of freedom.

Return the basic dof located on the border of a convex and which belong to only one convex, except the ones which are located on the border of the mesh. For example, if the convex 'a' and 'b' share a common face, 'a' has a P1 FEM, and 'b' has a P2 FEM, then the basic dof on the middle of the face will be returned by this function (this can be useful when searching the interfaces between classical FEM and hierarchical FEM).

non_conformal_dof(*CVids=None*)
Deprecated function. Use MeshFem.non_conformal_basic_dof() instead.

qdim()
Return the dimension Q of the field interpolated by the MeshFem.

By default, Q=1 (scalar field). This has an impact on the dof numbering.

reduce_meshfem(*RM*)
Set reduction mesh fem This function selects the degrees of freedom of the finite element method by selecting a set of independent vectors of the matrix RM. The number of columns of RM should correspond to the number of degrees of freedom of the finite element method.

reduce_vector(*V*)
Multiply the provided vector V with the extension matrix of the MeshFem.

reduction(*s*)
Set or unset the use of the reduction/extension matrices.

reduction_matrices(*R*, *E*)
Set the reduction and extension matrices and valid their use.

reduction_matrix()
Return the optional reduction matrix.

save(*filename*, *opt=None*)
Save a MeshFem in a text file (and optionally its linked mesh object if *opt* is the string 'with_mesh').

set_classical_discontinuous_fem(*k*, **args*)
Synopsis: MeshFem.set_classical_discontinuous_fem(self, int k[[, 'complete'], @tscalar alpha[, ivec CVIDX]])

Assigns a classical (Lagrange polynomial) discontinuous fem of order k.

Similar to MeshFem.set_classical_fem() except that FEM_PK_DISCONTINUOUS is used. Param *alpha* the node inset, $0 \leq alpha < 1$, where 0 implies usual dof nodes, greater values move the nodes toward the center of gravity, and 1 means that all degrees of freedom collapse on the center of gravity. The option 'complete' requests complete Langrange polynomial elements, even if the element geometric transformation is an incomplete one (e.g. 8-node quadrilateral or 20-node hexahedral).

set_classical_fem(*k*, **args*)
Synopsis: MeshFem.set_classical_fem(self, int k[[, 'complete'], ivec CVids])

Assign a classical (Lagrange polynomial) fem of order *k* to the MeshFem. The option 'complete' requests complete Langrange polynomial elements, even if the element geometric transformation is an incomplete one (e.g. 8-node quadrilateral or 20-node hexahedral).

Uses FEM_PK for simplexes, FEM_QK for parallelepipeds etc.

set_dof_partition(*DOFP*)
Change the 'dof_partition' array.

DOFP is a vector holding a integer value for each convex of the MeshFem. See MeshFem.dof_partition() for a description of "dof partition".

set_enriched_dofs(*DOFs*)
For a MeshFem product object only. Set te enriched dofs and adapt the MeshFem product.

set_fem(*f*, *CVids=None*)
Set the Finite Element Method.

Assign an FEM *f* to all convexes whose #ids are listed in *CVids*. If *CVids* is not given, the integration is assigned to all convexes.

See the help of Fem to obtain a list of available FEM methods.

set_partial(*DOFs*, *RCVs=None*)
Can only be applied to a partial MeshFem. Change the subset of the degrees of freedom of *mf*.

If *RCVs* is given, no FEM will be put on the convexes listed in *RCVs*.

set_qdim(*Q*)
Change the *Q* dimension of the field that is interpolated by the MeshFem.

$Q = 1$ means that the MeshFem describes a scalar field, $Q = N$ means that the MeshFem describes a vector field of dimension N.

8.11 MeshIm

class MeshIm(**args*)
GetFEM MeshIm object

This object represents an integration method defined on a whole mesh (and potentially on its boundaries).

General constructor for MeshIm objects

- `MIM = MeshIm('load', string fname[, Mesh m])` Load a MeshIm from a file.

  If the mesh *m* is not supplied (this kind of file does not store the mesh), then it is read from the file and its descriptor is returned as the second output argument.

- `MIM = MeshIm('from string', string s[, Mesh m])` Create a MeshIm object from its string description.

  See also `MeshIm.char()`

- `MIM = MeshIm('clone', MeshIm mim)` Create a copy of a MeshIm.

- `MIM = MeshIm('levelset', MeshLevelSet mls, string where, Integ im[, Integ im_tip[, Integ im_set]])` Build an integration method conformal to a partition defined implicitly by a levelset.

  The *where* argument defines the domain of integration with respect to the levelset, it has to be chosen among 'ALL', 'INSIDE', 'OUTSIDE' and 'BOUNDARY'.

  It can be completed by a string defining the boolean operation to define the integration domain when there is more than one levelset.

  The syntax is very simple, for example if there are 3 different levelsets,

  "a*b*c" is the intersection of the domains defined by each levelset (this is the default behaviour if this function is not called).

  "a+b+c" is the union of their domains.

  "c-(a+b)" is the domain of the third levelset minus the union of the domains of the two others.

  "!a" is the complementary of the domain of a (i.e. it is the domain where a(x)>0)

  The first levelset is always referred to with "a", the second with "b", and so on.

  for intance INSIDE(a*b*c)

CAUTION: this integration method will be defined only on elements cut by the level-set. For the 'ALL', 'INSIDE' and 'OUTSIDE' options it is mandatory to use the method `MeshIm.set_integ()` to define the integration method on the remaining elements.

- `MIM = MeshIm(Mesh m, [{Integ im|int im_degree}])` Build a new MeshIm object.

  For convenience, optional arguments (*im* or *im_degree*) can be provided, in that case a call to `MeshIm.integ()` is issued with these arguments.

adapt()
For a MeshIm levelset object only. Adapt the integration methods to a change of the levelset function.

char()
Output a string description of the MeshIm.

By default, it does not include the description of the linked Mesh object.

convex_index()
Return the list of convexes who have a integration method.

Convexes who have the dummy IM_NONE method are not listed.

display()
displays a short summary for a MeshIm object.

eltm(*em*, *cv*, *f=None*)
Return the elementary matrix (or tensor) integrated on the convex *cv*.

WARNING

Be sure that the fem used for the construction of *em* is compatible with the fem assigned to element *cv* ! This is not checked by the function ! If the argument *f* is given, then the elementary tensor is integrated on the face *f* of *cv* instead of the whole convex.

im_nodes(*CVids=None*)
Return the coordinates of the integration points, with their weights.

CVids may be a list of convexes, or a list of convex faces, such as returned by Mesh.region()

WARNING

Convexes which are not part of the mesh, or convexes which do not have an approximate integration method do not have their corresponding entry (this has no meaning for exact integration methods!).

integ(*CVids=None*)
Return a list of integration methods used by the MeshIm.

I is an array of all Integ objects found in the convexes given in *CVids*. If *CV2I* was supplied as an output argument, it contains, for each convex listed in *CVids*, the index of its correspounding integration method in *I*.

Convexes which are not part of the mesh, or convexes which do not have any integration method have their correspounding entry in *CV2I* set to -1.

linked_mesh()
Returns a reference to the Mesh object linked to *mim*.

memsize()
Return the amount of memory (in bytes) used by the MeshIm object.

The result does not take into account the linked Mesh object.

save(*filename*)
Saves a MeshIm in a text file (and optionally its linked mesh object).

set_integ(**args*)
Synopsis: MeshIm.set_integ(self,{Integ im|int im_degree}[, ivec CVids])

Set the integration method.

Assign an integration method to all convexes whose #ids are listed in *CVids*. If *CVids* is not given, the integration is assigned to all convexes. It is possible to assign a specific integration method with an integration method handle *im* obtained via Integ('IM_SOMETHING'), or to let getfem choose a suitable integration method with *im_degree* (choosen such that polynomials of degree $\leq$ im_degree are exactly integrated. If *im_degree=-1*, then the dummy integration method IM_NONE will be used.)

8.12 MeshImData

class MeshImData(**args*)
GetFEM MeshImData object

This object represents data defined on a mesh_im object.

General constructor for MeshImData objects

- `MIMD = MeshImData(MeshIm mim, int region, ivec size)` Build a new MeshImd object linked to a MeshIm object. If *region* is provided, considered integration points are filtered in this region. *size* is a vector of integers that specifies the dimensions of the stored data per integration point. If not given, the scalar stored data are considered.

display()
displays a short summary for a MeshImd object.

linked_mesh()
Returns a reference to the Mesh object linked to *mim*.

nb_tensor_elements()
Output the size of the stored data (per integration point).

nbpts()
Output the number of integration points (filtered in the considered region).

region()
Output the region that the MeshImd is restricted to.

set_region(*rnum*)
Set the considered region to *rnum*.

set_tensor_size()
Set the size of the data per integration point.

tensor_size()
Output the dimensions of the stored data (per integration point).

8.13 MeshLevelSet

class MeshLevelSet(**args*)

GetFEM MeshLevelSet object

General constructor for mesh_levelset objects. The role of this object is to provide a mesh cut by a certain number of level_set. This object is used to build conformal integration method (object mim and enriched finite element methods (Xfem)).

General constructor for MeshLevelSet objects

- `MLS = MeshLevelSet(Mesh m)` Build a new MeshLevelSet object from a Mesh and returns its handle.

adapt()

Do all the work (cut the convexes with the levelsets).

To initialice the MeshLevelSet object or to actualize it when the value of any levelset function is modified, one has to call this method.

add(*ls*)

Add a link to the LevelSet *ls*.

Only a reference is kept, no copy is done. In order to indicate that the linked Mesh is cut by a LevelSet one has to call this method, where *ls* is an LevelSet object. An arbitrary number of LevelSet can be added.

WARNING

The Mesh of *ls* and the linked Mesh must be the same.

char()

Output a (unique) string representation of the MeshLevelSetn.

This can be used to perform comparisons between two different MeshLevelSet objects. This function is to be completed.

crack_tip_convexes()

Return the list of convex #id's of the linked Mesh on which have a tip of any linked LevelSet's.

cut_mesh()

Return a Mesh cut by the linked LevelSet's.

display()

displays a short summary for a MeshLevelSet object.

levelsets()

Return a list of references to the linked LevelSet's.

linked_mesh()

Return a reference to the linked Mesh.

memsize()

Return the amount of memory (in bytes) used by the MeshLevelSet.

nb_ls()

Return the number of linked LevelSet's.

sup(*ls*)

Remove a link to the LevelSet *ls*.

8.14 MesherObject

class MesherObject(**args*)

GetFEM MesherObject object

This object represents a geometric object to be meshed by the experimental meshing procedure of Getfem.

General constructor for MesherObject objects

- `MF = MesherObject('ball', vec center, scalar radius)` Represents a ball of corresponding center and radius.
- `MF = MesherObject('half space', vec origin, vec normal_vector)` Represents an half space delimited by the plane which contains the origin and normal to *normal_vector*. The selected part is the part in the direction of the normal vector. This allows to cut a geometry with a plane for instance to build a polygon or a polyhedron.
- `MF = MesherObject('cylinder', vec origin, vec n, scalar length, scalar radius)` Represents a cylinder (in any dimension) of a certain radius whose axis is determined by the origin, a vector *n* and a certain length.
- `MF = MesherObject('cone', vec origin, vec n, scalar length, scalar half_angle)` Represents a cone (in any dimension) of a certain half-angle (in radians) whose axis is determined by the origin, a vector *n* and a certain length.
- `MF = MesherObject('torus', scalar R, scalar r)` Represents a torus in 3d of axis along the z axis with a great radius equal to *R* and small radius equal to *r*. For the moment, the possibility to change the axis is not given.
- `MF = MesherObject('rectangle', vec rmin, vec rmax)` Represents a rectangle (or parallelepiped in 3D) parallel to the axes.
- `MF = MesherObject('intersect', MesherObject object1 , MesherObject object2, ...)` Intersection of several objects.
- `MF = MesherObject('union', MesherObject object1 , MesherObject object2, ...)` Union of several objects.
- `MF = MesherObject('set minus', MesherObject object1 , MesherObject object2)` Geometric object being object1 minus object2.

char()

Output a (unique) string representation of the MesherObject.

This can be used to perform comparisons between two different MesherObject objects. This function is to be completed.

display()

displays a short summary for a MesherObject object.

8.15 Model

class Model(**args*)

GetFEM Model object

Model variables store the variables and the state data and the description of a model. This includes the global tangent matrix, the right hand side and the constraints. There are two kinds of models, the *real* and the *complex* models.

General constructor for Model objects

- `MD = Model('real')` Build a model for real unknowns.
- `MD = Model('complex')` Build a model for complex unknowns.

Neumann_term(*varname*, *region*)

Gives the assembly string corresponding to the Neumann term of the fem variable *varname* on *region*. It is deduced from the assembly string declared by the model bricks. *region* should be the index of a boundary region on the mesh where *varname* is defined. Care to call this function only after all the volumic bricks have been declared. Complains, if a brick omit to declare an assembly string.

add_Dirichlet_condition_with_Nitsche_method(*mim*, *varname*, *Neumannterm*, *datagamma0*, *region*, *theta=None*, **args*)

Synopsis: ind = Model.add_Dirichlet_condition_with_Nitsche_method(self, MeshIm mim, string varname, string Neumannterm, string datagamma0, int region[, scalar theta][, string dataname])

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This region should be a boundary. *Neumannterm* is the expression of the Neumann term (obtained by the Green formula) described as an expression of the high-level generic assembly language. This term can be obtained by Model.Neumann_term(varname, region) once all volumic bricks have been added to the model. The Dirichlet condition is prescribed with Nitsche's method. *datag* is the optional right hand side of the Dirichlet condition. *datagamma0* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* (default) is the simplest method for which the second derivative of the Neumann term is not necessary even for nonlinear problems. Return the brick index in the model.

add_Dirichlet_condition_with_multipliers(*mim*, *varname*, *mult_description*, *region*, *dataname=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with a multiplier variable described by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. Return the brick index in the model.

add_Dirichlet_condition_with_penalization(*mim*, *varname*, *coeff*, *region*, *dataname=None*, *mf_mult=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with penalization. The penalization coefficient is initially *coeff* and will be added to the data of the model. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. *mf_mult* is an optional parameter which allows to weaken the Dirichlet condition specifying a multiplier space. Return the brick index in the model.

add_Dirichlet_condition_with_simplification(*varname*, *region*, *dataname=None*)

Adds a (simple) Dirichlet condition on the variable *varname* and the mesh region *region*. The Dirichlet condition is prescribed by a simple post-treatment of the final linear system (tangent system for nonlinear problems) consisting of modifying the lines corresponding to the degree of freedom of the variable on *region* (0 outside the diagonal, 1 on the diagonal of the matrix and the expected value on the right hand side). The symmetry of the linear system is kept if all other bricks are symmetric. This brick is to be reserved for simple Dirichlet conditions (only dof declared on the corresponding boundary are prescribed). The application of this brick on reduced dof may be problematic. Intrinsic vectorial finite element method are not supported. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant (but in that case, it can only be applied to Lagrange f.e.m.) or (important) described on the same finite element method as *varname*. Returns the brick index in the model.

add_Fourier_Robin_brick(*mim*, *varname*, *dataexpr*, *region*)

Add a Fourier-Robin term to the model relatively to the variable *varname*. This corresponds to a weak term of the form $\int (qu).v$. *dataexpr* is the parameter q of the Fourier-Robin condition. It can be an arbitrary valid expression of the high-level generic assembly language (except for the complex version for which it should be a data of the model). *region* is the mesh region on which the term is added. Return the brick index in the model.

add_HHO_reconstructed_gradient(*transname*)

Add to the model the elementary transformation corresponding to the reconstruction of a gradient for HHO methods. The name is the name given to the elementary transformation.

add_HHO_reconstructed_symmetrized_gradient(*transname*)

Add to the model the elementary transformation corresponding to the reconstruction of a symmetrized gradient for HHO methods. The name is the name given to the elementary transformation.

add_HHO_reconstructed_symmetrized_value(*transname*)

Add to the model the elementary transformation corresponding to the reconstruction of the variable for HHO methods using a symmetrized gradient. The name is the name given to the elementary transformation.

add_HHO_reconstructed_value(*transname*)

Add to the model the elementary transformation corresponding to the reconstruction of the variable for HHO methods. The name is the name given to the elementary transformation.

add_HHO_stabilization(*transname*)

Add to the model the elementary transformation corresponding to the HHO stabilization operator. The name is the name given to the elementary transformation.

add_HHO_symmetrized_stabilization(*transname*)

Add to the model the elementary transformation corresponding to the HHO stabilization operator using a symmetrized gradient. The name is the name given to the elementary transfor-

mation.

add_Helmholtz_brick(*mim*, *varname*, *dataexpr*, *region=None*)
Add a Helmholtz term to the model relatively to the variable *varname*. *dataexpr* is the wave number. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_Houbolt_scheme(*varname*)
Attach a Houbolt method for the time discretization of the variable *varname*. Valid only if there is at most second order time derivative of the variable.

add_Kirchhoff_Love_Neumann_term_brick(*mim*, *varname*, *dataname_M*, *dataname_divM*, *region*)
Add a Neumann term brick for Kirchhoff-Love model on the variable *varname* and the mesh region *region*. *dataname_M* represents the bending moment tensor and *dataname_divM* its divergence. Return the brick index in the model.

add_Kirchhoff_Love_plate_brick(*mim*, *varname*, *dataname_D*, *dataname_nu*, *region=None*)
Add a bilaplacian brick on the variable *varname* and on the mesh region *region*. This represent a term $\Delta(D\Delta u)$ where $D(x)$ is a the flexion modulus determined by *dataname_D*. The term is integrated by part following a Kirchhoff-Love plate model with *dataname_nu* the poisson ratio. Return the brick index in the model.

add_Laplacian_brick(*mim*, *varname*, *region=None*)
Add a Laplacian term to the model relatively to the variable *varname* (in fact with a minus : $-\text{div}(\nabla u)$). If this is a vector valued variable, the Laplacian term is added componentwise. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_Mindlin_Reissner_plate_brick(*mim*, *mim_reduced*, *varname_u3*, *varname_theta*, *param_E*, *param_nu*, *param_epsilon*, *param_kappa*, *variant=None*, **args*)
Synopsis: ind = Model.add_Mindlin_Reissner_plate_brick(self, MeshIm mim, MeshIm mim_reduced, string varname_u3, string varname_theta , string param_E, string param_nu, string param_epsilon, string param_kappa [,int variant [, int region]])

Add a term corresponding to the classical Reissner-Mindlin plate model for which *varname_u3* is the transverse displacement, *varname_theta* the rotation of fibers normal to the midplane, 'param_E' the Young Modulus, *param_nu* the poisson ratio, *param_epsilon* the plate thickness, *param_kappa* the shear correction factor. Note that since this brick uses the high level generic assembly language, the parameter can be regular expression of this language. There are three variants. *variant = 0* corresponds to the an unreduced formulation and in that case only the integration method *mim* is used. Practically this variant is not usable since it is subject to a strong locking phenomenon. *variant = 1* corresponds to a reduced integration where *mim* is used for the rotation term and *mim_reduced* for the transverse shear term. *variant = 2* (default) corresponds to the projection onto a rotated RT0 element of the transverse shear term. For the moment, this is adapted to quadrilateral only (because it is not sufficient to remove the locking phenomenon on triangle elements). Note also that if you use high order elements, the projection on RT0 will reduce the order of the approximation. Returns the brick index in the model.

add_Newmark_scheme(*varname*, *beta*, *gamma*)
Attach a theta method for the time discretization of the variable *varname*. Valid only if there is at most second order time derivative of the variable.

add_Nitsche_contact_with_rigid_obstacle_brick(*mim*, *varname*, *Neumannterm*, *dataname_obstacle*, *gamma0name*, *region*, *theta=None*, **args*)

Synopsis: ind = Model.add_Nitsche_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname, string Neumannterm, string dataname_obstacle, string gamma0name, int region[, scalar theta[, string dataname_friction_coeff[, string dataname_alpha, string dataname_wt]]])

Adds a contact condition with or without Coulomb friction on the variable *varname* and the mesh boundary *region*. The contact condition is prescribed with Nitsche's method. The rigid obstacle should be described with the data *dataname_obstacle* being a signed distance to the obstacle (interpolated on a finite element method). *gamma0name* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* is the simplest method for which the second derivative of the Neumann term is not necessary. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method. CAUTION: This brick has to be added in the model after all the bricks corresponding to partial differential terms having a Neumann term. Moreover, This brick can only be applied to bricks declaring their Neumann terms. Returns the brick index in the model.

add_Nitsche_fictitious_domain_contact_brick(*mim*, *varname1*, *varname2*, *dataname_d1*, *dataname_d2*, *gamma0name*, *theta=None*, **args*)

