Matlab Interface

Release 5.2

Yves Renard, Julien Pommier

May 09, 2018

CONTENTS

1 Introduction **1**

2 Installation **3**

3 Preliminary **5**

4 *GetFEM++* organization **7**
4.1 Functions . . . 7
4.2 Objects . . . 8

5 Examples **11**
5.1 A step-by-step basic example . . . 11
5.2 Another Laplacian with exact solution . . . 14
5.3 Linear and non-linear elasticity . . . 15
5.4 Avoiding the bricks framework . . . 17
5.5 Other examples . . . 18
5.6 Using Matlab Object-Oriented features . . . 19

6 Draw Command reference **21**
6.1 gf_colormap . . . 21
6.2 gf_plot . . . 21
6.3 gf_plot_1D . . . 22
6.4 gf_plot_mesh . . . 22
6.5 gf_plot_slice . . . 23

7 Command reference **25**
7.1 gf_asm . . . 26
7.2 gf_compute . . . 31
7.3 gf_cont_struct . . . 34
7.4 gf_cont_struct_get . . . 36
7.5 gf_cvstruct_get . . . 37
7.6 gf_delete . . . 38
7.7 gf_eltm . . . 38
7.8 gf_fem . . . 39
7.9 gf_fem_get . . . 41
7.10 gf_geotrans . . . 42
7.11 gf_geotrans_get . . . 43
7.12 gf_global_function . . . 44
7.13 gf_global_function_get . . . 45
7.14 gf_integ . . . 45
7.15 gf_integ_get . . . 46

7.16 gf_levelset . . . 47
7.17 gf_levelset_get . . . 48
7.18 gf_levelset_set . . . 49
7.19 gf_linsolve . . . 49
7.20 gf_mesh . . . 50
7.21 gf_mesh_get . . . 52
7.22 gf_mesh_set . . . 58
7.23 gf_mesh_fem . . . 60
7.24 gf_mesh_fem_get . . . 61
7.25 gf_mesh_fem_set . . . 66
7.26 gf_mesh_im . . . 68
7.27 gf_mesh_im_get . . . 69
7.28 gf_mesh_im_set . . . 70
7.29 gf_mesh_im_data . . . 71
7.30 gf_mesh_im_data_get . . . 71
7.31 gf_mesh_im_data_set . . . 72
7.32 gf_mesh_levelset . . . 72
7.33 gf_mesh_levelset_get . . . 73
7.34 gf_mesh_levelset_set . . . 74
7.35 gf_mesher_object . . . 74
7.36 gf_mesher_object_get . . . 75
7.37 gf_model . . . 76
7.38 gf_model_get . . . 76
7.39 gf_model_set . . . 82
7.40 gf_poly . . . 106
7.41 gf_precond . . . 107
7.42 gf_precond_get . . . 108
7.43 gf_slice . . . 109
7.44 gf_slice_get . . . 110
7.45 gf_slice_set . . . 113
7.46 gf_spmat . . . 113
7.47 gf_spmat_get . . . 114
7.48 gf_spmat_set . . . 116
7.49 gf_util . . . 117
7.50 gf_workspace . . . 118

8 *GetFEM++* OO-commands **121**

Index **123**

CHAPTER

ONE

INTRODUCTION

This guide provides a reference about the *MatLab* 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.

CHAPTER

TWO

INSTALLATION

The installation of the *getfem-interface* toolbox can be somewhat tricky, since it combines a C++ compiler, libraries and *MatLab* interaction... In case of troubles with a non-GNU compiler, gcc/g++ (>= 4.8) should be a safe solution.

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 a 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:

```
>> gf_mesh_set(m,'optimize structure')
```

CHAPTER

FOUR

GETFEM++ ORGANIZATION

The *GetFEM++* toolbox is just a convenient interface to the *GetFEM++* library: you must have a working *GetFEM++* installed on your computer. This toolbox provides a big `mex-file` (c++ binary callable from *MatLab*) and some additional `m-files` (documentation and extra-functionalities). All the functions of *GetFEM++* are prefixed by `gf_` (hence typing `gf_` at the *MatLab* prompt and then pressing the `<tab>` key is a quick way to obtain the list of getfem functions).

Functions

- `gf_workspace` : workspace management.
- `gf_util` : miscellanous utility functions.
- `gf_delete` : destroy a *GetFEM++* object (gfMesh , gfMeshFem , gfMeshIm etc.).
- `gf_cvstruct_get` : retrieve informations from a gfCvStruct object.
- `gf_geotrans` : define a geometric transformation.
- `gf_geotrans_get` : retrieve informations from a gfGeoTrans object.
- `gf_mesh` : creates a new gfMesh object.
- `gf_mesh_get` : retrieve informations from a gfMesh object.
- `gf_mesh_set` : modify a gfMesh object.
- `gf_eltm` : define an elementary matrix.
- `gf_fem` : define a gfFem.
- `gf_fem_get` : retrieve informations from a gfFem object.
- `gf_integ` : define a integration method.
- `gf_integ_get` : retrieve informations from an gfInteg object.
- `gf_mesh_fem` : creates a new gfMeshFem object.
- `gf_mesh_fem_get` : retrieve informations from a gfMeshFem object.
- `gf_mesh_fem_set` : modify a gfMeshFem object.
- `gf_mesh_im` : creates a new gfMeshIm object.
- `gf_mesh_im_get` : retrieve informations from a gfMeshIm object.
- `gf_mesh_im_set` : modify a gfMeshIm object.
- `gf_slice` : create a new gfSlice object.

- `gf_slice_get` : retrieve informations from a gfSlice object.
- `gf_slice_set` : modify a gfSlice object.
- `gf_spmat` : create a gfSpMat object.
- `gf_spmat_get` : perform computations with the gfSpMat.
- `gf_spmat_set` : modify the gfSpMat.
- `gf_precond` : create a gfPrecond object.
- `gf_precond_get` : perform computations with the gfPrecond.
- `gf_linsolve` : interface to various linear solvers provided by getfem (*SuperLU*, conjugated gradient, etc.).
- `gf_asm` : assembly routines.
- `gf_solve` : various solvers for usual PDEs (obsoleted by the gfMdBrick objects).
- `gf_compute` : computations involving the solution of a PDE (norm, derivative, etc.).
- `gf_mdbrick` : create a ("model brick") gfMdBrick object.
- `gf_mdbrick_get` : retrieve information from a gfMdBrick object.
- `gf_mdbrick_set` : modify a gfMdBrick object.
- `gf_mdstate` : create a ("model state") gfMdState object.
- `gf_mdstate_get` : retrieve information from a gfMdState object.
- `gf_mdstate_set` : modify a gfMdState object.
- `gf_model` : create a gfModel object.
- `gf_model_get` : retrieve information from a gfModel object.
- `gf_model_set` : modify a gfModel object.
- `gf_global_function` : create a gfGlobalFunction object.
- `gf_model_get` : retrieve information from a gfGlobalFunction object.
- `gf_model_set` : modify a GlobalFunction object.
- `gf_plot_mesh` : plotting of mesh.
- `gf_plot` : plotting of 2D and 3D fields.
- `gf_plot_1D` : plotting of 1D fields.
- `gf_plot_slice` : plotting of a mesh slice.

Objects

Various "objects" can be manipulated by the *GetFEM++* toolbox, see fig. *GetFEM++ objects hierarchy.*. The MESH and MESHFEM objects are the two most important objects.

- `gfGeoTrans`: geometric transformations (defines the shape/position of the convexes), created with `gf_geotrans`
- `gfGlobalFunction`: represent a global function for the enrichment of finite element methods.
- `gfMesh` : mesh structure (nodes, convexes, geometric transformations for each convex), created with `gf_mesh`

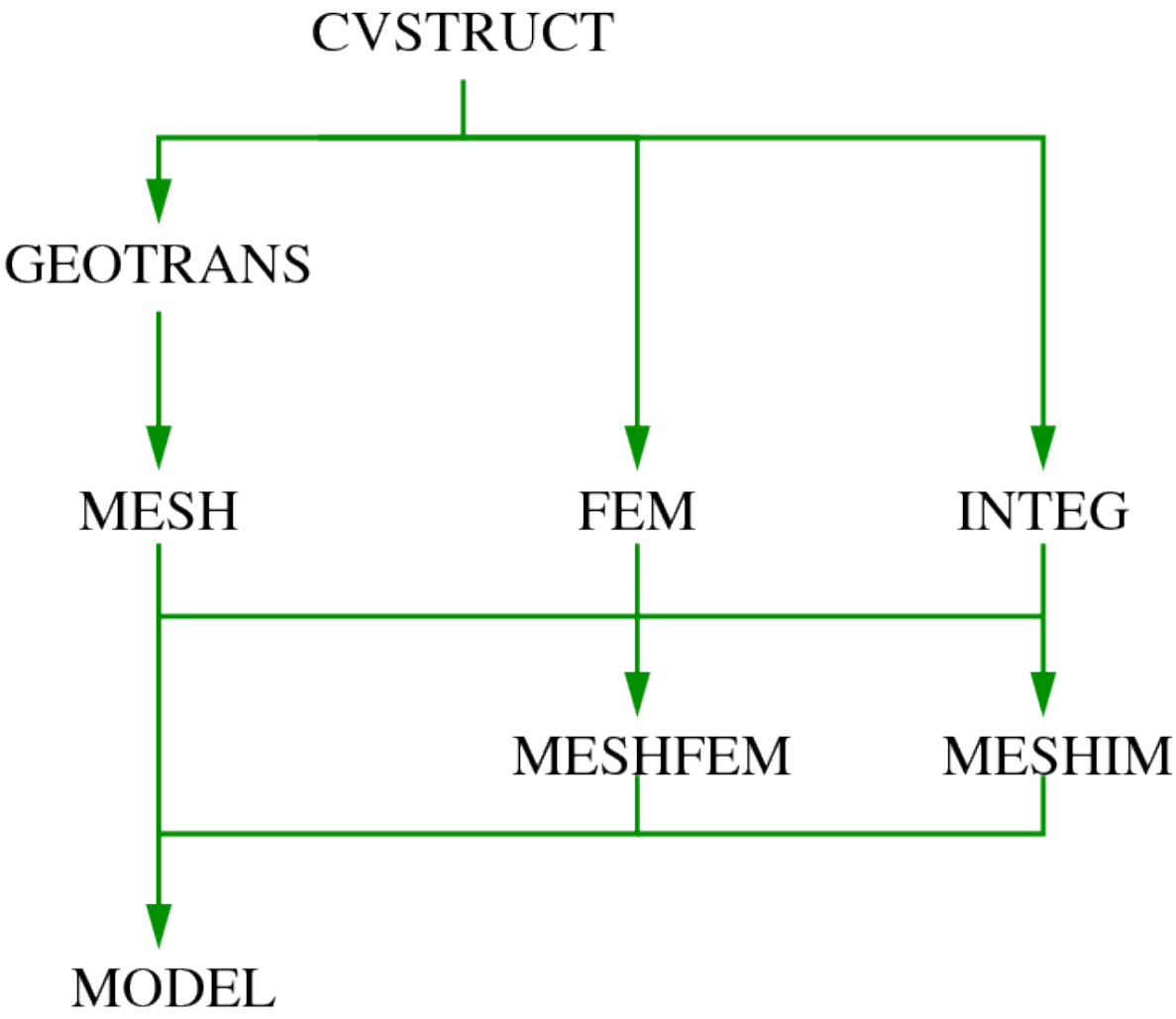

Figure 4.1: *GetFEM++* objects hierarchy.

- `gfInteg` : integration method (exact, quadrature formula...). Although not linked directly to GEOTRANS, an integration method is usually specific to a given convex structure. Created with `gf_integ`
- `gfFem` : the finite element method (one per convex, can be PK, QK, HERMITE, etc.). Created with `gf_fem`
- `gfCvStruct` : stores formal information convex structures (nb. of points, nb. of faces which are themselves convex structures).
- `gfMeshFem` : object linked to a mesh, where each convex has been assigned a FEM. Created with `gf_mesh_fem`.
- `gfMeshImM` : object linked to a mesh, where each convex has been assigned an integration method. Created with `gf_mesh_im`.
- `gfMeshSlice` : object linked to a mesh, very similar to a P1-discontinuous gfMeshFem. Used for fast interpolation and plotting.
- `gfMdBrick` : gfMdBrick , an abstraction of a part of solver (for example, the part which build the tangent matrix, the part which handles the dirichlet conditions, etc.). These objects are stacked to build a complete solver for a wide variety of problems. They typically use a number of gfMeshFem, gfMeshIm etc. Deprecated object, replaced now by gfModel.
- `gfMdState` : "model state", holds the global data for a stack of mdbricks (global tangent matrix, right hand side etc.). Deprecated object, replaced now by gfModel.
- `gfModel` : "model", holds the global data, variables and description of a model. Evolution of "model state" object for 4.0 version of *GetFEM++*.

The *GetFEM++* toolbox uses its own `memory management`. Hence *GetFEM++* objects are not cleared when a:

```
>> clear all
```

is issued at the *MatLab* prompt, but instead the function:

```
>> gf_workspace('clear all')
```

should be used. The various *GetFEM++* object can be accessed via *handles* (or *descriptors*), which are just *MatLab* structures containing 32-bits integer identifiers to the real objects. Hence the *MatLab* command:

```
>> whos
```

does not report the memory consumption of *GetFEM++* objects (except the marginal space used by the handle). Instead, you should use:

```
>> gf_workspace('stats')
```

There are two kinds of *GetFEM++* objects:

- static ones, which can not be deleted: ELTM, FEM, INTEG, GEOTRANS and CVSTRUCT. Hopefully their memory consumption is very low.
- dynamic ones, which can be destroyed, and are handled by the `gf_workspace` function: MESH, MESHFEM, MESHIM, SLICE, SPMAT, PRECOND.

The objects MESH and MESHFEM are not independent: a MESHFEM object is always linked to a MESH object, and a MESH object can be used by several MESHFEM objects. Hence when you request the destruction of a MESH object, its destruction might be delayed until it is not used anymore by any MESHFEM (these objects waiting for deletion are listed in the *anonymous workspace* section of `gf_workspace('stats')`).

CHAPTER

FIVE

EXAMPLES

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 **m-file** of this example under the name **demo_step_by_step.m** in the directory `interface/tests/matlab/` of the *GetFEM++* distribution.

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

```
>> % creation of a simple cartesian mesh
>> m = gf_mesh('cartesian',[0:.1:1],[0:.1:1]);
m =
     id: 0
    cid: 0
```

If you try to look at the value of `m`, you'll notice that it appears to be a structure containing two integers. The first one is its identifier, the second one is its class-id, i.e. an identifier of its type. This small structure is just an "handle" or "descriptor" to the real object, which is stored in the *GetFEM++* memory and cannot be represented via *MatLab* data structures. Anyway, you can still inspect the *GetFEM++* objects via the command `gf_workspace('stats')`.

Now we can try to have a **look at the mesh**, with its vertices numbering and the convexes numbering:

```
>> % we enable vertices and convexes labels
>> gf_plot_mesh(m, 'vertices', 'on', 'convexes', 'on');
```

As you can see, the mesh is regular, and the numbering of its nodes and convexes is also regular (this is guaranteed for cartesian meshes, but do not hope a similar numbering for the degrees of freedom).

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

```
>> % create a mesh_fem of for a field of dimension 1 (i.e. a scalar field)
>> mf = gf_mesh_fem(m,1);
>> gf_mesh_fem_set(mf,'fem',gf_fem('FEM_QK(2,2)'));
```

The first instruction builds a new gfMeshFem 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 $P^k \Rightarrow$ polynomials of degree k, while $Q^k \Rightarrow$ polynomials of degree $2k$). As Q^2 is a polynomial FEM, you can view the expression of its basis functions on the reference convex:

```
>> gf_fem_get(gf_fem('FEM_QK(2,2)'), 'poly_str');
ans =
    '1 - 3*x - 3*y + 2*x^2 + 9*x*y + 2*y^2 - 6*x^2*y - 6*x*y^2 + 4*x^2*y^2'
    '4*x - 4*x^2 - 12*x*y + 12*x^2*y + 8*x*y^2 - 8*x^2*y^2'
    '-x + 2*x^2 + 3*x*y - 6*x^2*y - 2*x*y^2 + 4*x^2*y^2'
    '4*y - 12*x*y - 4*y^2 + 8*x^2*y + 12*x*y^2 - 8*x^2*y^2'
    '16*x*y - 16*x^2*y - 16*x*y^2 + 16*x^2*y^2'
    '-4*x*y + 8*x^2*y + 4*x*y^2 - 8*x^2*y^2'
    '-y + 3*x*y + 2*y^2 - 2*x^2*y - 6*x*y^2 + 4*x^2*y^2'
    '-4*x*y + 4*x^2*y + 8*x*y^2 - 8*x^2*y^2'
    'x*y - 2*x^2*y - 2*x*y^2 + 4*x^2*y^2'
```

It is also possible to make use of the "object oriented" features of *MatLab*. As you may have noticed, when a class "foo" is provided by the *getfem-interface*, it is build with the function `gf_foo`, and manipulated with the functions `gf_foo_get` and `gf_foo_set`. But (with matlab 6.x and better) you may also create the object with the `gfFoo` constructor , and manipulated with the `get(..)` and `set(..)` methods. For example, the previous steps could have been:

```
>> gfFem('FEM_QK(2,2)');
gfFem object ID=0 dim=2, target_dim=1, nbdof=9,[EQUIV, POLY, LAGR], est.degree=4
  -> FEM_QK(2,2)
>> m=gfMesh('cartesian', [0:.1:1], [0:.1:1]);
gfMesh object ID=0 [16512 bytes], dim=2, nbpts=121, nbcvs=100
>> mf=gfMeshFem(m,1);
gfMeshFem object: ID=1 [804 bytes], qdim=1, nbdof=0,
  linked gfMesh object: dim=2, nbpts=121, nbcvs=100
>> set(mf, 'fem', gfFem('FEM_QK(2,2)'));
>> mf
gfMeshFem object: ID=1 [1316 bytes], qdim=1, nbdof=441,
  linked gfMesh object: dim=2, nbpts=121, nbcvs=100
```

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

```
>> % assign the same integration method on all convexes
>> mim = gf_mesh_im(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 = gf_mesh_get(m,'outer faces');
>> % mark it as boundary #42
>> gf_mesh_set(m, 'region', 42, border);
>> gf_plot_mesh(m, 'regions', [42]); % the boundary edges appears in red
```

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).

Remark:

we could have used `gf_mesh_get(m, 'OuTEr_faCes')`, as the interface is case-insensitive, and whitespaces can be replaced by underscores.

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 desribe the model and run the solver to get the solution! The "`model`" is created with the `gf_model` (or `gfModel`) 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 contraints, 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: $u = x(x-1)y(y-1) + x^5$, then we have $\Delta u = 2(x^2+y^2) - 2(x+y) + 20x^3$ (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`
>> gf_model_set(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"
>> gf_model_set(md, 'add Laplacian brick', mim, 'u');
```

Next we add a Dirichlet condition on the domain boundary:

```
>> % add Dirichlet condition
>> Uexact = gf_mesh_fem_get(mf, 'eval', {'(x-.5).^2 + (y-.5).^2 + x/5 - y/3'});
>> gf_model_set(md, 'add initialized fem data', 'DirichletData', mf, Uexact);
>> gf_model_set(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 gfMeshFem 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 gfMeshFem for the unknown and for the data such as `g`, but in the general case, `mf` won't be Lagrangian and another (Lagrangian) *mesh_fem* will be used for the description of Dirichlet conditions, source terms etc.