Synopsis: ind = Model.add_Nitsche_fictitious_domain_contact_brick(self, MeshIm mim, string varname1, string varname2, string dataname_d1, string dataname_d2, string gamma0name [, scalar theta[, string dataname_friction_coeff[, string dataname_alpha, string dataname_wt1,string dataname_wt2]]])

Adds a contact condition with or without Coulomb friction between two bodies in a fictitious domain. The contact condition is applied on the variable *varname_u1* corresponds with the first and slave body with Nitsche's method and on the variable *varname_u2* corresponds with the second and master body with Nitsche's method. The contact condition is evaluated on the fictitious slave boundary. The first body should be described by the level-set *dataname_d1* and the second body should be described by the level-set *dataname_d2*. *gamma0name* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* is the simplest method for which the second derivative of the Neumann term is not necessary. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method. CAUTION: This brick has to be added in the model after all the bricks corresponding to partial differential terms having a Neumann term. Moreover, This brick can only be applied to bricks declaring their Neumann terms. Returns the brick index in the model.

add_Nitsche_large_sliding_contact_brick_raytracing(*unbiased_version*, *dataname_r*, *release_distance*, *dataname_fr=None*, **args*)

Synopsis: ind = Model.add_Nitsche_large_sliding_contact_brick_raytracing(self, bool unbiased_version, string dataname_r, scalar release_distance[, string dataname_fr[, string dataname_alpha[, int version]]])

Adds a large sliding contact with friction brick to the model based on the Nitsche's method. This brick is able to deal with self-contact, contact between several deformable bodies and contact with rigid obstacles. It uses the high-level generic assembly. It adds to the model a raytracing_interpolate_transformation object. "unbiased_version" refers to the version of Nische's method to be used. (unbiased or biased one). For each slave boundary a material law should be defined as a function of the dispacement variable on this boundary. The release distance should be determined with care (generally a few times a mean element size, and less than the thickness of the body). Initially, the brick is added with no contact boundaries. The contact boundaries and rigid bodies are added with special functions. *version* is 0 (the default value) for the non-symmetric version and 1 for the more symmetric one (not fully symmetric even without friction).

add_Nitsche_midpoint_contact_with_rigid_obstacle_brick(*mim*, *varname*, *Neumannterm*, *Neumannterm_wt*, *dataname_obstacle*, *gamma0name*, *region*, *theta*, *dataname_friction_coeff*, *dataname_alpha*, *dataname_wt*)

EXPERIMENTAL BRICK: for midpoint scheme only !! Adds a contact condition with or without Coulomb friction on the variable *varname* and the mesh boundary *region*. The contact condition is prescribed with Nitsche's method. The rigid obstacle should be described with the data *dataname_obstacle* being a signed distance to the obstacle (interpolated on a finite element method). *gamma0name* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* is the simplest method for which the second derivative of the Neumann term is not necessary. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method. Returns the brick index in the model.

add_assembly_assignment(*dataname*, *expression*, *region=None*, **args*)

Synopsis: Model.add_assembly_assignment(self, string dataname, string expression[, int region[, int order[, int before]]])

Adds expression *expr* to be evaluated at assembly time and being assigned to the data *dataname* which has to be of im_data type. This allows for instance to store a sub-expression of an assembly computation to be used on an other assembly. It can be used for instance to store the plastic strain in plasticity models. *order* represents the order of assembly where this assignement has to be done (potential(0), weak form(1) or tangent system(2) or at each order(-1)). The default value is 1. If before = 1, the the assignement is perfromed before the computation of the other assembly terms, such that the data can be used in the remaining of the assembly as an intermediary result (be careful that it is still considered as a data, no derivation of the expression is performed for the tangent system). If before = 0 (default), the assignement is done after the assembly terms.

add_basic_contact_brick(*varname_u*, *multname_n*, *multname_t=None*, **args*)

Synopsis: ind = Model.add_basic_contact_brick(self, string varname_u, string multname_n[, string multname_t], string dataname_r, Spmat BN[, Spmat BT, string dataname_friction_coeff][, string dataname_gap[, string dataname_alpha[, int augmented_version[, string dataname_gamma, string dataname_wt]]])

Add a contact with or without friction brick to the model. If U is the vector of degrees of freedom on which the unilateral constraint is applied, the matrix *BN* have to be such that this constraint is defined by $B_N U \leq 0$. A friction condition can be considered by adding the three parameters *multname_t*, *BT* and *dataname_friction_coeff*. In this case, the tangential displacement is $B_T U$ and the matrix *BT* should have as many rows as *BN* multiplied by $d-1$ where d is the domain dimension. In this case also, *dataname_friction_coeff* is a data which represents the coefficient of friction. It can be a scalar or a vector representing a value on each contact condition. The unilateral constraint is prescribed thank to a multiplier *multname_n* whose dimension should be equal to the number of rows of *BN*. If a friction condition is added, it is prescribed with a multiplier *multname_t* whose dimension should be equal to the number of rows of *BT*. The augmentation parameter *r* should be chosen in a range of acceptabe values (see Getfem user documentation). *dataname_gap* is an optional parameter representing the initial gap. It can be a single value or a vector of value. *dataname_alpha* is an optional homogenization parameter for the augmentation parameter (see Getfem user documentation). The parameter *augmented_version* indicates the augmentation strategy : 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one (except for the coupling between contact and Coulomb friction), 3 for the unsymmetric method with augmented multipliers, 4 for the unsymmetric method with augmented multipliers and De Saxce projection.

add_basic_contact_brick_two_deformable_bodies(*varname_u1*, *varname_u2*, *multname_n*, *dataname_r*, *BN1*, *BN2*, *dataname_gap=None*, **args*)

Synopsis: ind = Model.add_basic_contact_brick_two_deformable_bodies(self, string varname_u1, string varname_u2, string multname_n, string dataname_r, Spmat BN1, Spmat BN2[, string dataname_gap[, string dataname_alpha[, int augmented_version]]])

Add a frictionless contact condition to the model between two deformable bodies. If U1, U2 are the vector of degrees of freedom on which the unilateral constraint is applied, the matrices *BN1* and *BN2* have to be such that this condition is defined by $B_{N1} U_1 B_{N2} U_2 + le gap$. The constraint is prescribed thank to a multiplier *multname_n* whose dimension should be equal to the number of lines of *BN*. The augmentation parameter *r* should be chosen in a range of acceptabe values (see Getfem user documentation). *dataname_gap* is an optional parameter representing the initial gap. It can be a single value or a vector of value. *dataname_alpha* is an optional homogenization parameter for the augmentation parameter (see Getfem user documentation). The parameter *aug_version* indicates the augmentation strategy : 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method with augmented multiplier.

add_bilaplacian_brick(*mim*, *varname*, *dataname*, *region=None*)

Add a bilaplacian brick on the variable *varname* and on the mesh region *region*. This represent a term $\Delta(D\Delta u)$. where $D(x)$ is a coefficient determined by *dataname* which could be constant or described on a f.e.m. The corresponding weak form is $\int D(x)\Delta u(x)\Delta v(x)dx$. Return the brick index in the model.

add_constraint_with_multipliers(*varname*, *multname*, *B*, **args*)

Synopsis: ind = Model.add_constraint_with_multipliers(self, string varname, string multname, Spmat B, {vec L | string dataname})

Add an additional explicit constraint on the variable *varname* thank to a multiplier *multname* peviously added to the model (should be a fixed size variable). The constraint is $BU = L$ with *B* being a rectangular sparse matrix. It is possible to change the constraint at any time with the methods Model.set_private_matrix() and Model.set_private_rhs(). If *dataname* is specified instead of *L*, the vector *L* is defined in the model as data with the given name. Return

the brick index in the model.

add_constraint_with_penalization(*varname*, *coeff*, *B*, **args*)
Synopsis: ind = Model.add_constraint_with_penalization(self, string varname, scalar coeff, Spmat B, {vec L | string dataname})

Add an additional explicit penalized constraint on the variable *varname*. The constraint is :math`BU=L` with *B* being a rectangular sparse matrix. Be aware that *B* should not contain a plain row, otherwise the whole tangent matrix will be plain. It is possible to change the constraint at any time with the methods Model.set_private_matrix() and Model.set_private_rhs(). The method Model.change_penalization_coeff() can be used. If *dataname* is specified instead of *L*, the vector *L* is defined in the model as data with the given name. Return the brick index in the model.

add_contact_boundary_to_unbiased_Nitsche_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *lambdaname*, *wname=None*)
Adds a contact boundary to an existing unbiased Nitschelarge sliding contact with friction brick which is both master and slave.

add_contact_with_rigid_obstacle_brick(*mim*, *varname_u*, *multname_n*, *multname_t=None*, **args*)
Synopsis: ind = Model.add_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname_u, string multname_n[, string multname_t], string dataname_r[, string dataname_friction_coeff], int region, string obstacle[, int augmented_version])

DEPRECATED FUNCTION. Use 'add nodal contact with rigid obstacle brick' instead.

add_data(*name*, *size*)
Add a fixed size data to the model. *sizes* is either a integer (for a scalar or vector data) or a vector of dimensions for a tensor data. *name* is the data name.

add_elastoplasticity_brick(*mim*, *projname*, *varname*, *previous_dep_name*, *datalambda*, *datamu*, *datathreshold*, *datasigma*, *region=None*)
Old (obsolete) brick which do not use the high level generic assembly. Add a nonlinear elastoplastic term to the model relatively to the variable *varname*, in small deformations, for an isotropic material and for a quasistatic model. *projname* is the type of projection that used: only the Von Mises projection is available with 'VM' or 'Von Mises'. *datasigma* is the variable representing the constraints on the material. *previous_dep_name* represents the displacement at the previous time step. Moreover, the finite element method on which *varname* is described is an K ordered mesh_fem, the *datasigma* one have to be at least an K-1 ordered mesh_fem. *datalambda* and *datamu* are the Lame coefficients of the studied material. *datathreshold* is the plasticity threshold of the material. The three last variables could be constants or described on the same finite element method. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_element_extrapolation_transformation(*transname*, *source_mesh*, *elt_corr*)
Add a special interpolation transformation which represents the identity transformation but

allows to evaluate the expression on another element than the current element by polynomial extrapolation. It is used for stabilization term in fictitious domain applications. the array elt_cor should be a two entry array whose first line contains the elements concerned by the transformation and the second line the respective elements on which the extrapolation has to be made. If an element is not listed in elt_cor the evaluation is just made on the current element.

add_elementary_P0_projection(*transname*)

Add the elementary transformation corresponding to the projection P0 element. The name is the name given to the elementary transformation.

add_elementary_rotated_RT0_projection(*transname*)

Add the elementary transformation corresponding to the projection on rotated RT0 element for two-dimensional elements to the model. The name is the name given to the elementary transformation.

add_enriched_Mindlin_Reissner_plate_brick(*mim*, *mim_reduced1*, *mim_reduced2*, *varname_ua*, *varname_theta*, *varname_u3*, *varname_theta3*, *param_E*, *param_nu*, *param_epsilon*, *variant=None*, **args*)

Synopsis: ind = Model.add_enriched_Mindlin_Reissner_plate_brick(self, MeshIm mim, MeshIm mim_reduced1, MeshIm mim_reduced2, string varname_ua, string varname_theta,string varname_u3, string varname_theta3 , string param_E, string param_nu, string param_epsilon [,int variant [, int region]])

Add a term corresponding to the enriched Reissner-Mindlin plate model for which *varname_ua* is the membrane displacements, *varname_u3* is the transverse displacement, *varname_theta* the rotation of fibers normal to the midplane, *varname_theta3* the pinching, 'param_E' the Young Modulus, *param_nu* the poisson ratio, *param_epsilon* the plate thickness. Note that since this brick uses the high level generic assembly language, the parameter can be regular expression of this language. There are four variants. *variant = 0* corresponds to the an unreduced formulation and in that case only the integration method *mim* is used. Practically this variant is not usable since it is subject to a strong locking phenomenon. *variant = 1* corresponds to a reduced integration where *mim* is used for the rotation term and *mim_reduced1* for the transverse shear term and *mim_reduced2* for the pinching term. *variant = 2* (default) corresponds to the projection onto a rotated RT0 element of the transverse shear term and a reduced integration for the pinching term. For the moment, this is adapted to quadrilateral only (because it is not sufficient to remove the locking phenomenon on triangle elements). Note also that if you use high order elements, the projection on RT0 will reduce the order of the approximation. *variant = 3* corresponds to the projection onto a rotated RT0 element of the transverse shear term and the projection onto P0 element of the pinching term. For the moment, this is adapted to quadrilateral only (because it is not sufficient to remove the locking phenomenon on triangle elements). Note also that if you use high order elements, the projection on RT0 will reduce the order of the approximation. Returns the brick index in the model.

add_explicit_matrix(*varname1*, *varname2*, *B*, *issymmetric=None*, **args*)

Synopsis: ind = Model.add_explicit_matrix(self, string varname1, string varname2, Spmat B[, int issymmetric[, int iscoercive]])

Add a brick representing an explicit matrix to be added to the tangent linear system relatively to the variables *varname1* and *varname2*. The given matrix should have has many rows as the dimension of *varname1* and as many columns as the dimension of *varname2*. If the two

variables are different and if *issymmetric* is set to 1 then the transpose of the matrix is also added to the tangent system (default is 0). Set *iscoercive* to 1 if the term does not affect the coercivity of the tangent system (default is 0). The matrix can be changed by the command Model.set_private_matrix(). Return the brick index in the model.

add_explicit_rhs(*varname*, *L*)
Add a brick representing an explicit right hand side to be added to the right hand side of the tangent linear system relatively to the variable *varname*. The given rhs should have the same size than the dimension of *varname*. The rhs can be changed by the command Model.set_private_rhs(). If *dataname* is specified instead of *L*, the vector *L* is defined in the model as data with the given name. Return the brick index in the model.

add_fem_data(*name*, *mf*, *sizes=None*)
Add a data to the model linked to a MeshFem. *name* is the data name, *sizes* an optional parameter which is either an integer or a vector of suplementary dimensions with respect to *mf*.

add_fem_variable(*name*, *mf*)
Add a variable to the model linked to a MeshFem. *name* is the variable name.

add_filtered_fem_variable(*name*, *mf*, *region*)
Add a variable to the model linked to a MeshFem. The variable is filtered in the sense that only the dof on the region are considered. *name* is the variable name.

add_finite_strain_elasticity_brick(*mim*, *constitutive_law*, *varname*, *params*, *region=None*)
Add a nonlinear elasticity term to the model relatively to the variable *varname*. *lawname* is the constitutive law which could be 'SaintVenant Kirchhoff', 'Mooney Rivlin', 'Neo Hookean', 'Ciarlet Geymonat' or 'Generalized Blatz Ko'. 'Mooney Rivlin' and 'Neo Hookean' law names have to be preceeded with the word 'Compressible' or 'Incompressible' to force using the corresponding version. The compressible version of these laws requires one additional material coefficient.

IMPORTANT : if the variable is defined on a 2D mesh, the plane strain approximation is automatically used. *params* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. This brick use the high-level generic assembly. Returns the brick index in the model.

add_finite_strain_elastoplasticity_brick(*mim*, *lawname*, *unknowns_type*, *varnames=None*, **args*)
Synopsis: ind = Model.add_finite_strain_elastoplasticity_brick(self, MeshIm mim , string lawname, string unknowns_type [, string varnames, ...] [, string params, ...] [, int region = -1])

Add a finite strain elastoplasticity brick to the model. For the moment there is only one supported law defined through *lawname* as "Simo_Miehe". This law supports to possibilities of unknown variables to solve for defined by means of *unknowns_type* set to either 'DISPLACEMENT_AND_PLASTIC_MULTIPLIER' (integer value 1) or 'DISPLACEMENT_AND_PLASTIC_MULTIPLIER_AND_PRESSURE' (integer value 3). The "Simo_Miehe" law expects as *varnames* a set of the following names that have to be defined as variables in the model:

- the displacement variable which has to be defined as an unknown,

- the plastic multiplier which has also defined as an unknown,
- optionally the pressure variable for a mixed displacement-pressure formulation for 'DISPLACEMENT_AND_PLASTIC_MULTIPLIER_AND_PRESSURE' as *unknowns_type*,
- the name of a (scalar) fem_data or im_data field that holds the plastic strain at the previous time step, and
- the name of a fem_data or im_data field that holds all non-repeated components of the inverse of the plastic right Cauchy-Green tensor at the previous time step (it has to be a 4 element vector for plane strain 2D problems and a 6 element vector for 3D problems).

The "Simo_Miehe" law also expects as *params* a set of the following three parameters:

- an expression for the initial bulk modulus K,
- an expression for the initial shear modulus G,
- the name of a user predefined function that decribes the yield limit as a function of the hardening variable (both the yield limit and the hardening variable values are assumed to be Frobenius norms of appropriate stress and strain tensors, respectively).

As usual, *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_finite_strain_incompressibility_brick(*mim*, *varname*, *multname_pressure*, *region=None*)

Add a finite strain incompressibility condition on *variable* (for large strain elasticity). *multname_pressure* is a variable which represent the pressure. Be aware that an inf-sup condition between the finite element method describing the pressure and the primal variable has to be satisfied. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model. This brick is equivalent to the `nonlinear incompressibility brick` but uses the high-level generic assembly adding the term `p*(1-Det(Id(meshdim)+Grad_u))` if `p` is the multiplier and `u` the variable which represent the displacement.

add_generalized_Dirichlet_condition_with_Nitsche_method(*mim*, *varname*, *Neumannterm*, *gamma0name*, *region*, *theta=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This version is for vector field. It prescribes a condition @f$ Hu = r @f$ where *H* is a matrix field. CAUTION : the matrix H should have all eigenvalues equal to 1 or 0. The region should be a boundary. *Neumannterm* is the expression of the Neumann term (obtained by the Green formula) described as an expression of the high-level generic assembly language. This term can be obtained by Model.Neumann_term(varname, region) once all volumic bricks have been added to the model. The Dirichlet condition is prescribed with Nitsche's method. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem. *gamma0name* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* is the simplest method for which the second derivative of the Neumann term is not necessary even for nonlinear problems. *Hname* is the data corresponding to the matrix field *H*. It has to be a constant matrix or described on a scalar fem. Returns the brick index in the model. (This brick is not fully tested)

add_generalized_Dirichlet_condition_with_multipliers(*mim*, *varname*, *mult_description*, *region*, *dataname*, *Hname*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This version is for vector field. It prescribes a condition $Hu = r$ where *H* is a matrix field. The region should be a boundary. The Dirichlet condition is prescribed with a multiplier variable described by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is the right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. *Hname* is the data corresponding to the matrix field *H*. Returns the brick index in the model.

add_generalized_Dirichlet_condition_with_penalization(*mim*, *varname*, *coeff*, *region*, *dataname*, *Hname*, *mf_mult=None*)

Add a Dirichlet condition on the variable *varname* and the mesh region *region*. This version is for vector field. It prescribes a condition $Hu = r$ where *H* is a matrix field. The region should be a boundary. The Dirichlet condition is prescribed with penalization. The penalization coefficient is intially *coeff* and will be added to the data of the model. *dataname* is the right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. *Hname* is the data corresponding to the matrix field *H*. It has to be a constant matrix or described on a scalar fem. *mf_mult* is an optional parameter which allows to weaken the Dirichlet condition specifying a multiplier space. Return the brick index in the model.

add_generic_elliptic_brick(*mim*, *varname*, *dataname*, *region=None*)

Add a generic elliptic term to the model relatively to the variable *varname*. The shape of the elliptic term depends both on the variable and the data. This corresponds to a term $-\text{div}(a\nabla u)$ where a is the data and u the variable. The data can be a scalar, a matrix or an order four tensor. The variable can be vector valued or not. If the data is a scalar or a matrix and the variable is vector valued then the term is added componentwise. An order four tensor data is allowed for vector valued variable only. The data can be constant or described on a fem. Of course, when the data is a tensor describe on a finite element method (a tensor field) the data can be a huge vector. The components of the matrix/tensor have to be stored with the fortran order (columnwise) in the data vector (compatibility with blas). The symmetry of the given matrix/tensor is not verified (but assumed). If this is a vector valued variable, the elliptic term is added componentwise. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Note that for the real version which uses the high-level generic assembly language, *dataname* can be any regular expression of the high-level generic assembly language (like "1", "sin(X(1))" or "Norm(u)" for instance) even depending on model variables. Return the brick index in the model.

add_im_data(*name*, *mimd*)

Add a data set to the model linked to a MeshImd. *name* is the data name.

add_im_variable(*name*, *mimd*)
Add a variable to the model linked to a MeshImd. *name* is the variable name.

add_initialized_data(*name*, *V*, *sizes=None*)
Add an initialized fixed size data to the model. *sizes* an optional parameter which is either an integer or a vector dimensions that describes the format of the data. By default, the data is considered to b a vector field. *name* is the data name and *V* is the value of the data.

add_initialized_fem_data(*name*, *mf*, *V*, *sizes=None*)
Add a data to the model linked to a MeshFem. *name* is the data name. The data is initiakized with *V*. The data can be a scalar or vector field. *sizes* an optional parameter which is either an integer or a vector of suplementary dimensions with respect to *mf*.

add_integral_contact_between_nonmatching_meshes_brick(*mim*, *varname_u1*, *varname_u2*, *multname*, *dataname_r*, *dataname_friction_coeff=None*, **args*)
Synopsis: ind = Model.add_integral_contact_between_nonmatching_meshes_brick(self, MeshIm mim, string varname_u1, string varname_u2, string multname, string dataname_r [, string dataname_friction_coeff], int region1, int region2 [, int option [, string dataname_alpha [, string dataname_wt1 , string dataname_wt2]]])

Add a contact with or without friction condition between nonmatching meshes to the model. This brick adds a contact which is defined in an integral way. It is the direct approximation of an augmented agrangian formulation (see Getfem user documentation) defined at the continuous level. The advantage should be a better scalability: the number of Newton iterations should be more or less independent of the mesh size. The condition is applied on the variables *varname_u1* and *varname_u2* on the boundaries corresponding to *region1* and *region2*. *multname* should be a fem variable representing the contact stress for the frictionless case and the contact and friction stress for the case with friction. An inf-sup condition between *multname* and *varname_u1* and *varname_u2* is required. The augmentation parameter *dataname_r* should be chosen in a range of acceptable values. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method on the same mesh as *varname_u1*. Possible values for *option* is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new unsymmetric method. The default value is 1. In case of contact with friction, *dataname_alpha*, *dataname_wt1* and *dataname_wt2* are optional parameters to solve evolutionary friction problems.

add_integral_contact_with_rigid_obstacle_brick(*mim*, *varname_u*, *multname*, *dataname_obstacle*, *dataname_r*, *dataname_friction_coeff=None*, **args*)
Synopsis: ind = Model.add_integral_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname_u, string multname, string dataname_obstacle, string dataname_r [, string dataname_friction_coeff], int region [, int option [, string dataname_alpha [, string dataname_wt [, string dataname_gamma [, string dataname_vt]]]]])

Add a contact with or without friction condition with a rigid obstacle to the model. This brick adds a contact which is defined in an integral way. It is the direct approximation of an augmented Lagrangian formulation (see Getfem user documentation) defined at the continuous level. The advantage is a better scalability: the number of Newton iterations should be

more or less independent of the mesh size. The contact condition is applied on the variable *varname_u* on the boundary corresponding to *region*. The rigid obstacle should be described with the data *dataname_obstacle* being a signed distance to the obstacle (interpolated on a finite element method). *multname* should be a fem variable representing the contact stress. An inf-sup condition beetween *multname* and *varname_u* is required. The augmentation parameter *dataname_r* should be chosen in a range of acceptabe values. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method. Possible values for *option* is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new unsymmetric method. The default value is 1. In case of contact with friction, *dataname_alpha* and *dataname_wt* are optional parameters to solve evolutionary friction problems. *dataname_gamma* and *dataname_vt* represent optional data for adding a parameter-dependent sliding velocity to the friction condition.

add_integral_large_sliding_contact_brick_raytracing(*dataname_r*, *release_distance*, *dataname_fr=None*, **args*)

Synopsis: ind = Model.add_integral_large_sliding_contact_brick_raytracing(self, string dataname_r, scalar release_distance, [, string dataname_fr[, string dataname_alpha[, int version]]])

Adds a large sliding contact with friction brick to the model. This brick is able to deal with self-contact, contact between several deformable bodies and contact with rigid obstacles. It uses the high-level generic assembly. It adds to the model a raytracing_interpolate_transformation object. For each slave boundary a multiplier variable should be defined. The release distance should be determined with care (generally a few times a mean element size, and less than the thickness of the body). Initially, the brick is added with no contact boundaries. The contact boundaries and rigid bodies are added with special functions. *version* is 0 (the default value) for the non-symmetric version and 1 for the more symmetric one (not fully symmetric even without friction).

add_internal_im_variable(*name*, *mimd*)

Add a variable to the model, which is linked to a MeshImd and will be condensed out during the assemblage of the tangent matrix. *name* is the variable name.

add_interpolate_transformation_from_expression(*transname*, *source_mesh*, *target_mesh*, *expr*)

Add a transformation to the model from mesh *source_mesh* to mesh *target_mesh* given by the expression *expr* which corresponds to a high-level generic assembly expression which may contains some variable of the model. CAUTION: the derivative of the transformation with used variable is taken into account in the computation of the tangen system. However, order two derivative is not implemented, so such tranformation is not allowed in the definition of a potential.

add_isotropic_linearized_elasticity_brick(*mim*, *varname*, *dataname_lambda*, *dataname_mu*, *region=None*)

Add an isotropic linearized elasticity term to the model relatively to the variable *varname*. *dataname_lambda* and *dataname_mu* should contain the Lame coefficients. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_isotropic_linearized_elasticity_pstrain_brick(*mim*, *varname*, *data_E*, *data_nu*, *region=None*)

Add an isotropic linearized elasticity term to the model relatively to the variable *varname*. *data_E* and *data_nu* should contain the Young modulus and Poisson ratio, respectively. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. On two-dimensional meshes, the term will correpsond to a plain strain approximation. On three-dimensional meshes, it will correspond to the standard model. Return the brick index in the model.

add_isotropic_linearized_elasticity_pstress_brick(*mim*, *varname*, *data_E*, *data_nu*, *region=None*)

Add an isotropic linearized elasticity term to the model relatively to the variable *varname*. *data_E* and *data_nu* should contain the Young modulus and Poisson ratio, respectively. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. On two-dimensional meshes, the term will correpsond to a plain stress approximation. On three-dimensional meshes, it will correspond to the standard model. Return the brick index in the model.

add_linear_generic_assembly_brick(*mim*, *expression*, *region=None*, **args*)

Synopsis: ind = Model.add_linear_generic_assembly_brick(self, MeshIm mim, string expression[, int region[, int is_symmetric[, int is_coercive]]])

Deprecated. Use Model.add_linear_term() instead.

add_linear_incompressibility_brick(*mim*, *varname*, *multname_pressure*, *region=None*, **args*)

Synopsis: ind = Model.add_linear_incompressibility_brick(self, MeshIm mim, string varname, string multname_pressure[, int region[, string dataexpr_coeff]])

Add a linear incompressibility condition on *variable*. *multname_pressure* is a variable which represent the pressure. Be aware that an inf-sup condition between the finite element method describing the pressure and the primal variable has to be satisfied. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. *dataexpr_coeff* is an optional penalization coefficient for nearly incompressible elasticity for instance. In this case, it is the inverse of the Lame coefficient λ. Return the brick index in the model.

add_linear_term(*mim*, *expression*, *region=None*, **args*)

Synopsis: ind = Model.add_linear_term(self, MeshIm mim, string expression[, int region[, int is_symmetric[, int is_coercive]]])

Adds a matrix term given by the assembly string *expr* which will be assembled in region *region* and with the integration method *mim*. Only the matrix term will be taken into account, assuming that it is linear. The advantage of declaring a term linear instead of nonlinear is that it will be assembled only once and no assembly is necessary for the residual. Take care that if the expression contains some variables and if the expression is a potential or of first order (i.e. describe the weak form, not the derivative of the weak form), the expression will be derivated with respect to all variables. You can specify if the term is symmetric, coercive or not. If you are not sure, the better is to declare the term not symmetric and not coercive. But some solvers (conjugate gradient for instance) are not allowed for non-coercive problems. *brickname* is an optional name for the brick.

add_linear_twodomain_term(*mim*, *expression*, *region*, *secondary_domain*, *is_symmetric=None*, **args*)

Synopsis: ind = Model.add_linear_twodomain_term(self, MeshIm mim, string expression,

int region, string secondary_domain[, int is_symmetric[, int is_coercive]])

Adds a linear term given by a weak form language expression like Model.add_linear_term() but for an integration on a direct product of two domains, a first specfied by `mim` and `region` and a second one by `secondary_domain` which has to be declared first into the model.

add_lumped_mass_for_first_order_brick(*mim*, *varname*, *dataexpr_rho=None*, **args*)
Synopsis: ind = Model.add_lumped_mass_for_first_order_brick(self, MeshIm mim, string varname[, string dataexpr_rho[, int region]])

Add lumped mass for first order term to the model relatively to the variable *varname*. If specified, the data *dataexpr_rho* is the density (1 if omitted). *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_macro(*name*, *expr*)
Define a new macro for the high generic assembly language. The name include the parameters. For instance name='sp(a,b)', expr='a.b' is a valid definition. Macro without parameter can also be defined. For instance name='x1', expr='X[1]' is valid. The form name='grad(u)', expr='Grad_u' is also allowed but in that case, the parameter 'u' will only be allowed to be a variable name when using the macro. Note that macros can be directly defined inside the assembly strings with the keyword 'Def'.

add_mass_brick(*mim*, *varname*, *dataexpr_rho=None*, **args*)
Synopsis: ind = Model.add_mass_brick(self, MeshIm mim, string varname[, string dataexpr_rho[, int region]])

Add mass term to the model relatively to the variable *varname*. If specified, the data *dataexpr_rho* is the density (1 if omitted). *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_master_contact_boundary_to_biased_Nitsche_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *wname=None*)
Adds a master contact boundary to an existing biased Nitsche's large sliding contact with friction brick.

add_master_contact_boundary_to_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *wname=None*)
Adds a master contact boundary to an existing large sliding contact with friction brick.

add_master_contact_boundary_to_projection_transformation(*transname*, *m*, *dispname*, *region*)
Add a master contact boundary with corresponding displacement variable *dispname* on a specific boundary *region* to an existing projection interpolate transformation called *transname*.

add_master_contact_boundary_to_raytracing_transformation(*transname*, *m*, *dispname*, *region*)
Add a master contact boundary with corresponding displacement variable *dispname* on a specific boundary *region* to an existing raytracing interpolate transformation called *transname*.

add_master_slave_contact_boundary_to_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *lambdaname*, *wname=None*)

Adds a contact boundary to an existing large sliding contact with friction brick which is both master and slave (allowing the self-contact).

add_multiplier(*name*, *mf*, *primalname*, *mim=None*, *region=None*)

Add a particular variable linked to a fem being a multiplier with respect to a primal variable. The dof will be filtered with the `gmm::range_basis` function applied on the terms of the model which link the multiplier and the primal variable. This in order to retain only linearly independent constraints on the primal variable. Optimized for boundary multipliers.

add_nodal_contact_between_nonmatching_meshes_brick(*mim1*, *mim2=None*, **args*)

Synopsis: ind = Model.add_nodal_contact_between_nonmatching_meshes_brick(self, MeshIm mim1[, MeshIm mim2], string varname_u1[, string varname_u2], string multname_n[, string multname_t], string dataname_r[, string dataname_fr], int rg1, int rg2[, int slave1, int slave2, int augmented_version])

Add a contact with or without friction condition between two faces of one or two elastic bodies. The condition is applied on the variable *varname_u1* or the variables *varname_u1* and *varname_u2* depending if a single or two distinct displacement fields are given. Integers *rg1* and *rg2* represent the regions expected to come in contact with each other. In the single displacement variable case the regions defined in both *rg1* and *rg2* refer to the variable *varname_u1*. In the case of two displacement variables, *rg1* refers to *varname_u1* and *rg2* refers to *varname_u2*. *multname_n* should be a fixed size variable whose size is the number of degrees of freedom on those regions among the ones defined in *rg1* and *rg2* which are characterized as "slaves". It represents the contact equivalent nodal normal forces. *multname_t* should be a fixed size variable whose size corresponds to the size of *multname_n* multiplied by qdim - 1 . It represents the contact equivalent nodal tangent (frictional) forces. The augmentation parameter *r* should be chosen in a range of acceptabe values (close to the Young modulus of the elastic body, see Getfem user documentation). The friction coefficient stored in the parameter *fr* is either a single value or a vector of the same size as *multname_n*. The optional parameters *slave1* and *slave2* declare if the regions defined in *rg1* and *rg2* are correspondingly considered as "slaves". By default *slave1* is true and *slave2* is false, i.e. *rg1* contains the slave surfaces, while 'rg2' the master surfaces. Preferrably only one of *slave1* and *slave2* is set to true. The parameter *augmented_version* indicates the augmentation strategy : 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one (except for the coupling between contact and Coulomb friction), 3 for the new unsymmetric method. Basically, this brick computes the matrices BN and BT and the vectors gap and alpha and calls the basic contact brick.

add_nodal_contact_with_rigid_obstacle_brick(*mim*, *varname_u*, *multname_n*, *multname_t=None*, **args*)

Synopsis: ind = Model.add_nodal_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname_u, string multname_n[, string multname_t], string dataname_r[, string dataname_friction_coeff], int region, string obstacle[, int augmented_version])

Add a contact with or without friction condition with a rigid obstacle to the model. The condition is applied on the variable *varname_u* on the boundary corresponding to *region*. The rigid obstacle should be described with the string *obstacle* being a signed distance to the

obstacle. This string should be an expression where the coordinates are 'x', 'y' in 2D and 'x', 'y', 'z' in 3D. For instance, if the rigid obstacle correspond to $z \leq 0$, the corresponding signed distance will be simply "z". *multname_n* should be a fixed size variable whose size is the number of degrees of freedom on boundary *region*. It represents the contact equivalent nodal forces. In order to add a friction condition one has to add the *multname_t* and *dataname_friction_coeff* parameters. *multname_t* should be a fixed size variable whose size is the number of degrees of freedom on boundary *region* multiplied by $d - 1$ where d is the domain dimension. It represents the friction equivalent nodal forces. The augmentation parameter *r* should be chosen in a range of acceptabe values (close to the Young modulus of the elastic body, see GetFEM user documentation). *dataname_friction_coeff* is the friction coefficient. It could be a scalar or a vector of values representing the friction coefficient on each contact node. The parameter *augmented_version* indicates the augmentation strategy : 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one (except for the coupling between contact and Coulomb friction), 3 for the new unsymmetric method. Basically, this brick compute the matrix BN and the vectors gap and alpha and calls the basic contact brick.

add_nonlinear_elasticity_brick(*mim*, *varname*, *constitutive_law*, *dataname*, *region=None*)

Add a nonlinear elasticity term to the model relatively to the variable *varname* (deprecated brick, use add_finite_strain_elaticity instead). *lawname* is the constitutive law which could be 'SaintVenant Kirchhoff', 'Mooney Rivlin', 'neo Hookean', 'Ciarlet Geymonat' or 'generalized Blatz Ko'. 'Mooney Rivlin' and 'neo Hookean' law names can be preceded with the word 'compressible' or 'incompressible' to force using the corresponding version. The compressible version of these laws requires one additional material coefficient. By default, the incompressible version of 'Mooney Rivlin' law and the compressible one of the 'neo Hookean' law are considered. In general, 'neo Hookean' is a special case of the 'Mooney Rivlin' law that requires one coefficient less. IMPORTANT : if the variable is defined on a 2D mesh, the plane strain approximation is automatically used. *dataname* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. This brick use the low-level generic assembly. Returns the brick index in the model.

add_nonlinear_generic_assembly_brick(*mim*, *expression*, *region=None*, **args*)

Synopsis: ind = Model.add_nonlinear_generic_assembly_brick(self, MeshIm mim, string expression[, int region[, int is_symmetric[, int is_coercive]]])

Deprecated. Use Model.add_nonlinear_term() instead.

add_nonlinear_incompressibility_brick(*mim*, *varname*, *multname_pressure*, *region=None*)

Add a nonlinear incompressibility condition on *variable* (for large strain elasticity). *multname_pressure* is a variable which represent the pressure. Be aware that an inf-sup condition between the finite element method describing the pressure and the primal variable has to be satisfied. *region* is an optional mesh region on which the term is added. If it is not specified, it is added on the whole mesh. Return the brick index in the model.

add_nonlinear_term(*mim*, *expression*, *region=None*, **args*)

Synopsis: ind = Model.add_nonlinear_term(self, MeshIm mim, string expression[, int region[, int is_symmetric[, int is_coercive]]])

Adds a nonlinear term given by the assembly string *expr* which will be assembled in region

region and with the integration method *mim*. The expression can describe a potential or a weak form. Second order terms (i.e. containing second order test functions, Test2) are not allowed. You can specify if the term is symmetric, coercive or not. If you are not sure, the better is to declare the term not symmetric and not coercive. But some solvers (conjugate gradient for instance) are not allowed for non-coercive problems. *brickname* is an optional name for the brick.

add_nonlinear_twodomain_term(*mim*, *expression*, *region*, *secondary_domain*, *is_symmetric=None*, **args*)

Synopsis: ind = Model.add_nonlinear_twodomain_term(self, MeshIm mim, string expression, int region, string secondary_domain[, int is_symmetric[, int is_coercive]])

Adds a nonlinear term given by a weak form language expression like Model.add_nonlinear_term() but for an integration on a direct product of two domains, a first specfied by `mim` and `region` and a second one by `secondary_domain` which has to be declared first into the model.

add_nonmatching_meshes_contact_brick(*mim1*, *mim2=None*, **args*)

Synopsis: ind = Model.add_nonmatching_meshes_contact_brick(self, MeshIm mim1[, MeshIm mim2], string varname_u1[, string varname_u2], string multname_n[, string multname_t], string dataname_r[, string dataname_fr], int rg1, int rg2[, int slave1, int slave2, int augmented_version])

DEPRECATED FUNCTION. Use 'add nodal contact between nonmatching meshes brick' instead.

add_normal_Dirichlet_condition_with_Nitsche_method(*mim*, *varname*, *Neumannterm*, *gamma0name*, *region*, *theta=None*, **args*)

Synopsis: ind = Model.add_normal_Dirichlet_condition_with_Nitsche_method(self, MeshIm mim, string varname, string Neumannterm, string gamma0name, int region[, scalar theta][, string dataname])

Add a Dirichlet condition to the normal component of the vector (or tensor) valued variable *varname* and the mesh region *region*. This region should be a boundary. *Neumannterm* is the expression of the Neumann term (obtained by the Green formula) described as an expression of the high-level generic assembly language. This term can be obtained by Model.Neumann_term(varname, region) once all volumic bricks have been added to the model. The Dirichlet condition is prescribed with Nitsche's method. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem. *gamma0name* is the Nitsche's method parameter. *theta* is a scalar value which can be positive or negative. *theta = 1* corresponds to the standard symmetric method which is conditionally coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionally coercive. *theta = 0* is the simplest method for which the second derivative of the Neumann term is not necessary even for nonlinear problems. Returns the brick index in the model. (This brick is not fully tested)

add_normal_Dirichlet_condition_with_multipliers(*mim*, *varname*, *mult_description*, *region*, *dataname=None*)

Add a Dirichlet condition to the normal component of the vector (or tensor) valued variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with a multiplier variable described by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be

added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed (scalar if the variable is vector valued, vector if the variable is tensor valued). Returns the brick index in the model.

add_normal_Dirichlet_condition_with_penalization(*mim*, *varname*, *coeff*, *region*, *dataname=None*, *mf_mult=None*)

Add a Dirichlet condition to the normal component of the vector (or tensor) valued variable *varname* and the mesh region *region*. This region should be a boundary. The Dirichlet condition is prescribed with penalization. The penalization coefficient is initially *coeff* and will be added to the data of the model. *dataname* is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed (scalar if the variable is vector valued, vector if the variable is tensor valued). *mf_mult* is an optional parameter which allows to weaken the Dirichlet condition specifying a multiplier space. Returns the brick index in the model.

add_normal_derivative_Dirichlet_condition_with_multipliers(*mim*, *varname*, *mult_description*, *region*, *dataname=None*, *R_must_be_derivated=None*)

Add a Dirichlet condition on the normal derivative of the variable *varname* and on the mesh region *region* (which should be a boundary). The general form is $\int \partial_n u(x)v(x) = \int r(x)v(x)\forall v$ where $r(x)$ is the right hand side for the Dirichlet condition (0 for homogeneous conditions) and v is in a space of multipliers defined by *mult_description*. If *mult_description* is a string this is assumed to be the variable name corresponding to the multiplier (which should be first declared as a multiplier variable on the mesh region in the model). If it is a finite element method (mesh_fem object) then a multiplier variable will be added to the model and build on this finite element method (it will be restricted to the mesh region *region* and eventually some conflicting dofs with some other multiplier variables will be suppressed). If it is an integer, then a multiplier variable will be added to the model and build on a classical finite element of degree that integer. *dataname* is an optional parameter which represents the right hand side of the Dirichlet condition. If *R_must_be_derivated* is set to *true* then the normal derivative of *dataname* is considered. Return the brick index in the model.

add_normal_derivative_Dirichlet_condition_with_penalization(*mim*, *varname*, *coeff*, *region*, *dataname=None*, *R_must_be_derivated=None*)

Add a Dirichlet condition on the normal derivative of the variable *varname* and on the mesh region *region* (which should be a boundary). The general form is $\int \partial_n u(x)v(x) = \int r(x)v(x)\forall v$ where $r(x)$ is the right hand side for the Dirichlet condition (0 for homogeneous conditions). The penalization coefficient is initially *coeff* and will be added to the data of the model. It can be changed with the command Model.change_penalization_coeff(). *dataname* is an optional parameter which represents the right hand side of the Dirichlet condition. If *R_must_be_derivated* is set to *true* then the normal derivative of *dataname* is con-

sidered. Return the brick index in the model.

add_normal_derivative_source_term_brick(*mim*, *varname*, *dataname*, *region*)
Add a normal derivative source term brick $F = \int b.\partial_n v$ on the variable *varname* and the mesh region *region*.