A source term can be added with the following lines:

```
>> % add source term
>> f = gf_mesh_fem_get(mf, 'eval', { '2(x^2+y^2)-2(x+y)+20x^3' });
```

```
>> gf_model_set(md, 'add initialized fem data', 'VolumicData', mf, f);
>> gf_model_set(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
>> gf_model_get(md, 'solve');
```

The model now contains the solution (as well as other things, such as the linear system which was solved). It is extracted, a display into a *MatLab* figure:

```
>> % extracted solution
>> u = gf_model_get(md, 'variable', 'u');
>> % display
>> gf_plot(mf, u, 'mesh','on');
```

Another Laplacian with exact solution

This is the `tests/matlab/demo_laplacian.m` example.

```
% trace on;
gf_workspace('clear all');
m = gf_mesh('cartesian',[0:.1:1],[0:.1:1]);
%m=gf_mesh('import','structured','GT="GT_QK(2,1)";SIZES=[1,1];NOISED=1;NSUBDIV=[1,1];')

% create a mesh_fem of for a field of dimension 1 (i.e. a scalar field)
mf = gf_mesh_fem(m,1);
% assign the Q2 fem to all convexes of the mesh_fem,
gf_mesh_fem_set(mf,'fem',gf_fem('FEM_QK(2,2)'));

% Integration which will be used
mim = gf_mesh_im(m, gf_integ('IM_GAUSS_PARALLELEPIPED(2,4)'));
%mim = gf_mesh_im(m, gf_integ('IM_STRUCTURED_COMPOSITE(IM_GAUSS_PARALLELEPIPED(2,5),4)'));
% detect the border of the mesh
border = gf_mesh_get(m,'outer faces');
% mark it as boundary #1
gf_mesh_set(m, 'boundary', 1, border);
gf_plot_mesh(m, 'regions', [1]); % the boundary edges appears in red
pause(1);

% interpolate the exact solution
Uexact = gf_mesh_fem_get(mf, 'eval', { 'y.*(y-1).*x.*(x-1)+x.^5' });
% its second derivative
F      = gf_mesh_fem_get(mf, 'eval', { '-(2*(x.^2+y.^2)-2*x-2*y+20*x.^3)' });


md=gf_model('real');
gf_model_set(md, 'add fem variable', 'u', mf);
gf_model_set(md, 'add Laplacian brick', mim, 'u');
gf_model_set(md, 'add initialized fem data', 'VolumicData', mf, F);
gf_model_set(md, 'add source term brick', mim, 'u', 'VolumicData');
gf_model_set(md, 'add initialized fem data', 'DirichletData', mf, Uexact);
gf_model_set(md, 'add Dirichlet condition with multipliers', mim, 'u', mf, 1, 'DirichletData');

gf_model_get(md, 'solve');
U = gf_model_get(md, 'variable', 'u');
```

```
% Version with old bricks
% b0=gf_mdbrick('generic elliptic',mim,mf);
% b1=gf_mdbrick('dirichlet', b0, 1, mf, 'penalized');
% gf_mdbrick_set(b1, 'param', 'R', mf, Uexact);
% b2=gf_mdbrick('source term',b1);
% gf_mdbrick_set(b2, 'param', 'source_term', mf, F);
% mds=gf_mdstate(b1);
% gf_mdbrick_get(b2, 'solve', mds)
% U=gf_mdstate_get(mds, 'state');

disp(sprintf('H1 norm of error: %g', gf_compute(mf,U-Uexact,'H1 norm',mim)));

subplot(2,1,1); gf_plot(mf,U,'mesh','on','contour',.01:.01:.1);
colorbar; title('computed solution');

subplot(2,1,2); gf_plot(mf,U-Uexact,'mesh','on');
colorbar;title('difference with exact solution');
```

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 `m-file` of this example under the name `demo_tripod.m` in the directory `tests/matlab` of the toolbox distribution.

```
disp('This demo is an adaption of the original tripod demo')
disp('which uses the new "brick" framework of getfem')
disp('The code is shorter, faster and much more powerful')
disp('You can easily switch between linear/non linear')
disp('compressible/incompressible elasticity!')

linear = 1
incompressible = 0


gf_workspace('clear all');
% import the mesh
m=gfMesh('import','gid','../meshes/tripod.GiD.msh');
mfu=gfMeshFem(m,3);      % mesh-fem supporting a 3D-vector field
mfd=gfMeshFem(m,1);      % scalar mesh_fem, for data fields.
% the mesh_im stores the integration methods for each tetrahedron
mim=gfMeshIm(m,gf_integ('IM_TETRAHEDRON(5)'));
% we choose a P2 fem for the main unknown
gf_mesh_fem_set(mfu,'fem',gf_fem('FEM_PK(3,2)'));
% the material is homogeneous, hence we use a P0 fem for the data
gf_mesh_fem_set(mfd,'fem',gf_fem('FEM_PK(3,0)'));
% display some informations about the mesh
disp(sprintf('nbcvs=%d, nbpts=%d, nbdof=%d',gf_mesh_get(m,'nbcvs'),...
             gf_mesh_get(m,'nbpts'),gf_mesh_fem_get(mfu,'nbdof')));
P=gf_mesh_get(m,'pts'); % get list of mesh points coordinates
pidtop=find(abs(P(2,:)-13)<1e-6); % find those on top of the object
pidbot=find(abs(P(2,:)+10)<1e-6); % find those on the bottom
% build the list of faces from the list of points
ftop=gf_mesh_get(m,'faces from pid',pidtop);
fbot=gf_mesh_get(m,'faces from pid',pidbot);
% assign boundary numbers
```

```
gf_mesh_set(m,'boundary',1,ftop);
gf_mesh_set(m,'boundary',2,fbot);

E = 1e3; Nu = 0.3;
% set the Lame coefficients
lambda = E*Nu/((1+Nu)*(1-2*Nu));
mu = E/(2*(1+Nu));

% create a meshfem for the pressure field (used if incompressible ~= 0)
mfp=gfMeshFem(m); set(mfp, 'fem',gfFem('FEM_PK_DISCONTINUOUS(3,0)'));
if (linear)
  % the linearized elasticity , for small displacements
  b0 = gfMdBrick('isotropic_linearized_elasticity',mim,mfu)
  set(b0, 'param','lambda', lambda);
  set(b0, 'param','mu', mu);
  if (incompressible)
    b1 = gfMdBrick('linear incompressibility term', b0, mfp);
  else
    b1 = b0;
  end;
else
  % See also demo_nonlinear_elasticity for a better example
  if (incompressible)
    b0 = gfMdBrick('nonlinear elasticity',mim, mfu, 'Mooney Rivlin');
    b1 = gfMdBrick('nonlinear elasticity incompressibility term',b0,mfp);
    set(b0, 'param','params',[lambda;mu]);
  else
    % large deformation with a linearized material law.. not
    % a very good choice!
    b0 = gfMdBrick('nonlinear elasticity',mim, mfu, 'SaintVenant Kirchhoff');
    set(b0, 'param','params',[lambda;mu]);
    %b0 = gfMdBrick('nonlinear elasticity',mim, mfu, 'Ciarlet Geymonat');
    b1 = b0;
  end;
end

% set a vertical force on the top of the tripod
b2 = gfMdBrick('source term', b1, 1);
set(b2, 'param', 'source_term', mfd, get(mfd, 'eval', {0;-10;0}));

% attach the tripod to the ground
b3 = gfMdBrick('dirichlet', b2, 2, mfu, 'penalized');

mds=gfMdState(b3)

disp('running solve...')

t0=cputime;

get(b3, 'solve', mds, 'noisy', 'max_iter', 1000, 'max_res', 1e-6, 'lsolver', 'superlu');
disp(sprintf('solve done in %.2f sec', cputime-t0));

mfdu=gf_mesh_fem(m,1);
% the P2 fem is not derivable across elements, hence we use a discontinuous
% fem for the derivative of U.
gf_mesh_fem_set(mfdu,'fem',gf_fem('FEM_PK_DISCONTINUOUS(3,1)'));
VM=get(b0, 'von mises',mds,mfdu);
```

```
U=get(mds, 'state'); U=U(1:get(mfu, 'nbdof'));

disp('plotting ... can also take some minutes!');

% we plot the von mises on the deformed object, in superposition
% with the initial mesh.
if (linear),
  gf_plot(mfdu,VM,'mesh','on', 'cvlst', get(m, 'outer faces'),...
          'deformation',U,'deformation_mf',mfu);
else
  gf_plot(mfdu,VM,'mesh','on', 'cvlst', get(m, 'outer faces'),...
          'deformation',U,'deformation_mf',mfu,'deformation_scale',1);
end;

caxis([0 100]);
colorbar; view(180,-50); camlight;
gf_colormap('tripod');

% the von mises stress is exported into a VTK file
% (which can be viewed with 'mayavi -d tripod.vtk -m BandedSurfaceMap')
% see http://mayavi.sourceforge.net/
gf_mesh_fem_get(mfdu,'export to vtk','tripod.vtk','ascii',VM,'vm')
```

Here is the final figure, displaying the `Von Mises` stress:

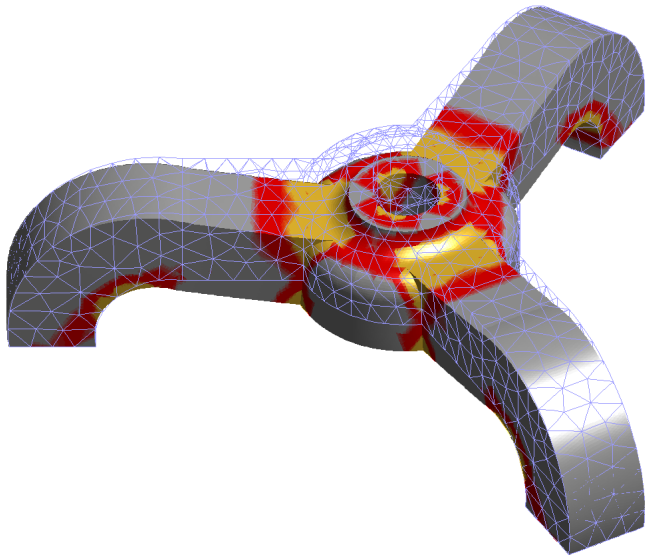

Figure 5.1: deformed tripod

Avoiding the bricks 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 *MatLab*. For example, the demonstration `demo_tripod_alt.m` is very similar to the `demo_tripod.m` except that the assembly is explicit:

```
nbd=get(mfd, 'nbdof');
F = gf_asm('boundary_source', 1, mim, mfu, mfd, repmat([0;-10;0],1,nbd));
K = gf_asm('linear_elasticity', mim, mfu, mfd, ...
           lambda*ones(1,nbd),mu*ones(1,nbd));

% handle Dirichlet condition
[H,R]=gf_asm('dirichlet', 2, mim, mfu, mfd, repmat(eye(3),[1,1,nbd]), zeros(3, nbd));
[N,U0]=gf_spmat_get(H, 'dirichlet_nullspace', R);
KK=N'*K*N;
FF=N'*F;
% solve ...
disp('solving...'); t0 = cputime;
lsolver = 1 % change this to compare the different solvers
if (lsolver == 1),     % conjugate gradient
  P=gfPrecond('ildlt',KK);
  UU=gf_linsolve('cg',KK,FF,P,'noisy','res',1e-9);
elseif (lsolver == 2), % superlu
  UU=gf_linsolve('superlu',KK,FF);
else                   % the matlab "slash" operator
  UU=KK \ FF;
end;
disp(sprintf('linear system solved in \%.2f sec', cputime-t0));
U=(N*UU).'+U0;
```

In *getfem-interface*, the assembly of vectors, and matrices is done via the `gf_asm` function. The Dirichlet condition $u(x) = r(x)$ 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.

Other examples

- the `demo_refine.m` 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.m` script shows a 3D bar which is 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.m` script shows a Stokes (viscous fluid) problem in a tank. The `demo_stokes_3D_tank_draw.m` shows how to draw a nice plot of the solution, with mesh slices and stream lines. Note that the `demo_stokes_3D_tank_alt.m` is the old example, which uses the deprecated `gf_solve` function.
- the `demo_bilaplacian.m` 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.m` 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.m` is a 2D scalar wave equation example (diffraction of a plane wave by a cylinder), with high order geometric transformations and high order FEMs.

Using Matlab Object-Oriented features

The basic functions of the *GetFEM++* toolbox do not use any advanced *MatLab* features (except that the handles to getfem objects are stored in a small *MatLab* structure). But the toolbox comes with a set of *MatLab* objects, which encapsulate the handles and make them look as real *MatLab* objects. The aim is not to provide extra-functionalities, but to have a better integration of the toolbox with *MatLab*.

Here is an example of its use:

```
>> m=gf_mesh('cartesian',0:.1:1,0:.1:1)
m =
     id: 0
    cid: 0

>> m2=gfMesh('cartesian',0:.1:1,0:.1:1)
gfMesh object ID=1 [17512 bytes], dim=2, nbpts=121, nbcvs=100
% while \kw{m} is a simple structure, \kw{m2} has been flagged by |mlab|
% as  an object of class gfMesh.  Since the \texttt{display} method for
% these  objects  have  been  overloaded,  the  toolbox  displays  some
% information about the mesh instead of the content of the structure.
>> gf_mesh_get(m,'nbpts')
ans =
   121
% pseudo member access (which calls ##gf_mesh_get(m2,'nbpts'))
>> m2.nbpts
ans =
   121
```

Refer to the OO-commands reference *GetFEM++ OO-commands* for more details.

CHAPTER

SIX

DRAW COMMAND REFERENCE

gf_colormap

Synopsis

```
c=gf_colormap(name)
```

Description :

return a colormap, or change the current colormap. name can be: 'tripod', 'chouette', 'froid', 'tank' or 'earth'.

gf_plot

Synopsis

```
[hsurf, hcontour, hquiver, hmesh, hdefmesh]=gf_plot(mesh_fem mf, U, ...)

The options are specified as pairs of "option name"/"option value"

'zplot',{'off'|'on'}       : values of ``U`` are mapped on the $z$-axis (only possible when qdim=1, r
'norm', {'off'|'on'}       : if qdim >= 2, color-plot the norm of the field
'dir',[]                    : or the scalar product of the field with 'dir' (can be a vector, or 'x',
'refine',8                  : nb of refinments for curved edges and surface plots
'interpolated',{'off'|'on'}: if triangular patch are interpolated
'pcolor',{'on'|'off'}      : if the field is scalar, a color plot of its values is plotted
'quiver',{'on'|'off'}      : if the field is vector, represent arrows
'quiver_density',50        : density of arrows in quiver plot
'quiver_scale',1           : scaling of arrows (0=>no scaling)
'mesh',{'off'|'on'}         : show the mesh ?
'meshopts',{cell(0)}          : cell array of options passed to gf_plot_slice for the mesh
'deformed_mesh', {'off'|'on'} : shows the deformed mesh (only when qdim == mdim)
'deformed_meshopts', {cell(0)}: cell array of options passed to gf_plot_slice for the deformed mesh
'deformation',[]            : plots on the deformed object (only when qdim == mdim)
'deformation_mf',[]        : plots on the deformed object (only when qdim == mdim)
'deformation_scale','10%'  : indicate the amplitude of the deformation. Can be a percentage of the me
'cvlst',[]                  : list of convexes to plot (empty=>all convexes)
'title',[]                 : set the title
'contour',[]               : list of contour values
'disp_options', {'off'|'on'} : shows the option or not.
```

Description :

The function expects `U` to be a row vector. If `U` is a scalar field, then `gf\_plot(mf,U)` will fill the mesh with colors representing the values of `U`. If `U` is a vector field, then the default behavior of `gf_plot` is to draw vectors representing the values of `U`.

On output, this function returns the handles to the various graphical objects created: `hmesh` is the handles to the mesh lines, `hbound` is the handles to the edges of the boundaries, `hfill` is the handle of the patch objects of faces, `hvert` (resp `hconv`, `hdof`) is the handles of the vertices (resp. convexes, dof) labels.

For example, plotting a scalar field on the border of a 3D mesh can be done with

```
% load the 'strange.mesh_fem' (found in the getfem_matlab/tests directory)
mf=gf_mesh_fem('load', 'strange.mesh_fem')
U=rand(1, gf_mesh_fem_get(mf, 'nbdof')); # random field that will be drawn
gf_plot(mf, U, 'refine', 25, 'cvlst', gf_mesh_get(mf,'outer faces'), 'mesh','on');
```

gf_plot_1D

Synopsis

```
gf_plot_1D(mesh_fem mf, U, ...)

The options are specified as pairs of "option name"/"option value"


'style', 'bo-'       : the line style and dof marker style (same syntax as in the matlab command 'plo
'color', []          : override the line color.
'dof_color', [1,0,0] : color of the markers for the degrees of freedom.
'width', 2           : line width.
```

Description :

This function plots a 1D finite elements field.

gf_plot_mesh

Synopsis

```
gf_plot_mesh(m, ...)

'vertices', {'off'|'on'}     : displays also vertices numbers.
'convexes', {'off'|'on'}     : displays also convexes numbers.
'dof',{'off'|'on'}           : displays also finite element nodes. In that case, ``m`` should be a ``m
'regions',BLST               : displays the boundaries listed in BLST.
'cvlst',CVLST                : display only the listed convexes. If CVLST has two rows, display only t
'edges', {'on' | 'off'}      : display edges ?
'faces', {'off'|'on'}        : fills each 2D-face of the mesh
'curved', {'off'|'on'}       : displays curved edges
'refine',N                   : refine curved edges and filled faces N times
'deformation', Udef          : optionnal deformation applied to the mesh (M must be a mesh_fem object)
'edges_color',[.6 .6 1]      : RGB values for the color of edges
'edges_width',1              : width of edges
'faces_color',[.75 .75 .75]): RGB values for the color of faces
'quality',{ 'off' | 'on' }   : Display the quality of the mesh.
```

Description :

This function is used to display a mesh.

Example

```
% the mesh is in the tests directory of the distribution
m=gf_mesh('import','gid','donut_with_quadratic_tetra_314_elements.msh');
gf_plot_mesh(m,'refine',15,'cvlst',gf_mesh_get(m,'outer faces'),'faces','on',\ldots, 'faces_colo
camlight % turn on the light!
```

gf_plot_slice

Synopsis

```
gf_plot_slice(sl, ...)

The options are specified as pairs of "option name"/"option value"


data    []          : data to be plotted (one value per slice node)
convex_data    []   : data to be plotted (one value per mesh convex)
mesh, ['auto']      : 'on' -> show the mesh (faces of edges), 'off' -> ignore mesh
mesh_edges, ['on']  : show mesh edges ?
mesh_edges_color, [0.60 0.60 1] : color of mesh edges
mesh_edges_width, [0.70] : width of mesh edges
mesh_slice_edges, ['on'] : show edges of the slice ?
mesh_slice_edges_color, [0.70 0 0] : color of slice edges
mesh_slice_edges_width, [0.50] : width of slice edges
mesh_faces, ['off'] : 'on' -> fill mesh faces (otherwise they are transparent)
mesh_faces_color, [0.75 0.75 0.75]
pcolor, ['on']      : if the field is scalar, a color plot of its values is plotted
quiver, ['on']      : if the field is vector, represent arrows
quiver_density, 50  : density of arrows in quiver plot
quiver_scale, 1     : density of arrows in quiver plot
tube, ['on']        : use tube plot for 'filar' (1D) parts of the slice
tube_color, ['red'] : color of tubes (ignored if 'data' is not empty and 'pcolor' is on)
tube_radius, ['0.5%'] : tube radius; you can use a constant, or a percentage (of the mesh size) or a
showoptions, ['on'] : display the list of options

the 'data' and 'convex_data' are mutually exclusive.
```

Description :

This function can be used to plot mesh slices. It is also used by the `gf_plot_mesh` and `gf_plot` functions.

Example : consider that you have a 3D mesh_fem `mf` and a vector field `U` defined on this mesh_fem, solution of the Stokes problem in a tank (see the demo `demo_stokes_3D_tank_draw.m` in the tests directory).

```
figure;
% slice the mesh with two half spaces, and take the boundary of the resulting quarter-cylinder
sl=gf_slice(\{'boundary',\{'intersection',\{'planar',+1,[0;0;0],[0;1;0]\},\ldots
                                          \{'planar',+1,[0;0;0],[1;0;0]\}\}\},m,6);
Usl=gf_compute(pde.mf_u,U,'interpolate on', sl);  % interpolate the solution on the slice
% show the norm of the displacement on this slice
gf_plot_slice(sl,'mesh','on','data',sqrt(sum(Usl.^2,1)),'mesh_slice_edges','off');

% another slice: now we take the lower part of the mesh
```

```
sl=gf_slice(\{'boundary',\{'intersection',\{'planar',+1,[0;0;6],[0;0;-1]\},\ldots
                                        \{'planar',+1,[0;0;0],[0;1;0]\}\}\},m,6);
Usl=gf_compute(pde.mf_u,U,'interpolate on', sl);
hold on;
gf_plot_slice(sl,'mesh','on','data',sqrt(sum(Usl.^2,1)),'mesh_slice_edges','off');

% this slice contains the transparent mesh faces displayed on the picture
sl2=gf_slice(\{'boundary',\{'planar',+1,[0;0;0],[0;1;0]\}\},\ldots
            m,6,setdiff(all_faces',TOPfaces','rows')');
gf_plot_slice(sl2,'mesh_faces','off','mesh','on','pcolor','off');

% last step is to plot the streamlines
hh=[1 5 9 12.5 16 19.5]; % vertical position of the different starting points of the streamlines
H=[zeros(2,numel(hh));hh];

% compute the streamlines
tsl=gf_slice('streamlines',pde.mf_u,U,H);
Utsl=gf_compute(pde.mf_u,U,'interpolate on', tsl);

% render them with "tube plot"
[a,h]=gf_plot_slice(tsl,'mesh','off','tube_radius',.2,'tube_color','white');
hold off;
% use a nice colormap
caxis([0 .7]);
c=[0 0 1; 0 .5 1; 0 1 .5; 0 1 0; .5 1 0; 1 .5 0; 1 .4 0; 1 0 0; 1 .2 0; 1 .4 0; 1 .6 0; 1 .8 0];
colormap(c);
```

CHAPTER

SEVEN

COMMAND 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.

The expected type of each function argument is indicated in this reference. Here is a list of these types:

int	integer value
hobj	a handle for any GetFEM++ object
scalar	scalar value
string	string
ivec	vector of integer values
vec	vector
imat	matrix of integer values
mat	matrix
spmat	sparse matrix (both matlab native sparse matrices, and GetFEM sparse matrices)
precond	GetFEM preconditioner object
mesh mesh	object descriptor (or gfMesh object)
mesh_fem	mesh fem object descriptor (or gfMeshFem object)
mesh_im	mesh im object descriptor (or gfMeshIm object)
mesh_im_data	mesh im data object descriptor (or gfMeshImData object)
mesh_slice	mesh slice object descriptor (or gfSlice object)
cvstruct	convex structure descriptor (or gfCvStruct object)
geotrans	geometric transformation descriptor (or gfGeoTrans object)
fem	fem descriptor (or gfFem object)
eltm	elementary matrix descriptor (or gfEltm object)
integ	integration method descriptor (or gfInteg object)
model	model descriptor (or gfModel object)
global_function	global function descriptor
mesher_object	mesher object descriptor
cont_struct	continuation-structure descriptor

Arguments listed between square brackets are optional. Lists between braces indicate that the argument must match one of the elements of the list. For example:

```
>> [X,Y]=dummy(int i, 'foo' | 'bar' [,vec v])
```

means that the dummy function takes two or three arguments, its first being an integer value, the second a string which is either 'foo' or 'bar', and a third optional argument. It returns two values (with the usual matlab meaning, i.e. the caller can always choose to ignore them).

gf_asm

Synopsis

```
{...} = gf_asm('generic', mesh_im mim, int order, string expression, int region, [model model,] [stri
M = gf_asm('mass matrix', mesh_im mim, mesh_fem mf1[, mesh_fem mf2[, int region]])
L = gf_asm('laplacian', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec a[, int region])
Le = gf_asm('linear elasticity', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec lambda_d, vec mu_d[,
TRHS = gf_asm('nonlinear elasticity', mesh_im mim, mesh_fem mf_u, vec U, string law, mesh_fem mf_d, m
A = gf_asm('helmholtz', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec k[, int region])
A = gf_asm('bilaplacian', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec a[, int region])
A = gf_asm('bilaplacian KL', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec a, vec nu[, int region])
V = gf_asm('volumic source', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec fd[, int region])
B = gf_asm('boundary source', int bnum, mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, vec G)
{HH, RR} = gf_asm('dirichlet', int bnum, mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, mat H, vec R [, s
Q = gf_asm('boundary qu term',int boundary_num, mesh_im mim, mesh_fem mf_u, mesh_fem mf_d, mat q)
gf_asm('define function', string name, int nb_args, string expression[, string expression_derivative_
gf_asm('undefine function', string name)
gf_asm('define linear hardening function', string name, scalar sigma_y0, scalar H, ... [string 'Frobe
gf_asm('define Ramberg Osgood hardening function', string name, scalar sigma_ref, {scalar eps_ref | s
gf_asm('expression analysis', string expression [, {@tm mesh | mesh_im mim}] [, der_order] [, model m
{...} = gf_asm('volumic' [,CVLST], expr [, mesh_ims, mesh_fems, data...])
{...} = gf_asm('boundary', int bnum, string expr [, mesh_im mim, mesh_fem mf, data...])
Mi = gf_asm('interpolation matrix', mesh_fem mf, {mesh_fem mfi | vec pts})
Me = gf_asm('extrapolation matrix',mesh_fem mf,  {mesh_fem mfe | vec pts})
B = gf_asm('integral contact Uzawa projection', int bnum, mesh_im mim, mesh_fem mf_u, vec U, mesh_fem
B = gf_asm('level set normal source term', int bnum, mesh_im mim, mesh_fem mf_u, mesh_fem mf_lambda,
M = gf_asm('lsneuman matrix', mesh_im mim, mesh_fem mf1, mesh_fem mf2, levelset ls[, int region])
M = gf_asm('nlsgrad matrix', mesh_im mim, mesh_fem mf1, mesh_fem mf2, levelset ls[, int region])
M = gf_asm('stabilization patch matrix', @tm mesh, mesh_fem mf,  mesh_im mim, real ratio, real h)
{Q, G, H, R, F} = gf_asm('pdetool boundary conditions', mf_u, mf_d, b, e[, f_expr])
```

Description :

General assembly function.

Many of the functions below use more than one mesh_fem: the main mesh_fem (mf_u) used for the main unknown, and data mesh_fem (mf_d) used for the data. It is always assumed that the Qdim of mf_d is equal to 1: if mf_d is used to describe vector or tensor data, you just have to "stack" (in fortran ordering) as many scalar fields as necessary.

Command list :

```
{...} = gf_asm('generic', mesh_im mim, int order, string expression,
int region, [model model,] [string varname, int is_variable[,
{mesh_fem mf, mesh_imd mimd}], value], ...)
```

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 the element 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.

The variables and constant (data) are listed after the region number (or optionally the model). For each variable/constant, first the variable/constant name should be given (as it is referred

in the assembly string), then 1 if it is a variable or 0 for 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 automatic differentiation is done with respect to variables only (see GetFEM++ user documentation). Test functions are only available for variables, not for constants.

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, and so on). If a model is provided, all degrees of freedom of the model will be counted first.

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,
```

```
M = gf_asm('mass matrix', mesh_im mim, mesh_fem mf1[, mesh_fem mf2[,
int region]])
```

Assembly of a mass matrix.

Return a spmat object.

```
L = gf_asm('laplacian', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d,
vec a[, int region])
```

Assembly of the matrix for the Laplacian problem.

$\nabla \cdot (a(x)\nabla u)$ with *a* a scalar.

Return a spmat object.

```
Le = gf_asm('linear elasticity', mesh_im mim, mesh_fem mf_u, mesh_fem
mf_d, vec lambda_d, vec mu_d[, int region])
```

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.

```
TRHS = gf_asm('nonlinear elasticity', mesh_im mim, mesh_fem
mf_u, vec U, string law, mesh_fem mf_d, mat params, {'tangent
matrix'|'rhs'|'incompressible tangent matrix', mesh_fem mf_p, vec
P|'incompressible rhs', mesh_fem 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 de-

fault, 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 mesh_fem *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 mesh_fem *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).

```
A = gf_asm('helmholtz', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d,
vec k[, int region])
```

Assembly of the matrix for the Helmholtz problem.

$\Delta u + k^2 u = 0$, with *k* complex scalar.

Return a spmat object.

```
A = gf_asm('bilaplacian', mesh_im mim, mesh_fem mf_u, mesh_fem mf_d,
vec a[, int region])
```

Assembly of the matrix for the Bilaplacian problem.

$\Delta(a(x)\Delta u) = 0$ with *a* scalar.

Return a spmat object.

```
A = gf_asm('bilaplacian KL', mesh_im mim, mesh_fem mf_u, mesh_fem
mf_d, vec a, vec nu[, int region])
```

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.

```
V = gf_asm('volumic source', mesh_im mim, mesh_fem mf_u, mesh_fem
mf_d, vec fd[, int region])
```

Assembly of a volumic source term.

Output a vector *V*, assembled on the mesh_fem *mf_u*, using the data vector *fd* defined on the data mesh_fem *mf_d*. *fd* may be real or complex-valued.

Return a vec object.

```
B = gf_asm('boundary source', int bnum, mesh_im mim, mesh_fem mf_u,
mesh_fem mf_d, vec 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 mesh_fem).

Return a vec object.

```
{HH, RR} = gf_asm('dirichlet', int bnum, mesh_im mim, mesh_fem mf_u,
mesh_fem mf_d, mat H, vec R [, scalar threshold])
```

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 gf_spmat_get(spmat S, 'Dirichlet_nullspace').

```
Q = gf_asm('boundary qu term',int boundary_num, mesh_im mim, mesh_fem
mf_u, mesh_fem mf_d, mat 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 mesh_fem).

Return a spmat object.

```
gf_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.

```
gf_asm('undefine function', string name)
```

Cancel the definition of a previously defined function *name* for the high level generic assembly.

```
gf_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.

```
gf_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.

```
gf_asm('expression analysis', string expression [, {@tm mesh |
mesh_im mim}] [, der_order] [, model model] [, string varname, int
is_variable[, {mesh_fem mf | mesh_imd mimd}], ...])
```

Analyse a high-level generic assembly expression and print information about the provided expression.

```
{...} = gf_asm('volumic' [,CVLST], expr [, mesh_ims, mesh_fems,
data...])
```

Low-level generic assembly procedure for volumic assembly.

The expression *expr* is evaluated over the mesh_fem'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))
```

```
{...} = gf_asm('boundary', int bnum, string expr [, mesh_im mim,
mesh_fem mf, data...])
```

Low-level generic boundary assembly.

See the help for gf_asm('volumic').

```
Mi = gf_asm('interpolation matrix', mesh_fem mf, {mesh_fem mfi | vec
pts})
```

Build the interpolation matrix from a mesh_fem onto another mesh_fem 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.

```
Me = gf_asm('extrapolation matrix',mesh_fem mf, {mesh_fem mfe | vec
pts})
```

Build the extrapolation matrix from a mesh_fem onto another mesh_fem 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.

```
B = gf_asm('integral contact Uzawa projection', int bnum, mesh_im
mim, mesh_fem mf_u, vec U, mesh_fem mf_lambda, vec vec_lambda,
mesh_fem mf_obstacle, vec obstacle, scalar r [, {scalar coeff |
mesh_fem 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.

```
B = gf_asm('level set normal source term', int bnum, mesh_im
mim, mesh_fem mf_u, mesh_fem mf_lambda, vec vec_lambda, mesh_fem
mf_levelset, vec 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.

```
M = gf_asm('lsneuman matrix', mesh_im mim, mesh_fem mf1, mesh_fem
mf2, levelset ls[, int region])
```

Assembly of a level set Neuman matrix.

Return a spmat object.

```
M = gf_asm('nlsgrad matrix', mesh_im mim, mesh_fem mf1, mesh_fem mf2,
levelset ls[, int region])
```

Assembly of a nlsgrad matrix.

Return a spmat object.

```
M = gf_asm('stabilization patch matrix', @tm mesh, mesh_fem mf,
mesh_im mim, real ratio, real h)
```

Assembly of stabilization patch matrix .

Return a spmat object.

```
{Q, G, H, R, F} = gf_asm('pdetool boundary conditions', mf_u, mf_d,
b, e[, f_expr])
```

Assembly of pdetool boundary conditions.

B is the boundary matrix exported by pdetool, and *E* is the edges array. *f_expr* is an optionnal expression (or vector) for the volumic term. On return *Q, G, H, R, F* contain the assembled boundary conditions (*Q* and *H* are matrices), similar to the ones returned by the function ASSEMB from PDETOOL.

gf_compute

Synopsis

```
n = gf_compute(mesh_fem MF, vec U, 'L2 norm', mesh_im mim[, mat CVids])
n = gf_compute(mesh_fem MF, vec U, 'L2 dist', mesh_im mim, mesh_fem mf2, vec U2[, mat CVids])
n = gf_compute(mesh_fem MF, vec U, 'H1 semi norm', mesh_im mim[, mat CVids])
n = gf_compute(mesh_fem MF, vec U, 'H1 semi dist', mesh_im mim, mesh_fem mf2, vec U2[, mat CVids])
n = gf_compute(mesh_fem MF, vec U, 'H1 norm', mesh_im mim[, mat CVids])
```

```
n = gf_compute(mesh_fem MF, vec U, 'H2 semi norm', mesh_im mim[, mat CVids])
n = gf_compute(mesh_fem MF, vec U, 'H2 norm', mesh_im mim[, mat CVids])
DU = gf_compute(mesh_fem MF, vec U, 'gradient', mesh_fem mf_du)
HU = gf_compute(mesh_fem MF, vec U, 'hessian', mesh_fem mf_h)
UP = gf_compute(mesh_fem MF, vec U, 'eval on triangulated surface', int Nrefine, [vec CVLIST])
Ui = gf_compute(mesh_fem MF, vec U, 'interpolate on', {mesh_fem mfi | slice sli | vec pts})
Ue = gf_compute(mesh_fem MF, vec U, 'extrapolate on', mesh_fem mfe)
E = gf_compute(mesh_fem MF, vec U, 'error estimate', mesh_im mim)
E = gf_compute(mesh_fem MF, vec U, 'error estimate nitsche', mesh_im mim, int GAMMAC, int GAMMAN, sca
gf_compute(mesh_fem MF, vec U, 'convect', mesh_fem mf_v, vec V, scalar dt, int nt[, string option[, v
[U2[,MF2,[,X[,Y[,Z]]]]] = gf_compute(mesh_fem MF, vec U, 'interpolate on Q1 grid', {'regular h', hxyz
```

Description :

Various computations involving the solution U to a finite element problem.

Command list :

`n = gf_compute(mesh_fem MF, vec U, 'L2 norm', mesh_im mim[, mat CVids])`

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.

`n = gf_compute(mesh_fem MF, vec U, 'L2 dist', mesh_im mim, mesh_fem mf2, vec U2[, mat CVids])`

Compute the L2 distance between *U* and *U2*.

If *CVids* is given, the norm will be computed only on the listed elements.

`n = gf_compute(mesh_fem MF, vec U, 'H1 semi norm', mesh_im mim[, mat CVids])`

Compute the L2 norm of grad(*U*).

If *CVids* is given, the norm will be computed only on the listed elements.

`n = gf_compute(mesh_fem MF, vec U, 'H1 semi dist', mesh_im mim, mesh_fem mf2, vec U2[, mat CVids])`

Compute the semi H1 distance between *U* and *U2*.

If *CVids* is given, the norm will be computed only on the listed elements.

`n = gf_compute(mesh_fem MF, vec U, 'H1 norm', mesh_im mim[, mat CVids])`

Compute the H1 norm of *U*.

If *CVids* is given, the norm will be computed only on the listed elements.

`n = gf_compute(mesh_fem MF, vec U, 'H2 semi norm', mesh_im mim[, mat CVids])`

Compute the L2 norm of D^2(*U*).

If *CVids* is given, the norm will be computed only on the listed elements.

`n = gf_compute(mesh_fem MF, vec U, 'H2 norm', mesh_im mim[, mat CVids])`

Compute the H2 norm of *U*.

If *CVids* is given, the norm will be computed only on the listed elements.

`DU = gf_compute(mesh_fem MF, vec U, 'gradient', mesh_fem mf_du)`

Compute the gradient of the field *U* defined on mesh_fem *mf_du*.

The gradient is interpolated on the mesh_fem *mf_du*, and returned in *DU*. For example, if *U* is defined on a P2 mesh_fem, *DU* should be evaluated on a P1-discontinuous mesh_fem. *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*.

`HU = gf_compute(mesh_fem MF, vec U, 'hessian', mesh_fem mf_h)`

Compute the hessian of the field *U* defined on mesh_fem *mf_h*.

See also gf_compute('gradient', mesh_fem mf_du).

`UP = gf_compute(mesh_fem MF, vec U, 'eval on triangulated surface', int Nrefine, [vec CVLIST])`

[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.

`Ui = gf_compute(mesh_fem MF, vec U, 'interpolate on', {mesh_fem mfi | slice sli | vec pts})`

Interpolate a field on another mesh_fem or a slice or a list of points.

- **Interpolation on another mesh_fem *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 gf_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')

`Ue = gf_compute(mesh_fem MF, vec U, 'extrapolate on', mesh_fem mfe)`

Extrapolate a field on another mesh_fem.

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')

`E = gf_compute(mesh_fem MF, vec U, 'error estimate', mesh_im 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.

```
E = gf_compute(mesh_fem MF, vec U, 'error estimate nitsche', mesh_im
mim, int GAMMAC, int GAMMAN, scalar lambda_, scalar mu_, scalar
gamma0, scalar f_coeff, scalar 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.

```
gf_compute(mesh_fem MF, vec U, 'convect', mesh_fem 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.

```
[U2[,MF2,[,X[,Y[,Z]]]]] = gf_compute(mesh_fem MF, vec U, 'interpolate
on Q1 grid', {'regular h', hxyz | 'regular N', Nxyz | X[,Y[,Z]]})
```

Creates a cartesian Q1 mesh fem and interpolates U on it. The returned field U2 is organized in a matrix such that in can be drawn via the MATLAB command 'pcolor'. The first dimension is the Qdim of MF (i.e. 1 if U is a scalar field)

example (mf_u is a 2D mesh_fem): >> Uq=gf_compute(mf_u, U, 'interpolate on Q1 grid', 'regular h', [.05, .05]); >> pcolor(squeeze(Uq(1,:,:)));

gf_cont_struct

Synopsis

```
S = gf_cont_struct(model md, string dataname_parameter[,string dataname_init, string dataname_final,
```

Description :

General constructor for cont_struct objects.

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).