Update the right hand side of the linear system. *dataname* represents *b* and *varname* represents *v*. Return the brick index in the model.

add_normal_source_term_brick(*mim*, *varname*, *dataname*, *region*)
Add a source term on the variable *varname* on a boundary *region*. This region should be a boundary. The source term is represented by the data *dataepxpr* which could be any regular expression of the high-level generic assembly language (except for the complex version where it has to be a declared data of the model). A scalar product with the outward normal unit vector to the boundary is performed. The main aim of this brick is to represent a Neumann condition with a vector data without performing the scalar product with the normal as a pre-processing. Return the brick index in the model.

add_penalized_contact_between_nonmatching_meshes_brick(*mim*, *varname_u1*, *varname_u2*, *dataname_r*, *dataname_coeff=None*, **args*)
Synopsis: ind = Model.add_penalized_contact_between_nonmatching_meshes_brick(self, MeshIm mim, string varname_u1, string varname_u2, string dataname_r [, string dataname_coeff], int region1, int region2 [, int option [, string dataname_lambda, [, string dataname_alpha [, string dataname_wt1, string dataname_wt2]]]])

Add a penalized contact condition with or without friction between nonmatching meshes to the model. The condition is applied on the variables *varname_u1* and *varname_u2* on the boundaries corresponding to *region1* and *region2*. The penalization parameter *dataname_r* should be chosen large enough to prescribe approximate non-penetration and friction conditions but not too large not to deteriorate too much the conditionning of the tangent system. The optional parameter *dataname_friction_coeff* is the friction coefficient which could be constant or defined on a finite element method on the same mesh as *varname_u1*. *dataname_lambda* is an optional parameter used if option is 2. In that case, the penalization term is shifted by lambda (this allows the use of an Uzawa algorithm on the corresponding augmented Lagrangian formulation) In case of contact with friction, *dataname_alpha*, *dataname_wt1* and *dataname_wt2* are optional parameters to solve evolutionary friction problems.

add_penalized_contact_with_rigid_obstacle_brick(*mim*, *varname_u*, *dataname_obstacle*, *dataname_r*, *dataname_coeff=None*, **args*)
Synopsis: ind = Model.add_penalized_contact_with_rigid_obstacle_brick(self, MeshIm mim, string varname_u, string dataname_obstacle, string dataname_r [, string dataname_coeff], int region [, int option, string dataname_lambda, [, string dataname_alpha [, string dataname_wt]]])

Add a penalized contact with or without friction condition with a rigid obstacle to the model. The condition is applied on the variable *varname_u* on the boundary corresponding to *region*. The rigid obstacle should be described with the data *dataname_obstacle* being a signed distance to the obstacle (interpolated on a finite element method). The penalization parameter *dataname_r* should be chosen large enough to prescribe approximate non-penetration and friction conditions but not too large not to deteriorate too much the conditionning of the tan-

gent system. *dataname_lambda* is an optional parameter used if option is 2. In that case, the penalization term is shifted by lambda (this allows the use of an Uzawa algorithm on the corresponding augmented Lagrangian formulation)

add_pointwise_constraints_with_given_multipliers(*varname*, *multname*, *dataname_pt*, *dataname_unitv=None*, **args*)

Synopsis: ind = Model.add_pointwise_constraints_with_given_multipliers(self, string varname, string multname, string dataname_pt[, string dataname_unitv] [, string dataname_val])

Add some pointwise constraints on the variable *varname* using a given multiplier *multname*. The conditions are prescribed on a set of points given in the data *dataname_pt* whose dimension is the number of points times the dimension of the mesh. The multiplier variable should be a fixed size variable of size the number of points. If the variable represents a vector field, one has to give the data *dataname_unitv* which represents a vector of dimension the number of points times the dimension of the vector field which should store some unit vectors. In that case the prescribed constraint is the scalar product of the variable at the corresponding point with the corresponding unit vector. The optional data *dataname_val* is the vector of values to be prescribed at the different points. This brick is specifically designed to kill rigid displacement in a Neumann problem. Returns the brick index in the model.

add_pointwise_constraints_with_multipliers(*varname*, *dataname_pt*, *dataname_unitv=None*, **args*)

Synopsis: ind = Model.add_pointwise_constraints_with_multipliers(self, string varname, string dataname_pt[, string dataname_unitv] [, string dataname_val])

Add some pointwise constraints on the variable *varname* using multiplier. The multiplier variable is automatically added to the model. The conditions are prescribed on a set of points given in the data *dataname_pt* whose dimension is the number of points times the dimension of the mesh. If the variable represents a vector field, one has to give the data *dataname_unitv* which represents a vector of dimension the number of points times the dimension of the vector field which should store some unit vectors. In that case the prescribed constraint is the scalar product of the variable at the corresponding point with the corresponding unit vector. The optional data *dataname_val* is the vector of values to be prescribed at the different points. This brick is specifically designed to kill rigid displacement in a Neumann problem. Returns the brick index in the model.

add_pointwise_constraints_with_penalization(*varname*, *coeff*, *dataname_pt*, *dataname_unitv=None*, **args*)

Synopsis: ind = Model.add_pointwise_constraints_with_penalization(self, string varname, scalar coeff, string dataname_pt[, string dataname_unitv] [, string dataname_val])

Add some pointwise constraints on the variable *varname* thanks to a penalization. The penalization coefficient is initially *penalization_coeff* and will be added to the data of the model. The conditions are prescribed on a set of points given in the data *dataname_pt* whose dimension is the number of points times the dimension of the mesh. If the variable represents a vector field, one has to give the data *dataname_unitv* which represents a vector of dimension the number of points times the dimension of the vector field which should store some unit vectors. In that case the prescribed constraint is the scalar product of the variable at the corresponding point with the corresponding unit vector. The optional data *dataname_val* is the vector of values to be prescribed at the different points. This brick is specifically designed to kill rigid displacement in a Neumann problem. Returns the brick index in the model.

add_projection_transformation(*transname*, *release_distance*)

Add a projection interpolate transformation called *transname* to a model to be used by the

generic assembly bricks. CAUTION: For the moment, the derivative of the transformation is not taken into account in the model solve.

add_raytracing_transformation(*transname*, *release_distance*)
Add a raytracing interpolate transformation called *transname* to a model to be used by the generic assembly bricks. CAUTION: For the moment, the derivative of the transformation is not taken into account in the model solve.

add_rigid_obstacle_to_Nitsche_large_sliding_contact_brick(*indbrick*, *expr*, *N*)
Adds a rigid obstacle to an existing large sliding contact with friction brick. *expr* is an expression using the high-level generic assembly language (where *x* is the current point n the mesh) which should be a signed distance to the obstacle. *N* is the mesh dimension.

add_rigid_obstacle_to_large_sliding_contact_brick(*indbrick*, *expr*, *N*)
Adds a rigid obstacle to an existing large sliding contact with friction brick. *expr* is an expression using the high-level generic assembly language (where *x* is the current point n the mesh) which should be a signed distance to the obstacle. *N* is the mesh dimension.

add_rigid_obstacle_to_projection_transformation(*transname*, *expr*, *N*)
Add a rigid obstacle whose geometry corresponds to the zero level-set of the high-level generic assembly expression *expr* to an existing projection interpolate transformation called *transname*.

add_rigid_obstacle_to_raytracing_transformation(*transname*, *expr*, *N*)
Add a rigid obstacle whose geometry corresponds to the zero level-set of the high-level generic assembly expression *expr* to an existing raytracing interpolate transformation called *transname*.

add_slave_contact_boundary_to_biased_Nitsche_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *lambdaname*, *wname=None*)
Adds a slave contact boundary to an existing biased Nitsche's large sliding contact with friction brick.

add_slave_contact_boundary_to_large_sliding_contact_brick(*indbrick*, *mim*, *region*, *dispname*, *lambdaname*, *wname=None*)
Adds a slave contact boundary to an existing large sliding contact with friction brick.

add_slave_contact_boundary_to_projection_transformation(*transname*, *m*, *dispname*, *region*)
Add a slave contact boundary with corresponding displacement variable *dispname* on a specific boundary *region* to an existing projection interpolate transformation called *transname*.

add_slave_contact_boundary_to_raytracing_transformation(*transname*, *m*, *dispname*, *region*)
Add a slave contact boundary with corresponding displacement variable *dispname* on a specific boundary *region* to an existing raytracing interpolate transformation called *transname*.

add_small_strain_elastoplasticity_brick(*mim*, *lawname*, *unknowns_type*, *varnames=None*, **args*)

Synopsis: ind = Model.add_small_strain_elastoplasticity_brick(self, MeshIm mim, string lawname, string unknowns_type [, string varnames, ...] [, string params, ...] [, string theta = '1' [, string dt = 'timestep']] [, int region = -1])

Adds a small strain plasticity term to the model *M*. This is the main GetFEM brick for small strain plasticity. *lawname* is the name of an implemented plastic law, *unknowns_type* indicates the choice between a discretization where the plastic multiplier is an unknown of the problem or (return mapping approach) just a data of the model stored for the next iteration. Remember that in both cases, a multiplier is stored anyway. *varnames* is a set of variable and data names with length which may depend on the plastic law (at least the displacement, the plastic multiplier and the plastic strain). *params* is a list of expressions for the parameters (at least elastic coefficients and the yield stress). These expressions can be some data names (or even variable names) of the model but can also be any scalar valid expression of the high level assembly language (such as '1/2', '2+sin(X[0])', '1+Norm(v)' ...). The last two parameters optionally provided in *params* are the *theta* parameter of the *theta*-scheme (generalized trapezoidal rule) used for the plastic strain integration and the time-step`dt`. The default value for *theta* if omitted is 1, which corresponds to the classical Backward Euler scheme which is first order consistent. *theta=1/2* corresponds to the Crank-Nicolson scheme (trapezoidal rule) which is second order consistent. Any value between 1/2 and 1 should be a valid value. The default value of *dt* is 'timestep' which simply indicates the time step defined in the model (by md.set_time_step(dt)). Alternatively it can be any expression (data name, constant value ...). The time step can be altered from one iteration to the next one. *region* is a mesh region.

The available plasticity laws are:

- 'Prandtl Reuss' (or 'isotropic perfect plasticity'). Isotropic elasto-plasticity with no hardening. The variables are the displacement, the plastic multiplier and the plastic strain. The displacement should be a variable and have a corresponding data having the same name preceded by 'Previous_' corresponding to the displacement at the previous time step (typically 'u' and 'Previous_u'). The plastic multiplier should also have two versions (typically 'xi' and 'Previous_xi') the first one being defined as data if *unknowns_type `is 'DISPLACEMENT_ONLY' or the integer value 0, or as a variable if `unknowns_type* is DISPLACEMENT_AND_PLASTIC_MULTIPLIER or the integer value 1. The plastic strain should represent a n x n data tensor field stored on mesh_fem or (preferably) on an im_data (corresponding to *mim*). The data are the first Lame coefficient, the second one (shear modulus) and the uniaxial yield stress. A typical call is Model.add_small_strain_elastoplasticity_brick(mim, 'Prandtl Reuss', 0, 'u', 'xi', 'Previous_Ep', 'lambda', 'mu', 'sigma_y', '1', 'timestep'); IMPORTANT: Note that this law implements the 3D expressions. If it is used in 2D, the expressions are just transposed to the 2D. For the plane strain approximation, see below.
- "plane strain Prandtl Reuss" (or "plane strain isotropic perfect plasticity") The same law as the previous one but adapted to the plane strain approximation. Can only be used in 2D.
- "Prandtl Reuss linear hardening" (or "isotropic plasticity linear hardening"). Isotropic elasto-plasticity with linear isotropic and kinematic hardening. An additional variable compared to "Prandtl Reuss" law: the accumulated plastic strain. Similarly to the plastic strain, it is only stored at the end of the time step, so a simple data is required (preferably on an im_data). Two additional parameters: the kinematic hardening modulus and the isotropic one. 3D expressions only. A typical call is Model.add_small_strain_elastoplasticity_brick(mim, 'Prandtl Reuss linear hardening',

0, ‘u’, ‘xi’, ‘Previous_Ep’, ‘Previous_alpha’, ‘lambda’, ‘mu’, ‘sigma_y’, ‘H_k’, H_i’, ‘1’, ‘timestep’);

- “plane strain Prandtl Reuss linear hardening” (or “plane strain isotropic plasticity linear hardening”). The same law as the previous one but adapted to the plane strain approximation. Can only be used in 2D.

See GetFEM user documentation for further explanations on the discretization of the plastic flow and on the implemented plastic laws. See also GetFEM user documentation on time integration strategy (integration of transient problems).

IMPORTANT : remember that *small_strain_elastoplasticity_next_iter* has to be called at the end of each time step, before the next one (and before any post-treatment : this sets the value of the plastic strain and plastic multiplier).

add_source_term(*mim*, *expression*, *region=None*)
Adds a source term given by the assembly string *expr* which will be assembled in region *region* and with the integration method *mim*. Only the residual term will be taken into account. Take care that if the expression contains some variables and if the expression is a potential, the expression will be derivated with respect to all variables. *brickname* is an optional name for the brick.

add_source_term_brick(*mim*, *varname*, *dataexpr*, *region=None*, **args*)
Synopsis: ind = Model.add_source_term_brick(self, MeshIm mim, string varname, string dataexpr[, int region[, string directdataname]])

Add a source term to the model relatively to the variable *varname*. The source term is represented by *dataexpr* which could be any regular expression of the high-level generic assembly language (except for the complex version where it has to be a declared data of the model). *region* is an optional mesh region on which the term is added. An additional optional data *directdataname* can be provided. The corresponding data vector will be directly added to the right hand side without assembly. Note that when region is a boundary, this brick allows to prescribe a nonzero Neumann boundary condition. Return the brick index in the model.

add_source_term_generic_assembly_brick(*mim*, *expression*, *region=None*)
Deprecated. Use Model.add_source_term() instead.

add_standard_secondary_domain(*name*, *mim*, *region=- 1*)
Add a secondary domain to the model which can be used in a weak-form language expression for integration on the product of two domains. *name* is the name of the secondary domain, *mim* is an integration method on this domain and *region* the region on which the integration is to be performed.

add_theta_method_for_first_order(*varname*, *theta*)
Attach a theta method for the time discretization of the variable *varname*. Valid only if there is at most first order time derivative of the variable.

add_theta_method_for_second_order(*varname*, *theta*)
Attach a theta method for the time discretization of the variable *varname*. Valid only if there is at most second order time derivative of the variable.

add_twodomain_source_term(*mim*, *expression*, *region*, *secondary_domain*)
Adds a source term given by a weak form language expression like Model.add_source_term() but for an integration on a direct product of two domains, a first specfied by `mim` and `region` and a second one by `secondary_domain` which has to be declared first into the model.

add_variable(*name*, *sizes*)
Add a variable to the model of constant sizes. *sizes* is either a integer (for a scalar or vector variable) or a vector of dimensions for a tensor variable. *name* is the variable name.

assembly(*option=None*)
Assembly of the tangent system taking into account the terms from all bricks. *option*, if specified, should be 'build_all', 'build_rhs', 'build_matrix', 'build_rhs_with_internal', 'build_matrix_condensed', 'build_all_condensed'. The default is to build the whole tangent linear system (matrix and rhs). This function is useful to solve your problem with you own solver.

brick_list()
print to the output the list of bricks of the model.

brick_term_rhs(*ind_brick*, *ind_term=None*, *sym=None*, *ind_iter=None*)
Gives the access to the part of the right hand side of a term of a particular nonlinear brick. Does not account of the eventual time dispatcher. An assembly of the rhs has to be done first. *ind_brick* is the brick index. *ind_term* is the index of the term inside the brick (default value : 0). *sym* is to access to the second right hand side of for symmetric terms acting on two different variables (default is 0). *ind_iter* is the iteration number when time dispatchers are used (default is 0).

change_penalization_coeff(*ind_brick*, *coeff*)
Change the penalization coefficient of a Dirichlet condition with penalization brick. If the brick is not of this kind, this function has an undefined behavior.

char()
Output a (unique) string representation of the Model.

This can be used to perform comparisons between two different Model objects. This function is to be completed.

clear()
Clear the model.

clear_assembly_assignment()
Delete all added assembly assignments

compute_Von_Mises_or_Tresca(*varname*, *lawname*, *dataname*, *mf_vm*, *version=None*)
Compute on *mf_vm* the Von-Mises stress or the Tresca stress of a field for nonlinear elasticity in 3D. *lawname* is the constitutive law which could be 'SaintVenant Kirchhoff', 'Mooney Rivlin', 'neo Hookean' or 'Ciarlet Geymonat'. *dataname* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients. *version* should be 'Von_Mises' or 'Tresca' ('Von_Mises' is the default).

compute_elastoplasticity_Von_Mises_or_Tresca(*datasigma*, *mf_vm*, *version=None*)
Compute on *mf_vm* the Von-Mises or the Tresca stress of a field for plasticity and return it into the vector V. *datasigma* is a vector which contains the stress constraints values supported by the mesh. *version* should be 'Von_Mises' or 'Tresca' ('Von_Mises' is the default).

compute_finite_strain_elasticity_Von_Mises(*lawname*, *varname*, *params*, *mf_vm*, *region=None*)
Compute on *mf_vm* the Von-Mises stress of a field *varname* for nonlinear elasticity in 3D. *lawname* is the constitutive law which should be a valid name. *params* are the parameters law. It could be a short vector of constant values or may depend on data or variables of the model. Uses the high-level generic assembly.

compute_finite_strain_elastoplasticity_Von_Mises(*mim*, *mf_vm*, *lawname*, *unknowns_type*, *varnames=None*, **args*)

Synopsis: V = Model.compute_finite_strain_elastoplasticity_Von_Mises(self, MeshIm mim, MeshFem mf_vm, string lawname, string unknowns_type, [, string varnames, ...] [, string params, ...] [, int region = -1])

Compute on *mf_vm* the Von-Mises or the Tresca stress of a field for plasticity and return it into the vector V. The first input parameters ar as in the function 'finite strain elastoplasticity next iter'.

compute_isotropic_linearized_Von_Mises_or_Tresca(*varname*, *dataname_lambda*, *dataname_mu*, *mf_vm*, *version=None*)

Compute the Von-Mises stress or the Tresca stress of a field (only valid for isotropic linearized elasticity in 3D). *version* should be 'Von_Mises' or 'Tresca' ('Von_Mises' is the default). Parametrized by Lame coefficients.

compute_isotropic_linearized_Von_Mises_pstrain(*varname*, *data_E*, *data_nu*, *mf_vm*)

Compute the Von-Mises stress of a displacement field for isotropic linearized elasticity in 3D or in 2D with plane strain assumption. Parametrized by Young's modulus and Poisson ratio.

compute_isotropic_linearized_Von_Mises_pstress(*varname*, *data_E*, *data_nu*, *mf_vm*)

Compute the Von-Mises stress of a displacement field for isotropic linearized elasticity in 3D or in 2D with plane stress assumption. Parametrized by Young's modulus and Poisson ratio.

compute_plastic_part(*mim*, *mf_pl*, *varname*, *previous_dep_name*, *projname*, *datalambda*, *datamu*, *datathreshold*, *datasigma*)

Compute on *mf_pl* the plastic part and return it into the vector V. *datasigma* is a vector which contains the stress constraints values supported by the mesh.

compute_second_Piola_Kirchhoff_tensor(*varname*, *lawname*, *dataname*, *mf_sigma*)

Compute on *mf_sigma* the second Piola Kirchhoff stress tensor of a field for nonlinear elasticity in 3D. *lawname* is the constitutive law which could be 'SaintVenant Kirchhoff', 'Mooney Rivlin', 'neo Hookean' or 'Ciarlet Geymonat'. *dataname* is a vector of parameters for the constitutive law. Its length depends on the law. It could be a short vector of constant values or a vector field described on a finite element method for variable coefficients.

contact_brick_set_BN(*indbrick*, *BN*)

Can be used to set the BN matrix of a basic contact/friction brick.

contact_brick_set_BT(*indbrick*, *BT*)

Can be used to set the BT matrix of a basic contact with friction brick.

define_variable_group(*name*, *varname=None*, **args*)

Synopsis: Model.define_variable_group(self, string name[, string varname, ...])