Command list :

```
S = gf_cont_struct(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 parametrisation is done via a vector datum, *dataname_init* and *dataname_final* should store two given

values of this datum determining the parametrisation, 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.).

gf_cont_struct_get

Synopsis

```
h = gf_cont_struct_get(cont_struct CS, 'init step size')
h = gf_cont_struct_get(cont_struct CS, 'min step size')
h = gf_cont_struct_get(cont_struct CS, 'max step size')
h = gf_cont_struct_get(cont_struct CS, 'step size decrement')
h = gf_cont_struct_get(cont_struct CS, 'step size increment')
[vec tangent_sol, scalar tangent_par] = gf_cont_struct_get(cont_struct CS, 'compute tangent', vec sol
E = gf_cont_struct_get(cont_struct CS, 'init Moore-Penrose continuation', vec solution, scalar parame
E = gf_cont_struct_get(cont_struct CS, 'Moore-Penrose continuation', vec solution, scalar parameter,
t = gf_cont_struct_get(cont_struct CS, 'non-smooth bifurcation test', vec solution1, scalar parameter
t = gf_cont_struct_get(cont_struct CS, 'bifurcation test function')
{X, gamma, T_X, T_gamma} = gf_cont_struct_get(cont_struct CS, 'sing_data')
s = gf_cont_struct_get(cont_struct CS, 'char')
gf_cont_struct_get(cont_struct CS, 'display')
```

Description :

General function for querying information about cont_struct objects and for applying them to numerical continuation.

Command list :

`h = gf_cont_struct_get(cont_struct CS, 'init step size')`

Return an initial step size for continuation.

`h = gf_cont_struct_get(cont_struct CS, 'min step size')`

Return the minimum step size for continuation.

`h = gf_cont_struct_get(cont_struct CS, 'max step size')`

Return the maximum step size for continuation.

`h = gf_cont_struct_get(cont_struct CS, 'step size decrement')`

Return the decrement ratio of the step size for continuation.

`h = gf_cont_struct_get(cont_struct CS, 'step size increment')`

Return the increment ratio of the step size for continuation.

`[vec tangent_sol, scalar tangent_par] = gf_cont_struct_get(cont_struct CS, 'compute tangent', vec solution, scalar parameter, vec tangent_sol, scalar tangent_par)`

Compute and return an updated tangent.

`E = gf_cont_struct_get(cont_struct CS, 'init Moore-Penrose continuation', vec solution, scalar parameter, scalar 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*.

`E = gf_cont_struct_get(cont_struct CS, 'Moore-Penrose continuation', vec solution, scalar parameter, vec tangent_sol, scalar tangent_par, scalar 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!

```
t = gf_cont_struct_get(cont_struct CS, 'non-smooth bifurcation
test', vec solution1, scalar parameter1, vec tangent_sol1, scalar
tangent_par1, vec solution2, scalar parameter2, vec tangent_sol2,
scalar 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*.

```
t = gf_cont_struct_get(cont_struct CS, 'bifurcation test function')
```

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

```
{X, gamma, T_X, T_gamma} = gf_cont_struct_get(cont_struct CS,
'sing_data')
```

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

```
s = gf_cont_struct_get(cont_struct CS, 'char')
```

Output a (unique) string representation of the cont_struct.

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

```
gf_cont_struct_get(cont_struct CS, 'display')
```

Display a short summary for a cont_struct object.

gf_cvstruct_get

Synopsis

```
n = gf_cvstruct_get(cvstruct CVS, 'nbpts')
d = gf_cvstruct_get(cvstruct CVS, 'dim')
cs = gf_cvstruct_get(cvstruct CVS, 'basic structure')
cs = gf_cvstruct_get(cvstruct CVS, 'face', int F)
I = gf_cvstruct_get(cvstruct CVS, 'facepts', int F)
s = gf_cvstruct_get(cvstruct CVS, 'char')
gf_cvstruct_get(cvstruct CVS, 'display')
```

Description :

General function for querying information about convex_structure objects.

The convex structures are internal structures of getfem++. They do not contain points positions. These structures are recursive, since the faces of a convex structures are convex structures.

Command list :

```
n = gf_cvstruct_get(cvstruct CVS, 'nbpts')
```

Get the number of points of the convex structure.

```
d = gf_cvstruct_get(cvstruct CVS, 'dim')
```

Get the dimension of the convex structure.

```
cs = gf_cvstruct_get(cvstruct CVS, 'basic structure')
```

Get the simplest convex structure.

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

```
cs = gf_cvstruct_get(cvstruct CVS, 'face', int F)
```

Return the convex structure of the face *F*.

```
I = gf_cvstruct_get(cvstruct CVS, 'facepts', int F)
```

Return the list of point indices for the face *F*.

```
s = gf_cvstruct_get(cvstruct CVS, 'char')
```

Output a string description of the cvstruct.

```
gf_cvstruct_get(cvstruct CVS, 'display')
```

displays a short summary for a cvstruct object.

gf_delete

Synopsis

```
gf_delete(I[, J, K,...])
```

Description :

Delete an existing getfem object from memory (mesh, mesh_fem, etc.).

SEE ALSO: gf_workspace, gf_mesh, gf_mesh_fem.

Command list :

```
gf_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.

gf_eltm

Synopsis

```
E = gf_eltm('base', fem FEM)
E = gf_eltm('grad', fem FEM)
E = gf_eltm('hessian', fem FEM)
E = gf_eltm('normal')
E = gf_eltm('grad_geotrans')
E = gf_eltm('grad_geotrans_inv')
E = gf_eltm('product', eltm A, eltm B)
```

Description :

General constructor for eltm objects.

This object represents a type of elementary matrix. In order to obtain a numerical value of these matrices, see gf_mesh_im_get(mesh_im MI, '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.

Command list :

`E = gf_eltm('base', fem FEM)`

return a descriptor for the integration of shape functions on elements, using the fem *FEM*.

`E = gf_eltm('grad', fem FEM)`

return a descriptor for the integration of the gradient of shape functions on elements, using the fem *FEM*.

`E = gf_eltm('hessian', fem FEM)`

return a descriptor for the integration of the hessian of shape functions on elements, using the fem *FEM*.

`E = gf_eltm('normal')`

return a descriptor for the unit normal of convex faces.

`E = gf_eltm('grad_geotrans')`

return a descriptor to the gradient matrix of the geometric transformation.

`E = gf_eltm('grad_geotrans_inv')`

return a descriptor to the inverse of the gradient matrix of the geometric transformation (this is rarely used).

`E = gf_eltm('product', eltm A, eltm B)`

return a descriptor for the integration of the tensorial product of elementary matrices *A* and *B*.

gf_fem

Synopsis

```
F = gf_fem('interpolated_fem', mesh_fem mf, mesh_im mim, [ivec blocked_dof])
F = gf_fem(string fem_name)
```

Description :

General constructor for fem objects.

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

Command list :

`F = gf_fem('interpolated_fem', mesh_fem mf, mesh_im mim, [ivec blocked_dof])`

Build a special fem which is interpolated from another mesh_fem.

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

Note that this finite element may be quite slow, and eats much memory.

```
F = gf_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.

gf_fem_get

Synopsis

```
n = gf_fem_get(fem F, 'nbdof'[, int cv])
n = gf_fem_get(fem F, 'index of global dof', cv)
d = gf_fem_get(fem F, 'dim')
td = gf_fem_get(fem F, 'target_dim')
P = gf_fem_get(fem F, 'pts'[, int cv])
b = gf_fem_get(fem F, 'is_equivalent')
b = gf_fem_get(fem F, 'is_lagrange')
b = gf_fem_get(fem F, 'is_polynomial')
d = gf_fem_get(fem F, 'estimated_degree')
E = gf_fem_get(fem F, 'base_value',mat p)
ED = gf_fem_get(fem F, 'grad_base_value',mat p)
EH = gf_fem_get(fem F, 'hess_base_value',mat p)
gf_fem_get(fem F, 'poly_str')
string = gf_fem_get(fem F, 'char')
gf_fem_get(fem F, 'display')
```

Description :

General function for querying information about FEM objects.

Command list :

`n = gf_fem_get(fem F, 'nbdof'[, int cv])`

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.

`n = gf_fem_get(fem F, 'index of global dof', cv)`

Return the index of global dof for special fems such as interpolated fem.

`d = gf_fem_get(fem F, 'dim')`

Return the dimension (dimension of the reference convex) of the fem.

`td = gf_fem_get(fem F, 'target_dim')`

Return the dimension of the target space.

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

`P = gf_fem_get(fem F, 'pts'[, int cv])`

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.

```
b = gf_fem_get(fem F, '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.

```
b = gf_fem_get(fem F, 'is_lagrange')
```

Return 0 if the fem is not of Lagrange type.

```
b = gf_fem_get(fem F, 'is_polynomial')
```

Return 0 if the basis functions are not polynomials.

```
d = gf_fem_get(fem F, 'estimated_degree')
```

Return an estimation of the polynomial degree of the fem.

This is an estimation for fem which are not polynomials.

```
E = gf_fem_get(fem F, 'base_value',mat p)
```

Evaluate all basis functions of the FEM at point *p*.

p is supposed to be in the reference convex!

```
ED = gf_fem_get(fem F, 'grad_base_value',mat p)
```

Evaluate the gradient of all base functions of the fem at point *p*.

p is supposed to be in the reference convex!

```
EH = gf_fem_get(fem F, 'hess_base_value',mat p)
```

Evaluate the Hessian of all base functions of the fem at point *p*.

p is supposed to be in the reference convex!.

```
gf_fem_get(fem F, 'poly_str')
```

Return the polynomial expressions of its basis functions in the reference convex.

The result is expressed as a cell array of strings. Of course this will fail on non-polynomial fem's.

```
string = gf_fem_get(fem F, 'char')
```

Ouput a (unique) string representation of the fem.

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

```
gf_fem_get(fem F, 'display')
```

displays a short summary for a fem object.

gf_geotrans

Synopsis

```
GT = gf_geotrans(string name)
```

Description :

General constructor for geotrans objects.

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..)

Command list :

```
GT = gf_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

gf_geotrans_get

Synopsis

```
d = gf_geotrans_get(geotrans GT, 'dim')
b = gf_geotrans_get(geotrans GT, 'is_linear')
n = gf_geotrans_get(geotrans GT, 'nbpts')
P = gf_geotrans_get(geotrans GT, 'pts')
N = gf_geotrans_get(geotrans GT, 'normals')
Pt = gf_geotrans_get(geotrans GT, 'transform',mat G, mat Pr)
s = gf_geotrans_get(geotrans GT, 'char')
gf_geotrans_get(geotrans GT, 'display')
```

Description :

General function for querying information about geometric transformations objects.

Command list :

```
d = gf_geotrans_get(geotrans GT, 'dim')
```

Get the dimension of the geotrans.

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

```
b = gf_geotrans_get(geotrans GT, 'is_linear')
```

Return 0 if the geotrans is not linear.

```
n = gf_geotrans_get(geotrans GT, 'nbpts')
```

Return the number of points of the geotrans.

```
P = gf_geotrans_get(geotrans GT, 'pts')
```

Return the reference convex points of the geotrans.

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

```
N = gf_geotrans_get(geotrans GT, '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.

```
Pt = gf_geotrans_get(geotrans GT, 'transform',mat G, mat 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.

```
s = gf_geotrans_get(geotrans GT, 'char')
```

Output a (unique) string representation of the geotrans.

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

```
gf_geotrans_get(geotrans GT, 'display')
```

displays a short summary for a geotrans object.

gf_global_function

Synopsis

```
GF = gf_global_function('cutoff', int fn, scalar r, scalar r1, scalar r0)
GF = gf_global_function('crack', int fn)
GF = gf_global_function('parser', string val[, string grad[, string hess]])
GF = gf_global_function('product', global_function F, global_function G)
GF = gf_global_function('add', global_function gf1, global_function gf2)
```

Description :

General constructor for global_function objects.

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).

Command list :

```
GF = gf_global_function('cutoff', int fn, scalar r, scalar r1, scalar
r0)
```

Create a cutoff global function.

```
GF = gf_global_function('crack', int fn)
```

Create a near-tip asymptotic global function for modelling cracks.

```
GF = gf_global_function('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 = gf_global_function('product', global_function F, global_function
G)
```

Create a product of two global functions.

`GF = gf_global_function('add', global_function gf1, global_function gf2)`

Create a add of two global functions.

gf_global_function_get

Synopsis

```
VALs = gf_global_function_get(global_function GF, 'val',mat PTs)
GRADs = gf_global_function_get(global_function GF, 'grad',mat PTs)
HESSs = gf_global_function_get(global_function GF, 'hess',mat PTs)
s = gf_global_function_get(global_function GF, 'char')
gf_global_function_get(global_function GF, 'display')
```

Description :

General function for querying information about global_function objects.

Command list :

`VALs = gf_global_function_get(global_function GF, 'val',mat PTs)`

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

`GRADs = gf_global_function_get(global_function GF, 'grad',mat PTs)`

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

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

`HESSs = gf_global_function_get(global_function GF, 'hess',mat PTs)`

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

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

`s = gf_global_function_get(global_function GF, 'char')`

Output a (unique) string representation of the global_function.

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

`gf_global_function_get(global_function GF, 'display')`

displays a short summary for a global_function object.

gf_integ

Synopsis

```
I = gf_integ(string method)
```

Description :

General constructor for integ objects.

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

Command list :

```
I = gf_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 = gf_integ('IM_PRODUCT(IM_GAUSS1D(5),IM_GAUSS1D(5))')

is the same as:

- I = gf_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.

gf_integ_get

Synopsis

```
b = gf_integ_get(integ I, 'is_exact')
d = gf_integ_get(integ I, 'dim')
n = gf_integ_get(integ I, 'nbpts')
Pp = gf_integ_get(integ I, 'pts')
Pf = gf_integ_get(integ I, 'face_pts',F)
Cp = gf_integ_get(integ I, 'coeffs')
```

```
Cf = gf_integ_get(integ I, 'face_coeffs',F)
s = gf_integ_get(integ I, 'char')
gf_integ_get(integ I, 'display')
```

Description :

General function for querying information about integration method objects.

Command list :

`b = gf_integ_get(integ I, 'is_exact')`

Return 0 if the integration is an approximate one.

`d = gf_integ_get(integ I, 'dim')`

Return the dimension of the reference convex of the method.

`n = gf_integ_get(integ I, '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!

`Pp = gf_integ_get(integ I, 'pts')`

Return the list of integration points

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

`Pf = gf_integ_get(integ I, 'face_pts',F)`

Return the list of integration points for a face.

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

`Cp = gf_integ_get(integ I, 'coeffs')`

Returns the coefficients associated to each integration point.

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

`Cf = gf_integ_get(integ I, '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!

`s = gf_integ_get(integ I, 'char')`

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

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

`gf_integ_get(integ I, 'display')`

displays a short summary for a integ object.

gf_levelset

Synopsis

```
LS = gf_levelset(mesh m, int d[, string 'ws'| string f1[, string f2 | string 'ws']])
```

Description :

General constructor for levelset objects.

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.

Command list :

`LS = gf_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 mesh_fem 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.

gf_levelset_get

Synopsis

```
V = gf_levelset_get(levelset LS, 'values', int nls)
d = gf_levelset_get(levelset LS, 'degree')
mf = gf_levelset_get(levelset LS, 'mf')
z = gf_levelset_get(levelset LS, 'memsize')
s = gf_levelset_get(levelset LS, 'char')
gf_levelset_get(levelset LS, 'display')
```

Description :

General function for querying information about LEVELSET objects.

Command list :

`V = gf_levelset_get(levelset LS, 'values', int nls)`

Return the vector of dof for *nls* funtion.

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

`d = gf_levelset_get(levelset LS, 'degree')`

Return the degree of lagrange representation.

`mf = gf_levelset_get(levelset LS, 'mf')`

Return a reference on the mesh_fem object.

`z = gf_levelset_get(levelset LS, 'memsize')`

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

```
s = gf_levelset_get(levelset LS, '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.

```
gf_levelset_get(levelset LS, 'display')
```

displays a short summary for a levelset.

gf_levelset_set

Synopsis

```
gf_levelset_set(levelset LS, 'values', {mat v1|string func_1}[, mat v2|string func_2])
gf_levelset_set(levelset LS, 'simplify'[, scalar eps=0.01])
```

Description :

General function for modification of LEVELSET objects.

Command list :

```
gf_levelset_set(levelset LS, 'values', {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*)

```
gf_levelset_set(levelset LS, 'simplify'[, scalar eps=0.01])
```

Simplify dof of level-set optionally with the parameter *eps*.

gf_linsolve

Synopsis

```
X = gf_linsolve('gmres', spmat M, vec b[, int restart][, precond P][,'noisy'][,'res', r][,'maxiter',
X = gf_linsolve('cg', spmat M, vec b [, precond P][,'noisy'][,'res', r][,'maxiter', n])
X = gf_linsolve('bicgstab', spmat M, vec b [, precond P][,'noisy'][,'res', r][,'maxiter', n])
{U, cond} = gf_linsolve('lu', spmat M, vec b)
{U, cond} = gf_linsolve('superlu', spmat M, vec b)
{U, cond} = gf_linsolve('mumps', spmat M, vec b)
```

Description :

Various linear system solvers.

Command list :

```
X = gf_linsolve('gmres', spmat M, vec b[, int restart][, precond
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.

```
X = gf_linsolve('cg', spmat M, vec b [, precond P][,'noisy'][,'res',
r][,'maxiter', n])
```

Solve *M.X = b* with the conjugated gradient method.

Optionally using *P* as preconditioner.

```
X = gf_linsolve('bicgstab', spmat M, vec b [, precond
P][,'noisy'][,'res', r][,'maxiter', n])
```

Solve *M.X = b* with the bi-conjugated gradient stabilized method.

Optionally using *P* as a preconditioner.

```
{U, cond} = gf_linsolve('lu', spmat M, vec b)
```

Alias for gf_linsolve('superlu',...)

```
{U, cond} = gf_linsolve('superlu', spmat M, vec b)
```

Solve *M.U = b* apply the SuperLU solver (sparse LU factorization).

The condition number estimate *cond* is returned with the solution *U*.

```
{U, cond} = gf_linsolve('mumps', spmat M, vec b)
```

Solve *M.U = b* using the MUMPS solver.

gf_mesh

Synopsis

```
M = gf_mesh('empty', int dim)
M = gf_mesh('cartesian', vec X[, vec Y[, vec Z,..]])
M = gf_mesh('pyramidal', vec X[, vec Y[, vec Z,..]])
M = gf_mesh('cartesian Q1', vec X, vec Y[, vec Z,..])
M = gf_mesh('triangles grid', vec X, vec Y)
M = gf_mesh('regular simplices', vec X[, vec Y[, vec Z,...]]['degree', int k]['noised'])
M = gf_mesh('curved', mesh m, vec F)
M = gf_mesh('prismatic', mesh m, int nl[, int degree])
M = gf_mesh('pt2D', mat P, imat T[, int n])
M = gf_mesh('ptND', mat P, imat T)
M = gf_mesh('load', string filename)
M = gf_mesh('from string', string s)
M = gf_mesh('import', string format, string filename)
M = gf_mesh('clone', mesh m2)
M = gf_mesh('generate', mesher_object mo, scalar h[, int K = 1[, mat vertices]])
```

Description :

General constructor for mesh objects.

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

Note that for recent (> 6.0) versions of matlab, you should replace the calls to 'gf_mesh' with 'gfMesh' (this will instruct Matlab to consider the getfem mesh as a regular matlab object that can be manipulated with get() and set() methods).

Command list :

```
M = gf_mesh('empty', int dim)
```

Create a new empty mesh.

```
M = gf_mesh('cartesian', vec X[, vec Y[, vec Z,..]])
```

Build quickly a regular mesh of quadrangles, cubes, etc.

```
M = gf_mesh('pyramidal', vec X[, vec Y[, vec Z,..]])
```

Build quickly a regular mesh of pyramids, etc.

```
M = gf_mesh('cartesian Q1', vec X, vec Y[, vec Z,..])
```

Build quickly a regular mesh of quadrangles, cubes, etc. with Q1 elements.

```
M = gf_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 `gf_mesh('ptND')`, `gf_mesh('regular simplices')` and `gf_mesh('cartesian')`).

```
M = gf_mesh('regular simplices', vec X[, vec Y[, vec
Z,...]]['degree', int k]['noised'])
```

Mesh a n-dimensionnal parallelepipeded with simplices (triangles, tetrahedrons etc) .

The optional degree may be used to build meshes with non linear geometric transformations.

```
M = gf_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 mesh_fem object, in that case its linked mesh will be used.

```
M = gf_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 = gf_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).

Can be used to Convert a "pdetool" triangulation exported in variables P and T into a GETFEM mesh.

```
M = gf_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 = gf_mesh('load', string filename)
```

Load a mesh from a getfem++ ascii mesh file.

See also `gf_mesh_get(mesh M, 'save', string filename)`.

`M = gf_mesh('from string', string s)`

Load a mesh from a string description.

For example, a string returned by `gf_mesh_get(mesh M, 'char')`.

`M = gf_mesh('import', string format, string filename)`

Import a mesh.

format may be:

- 'gmsh' for a mesh created with *Gmsh*
- 'gid' for a mesh created with *GiD*
- 'cdb' for a mesh created with *ANSYS*
- 'am_fmt' for a mesh created with *EMC2*

`M = gf_mesh('clone', mesh m2)`

Create a copy of a mesh.

`M = gf_mesh('generate', mesher_object 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 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 desactivated using *gf_util('trace level', 2)*. More information can be obtained by *gf_util('trace level', 4)*. See `gf_mesher_object` to manipulate geometric primitives in order to desribe the geometry.

gf_mesh_get

Synopsis

```
d = gf_mesh_get(mesh M, 'dim')
np = gf_mesh_get(mesh M, 'nbpts')
nc = gf_mesh_get(mesh M, 'nbcvs')
P = gf_mesh_get(mesh M, 'pts'[, ivec PIDs])
Pid = gf_mesh_get(mesh M, 'pid')
PIDs = gf_mesh_get(mesh M, 'pid in faces', imat CVFIDs)
PIDs = gf_mesh_get(mesh M, 'pid in cvids', imat CVIDs)
PIDs = gf_mesh_get(mesh M, 'pid in regions', imat RIDs)
PIDs = gf_mesh_get(mesh M, 'pid from coords', mat PTS[, scalar radius=0])
{Pid, IDx} = gf_mesh_get(mesh M, 'pid from cvid'[, imat CVIDs])
{Pts, IDx} = gf_mesh_get(mesh M, 'pts from cvid'[, imat CVIDs])
CVid = gf_mesh_get(mesh M, 'cvid')
m = gf_mesh_get(mesh M, 'max pid')
m = gf_mesh_get(mesh M, 'max cvid')
```

```
[E,C] = gf_mesh_get(mesh M, 'edges' [, CVLST][, 'merge'])
[E,C] = gf_mesh_get(mesh M, 'curved edges', int N [, CVLST])
PIDs = gf_mesh_get(mesh M, 'orphaned pid')
CVIDs = gf_mesh_get(mesh M, 'cvid from pid', ivec PIDs[, bool share=False])
CVFIDs = gf_mesh_get(mesh M, 'faces from pid', ivec PIDs)
CVFIDs = gf_mesh_get(mesh M, 'outer faces'[, CVIDs])
CVFIDs = gf_mesh_get(mesh M, 'inner faces'[, CVIDs])
CVFIDs = gf_mesh_get(mesh M, 'outer faces with direction', vec v, scalar angle [, CVIDs])
CVFIDs = gf_mesh_get(mesh M, 'outer faces in box', vec pmin, vec pmax [, CVIDs])
CVFIDs = gf_mesh_get(mesh M, 'adjacent face', int cvid, int fid)
CVFIDs = gf_mesh_get(mesh M, 'faces from cvid'[, ivec CVIDs][, 'merge'])
[mat T] = gf_mesh_get(mesh M, 'triangulated surface', int Nrefine [,CVLIST])
N = gf_mesh_get(mesh M, 'normal of face', int cv, int f[, int nfpt])
N = gf_mesh_get(mesh M, 'normal of faces', imat CVFIDs)
Q = gf_mesh_get(mesh M, 'quality'[, ivec CVIDs])
A = gf_mesh_get(mesh M, 'convex area'[, ivec CVIDs])
A = gf_mesh_get(mesh M, 'convex radius'[, ivec CVIDs])
{S, CV2S} = gf_mesh_get(mesh M, 'cvstruct'[, ivec CVIDs])
{GT, CV2GT} = gf_mesh_get(mesh M, 'geotrans'[, ivec CVIDs])
RIDs = gf_mesh_get(mesh M, 'boundaries')
RIDs = gf_mesh_get(mesh M, 'regions')
RIDs = gf_mesh_get(mesh M, 'boundary')
CVFIDs = gf_mesh_get(mesh M, 'region', ivec RIDs)
gf_mesh_get(mesh M, 'save', string filename)
s = gf_mesh_get(mesh M, 'char')
gf_mesh_get(mesh M, 'export to vtk', string filename, ... [,'ascii'][,'quality'])
gf_mesh_get(mesh M, 'export to dx', string filename, ... [,'ascii'][,'append'][,'as',string name,[,'s
gf_mesh_get(mesh M, 'export to pos', string filename[, string name])
z = gf_mesh_get(mesh M, 'memsize')
gf_mesh_get(mesh M, 'display')
```

Description :

General mesh inquiry function. All these functions accept also a mesh_fem argument instead of a mesh M (in that case, the mesh_fem linked mesh will be used). Note that when your mesh is recognized as a Matlab object , you can simply use "get(M, 'dim')" instead of "gf_mesh_get(M, 'dim')".

Command list :

`d = gf_mesh_get(mesh M, 'dim')`

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

`np = gf_mesh_get(mesh M, 'nbpts')`

Get the number of points of the mesh.

`nc = gf_mesh_get(mesh M, 'nbcvs')`

Get the number of convexes of the mesh.

`P = gf_mesh_get(mesh M, 'pts'[, ivec PIDs])`

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 gf_mesh_get(mesh M, 'max_pid') columns, which might be greater than gf_mesh_get(mesh M, 'nbpts') (if some points of the mesh have been destroyed and no call to gf_mesh_set(mesh M, 'optimize structure') have been issued). The columns corresponding to deleted points will be filled with NaN. You can use gf_mesh_get(mesh M, 'pid') to filter such invalid points.

```
Pid = gf_mesh_get(mesh M, 'pid')
```

Return the list of points #id of the mesh.

Note that their numbering is not supposed to be contiguous from 1 to gf_mesh_get(mesh M, 'nbpts'), especially if some points have been removed from the mesh. You can use gf_mesh_set(mesh M, 'optimize_structure') to enforce a contiguous numbering. Pid is a row vector.

```
PIDs = gf_mesh_get(mesh M, 'pid in faces', imat CVFIDs)
```

Search 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 row vector containing points #id.

```
PIDs = gf_mesh_get(mesh M, 'pid in cvids', imat CVIDs)
```

Search point #id listed in *CVIDs*.

PIDs is a row vector containing points #id.

```
PIDs = gf_mesh_get(mesh M, 'pid in regions', imat RIDs)
```

Search point #id listed in *RIDs*.

PIDs is a row vector containing points #id.

```
PIDs = gf_mesh_get(mesh M, 'pid from coords', mat PTS[, scalar
radius=0])
```

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

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

```
{Pid, IDx} = gf_mesh_get(mesh M, 'pid from cvid'[, imat CVIDs])
```

Return the points attached to each convex of the mesh.

If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs = gf_mesh_get(mesh M, 'max cvid')*). *IDx* is a row vector, length(IDx) = length(CVIDs)+1. *Pid* is a row 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)-1).

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

```
{Pts, IDx} = gf_mesh_get(mesh M, 'pts from cvid'[, imat CVIDs])
```

Search point listed in *CVID*.

If *CVIDs* is omitted, all the convexes will be considered (equivalent to *CVIDs = gf_mesh_get(mesh M, 'max cvid')*). *IDx* is a row vector, length(IDx) = length(CVIDs)+1. *Pts* is a row 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)-1).

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

```
CVid = gf_mesh_get(mesh M, 'cvid')
```

Return the list of all convex #id.

Note that their numbering is not supposed to be contiguous from 1 to gf_mesh_get(mesh M, 'nbcvs'), especially if some points have been removed from the mesh. You can use gf_mesh_set(mesh M, 'optimize_structure') to enforce a contiguous numbering. CVid is a row vector.

```
m = gf_mesh_get(mesh M, 'max pid')
```

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

```
m = gf_mesh_get(mesh M, 'max cvid')
```

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

```
[E,C] = gf_mesh_get(mesh M, 'edges' [, 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.

```
[E,C] = gf_mesh_get(mesh M, 'curved edges', int N [, CVLST])
```

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

More sophisticated version of gf_mesh_get(mesh M, 'edges') designed for curved elements. This one will return N (N>=2) points of the (curved) edges. With N==2, this is equivalent to gf_mesh_get(mesh M, '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.

```
PIDs = gf_mesh_get(mesh M, 'orphaned pid')
```

Search point #id which are not linked to a convex.

```
CVIDs = gf_mesh_get(mesh M, 'cvid from pid', ivec PIDs[, bool
share=False])
```

Search 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 row vector (possibly empty).

```
CVFIDs = gf_mesh_get(mesh M, 'faces from pid', ivec 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*.

```
CVFIDs = gf_mesh_get(mesh M, 'outer faces'[, 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 *CVIDs* is not given, all convexes are considered, and it basically returns the mesh boundary. If *CVIDs* is given, it returns the boundary of the convex set whose #ids are listed in *CVIDs*.

```
CVFIDs = gf_mesh_get(mesh M, 'inner faces'[, CVIDs])
```

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 neighbour elements.

```
CVFIDs = gf_mesh_get(mesh M, 'outer faces with direction', vec v,
scalar angle [, 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). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

```
CVFIDs = gf_mesh_get(mesh M, 'outer faces in box', vec pmin, vec pmax
[, 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). If *CVIDs* is given, it returns portion of the boundary of the convex set defined by the #ids listed in *CVIDs*.

```
CVFIDs = gf_mesh_get(mesh M, 'adjacent face', int cvid, int fid)
```

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

```
CVFIDs = gf_mesh_get(mesh M, 'faces from cvid'[, 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*.

```
[mat T] = gf_mesh_get(mesh M, 'triangulated surface', int Nrefine
[,CVLIST])
```

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

Similar function to gf_mesh_get(mesh M, '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'))

```
N = gf_mesh_get(mesh M, 'normal of face', int cv, int f[, int nfpt])
```

Evaluates the normal 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.

```
N = gf_mesh_get(mesh M, 'normal of faces', imat CVFIDs)
```

Evaluates (at face centers) the normals 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).

```
Q = gf_mesh_get(mesh M, 'quality'[, ivec CVIDs])
```

Return an estimation of the quality of each convex ($0 \leq Q \leq 1$).

```
A = gf_mesh_get(mesh M, 'convex area'[, ivec CVIDs])
```

Return an estimate of the area of each convex.

```
A = gf_mesh_get(mesh M, 'convex radius'[, ivec CVIDs])
```

Return an estimate of the radius of each convex.

```
{S, CV2S} = gf_mesh_get(mesh M, 'cvstruct'[, ivec CVIDs])
```

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*.

```
{GT, CV2GT} = gf_mesh_get(mesh M, 'geotrans'[, ivec CVIDs])
```

Returns an array of the geometric transformations.

See also gf_mesh_get(mesh M, 'cvstruct').

```
RIDs = gf_mesh_get(mesh M, 'boundaries')
```

DEPRECATED FUNCTION. Use 'regions' instead.

```
RIDs = gf_mesh_get(mesh M, 'regions')
```

Return the list of valid regions stored in the mesh.

```
RIDs = gf_mesh_get(mesh M, 'boundary')
```

DEPRECATED FUNCTION. Use 'region' instead.

```
CVFIDs = gf_mesh_get(mesh M, 'region', ivec 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 0 when the whole convex is in the regions).

```
gf_mesh_get(mesh M, 'save', string filename)
```

Save the mesh object to an ascii file.

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

```
s = gf_mesh_get(mesh M, 'char')
```

Output a string description of the mesh.

```
gf_mesh_get(mesh M, 'export to vtk', 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 gf_mesh_fem_get(mesh_fem MF, 'export to vtk'), gf_slice_get(slice S, 'export to vtk').

```
gf_mesh_get(mesh M, 'export to dx', string filename, ...
[,'ascii'][,'append'][,'as',string name,[,'serie',string
serie_name]][,'edges'])
```

Exports a mesh to an OpenDX file.

See also gf_mesh_fem_get(mesh_fem MF, 'export to dx'), gf_slice_get(slice S, 'export to dx').

```
gf_mesh_get(mesh M, 'export to pos', string filename[, string name])
```

Exports a mesh to a POS file .

See also gf_mesh_fem_get(mesh_fem MF, 'export to pos'), gf_slice_get(slice S, 'export to pos').

`z = gf_mesh_get(mesh M, 'memsize')`

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

`gf_mesh_get(mesh M, 'display')`

displays a short summary for a mesh object.

gf_mesh_set

Synopsis

```
PIDs = gf_mesh_set(mesh M, 'pts', mat PTS)
PIDs = gf_mesh_set(mesh M, 'add point', mat PTS)
gf_mesh_set(mesh M, 'del point', ivec PIDs)
CVIDs = gf_mesh_set(mesh M, 'add convex', geotrans GT, mat PTS)
gf_mesh_set(mesh M, 'del convex', mat CVIDs)
gf_mesh_set(mesh M, 'del convex of dim', ivec DIMs)
gf_mesh_set(mesh M, 'translate', vec V)
gf_mesh_set(mesh M, 'transform', mat T)
gf_mesh_set(mesh M, 'boundary', int rnum, mat CVFIDs)
gf_mesh_set(mesh M, 'region', int rnum, mat CVFIDs)
gf_mesh_set(mesh M, 'extend region', int rnum, mat CVFIDs)
gf_mesh_set(mesh M, 'region intersect', int r1, int r2)
gf_mesh_set(mesh M, 'region merge', int r1, int r2)
gf_mesh_set(mesh M, 'region subtract', int r1, int r2)
gf_mesh_set(mesh M, 'delete boundary', int rnum, mat CVFIDs)
gf_mesh_set(mesh M, 'delete region', ivec RIDs)
gf_mesh_set(mesh M, 'merge', mesh m2[, scalar  tol])
gf_mesh_set(mesh M, 'optimize structure'[, int with_renumbering])
gf_mesh_set(mesh M, 'refine'[, ivec CVIDs])
```

Description :

General function for modification of a mesh object.

Command list :

`PIDs = gf_mesh_set(mesh M, 'pts', mat PTS)`

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

`PIDs = gf_mesh_set(mesh M, 'add point', mat 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.

`gf_mesh_set(mesh M, 'del point', ivec 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.

`CVIDs = gf_mesh_set(mesh M, 'add convex', geotrans GT, mat PTS)`

Add a new convex into the mesh.

The convex structure (triangle, prism,...) is given by *GT* (obtained with gf_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).

`gf_mesh_set(mesh M, 'del convex', mat 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.

`gf_mesh_set(mesh M, 'del convex of dim', ivec DIMs)`

Remove all convexes of dimension listed in *DIMs*.

For example; `gf_mesh_set(mesh M, 'del convex of dim', [1,2])` remove all line segments, triangles and quadrangles.

`gf_mesh_set(mesh M, 'translate', vec V)`

Translates each point of the mesh from *V*.

`gf_mesh_set(mesh M, 'transform', mat T)`

Applies the matrix *T* to each point of the mesh.

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

`gf_mesh_set(mesh M, 'boundary', int rnum, mat CVFIDs)`

DEPRECATED FUNCTION. Use 'region' instead.

`gf_mesh_set(mesh M, 'region', int rnum, mat 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 0 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.

`gf_mesh_set(mesh M, 'extend region', int rnum, mat 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 `gf_mesh_set(mesh M, 'set region)`.

`gf_mesh_set(mesh M, 'region intersect', int r1, int r2)`

Replace the region number *r1* with its intersection with region number *r2*.

`gf_mesh_set(mesh M, 'region merge', int r1, int r2)`

Merge region number *r2* into region number *r1*.

`gf_mesh_set(mesh M, 'region subtract', int r1, int r2)`

Replace the region number *r1* with its difference with region number *r2*.

`gf_mesh_set(mesh M, 'delete boundary', int rnum, mat CVFIDs)`

DEPRECATED FUNCTION. Use 'delete region' instead.

`gf_mesh_set(mesh M, 'delete region', ivec RIDs)`

Remove the regions whose #ids are listed in *RIDs*

`gf_mesh_set(mesh M, 'merge', mesh m2[, scalar tol])`

Merge with the mesh *m2*.

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

`gf_mesh_set(mesh M, 'optimize structure'[, int with_renumbering])`

Reset point and convex numbering.

After optimisation, the points (resp. convexes) will be consecutively numbered from 1 to gf_mesh_get(mesh M, 'max pid') (resp. gf_mesh_get(mesh M, 'max cvid')).

`gf_mesh_set(mesh M, 'refine'[, ivec CVIDs])`

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 mesh_fem and mesh_im objects linked to this mesh will be automagically refined.

gf_mesh_fem

Synopsis

```
MF = gf_mesh_fem(mesh m[, int Qdim1=1[, int Qdim2=1, ...]])
MF = gf_mesh_fem('load', string fname[, mesh m])
MF = gf_mesh_fem('from string', string s[, mesh m])
MF = gf_mesh_fem('clone', mesh_fem mf)
MF = gf_mesh_fem('sum', mesh_fem mf1, mesh_fem mf2[, mesh_fem mf3[, ...]])
MF = gf_mesh_fem('product', mesh_fem mf1, mesh_fem mf2)
MF = gf_mesh_fem('levelset', mesh_levelset mls, mesh_fem mf)
MF = gf_mesh_fem('global function', mesh m, levelset ls, {global_function GF1,...}[, int Qdim_m])
MF = gf_mesh_fem('partial', mesh_fem mf, ivec DOFs[, ivec RCVs])
```

Description :

General constructor for mesh_fem objects.

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

Command list :

`MF = gf_mesh_fem(mesh m[, int Qdim1=1[, int Qdim2=1, ...]])`

Build a new mesh_fem 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 = gf_mesh_fem('load', string fname[, mesh m])`

Load a mesh_fem 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 = gf_mesh_fem('from string', string s[, mesh m])
```

Create a mesh_fem object from its string description.

See also `gf_mesh_fem_get(mesh_fem MF, 'char')`

```
MF = gf_mesh_fem('clone', mesh_fem mf)
```

Create a copy of a mesh_fem.

```
MF = gf_mesh_fem('sum', mesh_fem mf1, mesh_fem mf2[, mesh_fem mf3[,
...]])
```

Create a mesh_fem that spans two (or more) mesh_fem's.

All mesh_fem must share the same mesh.

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

```
MF = gf_mesh_fem('product', mesh_fem mf1, mesh_fem mf2)
```

Create a mesh_fem 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 = gf_mesh_fem('levelset', mesh_levelset mls, mesh_fem mf)
```

Create a mesh_fem that is conformal to implicit surfaces defined in mesh_levelset.