Defines a group of variables for the interpolation (mainly for the raytracing interpolation transformation.

del_macro(*name*)

Delete a previously defined macro for the high generic assembly language.

delete_brick(*ind_brick*)

Delete a variable or a data from the model.

delete_variable(*name*)
Delete a variable or a data from the model.

disable_bricks(*bricks_indices*)
Disable a brick (the brick will no longer participate to the building of the tangent linear system).

disable_variable(*varname*)
Disable a variable for a solve (and its attached multipliers). The next solve will operate only on the remaining variables. This allows to solve separately different parts of a model. If there is a strong coupling of the variables, a fixed point strategy can the be used.

displacement_group_name_of_Nitsche_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

displacement_group_name_of_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

display()
displays a short summary for a Model object.

elastoplasticity_next_iter(*mim*, *varname*, *previous_dep_name*, *projname*, *datalambda*, *datamu*, *datathreshold*, *datasigma*)
Used with the old (obsolete) elastoplasticity brick to pass from an iteration to the next one. Compute and save the stress constraints sigma for the next iterations. 'mim' is the integration method to use for the computation. 'varname' is the main variable of the problem. 'previous_dep_name' represents the displacement at the previous time step. 'projname' is the type of projection to use. For the moment it could only be 'Von Mises' or 'VM'. 'datalambda' and 'datamu' are the Lame coefficients of the material. 'datasigma' is a vector which will contain the new stress constraints values.

enable_bricks(*bricks_indices*)
Enable a disabled brick.

enable_variable(*varname*)
Enable a disabled variable (and its attached multipliers).

finite_strain_elastoplasticity_next_iter(*mim*, *lawname*, *unknowns_type*, *varnames=None*, **args*)
Synopsis: Model.finite_strain_elastoplasticity_next_iter(self, MeshIm mim, string lawname, string unknowns_type, [, string varnames, ...] [, string params, ...] [, int region = -1])

Function that allows to pass from a time step to another for the finite strain plastic brick. The parameters have to be exactly the same than the one of *add_finite_strain_elastoplasticity_brick*, so see the documentation of this function for the explanations. Basically, this brick computes the plastic strain and the plastic multiplier and stores them for the next step. For the Simo-Miehe law which is currently the only one implemented, this function updates the state variables defined in the last two entries of *varnames*, and resets the plastic multiplier field given as the second entry of *varnames*.

first_iter()
To be executed before the first iteration of a time integration scheme.

from_variables(*with_internal=None*)
Return the vector of all the degrees of freedom of the model consisting of the concatenation

of the variables of the model (useful to solve your problem with you own solver).

get_time()
Give the value of the data *t* corresponding to the current time.

get_time_step()
Gives the value of the time step.

interpolation(*expr*, **args*)
Synopsis: V = Model.interpolation(self, string expr, {MeshFem mf | MeshImd mimd | vec pts, Mesh m}[, int region[, int extrapolation[, int rg_source]]])

Interpolate a certain expression with respect to the mesh_fem *mf* or the mesh_im_data *mimd* or the set of points *pts* on mesh *m*. The expression has to be valid according to the high-level generic assembly language possibly including references to the variables and data of the model.

The options *extrapolation* and *rg_source* are specific to interpolations with respect to a set of points *pts*.

interval_of_variable(*varname*)
Gives the interval of the variable *varname* in the linear system of the model.

is_complex()
Return 0 is the model is real, 1 if it is complex.

list_residuals()
print to the output the residuals corresponding to all terms included in the model.

local_projection(*mim*, *expr*, *mf*, *region=None*)
Make an elementwise L2 projection of an expression with respect to the mesh_fem *mf*. This mesh_fem has to be a discontinuous one. The expression has to be valid according to the high-level generic assembly language possibly including references to the variables and data of the model.

matrix_term(*ind_brick*, *ind_term*)
Gives the matrix term ind_term of the brick ind_brick if it exists

memsize()
Return a rough approximation of the amount of memory (in bytes) used by the model.

mesh_fem_of_variable(*name*)
Gives access to the *mesh_fem* of a variable or data.

mult_varname_Dirichlet(*ind_brick*)
Gives the name of the multiplier variable for a Dirichlet brick. If the brick is not a Dirichlet condition with multiplier brick, this function has an undefined behavior

nbdof()
Return the total number of degrees of freedom of the model.

next_iter()
To be executed at the end of each iteration of a time integration scheme.

perform_init_time_derivative(*ddt*)
By calling this function, indicates that the next solve will compute the solution for a (very) small time step *ddt* in order to initalize the data corresponding to the derivatives needed by time integration schemes (mainly the initial time derivative for order one in time problems

and the second order time derivative for second order in time problems). The next solve will not change the value of the variables.

resize_variable(*name*, *sizes*)
Resize a constant size variable of the model. *sizes* is either a integer (for a scalar or vector variable) or a vector of dimensions for a tensor variable. *name* is the variable name.

rhs()
Return the right hand side of the tangent problem.

set_element_extrapolation_correspondence(*transname*, *elt_corr*)
Change the correspondence map of an element extrapolation interpolate transformation.

set_private_matrix(*indbrick*, *B*)
For some specific bricks having an internal sparse matrix (explicit bricks: 'constraint brick' and 'explicit matrix brick'), set this matrix.

set_private_rhs(*indbrick*, *B*)
For some specific bricks having an internal right hand side vector (explicit bricks: 'constraint brick' and 'explicit rhs brick'), set this rhs.

set_time(*t*)
Set the value of the data *t* corresponding to the current time to *t*.

set_time_step(*dt*)
Set the value of the time step to *dt*. This value can be change from a step to another for all one-step schemes (i.e. for the moment to all proposed time integration schemes).

set_variable(*name*, *V*)
Set the value of a variable or data. *name* is the data name.

shift_variables_for_time_integration()
Function used to shift the variables of a model to the data corresponding of ther value on the previous time step for time integration schemes. For each variable for which a time integration scheme has been declared, the scheme is called to perform the shift. This function has to be called between two time steps.

sliding_data_group_name_of_Nitsche_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

sliding_data_group_name_of_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

small_strain_elastoplasticity_Von_Mises(*mim*, *mf_vm*, *lawname*, *unknowns_type*, *varnames=None*, **args*)
Synopsis: V = Model.small_strain_elastoplasticity_Von_Mises(self, MeshIm mim, MeshFem mf_vm, string lawname, string unknowns_type [, string varnames, ...] [, string params, ...] [, string theta = '1' [, string dt = 'timestep']] [, int region])

This function computes the Von Mises stress field with respect to a small strain elastoplasticity term, approximated on *mf_vm*, and stores the result into *VM*. All other parameters have to be exactly the same as for *add_small_strain_elastoplasticity_brick*. Remember that *small_strain_elastoplasticity_next_iter* has to be called before any call of this function.

small_strain_elastoplasticity_next_iter(*mim*, *lawname*, *unknowns_type*, *varnames=None*, **args*)

Synopsis: Model.small_strain_elastoplasticity_next_iter(self, MeshIm mim, string lawname, string unknowns_type [, string varnames, ...] [, string params, ...] [, string theta = '1' [, string dt = 'timestep']] [, int region = -1])

Function that allows to pass from a time step to another for the small strain plastic brick. The parameters have to be exactly the same than the one of *add_small_strain_elastoplasticity_brick*, so see the documentation of this function for the explanations. Basically, this brick computes the plastic strain and the plastic multiplier and stores them for the next step. Additionaly, it copies the computed displacement to the data that stores the displacement of the previous time step (typically 'u' to 'Previous_u'). It has to be called before any use of *compute_small_strain_elastoplasticity_Von_Mises*.

solve(**args*)

Synopsis: (nbit, converged) = Model.solve(self[, ...])

Run the standard getfem solver.

Note that you should be able to use your own solver if you want (it is possible to obtain the tangent matrix and its right hand side with the Model.tangent_matrix() etc.).

Various options can be specified:

- **'noisy' or 'very_noisy'** the solver will display some information showing the progress (residual values etc.).
- **'max_iter', int NIT** set the maximum iterations numbers.
- **'max_res', @float RES** set the target residual value.
- **'diverged_res', @float RES** set the threshold value of the residual beyond which the iterative method is considered to diverge (default is 1e200).
- **'lsolver', string SOLVER_NAME** select explicitely the solver used for the linear systems (the default value is 'auto', which lets getfem choose itself). Possible values are 'superlu', 'mumps' (if supported), 'cg/ildlt', 'gmres/ilu' and 'gmres/ilut'.
- **'lsearch', string LINE_SEARCH_NAME** select explicitely the line search method used for the linear systems (the default value is 'default'). Possible values are 'simplest', 'systematic', 'quadratic' or 'basic'.

Return the number of iterations, if an iterative method is used, and whether the solver converged. For a linear problem, *converged* is 0 when the linear solver failed: singular matrix detected by the solver (zero pivot), or computed solution whose normwise backward error **|K U - F|** / (**|K|** **|U|** + **|F|**) exceeds max(*max_res*, 1e-8) (for instance a solver leaving a zero solution). On a singular but compatible system, a solution may be found while *converged* is 0: it is then not unique. A numerically singular or very ill-conditioned matrix is not always detected, since the backward error of a stable direct solver remains small: check **|K U - F|** / **|F|** if needed.

Note that it is possible to disable some variables (see Model.disable_variable()) in order to solve the problem only with respect to a subset of variables (the disabled variables are then considered as data) for instance to replace the global Newton strategy with a fixed point one.

tangent_matrix()

Return the tangent matrix stored in the model .

test_tangent_matrix(*EPS=None*, **args*)
Synopsis: Model.test_tangent_matrix(self[, scalar EPS[, int NB[, scalar scale]]])

Test the consistency of the tangent matrix in some random positions and random directions (useful to test newly created bricks). *EPS* is the value of the small parameter for the finite difference computation of the derivative is the random direction (default is 1E-6). *NN* is the number of tests (default is 100). *scale* is a parameter for the random position (default is 1, 0 is an acceptable value) around the current position. Each dof of the random position is chosen in the range [current-scale, current+scale].

test_tangent_matrix_term(*varname1*, *varname2*, *EPS=None*, **args*)
Synopsis: Model.test_tangent_matrix_term(self, string varname1, string varname2[, scalar EPS[, int NB[, scalar scale]]])

Test the consistency of a part of the tangent matrix in some random positions and random directions (useful to test newly created bricks). The increment is only made on variable *varname2* and tested on the part of the residual corresponding to *varname1*. This means that only the term (*varname1*, *varname2*) of the tangent matrix is tested. *EPS* is the value of the small parameter for the finite difference computation of the derivative is the random direction (default is 1E-6). *NN* is the number of tests (default is 100). *scale* is a parameter for the random position (default is 1, 0 is an acceptable value) around the current position. Each dof of the random position is chosen in the range [current-scale, current+scale].

to_variables(*V*, *with_internal=None*)
Set the value of the variables of the model with the vector *V*. Typically, the vector *V* results of the solve of the tangent linear system (useful to solve your problem with you own solver).

transformation_name_of_Nitsche_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

transformation_name_of_large_sliding_contact_brick(*indbrick*)
Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

variable(*name*)
Gives the value of a variable or data.

variable_list()
print to the output the list of variables and constants of the model.

8.16 MumpsContext

class MumpsContext(**args*)
GetFEM MumpsContext object

General constructor for MumpsContext objects

- `CTX = MumpsContext(string sym, string datatype)` The argument *sym* should be 'symmetric' or 'general symmetric', 'symmetric positive definite', 'unsymmetric', or empty. Default value (empty) is 'unsymmetric' (MUMPS option 0). The option 'symmetric' is equivalent to 'general symmetric' (MUMPS option 2). The argument *datatype* should be 'real' or 'complex' or empty. Default value (empty) is 'real'.

CNTL(*ind*, *val*)
Set the scalar option at 1-based index *ind* in the CNTL vector of the mumps_context object to value *val*.

Capital naming convention is used to imply Fortran indexing.

ICNTL(*ind*)
Output the ICNTL array entry at 1-based index *ind* stored in the mumps_context object.

Capital naming convention is used to imply Fortran indexing.

INFOG(*ind*)
Output the INFOG array entry at 1-based index *ind* stored in the mumps_context object.

Capital naming convention is used to imply Fortran indexing.

RINFO(*ind*)
Output the RINFO array entry at 1-based index *ind* stored in the mumps_context object.

Capital naming convention is used to imply Fortran indexing.

RINFOG(*ind*)
Output the RINFOG array entry at 1-based index *ind* stored in the mumps_context object.

Capital naming convention is used to imply Fortran indexing.

analyze()
Run the MUMPS analysis job for the mumps_context object.

analyze_and_factorize()
Run the MUMPS analysis and factorization jobs for the mumps_context object.

analyze_factorize_and_solve()
Run the MUMPS analysis, factorization and solve jobs for the mumps_context object. An error check is also performed after the solve.

It returns the solution vector (on all processes if MPI is used).

display()
Display a short summary for a mumps_context object.

distributed_matrix()
The mumps_context object in the scripting API cannot return a reference to the ija matrix stored in C++. It just displays information about the stored matrix.

error_check()
Check the error status of the mumps_context object.

factorize()
Run the MUMPS factorization job for the mumps_context object.

factorize_and_solve()
Run the MUMPS factorization and solve jobs for the mumps_context object.

The analysis job needs to be run first before calling this function. An error check is performed after the solve.

It returns the solution vector (on all processes if MPI is used).

matrix()
The mumps_context object in the scripting API cannot return a reference to the ija matrix stored in C++. It just displays information about the stored matrix.

set_ICNTL(*ind*, *val*)
Set the integer option at 1-based index *ind* in the ICNTL vector of the mumps_context object to value *val*.

Capital naming convention is used to imply Fortran indexing.

set_distributed_matrix(*A*, *rows=None*, **args*)
Synopsis: MumpsContext.set_distributed_matrix(self, mat A[, vec rows[, vec cols]])

Set mat A(rows,cols) as the matrix A for the mumps_context object, distributed over all processes. It also sets ICNTL(5) to 0 and ICNTL(18) to 3.

Optional vectors vec rows and vec cols are used for selecting and/or permuting rows and columns from input matrix mat A. They are 0-based in Python and 1-based in Matlab/Octave.

set_matrix(*A*, *rows=None*, **args*)
Synopsis: MumpsContext.set_matrix(self, mat A[, vec rows[, vec cols]])

Set mat A(rows,cols) as the matrix for the mumps_context object.

Optional vectors vec rows and vec cols are used for selecting and/or permuting rows and columns from input matrix mat A. They are 0-based in Python and 1-based in Matlab/Octave.

set_vector(*b*)
Set the right hand side vec b for the mumps_context object.

solve()
Run the MUMPS solve job (only) for the mumps_context object.

The analysis and factorization jobs need to be run first before calling this function. An error check is performed after the solve.

It returns the solution vector (on all processes if MPI is used).

vector()
Outputs a copy of the vector stored in the mumps_context object. Before the solution, it contains the right hand side of the system. After the solution, it contains the solution.

8.17 Precond

class Precond(**args*)
GetFEM Precond object

The preconditioners may store REAL or COMPLEX values. They accept getfem sparse matrices and Matlab sparse matrices.

General constructor for Precond objects

- `PC = Precond('identity')` Create a REAL identity precondioner.
- `PC = Precond('cidentity')` Create a COMPLEX identity precondioner.
- `PC = Precond('diagonal', vec D)` Create a diagonal precondioner.
- `PC = Precond('ildlt', SpMat m)` Create an ILDLT (Cholesky) preconditioner for the (symmetric) sparse matrix *m*. This preconditioner has the same sparsity pattern than *m* (no fill-in).

- `PC = Precond('ilu', SpMat m)` Create an ILU (Incomplete LU) preconditioner for the sparse matrix *m*. This preconditioner has the same sparsity pattern than *m* (no fill-in).
- `PC = Precond('ildltt', SpMat m[, int fillin[, scalar threshold]])` Create an ILDLTT (Cholesky with filling) preconditioner for the (symmetric) sparse matrix *m*. The preconditioner may add at most *fillin* additional non-zero entries on each line. The default value for *fillin* is 10, and the default threshold is1e-7.
- `PC = Precond('ilut', SpMat m[, int fillin[, scalar threshold]])` Create an ILUT (Incomplete LU with filling) preconditioner for the sparse matrix *m*. The preconditioner may add at most *fillin* additional non-zero entries on each line. The default value for *fillin* is 10, and the default threshold is 1e-7.
- `PC = Precond('superlu', SpMat m)` Uses SuperLU to build an exact factorization of the sparse matrix *m*. This preconditioner is only available if the getfem-interface was built with SuperLU support. Note that LU factorization is likely to eat all your memory for 3D problems.
- `PC = Precond('spmat', SpMat m)` Preconditioner given explicitely by a sparse matrix.

char()
Output a (unique) string representation of the Precond.

This can be used to perform comparisons between two different Precond objects. This function is to be completed.

display()
displays a short summary for a Precond object.

is_complex()
Return 1 if the preconditioner stores complex values.

mult(*V*)
Apply the preconditioner to the supplied vector.

size()
Return the dimensions of the preconditioner.

tmult(*V*)
Apply the transposed preconditioner to the supplied vector.

type()
Return a string describing the type of the preconditioner ('ilu', 'ildlt',..).

8.18 Slice

class Slice(**args*)
GetFEM Slice object

Creation of a mesh slice. Mesh slices are very similar to a P1-discontinuous MeshFem on which interpolation is very fast. The slice is built from a mesh object, and a description of the slicing operation, for example:

```
sl = Slice(('planar',+1,[[0],[0]],[[0],[1]]), m, 5)
```

cuts the original mesh with the half space {y>0}. Each convex of the original Mesh *m* is simplexified (for example a quadrangle is splitted into 2 triangles), and each simplex is refined 5 times.

Slicing operations can be:

- cutting with a plane, a sphere or a cylinder
- intersection or union of slices
- isovalues surfaces/volumes
- "points", "streamlines" (see below)

If the first argument is a MeshFem *mf* instead of a Mesh, and if it is followed by a *mf*-field *u*, then the deformation *u* will be applied to the mesh before the slicing operation.

The first argument can also be a slice.

General constructor for Slice objects

- `sl = Slice(sliceop, {Slice sl|{Mesh m| MeshFem mf, vec U}, int refine}[, mat CVfids])` Create a Slice using *sliceop* operation.

  sliceop operation is specified with Tuple or List, do not forget the extra parentheses!. The first element is the name of the operation, followed the slicing options:

  - ('none') : Does not cut the mesh.
  - ('planar', int orient, vec p, vec n) : Planar cut. *p* and *n* define a half-space, *p* being a point belong to the boundary of the half-space, and *n* being its normal. If *orient* is equal to -1 (resp. 0, +1), then the slicing operation will cut the mesh with the "interior" (resp. "boundary", "exterior") of the half-space. *orient* may also be set to +2 which means that the mesh will be sliced, but both the outer and inner parts will be kept.
  - ('ball', int orient, vec c, scalar r) : Cut with a ball of center *c* and radius *r*.
  - ('cylinder', int orient, vec p1, vec p2, scalar r) : Cut with a cylinder whose axis is the line *(p1, p2)* and whose radius is *r*.
  - ('isovalues', int orient, MeshFem mf, vec U, scalar s) : Cut using the isosurface of the field *U* (defined on the MeshFem *mf*). The result is the set *{x such that :math:`U(x) leq s}`* or *{x such that `U`(x)=`s}`* or *{x such that `U`(x) >= `s}`* depending on the value of *orient*.
  - ('boundary'[, SLICEOP]) : Return the boundary of the result of SLICEOP, where SLICEOP is any slicing operation. If SLICEOP is not specified, then the whole mesh is considered (i.e. it is equivalent to ('boundary',{'none'})).
  - ('explode', mat Coef) : Build an 'exploded' view of the mesh: each convex is shrinked ($0 < \text{Coef} \leq 1$). In the case of 3D convexes, only their faces are kept.
  - ('union', SLICEOP1, SLICEOP2) : Returns the union of slicing operations.
  - ('intersection', SLICEOP1, SLICEOP2) : Returns the intersection of slicing operations, for example:

    ```
    sl = Slice(('intersection',('planar',+1,[[0],[0],[0]],[[0],[0],
    ↪[1]]),
                                ('isovalues',-1,mf2,u2,0)),mf,u,5)
    ```

  - ('comp', SLICEOP) : Returns the complementary of slicing operations.
  - ('diff', SLICEOP1, SLICEOP2) : Returns the difference of slicing operations.