```
MF = gf_mesh_fem('global function', mesh m, levelset ls,
{global_function GF1,...}[, int Qdim_m])
```

Create a mesh_fem 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 = gf_mesh_fem('partial', mesh_fem mf, ivec DOFs[, ivec RCVs])
```

Build a restricted mesh_fem 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*.

gf_mesh_fem_get

Synopsis

```
n = gf_mesh_fem_get(mesh_fem MF, 'nbdof')
n = gf_mesh_fem_get(mesh_fem MF, 'nb basic dof')
DOF = gf_mesh_fem_get(mesh_fem MF, 'dof from cv',mat CVids)
DOF = gf_mesh_fem_get(mesh_fem MF, 'basic dof from cv',mat CVids)
{DOFs, IDx} = gf_mesh_fem_get(mesh_fem MF, 'dof from cvid'[, mat CVids])
{DOFs, IDx} = gf_mesh_fem_get(mesh_fem MF, 'basic dof from cvid'[, mat CVids])
gf_mesh_fem_get(mesh_fem MF, 'non conformal dof'[, mat CVids])
gf_mesh_fem_get(mesh_fem MF, 'non conformal basic dof'[, mat CVids])
gf_mesh_fem_get(mesh_fem MF, 'qdim')
{FEMs, CV2F} = gf_mesh_fem_get(mesh_fem MF, 'fem'[, mat CVids])
CVs = gf_mesh_fem_get(mesh_fem MF, 'convex_index')
```

```
bB = gf_mesh_fem_get(mesh_fem MF, 'is_lagrangian'[, mat CVids])
bB = gf_mesh_fem_get(mesh_fem MF, 'is_equivalent'[, mat CVids])
bB = gf_mesh_fem_get(mesh_fem MF, 'is_polynomial'[, mat CVids])
bB = gf_mesh_fem_get(mesh_fem MF, 'is_reduced')
bB = gf_mesh_fem_get(mesh_fem MF, 'reduction matrix')
bB = gf_mesh_fem_get(mesh_fem MF, 'extension matrix')
Vr = gf_mesh_fem_get(mesh_fem MF, 'reduce vector', vec V)
Ve = gf_mesh_fem_get(mesh_fem MF, 'extend vector', vec V)
DOFs = gf_mesh_fem_get(mesh_fem MF, 'basic dof on region',mat Rs)
DOFs = gf_mesh_fem_get(mesh_fem MF, 'dof on region',mat Rs)
DOFpts = gf_mesh_fem_get(mesh_fem MF, 'dof nodes'[, mat DOFids])
DOFpts = gf_mesh_fem_get(mesh_fem MF, 'basic dof nodes'[, mat DOFids])
DOFP = gf_mesh_fem_get(mesh_fem MF, 'dof partition')
gf_mesh_fem_get(mesh_fem MF, 'save',string filename[, string opt])
gf_mesh_fem_get(mesh_fem MF, 'char'[, string opt])
gf_mesh_fem_get(mesh_fem MF, 'display')
m = gf_mesh_fem_get(mesh_fem MF, 'linked mesh')
m = gf_mesh_fem_get(mesh_fem MF, 'mesh')
gf_mesh_fem_get(mesh_fem MF, 'export to vtk',string filename, ... ['ascii'], U, 'name'...)
gf_mesh_fem_get(mesh_fem MF, 'export to dx',string filename, ...['as', string mesh_name][,'edges']['s
gf_mesh_fem_get(mesh_fem MF, 'export to pos',string filename[, string name][[,mesh_fem mf1], mat U1,
gf_mesh_fem_get(mesh_fem MF, 'dof_from_im',mesh_im mim[, int p])
U = gf_mesh_fem_get(mesh_fem MF, 'interpolate_convex_data',mat Ucv)
z = gf_mesh_fem_get(mesh_fem MF, 'memsize')
gf_mesh_fem_get(mesh_fem MF, 'has_linked_mesh_levelset')
gf_mesh_fem_get(mesh_fem MF, 'linked_mesh_levelset')
U = gf_mesh_fem_get(mesh_fem MF, 'eval', expr [, DOFLST])
```

Description :

General function for inquiry about mesh_fem objects.

Command list :

`n = gf_mesh_fem_get(mesh_fem MF, 'nbdof')`

Return the number of degrees of freedom (dof) of the mesh_fem.

`n = gf_mesh_fem_get(mesh_fem MF, 'nb basic dof')`

Return the number of basic degrees of freedom (dof) of the mesh_fem.

`DOF = gf_mesh_fem_get(mesh_fem MF, 'dof from cv',mat CVids)`

Deprecated function. Use gf_mesh_fem_get(mesh_fem MF, 'basic dof from cv') instead.

`DOF = gf_mesh_fem_get(mesh_fem MF, 'basic dof from cv',mat CVids)`

Return the dof of the convexes listed in *CVids*.

WARNING: the Degree 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).

`{DOFs, IDx} = gf_mesh_fem_get(mesh_fem MF, 'dof from cvid'[, mat CVids])`

Deprecated function. Use gf_mesh_fem_get(mesh_fem MF, 'basic dof from cvid') instead.

`{DOFs, IDx} = gf_mesh_fem_get(mesh_fem MF, 'basic dof from cvid'[, mat CVids])`

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 ... gf_mesh_get(mesh M, 'max cvid')*).

IDx is a row vector, *length(IDx) = length(CVids)+1*. *DOFs* is a row 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)-1).

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

`gf_mesh_fem_get(mesh_fem MF, 'non conformal dof'[, mat CVids])`

Deprecated function. Use gf_mesh_fem_get(mesh_fem MF, 'non conformal basic dof') instead.

`gf_mesh_fem_get(mesh_fem MF, 'non conformal basic dof'[, mat CVids])`

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).

`gf_mesh_fem_get(mesh_fem MF, 'qdim')`

Return the dimension Q of the field interpolated by the mesh_fem.

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

`{FEMs, CV2F} = gf_mesh_fem_get(mesh_fem MF, 'fem'[, mat CVids])`

Return a list of FEM used by the mesh_fem.

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.

Example:

```
cvid=gf_mesh_get(mf,'cvid');
[f,c2f]=gf_mesh_fem_get(mf, 'fem');
for i=1:size(f), sf{i}=gf_fem_get('char',f(i)); end;
for i=1:size(c2f),
   disp(sprintf('the fem of convex %d is %s',...
      cvid(i),sf{i}));
end;
```

`CVs = gf_mesh_fem_get(mesh_fem MF, 'convex_index')`

Return the list of convexes who have a FEM.

`bB = gf_mesh_fem_get(mesh_fem MF, 'is_lagrangian'[, mat CVids])`

Test if the mesh_fem 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.

```
bB = gf_mesh_fem_get(mesh_fem MF, 'is_equivalent'[, mat CVids])
```

Test if the mesh_fem is equivalent.

See gf_mesh_fem_get(mesh_fem MF, 'is_lagrangian')

```
bB = gf_mesh_fem_get(mesh_fem MF, 'is_polynomial'[, mat CVids])
```

Test if all base functions are polynomials.

See gf_mesh_fem_get(mesh_fem MF, 'is_lagrangian')

```
bB = gf_mesh_fem_get(mesh_fem MF, 'is_reduced')
```

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

```
bB = gf_mesh_fem_get(mesh_fem MF, 'reduction matrix')
```

Return the optional reduction matrix.

```
bB = gf_mesh_fem_get(mesh_fem MF, 'extension matrix')
```

Return the optional extension matrix.

```
Vr = gf_mesh_fem_get(mesh_fem MF, 'reduce vector', vec V)
```

Multiply the provided vector V with the extension matrix of the mesh_fem.

```
Ve = gf_mesh_fem_get(mesh_fem MF, 'extend vector', vec V)
```

Multiply the provided vector V with the reduction matrix of the mesh_fem.

```
DOFs = gf_mesh_fem_get(mesh_fem MF, 'basic dof on region',mat 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).

```
DOFs = gf_mesh_fem_get(mesh_fem MF, 'dof on region',mat 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 correspondance between basic and reduced dofs.

```
DOFpts = gf_mesh_fem_get(mesh_fem MF, 'dof nodes'[, mat DOFids])
```

Deprecated function. Use gf_mesh_fem_get(mesh_fem MF, 'basic dof nodes') instead.

```
DOFpts = gf_mesh_fem_get(mesh_fem MF, 'basic dof nodes'[, mat
DOFids])
```

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).

```
DOFP = gf_mesh_fem_get(mesh_fem MF, 'dof partition')
```

Get the 'dof_partition' array.

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

```
gf_mesh_fem_get(mesh_fem MF, 'save',string filename[, string opt])
```

Save a mesh_fem in a text file (and optionaly its linked mesh object if *opt* is the string 'with_mesh').

```
gf_mesh_fem_get(mesh_fem MF, 'char'[, string opt])
```

Output a string description of the mesh_fem.

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

```
gf_mesh_fem_get(mesh_fem MF, 'display')
```

displays a short summary for a mesh_fem object.

```
m = gf_mesh_fem_get(mesh_fem MF, 'linked mesh')
```

Return a reference to the mesh object linked to *mf*.

```
m = gf_mesh_fem_get(mesh_fem MF, 'mesh')
```

Return a reference to the mesh object linked to *mf*. (identical to gf_mesh_get(mesh M, 'linked mesh'))

```
gf_mesh_fem_get(mesh_fem MF, 'export to vtk',string filename, ...
['ascii'], U, 'name'...)
```

Export a mesh_fem 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 gf_slice_get(slice S, 'export to vtk').

```
gf_mesh_fem_get(mesh_fem MF, 'export to dx',string filename,
...['as', string mesh_name][,'edges']['serie',string
serie_name][,'ascii'][,'append'], U, 'name'...)
```

Export a mesh_fem and some fields to an OpenDX file.

This function will fail if the mesh_fem 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 gf_slice_get(slice S, 'export to dx').

`gf_mesh_fem_get(mesh_fem MF, 'export to pos',string filename[, string name][[,mesh_fem mf1], mat U1, string nameU1[[,mesh_fem mf2], mat U2, string nameU2,...]])`

Export a mesh_fem 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).

`gf_mesh_fem_get(mesh_fem MF, 'dof_from_im',mesh_im mim[, int p])`

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!

`U = gf_mesh_fem_get(mesh_fem MF, 'interpolate_convex_data',mat Ucv)`

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

The last dimension of the input vector Ucv should have gf_mesh_get(mesh M, 'max cvid') elements.

Example of use: gf_mesh_fem_get(mesh_fem MF, 'interpolate_convex_data', gf_mesh_get(mesh M, 'quality'))

`z = gf_mesh_fem_get(mesh_fem MF, '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.

`gf_mesh_fem_get(mesh_fem MF, 'has_linked_mesh_levelset')`

Is a mesh_fem_level_set or not.

`gf_mesh_fem_get(mesh_fem MF, 'linked_mesh_levelset')`

if it is a mesh_fem_level_set gives the linked mesh_level_set.

`U = gf_mesh_fem_get(mesh_fem MF, 'eval', expr [, DOFLST])`

Call gf_mesh_fem_get_eval. This function interpolates an expression on a lagrangian mesh_fem (for all dof except if DOFLST is specified). The expression can be a numeric constant, or a cell array containing numeric constants, string expressions or function handles. For example:

```
U1=gf_mesh_fem_get(mf,'eval',1)
U2=gf_mesh_fem_get(mf,'eval',[1;0]) % output has two rows
U3=gf_mesh_fem_get(mf,'eval',[1 0]) % output has one row, only valid if qdim(mf)==2
U4=gf_mesh_fem_get(mf,'eval',{'x';'y.*z';4;@myfunctionofxyz})
```

gf_mesh_fem_set

Synopsis

```
gf_mesh_fem_set(mesh_fem MF, 'fem', fem f[, ivec CVids])
gf_mesh_fem_set(mesh_fem MF, 'classical fem', int k[, ivec CVids])
gf_mesh_fem_set(mesh_fem MF, 'classical discontinuous fem', int K[, @tscalar alpha[, ivec CVIDX]])
gf_mesh_fem_set(mesh_fem MF, 'qdim', int Q)
gf_mesh_fem_set(mesh_fem MF, 'reduction matrices', mat R, mat E)
gf_mesh_fem_set(mesh_fem MF, 'reduction', int s)
gf_mesh_fem_set(mesh_fem MF, 'reduce meshfem', mat RM)
gf_mesh_fem_set(mesh_fem MF, 'dof partition', ivec DOFP)
gf_mesh_fem_set(mesh_fem MF, 'set partial', ivec DOFs[, ivec RCVs])
gf_mesh_fem_set(mesh_fem MF, 'adapt')
gf_mesh_fem_set(mesh_fem MF, 'set enriched dofs', ivec DOFs)
```

Description :

General function for modifying mesh_fem objects.

Command list :

`gf_mesh_fem_set(mesh_fem MF, 'fem', fem f[, ivec CVids])`

Set the Finite Element Method.

Assign a 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 gf_fem to obtain a list of available FEM methods.

`gf_mesh_fem_set(mesh_fem MF, 'classical fem', int k[, ivec CVids])`

Assign a classical (Lagrange polynomial) fem of order *k* to the mesh_fem.

Uses FEM_PK for simplexes, FEM_QK for parallelepipeds etc.

`gf_mesh_fem_set(mesh_fem MF, 'classical discontinuous fem', int K[, @tscalar alpha[, ivec CVIDX]])`

Assigns a classical (Lagrange polynomial) discontinuous fem or order K.

Similar to gf_mesh_fem_set(mesh_fem MF, '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.

`gf_mesh_fem_set(mesh_fem MF, 'qdim', int Q)`

Change the *Q* dimension of the field that is interpolated by the mesh_fem.

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

`gf_mesh_fem_set(mesh_fem MF, 'reduction matrices', mat R, mat E)`

Set the reduction and extension matrices and valid their use.

`gf_mesh_fem_set(mesh_fem MF, 'reduction', int s)`

Set or unset the use of the reduction/extension matrices.

`gf_mesh_fem_set(mesh_fem MF, 'reduce meshfem', mat 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 numer of columns of RM should corresponds to the number of degrees of fredoom of the finite element method.

`gf_mesh_fem_set(mesh_fem MF, 'dof partition', ivec DOFP)`

Change the 'dof_partition' array.

DOFP is a vector holding a integer value for each convex of the mesh_fem. See gf_mesh_fem_get(mesh_fem MF, 'dof partition') for a description of "dof partition".

`gf_mesh_fem_set(mesh_fem MF, 'set partial', ivec DOFs[, ivec RCVs])`

Can only be applied to a partial mesh_fem. 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*.

`gf_mesh_fem_set(mesh_fem MF, 'adapt')`

For a mesh_fem levelset object only. Adapt the mesh_fem object to a change of the levelset function.

`gf_mesh_fem_set(mesh_fem MF, 'set enriched dofs', ivec DOFs)`

For a mesh_fem product object only. Set te enriched dofs and adapt the mesh_fem product.

gf_mesh_im

Synopsis

```
MIM = gf_mesh_im('load', string fname[, mesh m])
MIM = gf_mesh_im('from string', string s[, mesh m])
MIM = gf_mesh_im('clone', mesh_im mim)
MIM = gf_mesh_im('levelset', mesh_levelset mls, string where, integ im[, integ im_tip[, integ im_set]
MIM = gf_mesh_im(mesh m, [{integ im|int im_degree}])
```

Description :

General constructor for mesh_im objects.

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

Command list :

`MIM = gf_mesh_im('load', string fname[, mesh m])`

Load a mesh_im 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 = gf_mesh_im('from string', string s[, mesh m])`

Create a mesh_im object from its string description.

See also `gf_mesh_im_get(mesh_im MI, 'char')`

`MIM = gf_mesh_im('clone', mesh_im mim)`

Create a copy of a mesh_im.

`MIM = gf_mesh_im('levelset', mesh_levelset mls, string where, integ im[, integ im_tip[, integ im_set]])`

Build an integration method conformal to a partition defined implicitely by a levelset.

The *where* argument define 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 levelset,

"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 the element cut by the levelset. For the 'ALL', 'INSIDE' and 'OUTSIDE' options it is mandatory to use the method `gf_mesh_im_set(mesh_im MI, 'integ')` to define the integration method on the remaining elements.

`MIM = gf_mesh_im(mesh m, [{integ im|int im_degree}])`

Build a new mesh_im object.

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

gf_mesh_im_get

Synopsis

```
{I, CV2I} = gf_mesh_im_get(mesh_im MI, 'integ'[, mat CVids])
CVids = gf_mesh_im_get(mesh_im MI, 'convex_index')
M = gf_mesh_im_get(mesh_im MI, 'eltm', eltm em, int cv [, int f])
Ip = gf_mesh_im_get(mesh_im MI, 'im_nodes'[, mat CVids])
gf_mesh_im_get(mesh_im MI, 'save',string filename[, 'with mesh'])
gf_mesh_im_get(mesh_im MI, 'char'[,'with mesh'])
gf_mesh_im_get(mesh_im MI, 'display')
m = gf_mesh_im_get(mesh_im MI, 'linked mesh')
z = gf_mesh_im_get(mesh_im MI, 'memsize')
```

Description :

General function extracting information from mesh_im objects.

Command list :

`{I, CV2I} = gf_mesh_im_get(mesh_im MI, 'integ'[, mat CVids])`

Return a list of integration methods used by the mesh_im.

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.

Example:

```
cvid=gf_mesh_get(mim,'cvid');
[f,c2f]=gf_mesh_im_get(mim, 'integ');
for i=1:size(f), sf{i}=gf_integ_get('char',f(i)); end;
for i=1:size(c2f),
  disp(sprintf('the integration of convex %d is %s',...
  cvid(i),sf{i}));
end;
```

`CVids = gf_mesh_im_get(mesh_im MI, 'convex_index')`

Return the list of convexes who have a integration method.

Convexes who have the dummy IM_NONE method are not listed.

`M = gf_mesh_im_get(mesh_im MI, 'eltm', eltm em, int cv [, int f])`

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.

`Ip = gf_mesh_im_get(mesh_im MI, 'im_nodes'[, mat CVids])`

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 gf_mesh_get(mesh M, '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!).

`gf_mesh_im_get(mesh_im MI, 'save',string filename[, 'with mesh'])`

Saves a mesh_im in a text file (and optionaly its linked mesh object).

`gf_mesh_im_get(mesh_im MI, 'char'[,'with mesh'])`

Output a string description of the mesh_im.

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

`gf_mesh_im_get(mesh_im MI, 'display')`

displays a short summary for a mesh_im object.

`m = gf_mesh_im_get(mesh_im MI, 'linked mesh')`

Returns a reference to the mesh object linked to *mim*.

`z = gf_mesh_im_get(mesh_im MI, 'memsize')`

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

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

gf_mesh_im_set

Synopsis

```
gf_mesh_im_set(mesh_im MI, 'integ',{integ im|int im_degree}[, ivec CVids])
gf_mesh_im_set(mesh_im MI, 'adapt')
```

Description :

General function for modifying mesh_im objects

Command list :

`gf_mesh_im_set(mesh_im MI, 'integ',{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 gf_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.)

`gf_mesh_im_set(mesh_im MI, 'adapt')`

For a mesh_im levelset object only. Adapt the integration methods to a change of the levelset function.

gf_mesh_im_data

Synopsis

```
MIMD = gf_mesh_im_data(mesh_im mim, int region, ivec size)
```

Description :

General constructor for mesh_im_data objects.

This object represents data defined on a mesh_im object.

Command list :

`MIMD = gf_mesh_im_data(mesh_im mim, int region, ivec size)`

Build a new mesh_imd object linked to a mesh_im 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.

gf_mesh_im_data_get

Synopsis

```
gf_mesh_im_data_get(mesh_im_data MID, 'region')
gf_mesh_im_data_get(mesh_im_data MID, 'nbpts')
gf_mesh_im_data_get(mesh_im_data MID, 'nb tensor elements')
gf_mesh_im_data_get(mesh_im_data MID, 'tensor size')
gf_mesh_im_data_get(mesh_im_data MID, 'display')
m = gf_mesh_im_data_get(mesh_im_data MID, 'linked mesh')
```

Description :

General function extracting information from mesh_im_data objects.

Command list :

`gf_mesh_im_data_get(mesh_im_data MID, 'region')`

Output the region that the mesh_imd is restricted to.

`gf_mesh_im_data_get(mesh_im_data MID, 'nbpts')`

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

`gf_mesh_im_data_get(mesh_im_data MID, 'nb tensor elements')`

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

`gf_mesh_im_data_get(mesh_im_data MID, 'tensor size')`

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

`gf_mesh_im_data_get(mesh_im_data MID, 'display')`

displays a short summary for a mesh_imd object.