  - ('mesh', Mesh m) : Build a slice which is the intersection of the sliced mesh with another mesh. The slice is such that all of its simplexes are stricly contained into a convex of each mesh.

- `sl = Slice('streamlines', MeshFem mf, mat U, mat S)` Compute streamlines of the (vector) field *U*, with seed points given by the columns of *S*.
- `sl = Slice('points', Mesh m, mat Pts)` Return the "slice" composed of points given by the columns of *Pts* (useful for interpolation on a given set of sparse points, see `gf_compute('interpolate on',sl)`).
- `sl = Slice('load', string filename[, Mesh m])` Load the slice (and its linked mesh if it is not given as an argument) from a text file.

area()
Return the area of the slice.

char()
Output a (unique) string representation of the Slice.

This can be used to perform comparisons between two different Slice objects. This function is to be completed.

cvs()
Return the list of convexes of the original mesh contained in the slice.

dim()
Return the dimension of the slice (2 for a 2D mesh, etc..).

display()
displays a short summary for a Slice object.

edges()
Return the edges of the linked mesh contained in the slice.

P contains the list of all edge vertices, *E1* contains the indices of each mesh edge in *P*, and *E2* contains the indices of each "edges" which is on the border of the slice. This function is useless except for post-processing purposes.

export_to_dx(*filename*, **args*)
Synopsis: Slice.export_to_dx(self, string filename, ...)

Export a slice to OpenDX.

Following the *filename*, you may use any of the following options:

- if 'ascii' is not used, the file will contain binary data (non portable, but fast).
- if 'edges' is used, the edges of the original mesh will be written instead of the slice content.
- if 'append' is used, the opendx file will not be overwritten, and the new data will be added at the end of the file.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor), followed by an optional name.
- a mesh_fem and a field, followed by an optional name.

export_to_pos(*filename*, *name=None*, **args*)
Synopsis: Slice.export_to_pos(self, string filename[, string name][[,MeshFem mf1], mat U1, string nameU1[[,MeshFem mf1], mat U2, string nameU2,...])

Export a slice to Gmsh.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor).
- a mesh_fem and a field.

export_to_pov(*filename*)
Export a the triangles of the slice to POV-RAY.

export_to_vtk(*filename*, **args*)
Synopsis: Slice.export_to_vtk(self, string filename, ...)

Export a slice to VTK.

Following the *filename*, you may use any of the following options:

- if 'ascii' is not used, the file will contain binary data (non portable, but fast).
- if 'edges' is used, the edges of the original mesh will be written instead of the slice content.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor), followed by an optional name.
- a mesh_fem and a field, followed by an optional name.

Examples:

- Slice.export_to_vtk('test.vtk', Usl, 'first_dataset', mf, U2, 'second_dataset')
- Slice.export_to_vtk('test.vtk', 'ascii', mf,U2)
- Slice.export_to_vtk('test.vtk', 'edges', 'ascii', Uslice)

export_to_vtu(*filename*, **args*)
Synopsis: Slice.export_to_vtu(self, string filename, ...)

Export a slice to VTK(XML).

Following the *filename*, you may use any of the following options:

- if 'ascii' is not used, the file will contain binary data (non portable, but fast).
- if 'edges' is used, the edges of the original mesh will be written instead of the slice content.

More than one dataset may be written, just list them. Each dataset consists of either:

- a field interpolated on the slice (scalar, vector or tensor), followed by an optional name.
- a mesh_fem and a field, followed by an optional name.

Examples:

- Slice.export_to_vtu('test.vtu', Usl, 'first_dataset', mf, U2, 'second_dataset')
- Slice.export_to_vtu('test.vtu', 'ascii', mf,U2)
- Slice.export_to_vtu('test.vtu', 'edges', 'ascii', Uslice)

interpolate_convex_data(*Ucv*)
Interpolate data given on each convex of the mesh to the slice nodes.

The input array *Ucv* may have any number of dimensions, but its last dimension should be equal to Mesh.max_cvid().

Example of use: Slice.interpolate_convex_data(Mesh.quality()).

linked_mesh()
Return the mesh on which the slice was taken.

memsize()
Return the amount of memory (in bytes) used by the slice object.

mesh()
Return the mesh on which the slice was taken (identical to 'linked mesh')

nbpts()
Return the number of points in the slice.

nbsplxs(*dim=None*)
Return the number of simplexes in the slice.

Since the slice may contain points (simplexes of dim 0), segments (simplexes of dimension 1), triangles etc., the result is a vector of size Slice.dim()+1, except if the optional argument *dim* is used.

pts()
Return the list of point coordinates.

set_pts(*P*)
Replace the points of the slice.

The new points *P* are stored in the columns the matrix. Note that you can use the function to apply a deformation to a slice, or to change the dimension of the slice (the number of rows of *P* is not required to be equal to Slice.dim()).

splxs(*dim*)
Return the list of simplexes of dimension *dim*.

On output, S has 'dim+1' rows, each column contains the point numbers of a simplex. The vector *CV2S* can be used to find the list of simplexes for any convex stored in the slice. For example 'S[:,CV2S[4]:CV2S[5]]' gives the list of simplexes for the fourth convex.

8.19 Spmat

class Spmat(**args*)
GetFEM Spmat object

Create a new sparse matrix in GetFEM format. These sparse matrix can be stored as CSC (compressed column sparse), which is the format used by Matlab, or they can be stored as WSC (internal format to getfem). The CSC matrices are not writable (it would be very inefficient), but they are optimized for multiplication with vectors, and memory usage. The WSC are writable, they are very fast with respect to random read/write operation. However their memory overhead is higher than CSC matrices, and they are a little bit slower for matrix-vector multiplications.

By default, all newly created matrices are build as WSC matrices. This can be changed later with `Spmat.to_csc(...)`, or may be changed automatically by getfem (for example `gf_linsolve()` converts the matrices to CSC).

The matrices may store REAL or COMPLEX values.

General constructor for Spmat objects

- `SM = Spmat('empty', int m [, int n])` Create a new empty (i.e. full of zeros) sparse matrix, of dimensions *m x n*. If *n* is omitted, the matrix dimension is *m x m*.
- `SM = Spmat('copy', mat K [, list I [, list J=I]])` Duplicate a matrix *K* (which might be an SpMat). If index *I* and/or *J* are given, the matrix will be a submatrix of *K*. For example:

```
m = Spmat('copy', Spmat('empty',50,50), range(40), [6, 7, 8, 3, 10])
```

  will return a 40x5 matrix.
- `SM = Spmat('identity', int n)` Create a *n x n* identity matrix.
- `SM = Spmat('mult', Spmat A, Spmat B)` Create a sparse matrix as the product of the sparse matrices *A* and *B*. It requires that *A* and *B* be both real or both complex, you may have to use `Spmat.to_complex()`.
- `SM = Spmat('add', Spmat A, Spmat B)` Create a sparse matrix as the sum of the sparse matrices *A* and *B*. Adding a real matrix with a complex matrix is possible.
- `SM = Spmat('diag', mat D [, ivec E [, int n [,int m]]])` Create a diagonal matrix. If *E* is given, *D* might be a matrix and each column of *E* will contain the sub-diagonal number that will be filled with the corresponding column of *D*.
- `SM = Spmat('load','hb'|'harwell-boeing'|'mm'|'matrix-market', string filename)` Read a sparse matrix from an Harwell-Boeing or a Matrix-Market file .

add(*I*, *J*, *V*)
Add *V* to the sub-matrix 'M(I,J)'.

V might be a sparse matrix or a full matrix.

assign(*I*, *J*, *V*)
Copy V into the sub-matrix 'M(I,J)'.

V might be a sparse matrix or a full matrix.

char()
Output a (unique) string representation of the Spmat.

This can be used to perform comparisons between two different Spmat objects. This function is to be completed.

clear(*I=None*, **args*)
Synopsis: Spmat.clear(self[, list I[, list J]])

Erase the non-zero entries of the matrix.

The optional arguments *I* and *J* may be specified to clear a sub-matrix instead of the entire matrix.

conjugate()
Conjugate each element of the matrix.

csc_ind()
Return the two usual index arrays of CSC storage.

If *M* is not stored as a CSC matrix, it is converted into CSC.

csc_val()
Return the array of values of all non-zero entries of *M*.

If *M* is not stored as a CSC matrix, it is converted into CSC.

determinant()
returns the matrix determinant calculated using MUMPS.

diag(*E=None*)
Return the diagonal of *M* as a vector.

If *E* is used, return the sub-diagonals whose ranks are given in E.

dirichlet_nullspace(*R*)
Solve the dirichlet conditions *M.U=R*.

A solution *U0* which has a minimum L2-norm is returned, with a sparse matrix *N* containing an orthogonal basis of the kernel of the (assembled) constraints matrix *M* (hence, the PDE linear system should be solved on this subspace): the initial problem

$K.U = B$ with constraints $M.U = R$

is replaced by

$(N'.K.N).UU = N'.B$ with $U = N.UU + U0$

display()
displays a short summary for a Spmat object.

full(*I=None*, **args*)
Synopsis: Sm = Spmat.full(self[, list I[, list J]])

Return a full (sub-)matrix.

The optional arguments *I* and *J*, are the sub-intervals for the rows and columns that are to be extracted.

is_complex()
Return 1 if the matrix contains complex values.

mult(*V*)
Product of the sparse matrix *M* with a vector *V*.

For matrix-matrix multiplications, see Spmat('mult').

nnz()
Return the number of non-null values stored in the sparse matrix.

save(*format*, *filename*)
Export the sparse matrix.

the format of the file may be 'hb' for Harwell-Boeing, or 'mm' for Matrix-Market.

scale(*v*)
Multiplies the matrix by a scalar value *v*.

set_diag(*D*, *E=None*)
Change the diagonal (or sub-diagonals) of the matrix.

If *E* is given, *D* might be a matrix and each column of *E* will contain the sub-diagonal number that will be filled with the corresponding column of *D*.

size()
Return a vector where *ni* and *nj* are the dimensions of the matrix.

storage()
Return the storage type currently used for the matrix.

The storage is returned as a string, either 'CSC' or 'WSC'.

tmult(*V*)
Product of *M* transposed (conjugated if *M* is complex) with the vector *V*.

to_complex()
Store complex numbers.

to_csc()
Convert the matrix to CSC storage.

CSC storage is recommended for matrix-vector multiplications.

to_wsc()
Convert the matrix to WSC storage.

Read and write operation are quite fast with WSC storage.

transconj()
Transpose and conjugate the matrix.

transpose()
Transpose the matrix.

8.20 Module asm

asm_generic(*mim*, *order*, *expression*, *region*, *model=None*, **args*)
Synopsis: (...) = asm_generic(MeshIm mim, int order, string expression, int region, [Model model, ['Secondary_domain', 'name',]] [string varname, int is_variable[, {MeshFem mf, MeshImd mimd}], value], ['select_output', 'varname1'[, 'varname2]], ...)

High-level generic assembly procedure for volumic or boundary assembly.

Performs the generic assembly of *expression* with the integration method *mim* on the mesh region of index *region* (-1 means all elements of the mesh). The same mesh should be shared by the integration method and all the finite element methods or mesh_im_data corresponding to the variables.

order indicates either that the (scalar) potential (order = 0) or the (vector) residual (order = 1) or the tangent (matrix) (order = 2) is to be computed.

model is an optional parameter allowing to take into account all variables and data of a model. Note that all enabled variables of the model will occupy space in the returned vector/matrix corresponding to their degrees of freedom in the global system, even if they are not present in *expression*.

The variables and constants (data) are listed after the region number (or optionally the model). For each variable/constant, a name must be given first (as it is referred in the assembly string), then an integer equal to 1 or 0 is expected respectively for declaring a variable or a constant, then the finite element method if it is a fem variable/constant or the mesh_im_data if it is data defined on integration points, and the vector representing the value of the variable/constant. It is possible to give an arbitrary number of variable/constant. The difference between a variable and a constant is that test functions are only available for variables, not for constants.

select_output is an optional parameter which allows to reduce the output vector (for *order* equal to 1) or the matrix (for *order* equal to 2) to the degrees of freedom of the specified variables. One variable has to be specified for a vector output and two for a matrix output.

Note that if several variables are given, the assembly of the tangent matrix/residual vector will be done considering the order in the call of the function (the degrees of freedom of the first variable, then of the second one, and so on). If a model is provided, all degrees of freedom of the model will be counted first, even if some of the model variables do not appear in *expression*.

For example, the L2 norm of a vector field "u" can be computed with:

```
gf_compute('L2 norm') or with the square root of:

gf_asm('generic', mim, 0, 'u.u', -1, 'u', 1, mf, U);
```

The nonhomogeneous Laplacian stiffness matrix of a scalar field can be evaluated with:

```
gf_asm('laplacian', mim, mf, mf_data, A) or equivalently with:

gf_asm('generic', mim, 2, 'A*Grad_Test2_u.Grad_Test_u', -1, 'u', 1, mf, U,
↪ 'A', 0, mf_data, A);
```

asm_mass_matrix(*mim*, *mf1*, *mf2=None*, **args*)

Synopsis: M = asm_mass_matrix(MeshIm mim, MeshFem mf1[, MeshFem mf2[, int region]])

Assembly of a mass matrix.

Return a SpMat object.

asm_laplacian(*mim*, *mf_u*, *mf_d*, *a*, *region=None*)

Assembly of the matrix for the Laplacian problem.

$\nabla \cdot (a(x)\nabla u)$ with *a* a scalar.

Return a SpMat object.

asm_linear_elasticity(*mim*, *mf_u*, *mf_d*, *lambda_d*, *mu_d*, *region=None*)

Assembles of the matrix for the linear (isotropic) elasticity problem.

$\nabla \cdot (C(x) : \nabla u)$ with C defined via *lambda_d* and *mu_d*.

Return a SpMat object.

asm_nonlinear_elasticity(*mim*, *mf_u*, *U*, *law*, *mf_d*, *params*, **args*)

Synopsis: TRHS = asm_nonlinear_elasticity(MeshIm mim, MeshFem mf_u, vec U, string law, MeshFem mf_d, mat params, {'tangent matrix'|'rhs'|'incompressible tangent matrix', MeshFem mf_p, vec P|'incompressible rhs', MeshFem mf_p, vec P})

Assembles terms (tangent matrix and right hand side) for nonlinear elasticity.

The solution U is required at the current time-step. The *law* may be choosen among:

- 'SaintVenant Kirchhoff': Linearized law, should be avoided. This law has the two usual Lame coefficients as parameters, called lambda and mu.
- 'Mooney Rivlin': This law has three parameters, called C1, C2 and D1. Can be preceded with the words 'compressible' or 'incompressible' to force a specific version. By default, the incompressible version is considered which requires only the first two material coefficients.
- 'neo Hookean': A special case of the 'Mooney Rivlin' law that requires one material coefficient less (C2 = 0). By default, its compressible version is used.
- 'Ciarlet Geymonat': This law has 3 parameters, called lambda, mu and gamma, with gamma chosen such that gamma is in]-lambda/2-mu, -mu[.

The parameters of the material law are described on the MeshFem *mf_d*. The matrix *params* should have *nbdof(mf_d)* columns, each row correspounds to a parameter.

The last argument selects what is to be built: either the tangent matrix, or the right hand side. If the incompressibility is considered, it should be followed by a MeshFem *mf_p*, for the pression.

Return a SpMat object (tangent matrix), vec object (right hand side), tuple of SpMat objects (incompressible tangent matrix), or tuple of vec objects (incompressible right hand side).

asm_helmholtz(*mim*, *mf_u*, *mf_d*, *k*, *region=None*)
Assembly of the matrix for the Helmholtz problem.

$\Delta u + k^2 u = 0$, with *k* complex scalar.

Return a SpMat object.

asm_bilaplacian(*mim*, *mf_u*, *mf_d*, *a*, *region=None*)
Assembly of the matrix for the Bilaplacian problem.

$\Delta(a(x)\Delta u) = 0$ with *a* scalar.

Return a SpMat object.

asm_bilaplacian_KL(*mim*, *mf_u*, *mf_d*, *a*, *nu*, *region=None*)
Assembly of the matrix for the Bilaplacian problem with Kirchhoff-Love formulation.

$\Delta(a(x)\Delta u) = 0$ with *a* scalar.

Return a SpMat object.

asm_volumic_source(*mim*, *mf_u*, *mf_d*, *fd*, *region=None*)
Assembly of a volumic source term.

Output a vector *V*, assembled on the MeshFem *mf_u*, using the data vector *fd* defined on the data MeshFem *mf_d*. *fd* may be real or complex-valued.

Return a vec object.

asm_boundary_source(*bnum*, *mim*, *mf_u*, *mf_d*, *G*)
Assembly of a boundary source term.

G should be a [Qdim x N] matrix, where N is the number of dof of *mf_d*, and Qdim is the dimension of the unkown u (that is set when creating the MeshFem).

Return a vec object.

asm_dirichlet(*bnum*, *mim*, *mf_u*, *mf_d*, *H*, *R*, *threshold=None*)
Assembly of Dirichlet conditions of type *h.u = r*.

Handle *h.u = r* where h is a square matrix (of any rank) whose size is equal to the dimension of the unkown u. This matrix is stored in *H*, one column per dof in *mf_d*, each column containing the values of the matrix h stored in fortran order:

$$`H(:,j) = [h11(x_j)h21(x_j)h12(x_j)h22(x_j)]`$$

if u is a 2D vector field.

Of course, if the unknown is a scalar field, you just have to set *H = ones(1, N)*, where N is the number of dof of *mf_d*.

This is basically the same than calling gf_asm('boundary qu term') for *H* and calling gf_asm('neumann') for *R*, except that this function tries to produce a 'better' (more diagonal) constraints matrix (when possible).

See also Spmat.Dirichlet_nullspace().

asm_boundary_qu_term(*boundary_num*, *mim*, *mf_u*, *mf_d*, *q*)
Assembly of a boundary qu term.

q should be be a [Qdim x Qdim x N] array, where N is the number of dof of *mf_d*, and Qdim is the dimension of the unkown u (that is set when creating the MeshFem).

Return a SpMat object.

asm_define_function(*name*, *nb_args*, *expression*, *expression_derivative_t=None*, **args*)
Synopsis: asm_define_function(string name, int nb_args, string expression[, string expression_derivative_t[, string expression_derivative_u]])

Define a new function *name* which can be used in high level generic assembly. The function can have one or two parameters. In *expression* all available predefined function or operation of the generic assembly can be used. However, no reference to some variables or data can be specified. The argument of the function is *t* for a one parameter function and *t* and *u* for a two parameter function. For instance 'sin(pi*t)+2*t*t' is a valid expression for a one parameter function and 'sin(max(t,u)*pi)' is a valid expression for a two parameters function. *expression_derivative_t* and *expression_derivative_u* are optional expressions for the derivatives with respect to *t* and *u*. If they are not furnished, a symbolic derivation is used.

asm_undefine_function(*name*)
Cancel the definition of a previously defined function *name* for the high level generic assembly.

asm_define_linear_hardening_function(*name*, *sigma_y0*, *H*, **args*)
Synopsis: asm_define_linear_hardening_function(string name, scalar sigma_y0, scalar H, ... [string 'Frobenius'])

Define a new linear hardening function under the name *name*, with initial yield stress *sigma_y0* and hardening modulus H. If an extra string argument with the value 'Frobenius' is provided, the hardening function is expressed in terms of Frobenius norms of its input strain and output stress, instead of their Von-Mises equivalents.

asm_define_Ramberg_Osgood_hardening_function(*name*, *sigma_ref*, **args*)
Synopsis: asm_define_Ramberg_Osgood_hardening_function(string name, scalar sigma_ref, {scalar eps_ref | scalar E, scalar alpha}, scalar n[, string 'Frobenius'])

Define a new Ramberg Osgood hardening function under the name *name*, with initial yield stress *sigma_y0* and hardening modulus H. If an extra string argument with the value 'Frobenius' is provided, the hardening function is expressed in terms of Frobenius norms of its input strain and output stress, instead of their Von-Mises equivalents.

asm_expression_analysis(*expression*, **args*)

Synopsis: asm_expression_analysis(string expression [, {Mesh mesh | MeshIm mim}] [, der_order] [, Model model] [, string varname, int is_variable[, {MeshFem mf | MeshImd mimd}], ...])

Analyse a high-level generic assembly expression and print information about the provided expression.

asm_volumic(*CVLST=None*, **args*)

Synopsis: (...) = asm_volumic(,CVLST], expr [, mesh_ims, mesh_fems, data...])

Low-level generic assembly procedure for volumic assembly.

The expression *expr* is evaluated over the MeshFem's listed in the arguments (with optional data) and assigned to the output arguments. For details about the syntax of assembly expressions, please refer to the getfem user manual (or look at the file getfem_assembling.h in the GetFEM sources).