`m = gf_mesh_im_data_get(mesh_im_data MID, 'linked mesh')`

Returns a reference to the mesh object linked to *mim*.

gf_mesh_im_data_set

Synopsis

```
gf_mesh_im_data_set(mesh_im_data MID, 'region', int rnum)
gf_mesh_im_data_set(mesh_im_data MID, 'tensor size',)
```

Description :

General function for modifying mesh_im objects

Command list :

`gf_mesh_im_data_set(mesh_im_data MID, 'region', int rnum)`

Set the considered region to *rnum*.

`gf_mesh_im_data_set(mesh_im_data MID, 'tensor size',)`

Set the size of the data per integration point.

gf_mesh_levelset

Synopsis

```
MLS = gf_mesh_levelset(mesh m)
```

Description :

General constructor for mesh_levelset objects.

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)).

Command list :

`MLS = gf_mesh_levelset(mesh m)`

Build a new mesh_levelset object from a mesh and returns its handle.

gf_mesh_levelset_get

Synopsis

```
M = gf_mesh_levelset_get(mesh_levelset MLS, 'cut_mesh')
LM = gf_mesh_levelset_get(mesh_levelset MLS, 'linked_mesh')
nbls = gf_mesh_levelset_get(mesh_levelset MLS, 'nb_ls')
LS = gf_mesh_levelset_get(mesh_levelset MLS, 'levelsets')
CVIDs = gf_mesh_levelset_get(mesh_levelset MLS, 'crack_tip_convexes')
SIZE = gf_mesh_levelset_get(mesh_levelset MLS, 'memsize')
s = gf_mesh_levelset_get(mesh_levelset MLS, 'char')
gf_mesh_levelset_get(mesh_levelset MLS, 'display')
```

Description :

General function for querying information about mesh_levelset objects.

Command list :

`M = gf_mesh_levelset_get(mesh_levelset MLS, 'cut_mesh')`

Return a mesh cut by the linked levelset's.

`LM = gf_mesh_levelset_get(mesh_levelset MLS, 'linked_mesh')`

Return a reference to the linked mesh.

`nbls = gf_mesh_levelset_get(mesh_levelset MLS, 'nb_ls')`

Return the number of linked levelset's.

`LS = gf_mesh_levelset_get(mesh_levelset MLS, 'levelsets')`

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

`CVIDs = gf_mesh_levelset_get(mesh_levelset MLS, 'crack_tip_convexes')`

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

`SIZE = gf_mesh_levelset_get(mesh_levelset MLS, 'memsize')`

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

`s = gf_mesh_levelset_get(mesh_levelset MLS, 'char')`

Output a (unique) string representation of the mesh_levelsetn.

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

`gf_mesh_levelset_get(mesh_levelset MLS, 'display')`

displays a short summary for a mesh_levelset object.

gf_mesh_levelset_set

Synopsis

```
gf_mesh_levelset_set(mesh_levelset MLS, 'add', levelset ls)
gf_mesh_levelset_set(mesh_levelset MLS, 'sup', levelset ls)
gf_mesh_levelset_set(mesh_levelset MLS, 'adapt')
```

Description :

General function for modification of mesh_levelset objects.

Command list :

`gf_mesh_levelset_set(mesh_levelset MLS, 'add', levelset 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.

`gf_mesh_levelset_set(mesh_levelset MLS, 'sup', levelset ls)`

Remove a link to the levelset *ls*.

`gf_mesh_levelset_set(mesh_levelset MLS, 'adapt')`

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

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

gf_mesher_object

Synopsis

```
MF = gf_mesher_object('ball', vec center, scalar radius)
MF = gf_mesher_object('half space', vec origin, vec normal_vector)
MF = gf_mesher_object('cylinder', vec origin, vec n, scalar length, scalar radius)
MF = gf_mesher_object('cone', vec origin, vec n, scalar length, scalar half_angle)
MF = gf_mesher_object('torus', scalar R, scalar r)
MF = gf_mesher_object('rectangle', vec rmin, vec rmax)
MF = gf_mesher_object('intersect', mesher_object object1 , mesher_object object2, ...)
MF = gf_mesher_object('union', mesher_object object1 , mesher_object object2, ...)
MF = gf_mesher_object('set minus', mesher_object object1 , mesher_object object2)
```

Description :

General constructor for mesher_object objects.

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

Command list :

`MF = gf_mesher_object('ball', vec center, scalar radius)`

Represents a ball of corresponding center and radius.

`MF = gf_mesher_object('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 = gf_mesher_object('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 = gf_mesher_object('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 = gf_mesher_object('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 = gf_mesher_object('rectangle', vec rmin, vec rmax)`

Represents a rectangle (or parallelepiped in 3D) parallel to the axes.

`MF = gf_mesher_object('intersect', mesher_object object1 , mesher_object object2, ...)`

Intersection of several objects.

`MF = gf_mesher_object('union', mesher_object object1 , mesher_object object2, ...)`

Union of several objects.

`MF = gf_mesher_object('set minus', mesher_object object1 , mesher_object object2)`

Geometric object being object1 minus object2.

gf_mesher_object_get

Synopsis

```
s = gf_mesher_object_get(mesher_object MO, 'char')
gf_mesher_object_get(mesher_object MO, 'display')
```

Description :

General function for querying information about mesher_object objects.

Command list :

`s = gf_mesher_object_get(mesher_object MO, 'char')`

Output a (unique) string representation of the mesher_object.

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

`gf_mesher_object_get(mesher_object MO, 'display')`

displays a short summary for a mesher_object object.

gf_model

Synopsis

```
MD = gf_model('real')
MD = gf_model('complex')
```

Description :

General constructor for model objects.

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.

Command list :

`MD = gf_model('real')`

Build a model for real unknowns.

`MD = gf_model('complex')`

Build a model for complex unknowns.

gf_model_get

Synopsis

```
b = gf_model_get(model M, 'is_complex')
T = gf_model_get(model M, 'nbdof')
dt = gf_model_get(model M, 'get time step')
t = gf_model_get(model M, 'get time')
T = gf_model_get(model M, 'tangent_matrix')
gf_model_get(model M, 'rhs')
gf_model_get(model M, 'brick term rhs', int ind_brick[, int ind_term, int sym, int ind_iter])
z = gf_model_get(model M, 'memsize')
gf_model_get(model M, 'variable list')
gf_model_get(model M, 'brick list')
gf_model_get(model M, 'list residuals')
V = gf_model_get(model M, 'variable', string name)
V = gf_model_get(model M, 'interpolation', string expr, {mesh_fem mf | mesh_imd mimd | vec pts,  mesh
V = gf_model_get(model M, 'local_projection', mesh_im mim, string expr, mesh_fem mf[, int region])
mf = gf_model_get(model M, 'mesh fem of variable', string name)
name = gf_model_get(model M, 'mult varname Dirichlet', int ind_brick)
I = gf_model_get(model M, 'interval of variable', string varname)
V = gf_model_get(model M, 'from variables')
gf_model_get(model M, 'assembly'[, string option])
{nbit, converged} = gf_model_get(model M, 'solve'[, ...])
gf_model_get(model M, 'test tangent matrix'[, scalar EPS[, int NB[, scalar scale]]])
gf_model_get(model M, 'test tangent matrix term', string varname1, string varname2[, scalar EPS[, int
expr = gf_model_get(model M, 'Neumann term', string varname, int region)
V = gf_model_get(model M, 'compute isotropic linearized Von Mises or Tresca', string varname, string
V = gf_model_get(model M, 'compute isotropic linearized Von Mises pstrain', string varname, string da
V = gf_model_get(model M, 'compute isotropic linearized Von Mises pstress', string varname, string da
```

```
V = gf_model_get(model M, 'compute Von Mises or Tresca', string varname, string lawname, string datar
V = gf_model_get(model M, 'compute finite strain elasticity Von Mises',  string lawname, string varna
V = gf_model_get(model M, 'compute second Piola Kirchhoff tensor', string varname, string lawname, st
gf_model_get(model M, 'elastoplasticity next iter', mesh_im mim, string varname, string previous_dep_
gf_model_get(model M, 'small strain elastoplasticity next iter', mesh_im mim,  string lawname, string
V = gf_model_get(model M, 'small strain elastoplasticity Von Mises', mesh_im mim, mesh_fem mf_vm, str
V = gf_model_get(model M, 'compute elastoplasticity Von Mises or Tresca', string datasigma, mesh_fem
V = gf_model_get(model M, 'compute plastic part', mesh_im mim, mesh_fem mf_pl, string varname, string
gf_model_get(model M, 'finite strain elastoplasticity next iter', mesh_im mim, string lawname, string
V = gf_model_get(model M, 'compute finite strain elastoplasticity Von Mises', mesh_im mim, mesh_fem m
V = gf_model_get(model M, 'sliding data group name of large sliding contact brick', int indbrick)
V = gf_model_get(model M, 'displacement group name of large sliding contact brick', int indbrick)
V = gf_model_get(model M, 'transformation name of large sliding contact brick', int indbrick)
V = gf_model_get(model M, 'sliding data group name of Nitsche large sliding contact brick', int indbr
V = gf_model_get(model M, 'displacement group name of Nitsche large sliding contact brick', int indbr
V = gf_model_get(model M, 'transformation name of Nitsche large sliding contact brick', int indbrick)
M = gf_model_get(model M, 'matrix term', int ind_brick, int ind_term)
s = gf_model_get(model M, 'char')
gf_model_get(model M, 'display')
```

Description :

Get information from a model object.

Command list :

`b = gf_model_get(model M, 'is_complex')`

Return 0 is the model is real, 1 if it is complex.

`T = gf_model_get(model M, 'nbdof')`

Return the total number of degrees of freedom of the model.

`dt = gf_model_get(model M, 'get time step')`

Gives the value of the time step.

`t = gf_model_get(model M, 'get time')`

Give the value of the data *t* corresponding to the current time.

`T = gf_model_get(model M, 'tangent_matrix')`

Return the tangent matrix stored in the model .

`gf_model_get(model M, 'rhs')`

Return the right hand side of the tangent problem.

`gf_model_get(model M, 'brick term rhs', int ind_brick[, int ind_term, int sym, int ind_iter])`

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 : 1). *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 1).

`z = gf_model_get(model M, 'memsize')`

Return a rough approximation of the amount of memory (in bytes) used by the model.

`gf_model_get(model M, 'variable list')`

print to the output the list of variables and constants of the model.

```
gf_model_get(model M, 'brick list')
```

print to the output the list of bricks of the model.

```
gf_model_get(model M, 'list residuals')
```

print to the output the residuals corresponding to all terms included in the model.

```
V = gf_model_get(model M, 'variable', string name)
```

Gives the value of a variable or data.

```
V = gf_model_get(model M, 'interpolation', string expr, {mesh_fem mf
| mesh_imd 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*.

```
V = gf_model_get(model M, 'local_projection', mesh_im mim, string
expr, mesh_fem mf[, int region])
```

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.

```
mf = gf_model_get(model M, 'mesh fem of variable', string name)
```

Gives access to the *mesh_fem* of a variable or data.

```
name = gf_model_get(model M, 'mult varname Dirichlet', int 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

```
I = gf_model_get(model M, 'interval of variable', string varname)
```

Gives the interval of the variable *varname* in the linear system of the model.

```
V = gf_model_get(model M, 'from variables')
```

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).

```
gf_model_get(model M, 'assembly'[, string option])
```

Assembly of the tangent system taking into account the terms from all bricks. *option*, if specified, should be 'build_all', 'build_rhs', 'build_matrix'. 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.

```
{nbit, converged} = gf_model_get(model M, 'solve'[, ...])
```

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 gf_model_get(model M, '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.

Note that it is possible to disable some variables (see gf_model_set(model M, '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.

```
gf_model_get(model M, 'test tangent matrix'[, 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].

```
gf_model_get(model M, 'test tangent matrix term', 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].

```
expr = gf_model_get(model M, 'Neumann term', string varname, int
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.

```
V = gf_model_get(model M, 'compute isotropic linearized Von Mises or
Tresca', string varname, string dataname_lambda, string dataname_mu,
mesh_fem mf_vm[, string version])
```

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.

```
V = gf_model_get(model M, 'compute isotropic linearized Von Mises
pstrain', string varname, string data_E, string data_nu, mesh_fem
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 modulus and Poisson ratio.

```
V = gf_model_get(model M, 'compute isotropic linearized Von Mises
pstress', string varname, string data_E, string data_nu, mesh_fem
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 modulus and Poisson ratio.

```
V = gf_model_get(model M, 'compute Von Mises or Tresca', string
varname, string lawname, string dataname, mesh_fem mf_vm[, string
version])
```

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).

```
V = gf_model_get(model M, 'compute finite strain elasticity Von
Mises', string lawname, string varname, string params, mesh_fem
mf_vm[, int region])
```

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.

```
V = gf_model_get(model M, 'compute second Piola Kirchhoff tensor',
string varname, string lawname, string dataname, mesh_fem 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.

```
gf_model_get(model M, 'elastoplasticity next iter', mesh_im mim,
string varname, string previous_dep_name, string projname, string
datalambda, string datamu, string datathreshold, string 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.

```
gf_model_get(model M, 'small strain elastoplasticity next iter',
mesh_im 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*.

```
V = gf_model_get(model M, 'small strain elastoplasticity Von Mises',
mesh_im mim, mesh_fem 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.

```
V = gf_model_get(model M, 'compute elastoplasticity Von Mises or
Tresca', string datasigma, mesh_fem mf_vm[, string version])
```

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).

```
V = gf_model_get(model M, 'compute plastic part', mesh_im mim,
mesh_fem mf_pl, string varname, string previous_dep_name, string
projname, string datalambda, string datamu, string datathreshold,
string 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.

```
gf_model_get(model M, 'finite strain elastoplasticity next iter',
mesh_im 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*.

```
V = gf_model_get(model M, 'compute finite strain elastoplasticity
Von Mises', mesh_im mim, mesh_fem 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'.

`V = gf_model_get(model M, 'sliding data group name of large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`V = gf_model_get(model M, 'displacement group name of large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`V = gf_model_get(model M, 'transformation name of large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`V = gf_model_get(model M, 'sliding data group name of Nitsche large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`V = gf_model_get(model M, 'displacement group name of Nitsche large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`V = gf_model_get(model M, 'transformation name of Nitsche large sliding contact brick', int indbrick)`

Gives the name of the group of variables corresponding to the sliding data for an existing large sliding contact brick.

`M = gf_model_get(model M, 'matrix term', int ind_brick, int ind_term)`

Gives the matrix term ind_term of the brick ind_brick if it exists

`s = gf_model_get(model M, '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.

`gf_model_get(model M, 'display')`

displays a short summary for a model object.