For example, the L2 norm of a field can be computed with:

```
gf_compute('L2 norm') or with the square root of:

gf_asm('volumic','u=data(#1); V()+=u(i).u(j).comp(Base(#1).Base(#1))(i,j)
↪',mim,mf,U)
```

The Laplacian stiffness matrix can be evaluated with:

```
gf_asm('laplacian',mim, mf, mf_data, A) or equivalently with:

gf_asm('volumic','a=data(#2);M(#1,#1)+=sym(comp(Grad(#1).Grad(#1).Base(
↪#2))(:,i,:,i,j).a(j))', mim,mf,mf_data,A);
```

asm_boundary(*bnum*, *expr*, *mim=None*, *mf=None*, *data=None*, **args*)

Synopsis: (...) = asm_boundary(int bnum, string expr [, MeshIm mim, MeshFem mf, data...])

Low-level generic boundary assembly.

See the help for gf_asm('volumic').

asm_interpolation_matrix(*mf*, **args*)

Synopsis: Mi = asm_interpolation_matrix(MeshFem mf, {MeshFem mfi | vec pts})

Build the interpolation matrix from a MeshFem onto another MeshFem or a set of points.

Return a matrix *Mi*, such that $V = Mi.U$ is equal to gf_compute('interpolate_on',mfi). Useful for repeated interpolations. Note that this is just interpolation, no elementary integrations are involved here, and *mfi* has to be lagrangian. In the more general case, you would have to do a L2 projection via the mass matrix.

Mi is a SpMat object.

asm_extrapolation_matrix(*mf*, **args*)

Synopsis: Me = asm_extrapolation_matrix(MeshFem mf, {MeshFem mfe | vec pts})

Build the extrapolation matrix from a MeshFem onto another MeshFem or a set of points.

Return a matrix *Me*, such that $V = Me.U$ is equal to gf_compute('extrapolate_on',mfe). Useful for repeated extrapolations.

Me is a SpMat object.

asm_integral_contact_Uzawa_projection(*bnum*, *mim*, *mf_u*, *U*, *mf_lambda*, *vec_lambda*, *mf_obstacle*, *obstacle*, *r*, **args*)

Synopsis: B = asm_integral_contact_Uzawa_projection(int bnum, MeshIm mim, MeshFem mf_u, vec U, MeshFem mf_lambda, vec vec_lambda, MeshFem mf_obstacle, vec obstacle, scalar r [, {scalar coeff | MeshFem mf_coeff, vec coeff} [, int option[, scalar alpha, vec W]]])

Specific assembly procedure for the use of an Uzawa algorithm to solve contact problems. Projects the term $-(lambda - r (u_N-g))_-$ on the finite element space of $lambda$.

Return a vec object.

asm_level_set_normal_source_term(*bnum*, *mim*, *mf_u*, *mf_lambda*, *vec_lambda*, *mf_levelset*, *levelset*)

Performs an assembly of the source term represented by *vec_lambda* on *mf_lambda* considered to be a component in the direction of the gradient of a levelset function (normal to the levelset) of a vector field defined on *mf_u* on the boundary *bnum*.

Return a vec object.

asm_lsneuman_matrix(*mim*, *mf1*, *mf2*, *ls*, *region=None*)

Assembly of a level set Neuman matrix.

Return a SpMat object.

asm_nlsgrad_matrix(*mim*, *mf1*, *mf2*, *ls*, *region=None*)

Assembly of a nlsgrad matrix.

Return a SpMat object.

asm_stabilization_patch_matrix(*mesh*, *mf*, *mim*, *ratio*, *h*)

Assembly of stabilization patch matrix .

Return a SpMat object.

8.21 Module compute

compute_L2_norm(*MF*, *U*, *mim*, *CVids=None*)

Compute the L2 norm of the (real or complex) field *U*.

If *CVids* is given, the norm will be computed only on the listed elements.

compute_L2_dist(*MF*, *U*, *mim*, *mf2*, *U2*, *CVids=None*)

Compute the L2 distance between *U* and *U2*.

If *CVids* is given, the norm will be computed only on the listed elements.

compute_H1_semi_norm(*MF*, *U*, *mim*, *CVids=None*)

Compute the L2 norm of grad(*U*).

If *CVids* is given, the norm will be computed only on the listed elements.

compute_H1_semi_dist(*MF*, *U*, *mim*, *mf2*, *U2*, *CVids=None*)

Compute the semi H1 distance between *U* and *U2*.

If *CVids* is given, the norm will be computed only on the listed elements.

compute_H1_norm(*MF*, *U*, *mim*, *CVids=None*)

Compute the H1 norm of *U*.

If *CVids* is given, the norm will be computed only on the listed elements.

compute_H2_semi_norm(*MF*, *U*, *mim*, *CVids=None*)
Compute the L2 norm of D^2(*U*).

If *CVids* is given, the norm will be computed only on the listed elements.

compute_H2_norm(*MF*, *U*, *mim*, *CVids=None*)
Compute the H2 norm of *U*.

If *CVids* is given, the norm will be computed only on the listed elements.

compute_gradient(*MF*, *U*, *mf_du*)
Compute the gradient of the field *U* defined on MeshFem *mf_du*.

The gradient is interpolated on the MeshFem *mf_du*, and returned in *DU*. For example, if *U* is defined on a P2 MeshFem, *DU* should be evaluated on a P1-discontinuous MeshFem. *mf* and *mf_du* should share the same mesh.

U may have any number of dimensions (i.e. this function is not restricted to the gradient of scalar fields, but may also be used for tensor fields). However the last dimension of *U* has to be equal to the number of dof of *mf*. For example, if *U* is a [3x3xNmf] array (where Nmf is the number of dof of *mf*), *DU* will be a [Nx3x3[xQ]xNmf_du] array, where N is the dimension of the mesh, Nmf_du is the number of dof of *mf_du*, and the optional Q dimension is inserted if *Qdim_mf != Qdim_mf_du*, where Qdim_mf is the Qdim of *mf* and Qdim_mf_du is the Qdim of *mf_du*.

compute_hessian(*MF*, *U*, *mf_h*)
Compute the hessian of the field *U* defined on MeshFem *mf_h*.

See also gf_compute('gradient', MeshFem mf_du).

compute_eval_on_triangulated_surface(*MF*, *U*, *Nrefine*, *CVLIST=None*)
[OBSOLETE FUNCTION! will be removed in a future release] Utility function designed for 2D triangular meshes : returns a list of triangles coordinates with interpolated U values. This can be used for the accurate visualization of data defined on a discontinous high order element. On output, the six first rows of UP contains the triangle coordinates, and the others rows contain the interpolated values of U (one for each triangle vertex) CVLIST may indicate the list of convex number that should be consider, if not used then all the mesh convexes will be used. U should be a row vector.

compute_interpolate_on(*MF*, *U*, **args*)
Synopsis: Ui = compute_interpolate_on(MeshFem MF, vec U, {MeshFem mfi | Slice sli | vec pts})

Interpolate a field on another MeshFem or a Slice or a list of points.

- **Interpolation on another MeshFem *mfi*:** *mfi* has to be Lagrangian. If *mf* and *mfi* share the same mesh object, the interpolation will be much faster.
- **Interpolation on a Slice *sli*:** this is similar to interpolation on a refined P1-discontinuous mesh, but it is much faster. This can also be used with Slice('points') to obtain field values at a given set of points.
- Interpolation on a set of points *pts*

See also gf_asm('interpolation matrix')

compute_extrapolate_on(*MF*, *U*, *mfe*)
Extrapolate a field on another MeshFem.

If the mesh of *mfe* is stricly included in the mesh of *mf*, this function does stricly the same job as gf_compute('interpolate_on'). However, if the mesh of *mfe* is not exactly included in *mf* (imagine interpolation between a curved refined mesh and a coarse mesh), then values which are outside *mf* will be extrapolated.

See also gf_asm('extrapolation matrix')

compute_error_estimate(*MF*, *U*, *mim*)
Compute an a posteriori error estimate.

Currently there is only one which is available: for each convex, the jump of the normal derivative is integrated on its faces.

compute_error_estimate_nitsche(*MF*, *U*, *mim*, *GAMMAC*, *GAMMAN*, *lambda_*, *mu_*, *gamma0*, *f_coeff*, *vertical_force*)
Compute an a posteriori error estimate in the case of Nitsche method.

Currently there is only one which is available: for each convex, the jump of the normal derivative is integrated on its faces.

compute_convect(*MF*, *U*, *mf_v*, *V*, *dt*, *nt*, *option=None*, **args*)
Synopsis: compute_convect(MeshFem MF, vec U, MeshFem mf_v, vec V, scalar dt, int nt[, string option[, vec per_min, vec per_max]])

Compute a convection of *U* with regards to a steady state velocity field *V* with a Characteristic-Galerkin method. The result is returned in-place in *U*. This method is restricted to pure Lagrange fems for U. *mf_v* should represent a continuous finite element method. *dt* is the integration time and *nt* is the number of integration step on the caracteristics. *option* is an option for the part of the boundary where there is a re-entrant convection. *option* = *'extrapolation'* for an extrapolation on the nearest element, *option* = *'unchanged'* for a constant value on that boundary or *option* = *'periodicity'* for a peridiodic boundary. For this latter option the two vectors per_min, per_max has to be given and represent the limits of the periodic domain (on components where per_max[k] < per_min[k] no operation is done). This method is rather dissipative, but stable.

8.22 Module delete

delete(*I*, *J=None*, *K=None*, **args*)
Synopsis: delete(I[, J, K,...])

I should be a descriptor given by gf_mesh(), gf_mesh_im(), gf_slice() etc.

Note that if another object uses I, then object I will be deleted only when both have been asked for deletion.

Only objects listed in the output of gf_workspace('stats') can be deleted (for example gf_fem objects cannot be destroyed).

You may also use gf_workspace('clear all') to erase everything at once.

8.23 Module linsolve

linsolve_gmres(*M*, *b*, *restart=None*, **args*)
Synopsis: X = linsolve_gmres(SpMat M, vec b[, int restart][, Mrecond P][,'noisy'][,'res', r][,'maxiter', n])

Solve $M.X = b$ with the generalized minimum residuals method.

Optionally using P as preconditioner. The default value of the restart parameter is 50.

linsolve_cg(*M*, *b*, *P=None*, **args*)
Synopsis: X = linsolve_cg(SpMat M, vec b [, Mrecond P][,'noisy'][,'res', r][,'maxiter', n])

Solve $M.X = b$ with the conjugated gradient method.

Optionally using P as preconditioner.

linsolve_bicgstab(*M*, *b*, *P=None*, **args*)
Synopsis: X = linsolve_bicgstab(SpMat M, vec b [, Mrecond P][,'noisy'][,'res', r][,'maxiter', n])

Solve $M.X = b$ with the bi-conjugated gradient stabilized method.

Optionally using P as a preconditioner.

linsolve_lu(*M*, *b*)
Alias for gf_linsolve('superlu',...)

linsolve_superlu(*M*, *b*)
Solve $M.U = b$ apply the SuperLU solver (sparse LU factorization).

The condition number estimate *cond* is returned with the solution U.

linsolve_mumps(*M*, *b*, **args*)
Synopsis: (U, cond) = linsolve_mumps(SpMat M, vec b, ... ['sym'])

Solve $M.U = b$ using the MUMPS solver.

The right hand side b can optionally by a matrix with several columns in order to solve multiple right hand sides at once.

If the option *sym* is provided, the symmetric version of the MUMPS solver will be used.

8.24 Module poly

poly_print(*P*)
Prints the content of P.

poly_product(*P*)
To be done ... !

8.25 Module util

util_save_matrix(*FMT*, *FILENAME*, *A*)
Exports a sparse matrix into the file named FILENAME, using Harwell-Boeing (FMT='hb') or Matrix-Market (FMT='mm') formatting.

util_load_matrix(*FMT*, *FILENAME*)
Imports a sparse matrix from a file.

util_trace_level(*level=None*)
Set the verbosity of some GetFEM routines.

Typically the messages printed by the model bricks, 0 means no trace message (default is 3). if no level is given, the current trace level is returned.

util_warning_level(*level=None*)
Filter the less important warnings displayed by getfem.

0 means no warnings, default level is 3. if no level is given, the current warning level is returned.

util_set_num_threads(*nb_threads*)
Sets the number of threads for the multithreaded GetFEM version. It is available only when GetFEM is compiled with openmp support.

util_mpi_parallelism_level()
Returns the level of MPI parallelism GetFEM is compiled with.

0 means no MPI parallelism, 1 means assembly is parallelized, and 2 means that both assembly and solve (with MUMPS) are MPI parallel.

util_mumps_linked()
Returns 1 if mumps is linked in GetFEM and 0 otherwise.

INDEX

A

adapt() (*MeshFem method*), 52
adapt() (*MeshIm method*), 58
adapt() (*MeshLevelSet method*), 60
add() (*MeshLevelSet method*), 60
add() (*Spmat method*), 100
add_assembly_assignment() (*Model method*), 66
add_basic_contact_brick() (*Model method*), 66
add_basic_contact_brick_two_deformable_bodies() (*Model method*), 67
add_bilaplacian_brick() (*Model method*), 67
add_constraint_with_multipliers() (*Model method*), 67
add_constraint_with_penalization() (*Model method*), 68
add_contact_boundary_to_unbiased_Nitsche_large_sliding_contact_brick() (*Model method*), 68
add_contact_with_rigid_obstacle_brick() (*Model method*), 68
add_convex() (*Mesh method*), 44
add_data() (*Model method*), 68
add_Dirichlet_condition_with_multipliers() (*Model method*), 62
add_Dirichlet_condition_with_Nitsche_method() (*Model method*), 62
add_Dirichlet_condition_with_penalization() (*Model method*), 63
add_Dirichlet_condition_with_simplification() (*Model method*), 63
add_elastoplasticity_brick() (*Model method*), 68
add_element_extrapolation_transformation() (*Model method*), 68
add_elementary_P0_projection() (*Model method*), 69
add_elementary_rotated_RT0_projection() (*Model method*), 69
add_enriched_Mindlin_Reissner_plate_brick() (*Model method*), 69
add_explicit_matrix() (*Model method*), 69
add_explicit_rhs() (*Model method*), 70
add_fem_data() (*Model method*), 70
add_fem_variable() (*Model method*), 70
add_filtered_fem_variable() (*Model method*), 70
add_finite_strain_elasticity_brick() (*Model method*), 70
add_finite_strain_elastoplasticity_brick() (*Model method*), 70
add_finite_strain_incompressibility_brick() (*Model method*), 71
add_Fourier_Robin_brick() (*Model method*), 63
add_generalized_Dirichlet_condition_with_multiplier (*Model method*), 71
add_generalized_Dirichlet_condition_with_Nitsche_me (*Model method*), 71
add_generalized_Dirichlet_condition_with_penalizati (*Model method*), 72
add_generic_elliptic_brick() (*Model method*), 72
add_Helmholtz_brick() (*Model method*), 64
add_HHO_reconstructed_gradient() (*Model method*), 63
add_HHO_reconstructed_symmetrized_gradient() (*Model method*), 63
add_HHO_reconstructed_symmetrized_value() (*Model method*), 63
add_HHO_reconstructed_value() (*Model method*), 63
add_HHO_stabilization() (*Model method*), 63
add_HHO_symmetrized_stabilization() (*Model method*), 63
add_Houbolt_scheme() (*Model method*), 64
add_im_data() (*Model method*), 72
add_im_variable() (*Model method*), 72
add_initialized_data() (*Model method*), 73
add_initialized_fem_data() (*Model method*), 73
add_integral_contact_between_nonmatching_meshes_bri

(*Model method*), 73
add_integral_contact_with_rigid_obstacle_brick() (*Model method*), 73
add_integral_large_sliding_contact_brick_raytracing() (*Model method*), 74
add_internal_im_variable() (*Model method*), 74
add_interpolate_transformation_from_expression() (*Model method*), 74
add_isotropic_linearized_elasticity_brick() (*Model method*), 74
add_isotropic_linearized_elasticity_pstrain_brick() (*Model method*), 74
add_isotropic_linearized_elasticity_pstress_brick() (*Model method*), 75
add_Kirchhoff_Love_Neumann_term_brick() (*Model method*), 64
add_Kirchhoff_Love_plate_brick() (*Model method*), 64
add_Laplacian_brick() (*Model method*), 64
add_linear_generic_assembly_brick() (*Model method*), 75
add_linear_incompressibility_brick() (*Model method*), 75
add_linear_term() (*Model method*), 75
add_linear_twodomain_term() (*Model method*), 75
add_lumped_mass_for_first_order_brick() (*Model method*), 76
add_macro() (*Model method*), 76
add_mass_brick() (*Model method*), 76
add_master_contact_boundary_to_biased_Nitsche_large_sliding_contact_brick() (*Model method*), 76
add_master_contact_boundary_to_large_sliding_contact_brick() (*Model method*), 76
add_master_contact_boundary_to_projection_transformation() (*Model method*), 76
add_master_contact_boundary_to_raytracing_transformation() (*Model method*), 76
add_master_slave_contact_boundary_to_large_sliding_contact_brick() (*Model method*), 76
add_Mindlin_Reissner_plate_brick() (*Model method*), 64
add_multiplier() (*Model method*), 77
add_Newmark_scheme() (*Model method*), 64
add_Nitsche_contact_with_rigid_obstacle_brick() (*Model method*), 64
add_Nitsche_fictitious_domain_contact_brick() (*Model method*), 65
add_Nitsche_large_sliding_contact_brick_raytracing() (*Model method*), 65
add_Nitsche_midpoint_contact_with_rigid_obstacle_brick() (*Model method*), 66
add_nodal_contact_between_nonmatching_meshes_brick() (*Model method*), 77
add_nodal_contact_with_rigid_obstacle_brick() (*Model method*), 77
add_nonlinear_elasticity_brick() (*Model method*), 78
add_nonlinear_generic_assembly_brick() (*Model method*), 78
add_nonlinear_incompressibility_brick() (*Model method*), 78
add_nonlinear_term() (*Model method*), 78
add_nonlinear_twodomain_term() (*Model method*), 79
add_nonmatching_meshes_contact_brick() (*Model method*), 79
add_normal_derivative_Dirichlet_condition_with_multipliers() (*Model method*), 80
add_normal_derivative_Dirichlet_condition_with_penalization() (*Model method*), 80
add_normal_derivative_source_term_brick() (*Model method*), 81
add_normal_Dirichlet_condition_with_multipliers() (*Model method*), 79
add_normal_Dirichlet_condition_with_Nitsche_method() (*Model method*), 79
add_normal_Dirichlet_condition_with_penalization() (*Model method*), 80
add_normal_source_term_brick() (*Model method*), 81
add_penalized_contact_between_nonmatching_meshes_brick() (*Model method*), 81
add_penalized_contact_with_rigid_obstacle_brick() (*Model method*), 81
add_proj() (*World method*), 44
add_pointwise_constraints_with_given_multipliers() (*Model method*), 82
add_pointwise_constraints_with_multipliers() (*Model method*), 82
add_pointwise_constraints_with_penalization() (*Model method*), 82
add_projection_transformation() (*Model method*), 82
add_raytracing_transformation() (*Model method*), 83
add_rigid_obstacle_to_large_sliding_contact_brick() (*Model method*), 83
add_rigid_obstacle_to_Nitsche_large_sliding_contact_brick() (*Model method*), 83
add_rigid_obstacle_to_projection_transformation()

(*Model method*), 83
add_rigid_obstacle_to_raytracing_transformation() (*Model method*), 83
add_slave_contact_boundary_to_biased_Nitsche_integral_contact() (*Model method*), 83
add_slave_contact_boundary_to_large_sliding_contact() (*Model method*), 83
add_slave_contact_boundary_to_projection_simplified() (*Model method*), 83
add_slave_contact_boundary_to_raytracing_transformation() (*Model method*), 83
add_small_strain_elastoplasticity_brick() (*Model method*), 83
add_source_term() (*Model method*), 85
add_source_term_brick() (*Model method*), 85
add_source_term_generic_assembly_brick() (*Model method*), 85
add_standard_secondary_domain() (*Model method*), 85
add_theta_method_for_first_order() (*Model method*), 85
add_theta_method_for_second_order() (*Model method*), 85
add_twodomain_source_term() (*Model method*), 85
add_variable() (*Model method*), 85
adjacent_face() (*Mesh method*), 44
all_faces() (*Mesh method*), 44
analyze() (*MumpsContext method*), 93
analyze_and_factorize() (*MumpsContext method*), 93
analyze_factorize_and_solve() (*MumpsContext method*), 93
area() (*Slice method*), 97
asm_bilaplacian() (*in module getfem*), 104
asm_bilaplacian_KL() (*in module getfem*), 104
asm_boundary() (*in module getfem*), 106
asm_boundary_qu_term() (*in module getfem*), 105
asm_boundary_source() (*in module getfem*), 104
asm_define_function() (*in module getfem*), 105
asm_define_linear_hardening_function() (*in module getfem*), 105
asm_define_Ramberg_Osgood_hardening_function() (*in module getfem*), 105
asm_dirichlet() (*in module getfem*), 104
asm_expression_analysis() (*in module getfem*), 105
asm_extrapolation_matrix() (*in module getfem*), 106
asm_generic() (*in module getfem*), 102
asm_helmholtz() (*in module getfem*), 104
asm_integral_contact_Uzawa_projection() (*in module getfem*), 106
asm_interpolation_matrix() (*in module getfem*), 106
asm_laplacian() (*in module getfem*), 103
asm_level_set_normal_source_term() (*in module getfem*), 107
asm_linear_elasticity() (*in module getfem*), 103
asm_lsneuman_matrix() (*in module getfem*), 107
asm_mass_matrix() (*in module getfem*), 103
asm_nlsgrad_matrix() (*in module getfem*), 107
asm_nonlinear_elasticity() (*in module getfem*), 103
asm_stabilization_patch_matrix() (*in module getfem*), 107
asm_undefine_function() (*in module getfem*), 105
asm_volumic() (*in module getfem*), 106
asm_volumic_source() (*in module getfem*), 104
assembly, 5
assembly() (*Model method*), 86
assign() (*Spmat method*), 100

B

base_value() (*Fem method*), 36
basic_dof_from_cv() (*MeshFem method*), 52
basic_dof_from_cvid() (*MeshFem method*), 52
basic_dof_nodes() (*MeshFem method*), 52
basic_dof_on_region() (*MeshFem method*), 52
basic_structure() (*CvStruct method*), 33
bifurcation_test_function() (*ContStruct method*), 32
boundaries() (*Mesh method*), 44
boundary, 12
boundary() (*Mesh method*), 44
brick_list() (*Model method*), 86
brick_term_rhs() (*Model method*), 86

C

change_penalization_coeff() (*Model method*), 86
char() (*ContStruct method*), 32
char() (*CvStruct method*), 33
char() (*Fem method*), 36
char() (*GeoTrans method*), 38

char() (*GlobalFunction method*), 39
char() (*Integ method*), 40
char() (*LevelSet method*), 41
char() (*Mesh method*), 44
char() (*MesherObject method*), 61
char() (*MeshFem method*), 52
char() (*MeshIm method*), 58
char() (*MeshLevelSet method*), 60
char() (*Model method*), 86
char() (*Precond method*), 95
char() (*Slice method*), 97
char() (*Spmat method*), 100
clear() (*Model method*), 86
clear() (*Spmat method*), 100
clear_assembly_assignment() (*Model method*), 86
CNTL() (*MumpsContext method*), 92
coeffs() (*Integ method*), 40
compute_convect() (*in module getfem*), 109
compute_elastoplasticity_Von_Mises_or_Tresca() (*Model method*), 86
compute_error_estimate() (*in module getfem*), 109
compute_error_estimate_nitsche() (*in module getfem*), 109
compute_eval_on_triangulated_surface() (*in module getfem*), 108
compute_extrapolate_on() (*in module getfem*), 108
compute_finite_strain_elasticity_Von_Mises() (*Model method*), 86
compute_finite_strain_elastoplasticity_Von_Mises() (*Model method*), 86
compute_gradient() (*in module getfem*), 108
compute_H1_norm() (*in module getfem*), 107
compute_H1_semi_dist() (*in module getfem*), 107
compute_H1_semi_norm() (*in module getfem*), 107
compute_H2_norm() (*in module getfem*), 108
compute_H2_semi_norm() (*in module getfem*), 108
compute_hessian() (*in module getfem*), 108
compute_interpolate_on() (*in module getfem*), 108
compute_isotropic_linearized_Von_Mises_or_Tresca() (*Model method*), 87
compute_isotropic_linearized_Von_Mises_pstrain() (*Model method*), 87
compute_isotropic_linearized_Von_Mises_pstress() (*Model method*), 87
compute_L2_dist() (*in module getfem*), 107
compute_L2_norm() (*in module getfem*), 107
compute_plastic_part() (*Model method*), 87
compute_second_Piola_Kirchhoff_tensor() (*Model method*), 87
compute_tangent() (*ContStruct method*), 32
compute_Von_Mises_or_Tresca() (*Model method*), 86
conjugate() (*Spmat method*), 100
contact_brick_set_BN() (*Model method*), 87
contact_brick_set_BT() (*Model method*), 87
ContStruct (*class in getfem*), 31
convex id, 6
convex_area() (*Mesh method*), 44
convex_index() (*MeshFem method*), 52
convex_index() (*MeshIm method*), 58
convex_radius() (*Mesh method*), 44
convexes, 5
convexes_in_box() (*Mesh method*), 44
crack_tip_convexes() (*MeshLevelSet method*), 60
csc_ind() (*Spmat method*), 100
csc_val() (*Spmat method*), 101
curved_edges() (*Mesh method*), 45
cut_mesh() (*MeshLevelSet method*), 60
cvid, 6
cvid() (*Mesh method*), 45
cvid_from_pid() (*Mesh method*), 45
cvs() (*Slice method*), 97
CvStruct (*built-in class*), 8
CvStruct (*class in getfem*), 33
cvs_faces() (*Mesh method*), 45

D

define_variable_group() (*Model method*), 87
degree() (*LevelSet method*), 41
degrees of freedom, 5
del_convex() (*Mesh method*), 45
del_convex_of_dim() (*Mesh method*), 45
del_macro() (*Model method*), 87
del_point() (*Mesh method*), 45
delete() (*in module getfem*), 109
delete_boundary() (*Mesh method*), 45
delete_brick() (*Model method*), 87
delete_region() (*Mesh method*), 45
delete_variable() (*Model method*), 87
determinant() (*Spmat method*), 101
diag() (*Spmat method*), 101
dim() (*CvStruct method*), 34
dim() (*Fem method*), 36
dim() (*GeoTrans method*), 38

dim() (*Integ method*), 40
dim() (*Mesh method*), 45
dim() (*Slice method*), 97
dirichlet_nullspace() (*Spmat method*), 101
disable_bricks() (*Model method*), 88
disable_variable() (*Model method*), 88
displacement_group_name_of_large_sliding_contact_brick() (*Model method*), 88
displacement_group_name_of_Nitsche_large_sliding_contact_brick() (*Model method*), 88
display() (*ContStruct method*), 32
display() (*CvStruct method*), 34
display() (*Fem method*), 36
display() (*GeoTrans method*), 38
display() (*GlobalFunction method*), 39
display() (*Integ method*), 40
display() (*LevelSet method*), 41
display() (*Mesh method*), 46
display() (*MesherObject method*), 61
display() (*MeshFem method*), 52
display() (*MeshIm method*), 58
display() (*MeshImData method*), 59
display() (*MeshLevelSet method*), 60
display() (*Model method*), 88
display() (*MumpsContext method*), 93
display() (*Precond method*), 95
display() (*Slice method*), 97
display() (*Spmat method*), 101
distributed_matrix() (*MumpsContext method*), 93
dof, 5
dof_from_cv() (*MeshFem method*), 52
dof_from_cvid() (*MeshFem method*), 53
dof_from_im() (*MeshFem method*), 53
dof_nodes() (*MeshFem method*), 53
dof_on_region() (*MeshFem method*), 53
dof_partition() (*MeshFem method*), 53

E

edges() (*Mesh method*), 46
edges() (*Slice method*), 97
elastoplasticity_next_iter() (*Model method*), 88
Eltm (*class in getfem*), 34
eltm() (*MeshIm method*), 58
enable_bricks() (*Model method*), 88
enable_variable() (*Model method*), 88
environment variable
    assembly, 5
    boundary, 12
    convex id, 6
    convexes, 5
    cvid, 6
    degrees of freedom, 5
    dof, 5
    FEM, 5
    geometric transformation, 5
    geometrical nodes, 5
    integration method, 5
    interpolate, 5
    Lagrangian, 5
    Laplacian, 11
    mesh, 5
    mesh nodes, 5
    mesh_fem, 5
    mesh_im, 5
    model, 12
    pid, 6
    point id, 6
    quadrature formula, 12
    quadrature formulas, 5
    reference convex, 5
    Von Mises, 20
error_check() (*MumpsContext method*), 93
estimated_degree() (*Fem method*), 36
eval() (*MeshFem method*), 53
export_to_dx() (*Mesh method*), 46
export_to_dx() (*MeshFem method*), 53
export_to_dx() (*Slice method*), 97
export_to_pos() (*Mesh method*), 46
export_to_pos() (*MeshFem method*), 54
export_to_pos() (*Slice method*), 97
export_to_pov() (*Slice method*), 98
export_to_vtk() (*Mesh method*), 46
export_to_vtk() (*MeshFem method*), 54
export_to_vtk() (*Slice method*), 98
export_to_vtu() (*Mesh method*), 46
export_to_vtu() (*MeshFem method*), 54
export_to_vtu() (*Slice method*), 98
extend_region() (*Mesh method*), 46
extend_vector() (*MeshFem method*), 54
extension_matrix() (*MeshFem method*), 54

F

face() (*CvStruct method*), 34
face_coeffs() (*Integ method*), 40
face_pts() (*Integ method*), 40
facepts() (*CvStruct method*), 34
faces_from_cvid() (*Mesh method*), 46
faces_from_pid() (*Mesh method*), 46
factorize() (*MumpsContext method*), 93

factorize_and_solve() (*MumpsContext method*), 93
FEM, 5
Fem (*built-in class*), 8
Fem (*class in getfem*), 35
fem() (*MeshFem method*), 54
finite_strain_elastoplasticity_next_iter() (*Model method*), 88
first_iter() (*Model method*), 88
from_variables() (*Model method*), 88
full() (*Spmat method*), 101

G

geometric transformation, 5
geometrical nodes, 5
GeoTrans (*built-in class*), 8
GeoTrans (*class in getfem*), 37
geotrans() (*Mesh method*), 47
get_time() (*Model method*), 89
get_time_step() (*Model method*), 89
GlobalFunction (*class in getfem*), 38
grad() (*GlobalFunction method*), 39
grad_base_value() (*Fem method*), 36

H

has_linked_mesh_levelset() (*MeshFem method*), 54
hess() (*GlobalFunction method*), 39
hess_base_value() (*Fem method*), 37

I

ICNTL() (*MumpsContext method*), 93
im_nodes() (*MeshIm method*), 58
index_of_global_dof() (*Fem method*), 37
INFOG() (*MumpsContext method*), 93
init_Moore_Penrose_continuation() (*ContStruct method*), 33
init_step_size() (*ContStruct method*), 33
inner_faces() (*Mesh method*), 47
Integ (*built-in class*), 10
Integ (*class in getfem*), 39
integ() (*MeshIm method*), 58
integration method, 5
interpolate, 5
interpolate_convex_data() (*MeshFem method*), 54
interpolate_convex_data() (*Slice method*), 98
interpolation() (*Model method*), 89
interval_of_variable() (*Model method*), 89
is_complex() (*Model method*), 89
is_complex() (*Precond method*), 95
is_complex() (*Spmat method*), 101
is_equivalent() (*Fem method*), 37
is_equivalent() (*MeshFem method*), 55
is_exact() (*Integ method*), 40
is_lagrange() (*Fem method*), 37
is_lagrangian() (*MeshFem method*), 55
is_linear() (*GeoTrans method*), 38
is_polynomial() (*Fem method*), 37
is_polynomial() (*MeshFem method*), 55
is_reduced() (*MeshFem method*), 55

L

Lagrangian, 5
Laplacian, 11
LevelSet (*class in getfem*), 41
levelsets() (*MeshLevelSet method*), 60
linked_mesh() (*MeshFem method*), 55
linked_mesh() (*MeshIm method*), 58
linked_mesh() (*MeshImData method*), 59
linked_mesh() (*MeshLevelSet method*), 60
linked_mesh() (*Slice method*), 99
linked_mesh_levelset() (*MeshFem method*), 55
linsolve_bicgstab() (*in module getfem*), 110
linsolve_cg() (*in module getfem*), 110
linsolve_gmres() (*in module getfem*), 110
linsolve_lu() (*in module getfem*), 110
linsolve_mumps() (*in module getfem*), 110
linsolve_superlu() (*in module getfem*), 110
list_residuals() (*Model method*), 89
local_projection() (*Model method*), 89

M

matrix() (*MumpsContext method*), 93
matrix_term() (*Model method*), 89
max_cvid() (*Mesh method*), 47
max_pid() (*Mesh method*), 47
max_step_size() (*ContStruct method*), 33
memsize() (*LevelSet method*), 41
memsize() (*Mesh method*), 47
memsize() (*MeshFem method*), 55
memsize() (*MeshIm method*), 58
memsize() (*MeshLevelSet method*), 60
memsize() (*Model method*), 89
memsize() (*Slice method*), 99
merge() (*Mesh method*), 47
mesh, 5
Mesh (*built-in class*), 8
Mesh (*class in getfem*), 42
mesh nodes, 5
mesh() (*MeshFem method*), 55
mesh() (*Slice method*), 99

mesh_fem, 5
mesh_fem_of_variable() (*Model method*), 89
mesh_im, 5
MesherObject (*class in getfem*), 61
MeshFem (*built-in class*), 10
MeshFem (*class in getfem*), 51
MeshIm (*built-in class*), 10
MeshIm (*class in getfem*), 57
MeshImData (*class in getfem*), 59
MeshLevelSet (*class in getfem*), 60
mf() (*LevelSet method*), 41
min_step_size() (*ContStruct method*), 33
model, 12
Model (*built-in class*), 10
Model (*class in getfem*), 62
Moore_Penrose_continuation() (*ContStruct method*), 32
mult() (*Precond method*), 95
mult() (*Spmat method*), 101
mult_varname_Dirichlet() (*Model method*), 89
MumpsContext (*class in getfem*), 92

N

nb_basic_dof() (*MeshFem method*), 55
nb_ls() (*MeshLevelSet method*), 60
nb_tensor_elements() (*MeshImData method*), 59
nbcvs() (*Mesh method*), 47
nbdof() (*Fem method*), 37
nbdof() (*MeshFem method*), 55
nbdof() (*Model method*), 89
nbpts() (*CvStruct method*), 34
nbpts() (*GeoTrans method*), 38
nbpts() (*Integ method*), 41
nbpts() (*Mesh method*), 47
nbpts() (*MeshImData method*), 59
nbpts() (*Slice method*), 99
nbsplxs() (*Slice method*), 99
Neumann_term() (*Model method*), 62
next_iter() (*Model method*), 89
nnz() (*Spmat method*), 101
non_conformal_basic_dof() (*MeshFem method*), 55
non_conformal_dof() (*MeshFem method*), 55
non_smooth_bifurcation_test() (*ContStruct method*), 33
non_smooth_branching() (*ContStruct method*), 33
normal_of_face() (*Mesh method*), 47
normal_of_faces() (*Mesh method*), 47
normals() (*GeoTrans method*), 38

O

optimize_structure() (*Mesh method*), 47
orphaned_pid() (*Mesh method*), 47
outer_faces() (*Mesh method*), 47
outer_faces_in_ball() (*Mesh method*), 48
outer_faces_in_box() (*Mesh method*), 48
outer_faces_with_direction() (*Mesh method*), 48

P

perform_init_time_derivative() (*Model method*), 89
pid, 6
pid() (*Mesh method*), 48
pid_from_coords() (*Mesh method*), 48
pid_from_cvid() (*Mesh method*), 48
pid_in_cvids() (*Mesh method*), 49
pid_in_faces() (*Mesh method*), 49
pid_in_regions() (*Mesh method*), 49
point id, 6
poly_print() (*in module getfem*), 110
poly_product() (*in module getfem*), 110
poly_str() (*Fem method*), 37
Precond (*class in getfem*), 94
pts() (*Fem method*), 37
pts() (*GeoTrans method*), 38
pts() (*Integ method*), 41
pts() (*Mesh method*), 49
pts() (*Slice method*), 99
pts_from_cvid() (*Mesh method*), 49

Q

qdim() (*MeshFem method*), 55
quadrature formula, 12
quadrature formulas, 5
quality() (*Mesh method*), 49

R

reduce_meshfem() (*MeshFem method*), 55
reduce_vector() (*MeshFem method*), 56
reduction() (*MeshFem method*), 56
reduction_matrices() (*MeshFem method*), 56
reduction_matrix() (*MeshFem method*), 56
reference convex, 5
refine() (*Mesh method*), 49
region() (*Mesh method*), 49
region() (*MeshImData method*), 59
region_intersect() (*Mesh method*), 49
region_merge() (*Mesh method*), 50
region_subtract() (*Mesh method*), 50

regions() (*Mesh method*), 50
resize_variable() (*Model method*), 90
rhs() (*Model method*), 90
RINFO() (*MumpsContext method*), 93
RINFOG() (*MumpsContext method*), 93

S

save() (*Mesh method*), 50
save() (*MeshFem method*), 56
save() (*MeshIm method*), 59
save() (*Spmat method*), 101
scale() (*Spmat method*), 101
set_boundary() (*Mesh method*), 50
set_classical_discontinuous_fem() (*MeshFem method*), 56
set_classical_fem() (*MeshFem method*), 56
set_diag() (*Spmat method*), 101
set_distributed_matrix() (*MumpsContext method*), 94
set_dof_partition() (*MeshFem method*), 56
set_element_extrapolation_correspondence() (*Model method*), 90
set_enriched_dofs() (*MeshFem method*), 56
set_fem() (*MeshFem method*), 56
set_ICNTL() (*MumpsContext method*), 93
set_integ() (*MeshIm method*), 59
set_matrix() (*MumpsContext method*), 94
set_partial() (*MeshFem method*), 56
set_private_matrix() (*Model method*), 90
set_private_rhs() (*Model method*), 90
set_pts() (*Mesh method*), 50
set_pts() (*Slice method*), 99
set_qdim() (*MeshFem method*), 57
set_region() (*Mesh method*), 50
set_region() (*MeshImData method*), 59
set_tensor_size() (*MeshImData method*), 59
set_time() (*Model method*), 90
set_time_step() (*Model method*), 90
set_values() (*LevelSet method*), 42
set_variable() (*Model method*), 90
set_vector() (*MumpsContext method*), 94
shift_variables_for_time_integration() (*Model method*), 90
simplify() (*LevelSet method*), 42
sing_data() (*ContStruct method*), 33
size() (*Precond method*), 95
size() (*Spmat method*), 102
Slice (*class in getfem*), 95
sliding_data_group_name_of_large_sliding_contact_brick() (*Model method*), 90
sliding_data_group_name_of_Nitsche_large_sliding_co (*Model method*), 90
small_strain_elastoplasticity_next_iter() (*Model method*), 90
small_strain_elastoplasticity_Von_Mises() (*Model method*), 90
solve() (*Model method*), 91
solve() (*MumpsContext method*), 94
splxs() (*Slice method*), 99
Spmat (*class in getfem*), 99
step_size_decrement() (*ContStruct method*), 33
step_size_increment() (*ContStruct method*), 33
storage() (*Spmat method*), 102
sup() (*MeshLevelSet method*), 60

T

tangent_matrix() (*Model method*), 91
target_dim() (*Fem method*), 37
tensor_size() (*MeshImData method*), 59
test_tangent_matrix() (*Model method*), 91
test_tangent_matrix_term() (*Model method*), 92
tmult() (*Precond method*), 95
tmult() (*Spmat method*), 102
to_complex() (*Spmat method*), 102
to_csc() (*Spmat method*), 102
to_variables() (*Model method*), 92
to_wsc() (*Spmat method*), 102
transconj() (*Spmat method*), 102
transform() (*GeoTrans method*), 38
transform() (*Mesh method*), 50
transformation_name_of_large_sliding_contact_brick((*Model method*), 92
transformation_name_of_Nitsche_large_sliding_contac (*Model method*), 92
translate() (*Mesh method*), 50
transpose() (*Spmat method*), 102
triangulated_surface() (*Mesh method*), 50
type() (*Precond method*), 95

U

util_load_matrix() (*in module getfem*), 111
util_mpi_parallelism_level() (*in module getfem*), 111
util_mumps_linked() (*in module getfem*), 111
util_save_matrix() (*in module getfem*), 111
util_set_num_threads() (*in module getfem*), 111
util_trace_level() (*in module getfem*), 111
util_warning_level() (*in module getfem*), 111

V

val() (*GlobalFunction method*), 39
values() (*LevelSet method*), 42
variable() (*Model method*), 92
variable_list() (*Model method*), 92
vector() (*MumpsContext method*), 94
Von Mises, 20