gf_model_set

Synopsis

```
gf_model_set(model M, 'clear')
gf_model_set(model M, 'add fem variable', string name, mesh_fem mf)
gf_model_set(model M, 'add filtered fem variable', string name, mesh_fem mf, int region)
gf_model_set(model M, 'add variable', string name, sizes)
gf_model_set(model M, 'delete variable', string name)
gf_model_set(model M, 'resize variable', string name, sizes)
gf_model_set(model M, 'add multiplier', string name, mesh_fem mf, string primalname[, mesh_im mim, in
```

```
gf_model_set(model M, 'add im data', string name, mesh_imd mimd)
gf_model_set(model M, 'add fem data', string name, mesh_fem mf[, sizes])
gf_model_set(model M, 'add initialized fem data', string name, mesh_fem mf, vec V[, sizes])
gf_model_set(model M, 'add data', string name, int size)
gf_model_set(model M, 'add macro', string name, string expr)
gf_model_set(model M, 'del macro', string name)
gf_model_set(model M, 'add initialized data', string name, vec V[, sizes])
gf_model_set(model M, 'variable', string name, vec V)
gf_model_set(model M, 'to variables', vec V)
gf_model_set(model M, 'delete brick', int ind_brick)
gf_model_set(model M, 'define variable group', string name[, string varname, ...])
gf_model_set(model M, 'add elementary rotated RT0 projection', string transname)
gf_model_set(model M, 'add interpolate transformation from expression', string transname, mesh source
gf_model_set(model M, 'add element extrapolation transformation', string transname, mesh source_mesh,
gf_model_set(model M, 'set element extrapolation correspondance', string transname, mat elt_corr)
gf_model_set(model M, 'add raytracing transformation', string transname, scalar release_distance)
gf_model_set(model M, 'add master contact boundary to raytracing transformation', string transname, r
gf_model_set(model M, 'add slave contact boundary to raytracing transformation', string transname, me
gf_model_set(model M, 'add rigid obstacle to raytracing transformation', string transname, string exp
gf_model_set(model M, 'add projection transformation', string transname, scalar release_distance)
gf_model_set(model M, 'add master contact boundary to projection transformation', string transname, r
gf_model_set(model M, 'add slave contact boundary to projection transformation', string transname, me
gf_model_set(model M, 'add rigid obstacle to projection transformation', string transname, string exp
ind = gf_model_set(model M, 'add linear generic assembly brick', mesh_im mim, string expression[, int
ind = gf_model_set(model M, 'add nonlinear generic assembly brick', mesh_im mim, string expression[,
ind = gf_model_set(model M, 'add source term generic assembly brick', mesh_im mim, string expression
gf_model_set(model M, 'add assembly assignment', string dataname, string expression[, int region[, in
gf_model_set(model M, 'clear assembly assignment')
ind = gf_model_set(model M, 'add Laplacian brick', mesh_im mim, string varname[, int region])
ind = gf_model_set(model M, 'add generic elliptic brick', mesh_im mim, string varname, string datanam
ind = gf_model_set(model M, 'add source term brick', mesh_im mim, string varname, string dataexpr[, i
ind = gf_model_set(model M, 'add normal source term brick', mesh_im mim, string varname, string datan
ind = gf_model_set(model M, 'add Dirichlet condition with simplification', string varname, int regio
ind = gf_model_set(model M, 'add Dirichlet condition with multipliers', mesh_im mim, string varname,
ind = gf_model_set(model M, 'add Dirichlet condition with Nitsche method', mesh_im mim, string varna
ind = gf_model_set(model M, 'add Dirichlet condition with penalization', mesh_im mim, string varname,
ind = gf_model_set(model M, 'add normal Dirichlet condition with multipliers', mesh_im mim, string va
ind = gf_model_set(model M, 'add normal Dirichlet condition with penalization', mesh_im mim, string v
ind = gf_model_set(model M, 'add normal Dirichlet condition with Nitsche method', mesh_im mim, string
ind = gf_model_set(model M, 'add generalized Dirichlet condition with multipliers', mesh_im mim, stri
ind = gf_model_set(model M, 'add generalized Dirichlet condition with penalization', mesh_im mim, str
ind = gf_model_set(model M, 'add generalized Dirichlet condition with Nitsche method', mesh_im mim, s
ind = gf_model_set(model M, 'add pointwise constraints with multipliers', string varname, string data
ind = gf_model_set(model M, 'add pointwise constraints with given multipliers', string varname, strin
ind = gf_model_set(model M, 'add pointwise constraints with penalization', string varname, scalar coe
gf_model_set(model M, 'change penalization coeff', int ind_brick, scalar coeff)
ind = gf_model_set(model M, 'add Helmholtz brick', mesh_im mim, string varname, string dataexpr[, int
ind = gf_model_set(model M, 'add Fourier Robin brick', mesh_im mim, string varname, string dataexpr,
ind = gf_model_set(model M, 'add constraint with multipliers', string varname, string multname, spmat
ind = gf_model_set(model M, 'add constraint with penalization', string varname, scalar coeff, spmat B
ind = gf_model_set(model M, 'add explicit matrix', string varname1, string varname2, spmat B[, int is
ind = gf_model_set(model M, 'add explicit rhs', string varname, vec L)
gf_model_set(model M, 'set private matrix', int indbrick, spmat B)
gf_model_set(model M, 'set private rhs', int indbrick, vec B)
ind = gf_model_set(model M, 'add isotropic linearized elasticity brick', mesh_im mim, string varname,
ind = gf_model_set(model M, 'add isotropic linearized elasticity brick pstrain', mesh_im mim, string
ind = gf_model_set(model M, 'add isotropic linearized elasticity brick pstress', mesh_im mim, string
ind = gf_model_set(model M, 'add linear incompressibility brick', mesh_im mim, string varname, string
```

```
ind = gf_model_set(model M, 'add nonlinear elasticity brick', mesh_im mim, string varname, string cor
ind = gf_model_set(model M, 'add finite strain elasticity brick', mesh_im mim, string constitutive_la
ind = gf_model_set(model M, 'add small strain elastoplasticity brick', mesh_im mim,  string lawname,
ind = gf_model_set(model M, 'add elastoplasticity brick', mesh_im mim ,string projname, string varnar
ind = gf_model_set(model M, 'add finite strain elastoplasticity brick', mesh_im mim , string lawname,
ind = gf_model_set(model M, 'add nonlinear incompressibility brick', mesh_im mim, string varname, str
ind = gf_model_set(model M, 'add finite strain incompressibility brick', mesh_im mim, string varname,
ind = gf_model_set(model M, 'add bilaplacian brick', mesh_im mim, string varname, string dataname [,
ind = gf_model_set(model M, 'add Kirchhoff-Love plate brick', mesh_im mim, string varname, string dat
ind = gf_model_set(model M, 'add normal derivative source term brick', mesh_im mim, string varname, s
ind = gf_model_set(model M, 'add Kirchhoff-Love Neumann term brick', mesh_im mim, string varname, str
ind = gf_model_set(model M, 'add normal derivative Dirichlet condition with multipliers', mesh_im mir
ind = gf_model_set(model M, 'add normal derivative Dirichlet condition with penalization', mesh_im mi
ind = gf_model_set(model M, 'add Mindlin Reissner plate brick', mesh_im mim, mesh_im mim_reduced, str
ind = gf_model_set(model M, 'add mass brick', mesh_im mim, string varname[, string dataexpr_rho[, int
gf_model_set(model M, 'shift variables for time integration')
gf_model_set(model M, 'perform init time derivative', scalar ddt)
gf_model_set(model M, 'set time step', scalar dt)
gf_model_set(model M, 'set time', scalar t)
gf_model_set(model M, 'add theta method for first order', string varname, scalar theta)
gf_model_set(model M, 'add theta method for second order', string varname, scalar theta)
gf_model_set(model M, 'add Newmark scheme', string varname, scalar beta, scalar gamma)
gf_model_set(model M, 'disable bricks', ivec bricks_indices)
gf_model_set(model M, 'enable bricks', ivec bricks_indices)
gf_model_set(model M, 'disable variable', string varname)
gf_model_set(model M, 'enable variable', string varname)
gf_model_set(model M, 'first iter')
gf_model_set(model M, 'next iter')
ind = gf_model_set(model M, 'add basic contact brick', string varname_u, string multname_n[, string m
ind = gf_model_set(model M, 'add basic contact brick two deformable bodies', string varname_u1, strin
gf_model_set(model M, 'contact brick set BN', int indbrick, spmat BN)
gf_model_set(model M, 'contact brick set BT', int indbrick, spmat BT)
ind = gf_model_set(model M, 'add nodal contact with rigid obstacle brick',  mesh_im mim, string varna
ind = gf_model_set(model M, 'add contact with rigid obstacle brick',  mesh_im mim, string varname_u,
ind = gf_model_set(model M, 'add integral contact with rigid obstacle brick',  mesh_im mim, string va
ind = gf_model_set(model M, 'add penalized contact with rigid obstacle brick',  mesh_im mim, string v
ind = gf_model_set(model M, 'add Nitsche contact with rigid obstacle brick', mesh_im mim, string varn
ind = gf_model_set(model M, 'add Nitsche midpoint contact with rigid obstacle brick', mesh_im mim, st
ind = gf_model_set(model M, 'add Nitsche fictitious domain contact brick', mesh_im mim, string varnar
ind = gf_model_set(model M, 'add nodal contact between nonmatching meshes brick',  mesh_im mim1[, mes
ind = gf_model_set(model M, 'add nonmatching meshes contact brick',  mesh_im mim1[, mesh_im mim2], st
ind = gf_model_set(model M, 'add integral contact between nonmatching meshes brick',  mesh_im mim, st
ind = gf_model_set(model M, 'add penalized contact between nonmatching meshes brick',  mesh_im mim, s
ind = gf_model_set(model M, 'add integral large sliding contact brick raytracing', string dataname_r,
gf_model_set(model M, 'add rigid obstacle to large sliding contact brick', int indbrick, string expr,
gf_model_set(model M, 'add master contact boundary to large sliding contact brick', int indbrick, mes
gf_model_set(model M, 'add slave contact boundary to large sliding contact brick', int indbrick, mesh
gf_model_set(model M, 'add master slave contact boundary to large sliding contact brick', int indbric
ind = gf_model_set(model M, 'add Nitsche large sliding contact brick raytracing', bool unbiased_versi
gf_model_set(model M, 'add rigid obstacle to Nitsche large sliding contact brick', int indbrick, stri
gf_model_set(model M, 'add master contact boundary to biased Nitsche large sliding contact brick', in
gf_model_set(model M, 'add slave contact boundary to biased Nitsche large sliding contact brick', int
gf_model_set(model M, 'add contact boundary to unbiased Nitsche large sliding contact brick', int ind
```

Description :

Modifies a model object.

Command list :

`gf_model_set(model M, 'clear')`

Clear the model.

`gf_model_set(model M, 'add fem variable', string name, mesh_fem mf)`

Add a variable to the model linked to a mesh_fem. *name* is the variable name.

`gf_model_set(model M, 'add filtered fem variable', string name, mesh_fem mf, int region)`

Add a variable to the model linked to a mesh_fem. The variable is filtered in the sense that only the dof on the region are considered. *name* is the variable name.

`gf_model_set(model M, 'add variable', string 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.

`gf_model_set(model M, 'delete variable', string name)`

Delete a variable or a data from the model.

`gf_model_set(model M, 'resize variable', string 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.

`gf_model_set(model M, 'add multiplier', string name, mesh_fem mf, string primalname[, mesh_im mim, int region])`

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.

`gf_model_set(model M, 'add im data', string name, mesh_imd mimd)`

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

`gf_model_set(model M, 'add fem data', string name, mesh_fem mf[, sizes])`

Add a data to the model linked to a mesh_fem. *name* is the data name, *sizes* an optional parameter which is either an integer or a vector of suplementary dimensions with respect to *mf*.

`gf_model_set(model M, 'add initialized fem data', string name, mesh_fem mf, vec V[, sizes])`

Add a data to the model linked to a mesh_fem. *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*.

`gf_model_set(model M, 'add data', string name, int 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.

`gf_model_set(model M, 'add macro', string name, string 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. Teh 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'.

```
gf_model_set(model M, 'del macro', string name)
```

Delete a previously defined macro for the high generic assembly language.

```
gf_model_set(model M, 'add initialized data', string name, vec V[,
sizes])
```

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.

```
gf_model_set(model M, 'variable', string name, vec V)
```

Set the value of a variable or data. *name* is the data name.

```
gf_model_set(model M, 'to variables', vec V)
```

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).

```
gf_model_set(model M, 'delete brick', int ind_brick)
```

Delete a variable or a data from the model.

```
gf_model_set(model M, 'define variable group', string name[, string
varname, ...])
```

Defines a group of variables for the interpolation (mainly for the raytracing interpolation transformation.

```
gf_model_set(model M, 'add elementary rotated RT0 projection', string
transname)
```

Experimental method ...

```
gf_model_set(model M, 'add interpolate transformation from
expression', string transname, mesh source_mesh, mesh target_mesh,
string 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.

```
gf_model_set(model M, 'add element extrapolation transformation',
string transname, mesh source_mesh, mat 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.

```
gf_model_set(model M, 'set element extrapolation correspondance',
string transname, mat elt_corr)
```

Change the correspondance map of an element extrapolation interpolate transformation.

```
gf_model_set(model M, 'add raytracing transformation', string
transname, scalar 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.

```
gf_model_set(model M, 'add master contact boundary to raytracing
transformation', string transname, mesh m, string dispname, int
region)
```

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

```
gf_model_set(model M, 'add slave contact boundary to raytracing
transformation', string transname, mesh m, string dispname, int
region)
```

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

```
gf_model_set(model M, 'add rigid obstacle to raytracing
transformation', string transname, string expr, int 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*.

```
gf_model_set(model M, 'add projection transformation', string
transname, scalar 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.

```
gf_model_set(model M, 'add master contact boundary to projection
transformation', string transname, mesh m, string dispname, int
region)
```

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

```
gf_model_set(model M, 'add slave contact boundary to projection
transformation', string transname, mesh m, string dispname, int
region)
```

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

```
gf_model_set(model M, 'add rigid obstacle to projection
transformation', string transname, string expr, int 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*.

```
ind = gf_model_set(model M, 'add linear generic assembly brick',
mesh_im 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 otpional name for the brick.

```
ind = gf_model_set(model M, 'add nonlinear generic assembly brick',
mesh_im 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 otpional name for the brick.

```
ind = gf_model_set(model M, 'add source term generic assembly brick',
mesh_im mim, string expression[, int region])
```

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 otpional name for the brick.

```
gf_model_set(model M, 'add assembly assignment', 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.

```
gf_model_set(model M, 'clear assembly assignment')
```

Delete all added assembly assignments

```
ind = gf_model_set(model M, 'add Laplacian brick', mesh_im mim,
string varname[, int region])
```

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.

```
ind = gf_model_set(model M, 'add generic elliptic brick', mesh_im
mim, string varname, string dataname[, int region])
```

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 describbed 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.

```
ind = gf_model_set(model M, 'add source term brick', mesh_im
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.

```
ind = gf_model_set(model M, 'add normal source term brick', mesh_im
mim, string varname, string dataname, int 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.

```
ind = gf_model_set(model M, 'add Dirichlet condition with
simplification', string varname, int region[, string dataname])
```

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 correspodning 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.

```
ind = gf_model_set(model M, 'add Dirichlet condition with
multipliers', mesh_im mim, string varname, mult_description, int
region[, string dataname])
```

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.

```
ind = gf_model_set(model M, 'add Dirichlet condition with Nitsche
method', mesh_im 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 gf_model_get(model M, '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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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.

```
ind = gf_model_set(model M, 'add Dirichlet condition with
penalization', mesh_im mim, string varname, scalar coeff, int
region[, string dataname, mesh_fem mf_mult])
```

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.

```
ind = gf_model_set(model M, 'add normal Dirichlet condition with
multipliers', mesh_im mim, string varname, mult_description, int
region[, 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. 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.

```
ind = gf_model_set(model M, 'add normal Dirichlet condition with
penalization', mesh_im mim, string varname, scalar coeff, int
region[, string dataname, mesh_fem mf_mult])
```

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.

```
ind = gf_model_set(model M, 'add normal Dirichlet condition with
Nitsche method', mesh_im 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 gf_model_get(model M, '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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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)

```
ind = gf_model_set(model M, 'add generalized Dirichlet condition
with multipliers', mesh_im mim, string varname, mult_description, int
region, string dataname, string 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.

```
ind = gf_model_set(model M, 'add generalized Dirichlet condition with
penalization', mesh_im mim, string varname, scalar coeff, int region,
string dataname, string Hname[, mesh_fem mf_mult])
```

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.

```
ind = gf_model_set(model M, 'add generalized Dirichlet condition with
Nitsche method', mesh_im mim, string varname, string Neumannterm,
string gamma0name, int region[, scalar theta], string dataname,
string Hname)
```

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 gf_model_get(model M, '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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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)

```
ind = gf_model_set(model M, 'add pointwise constraints with
multipliers', 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.

```
ind = gf_model_set(model M, 'add pointwise constraints with given
multipliers', 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.

```
ind = gf_model_set(model M, 'add pointwise constraints with
penalization', 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.

```
gf_model_set(model M, 'change penalization coeff', int ind_brick,
scalar 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.

```
ind = gf_model_set(model M, 'add Helmholtz brick', mesh_im mim,
string varname, string dataexpr[, int region])
```

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.

```
ind = gf_model_set(model M, 'add Fourier Robin brick', mesh_im mim,
string varname, string dataexpr, int 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.

```
ind = gf_model_set(model M, 'add constraint with multipliers', 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 gf_model_set(model M, 'set private matrix') and gf_model_set(model M, '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.

```
ind = gf_model_set(model M, 'add constraint with penalization',
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 palin row, otherwise the whole tangent matrix will be plain. It is possible to change the constraint at any time with the methods gf_model_set(model M, 'set private matrix') and gf_model_set(model M, 'set private rhs'). The method gf_model_set(model M, '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.

```
ind = gf_model_set(model M, 'add explicit matrix', 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 gf_model_set(model M, 'set private matrix'). Return the brick index in the model.

```
ind = gf_model_set(model M, 'add explicit rhs', string varname, vec
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 gf_model_set(model M, '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.

```
gf_model_set(model M, 'set private matrix', int indbrick, spmat B)
```

For some specific bricks having an internal sparse matrix (explicit bricks: 'constraint brick' and 'explicit matrix brick'), set this matrix.

```
gf_model_set(model M, 'set private rhs', int indbrick, vec B)
```

For some specific bricks having an internal right hand side vector (explicit bricks: 'constraint brick' and 'explicit rhs brick'), set this rhs.

```
ind = gf_model_set(model M, 'add isotropic linearized elasticity
brick', mesh_im mim, string varname, string dataname_lambda, string
dataname_mu[, int region])
```

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.

```
ind = gf_model_set(model M, 'add isotropic linearized elasticity
brick pstrain', mesh_im mim, string varname, string data_E, string
data_nu[, int region])
```

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.

```
ind = gf_model_set(model M, 'add isotropic linearized elasticity
brick pstress', mesh_im mim, string varname, string data_E, string
data_nu[, int region])
```

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.

```
ind = gf_model_set(model M, 'add linear incompressibility brick',
mesh_im 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.

```
ind = gf_model_set(model M, 'add nonlinear elasticity brick', mesh_im
mim, string varname, string constitutive_law, string dataname[, int
region])
```

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.

```
ind = gf_model_set(model M, 'add finite strain elasticity brick',
mesh_im mim, string constitutive_law, string varname, string params[,
int region])
```

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.

```
ind = gf_model_set(model M, 'add small strain elastoplasticity
brick', mesh_im 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 gf_model_get(model M, '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 gf_model_get(model M, '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).

```
ind = gf_model_set(model M, 'add elastoplasticity brick', mesh_im mim
,string projname, string varname, string previous_dep_name, string
datalambda, string datamu, string datathreshold, string datasigma[,
int region])
```

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.

```
ind = gf_model_set(model M, 'add finite strain elastoplasticity
brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add nonlinear incompressibility brick',
mesh_im mim, string varname, string multname_pressure[, int region])
```

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.

```
ind = gf_model_set(model M, 'add finite strain incompressibility
brick', mesh_im mim, string varname, string multname_pressure[, int
region])
```

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.

```
ind = gf_model_set(model M, 'add bilaplacian brick', mesh_im mim,
string varname, string dataname [, int region])
```

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.

```
ind = gf_model_set(model M, 'add Kirchhoff-Love plate brick', mesh_im
mim, string varname, string dataname_D, string dataname_nu [, int
region])
```

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.

```
ind = gf_model_set(model M, 'add normal derivative source term
brick', mesh_im mim, string varname, string dataname, int 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.

```
ind = gf_model_set(model M, 'add Kirchhoff-Love Neumann term brick',
mesh_im mim, string varname, string dataname_M, string dataname_divM,
int 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.

```
ind = gf_model_set(model M, 'add normal derivative Dirichlet
condition with multipliers', mesh_im mim, string varname,
mult_description, int region [, string dataname, int
R_must_be_derivated])
```

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.

```
ind = gf_model_set(model M, 'add normal derivative Dirichlet
condition with penalization', mesh_im mim, string varname, scalar
coeff, int region [, string dataname, int R_must_be_derivated])
```

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 gf_model_set(model M, '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 considered. Return the brick index in the model.

```
ind = gf_model_set(model M, 'add Mindlin Reissner plate
brick', mesh_im mim, mesh_im 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.

```
ind = gf_model_set(model M, 'add mass brick', mesh_im 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.

```
gf_model_set(model M, '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.

```
gf_model_set(model M, 'perform init time derivative', scalar 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.

```
gf_model_set(model M, 'set time step', scalar 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).

```
gf_model_set(model M, 'set time', scalar t)
```

Set the value of the data *t* corresponding to the current time to *t*.

```
gf_model_set(model M, 'add theta method for first order', string
varname, scalar 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.

```
gf_model_set(model M, 'add theta method for second order', string
varname, scalar 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.

```
gf_model_set(model M, 'add Newmark scheme', string varname, scalar
beta, scalar 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.

```
gf_model_set(model M, 'disable bricks', ivec bricks_indices)
```

Disable a brick (the brick will no longer participate to the building of the tangent linear system).

```
gf_model_set(model M, 'enable bricks', ivec bricks_indices)
```

Enable a disabled brick.

```
gf_model_set(model M, 'disable variable', string 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.

```
gf_model_set(model M, 'enable variable', string varname)
```

Enable a disabled variable (and its attached multipliers).

```
gf_model_set(model M, 'first iter')
```

To be executed before the first iteration of a time integration scheme.

```
gf_model_set(model M, 'next iter')
```

To be executed at the end of each iteration of a time integration scheme.

```
ind = gf_model_set(model M, 'add basic contact brick', 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.

```
ind = gf_model_set(model M, 'add basic contact brick two deformable
bodies', 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.

```
gf_model_set(model M, 'contact brick set BN', int indbrick, spmat BN)
```

Can be used to set the BN matrix of a basic contact/friction brick.

```
gf_model_set(model M, 'contact brick set BT', int indbrick, spmat BT)
```

Can be used to set the BT matrix of a basic contact with friction brick.

```
ind = gf_model_set(model M, 'add nodal contact with rigid obstacle
brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add contact with rigid obstacle
brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add integral contact with rigid
obstacle brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add penalized contact with
rigid obstacle brick', mesh_im 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 tangent 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)

```
ind = gf_model_set(model M, 'add Nitsche contact with rigid obstacle
brick', mesh_im 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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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.

```
ind = gf_model_set(model M, 'add Nitsche midpoint contact with rigid
obstacle brick', mesh_im mim, string varname, string Neumannterm,
string Neumannterm_wt, string dataname_obstacle, string gamma0name,
int region, scalar theta, string dataname_friction_coeff, string
dataname_alpha, string 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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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.

```
ind = gf_model_set(model M, 'add Nitsche fictitious domain contact
brick', mesh_im 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 conditionnaly coercive for *gamma0* small. *theta = -1* corresponds to the skew-symmetric method which is inconditionnaly 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.

```
ind = gf_model_set(model M, 'add nodal contact between nonmatching
meshes brick', mesh_im mim1[, mesh_im 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.

```
ind = gf_model_set(model M, 'add nonmatching meshes contact brick',
mesh_im mim1[, mesh_im 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.

```
ind = gf_model_set(model M, 'add integral contact between
nonmatching meshes brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add penalized contact between
nonmatching meshes brick', mesh_im 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.

```
ind = gf_model_set(model M, 'add integral large sliding contact brick
raytracing', 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).

```
gf_model_set(model M, 'add rigid obstacle to large sliding contact
brick', int indbrick, string expr, int 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.

```
gf_model_set(model M, 'add master contact boundary to large sliding
contact brick', int indbrick, mesh_im mim, int region, string
dispname[, string wname])
```

Adds a master contact boundary to an existing large sliding contact with friction brick.

```
gf_model_set(model M, 'add slave contact boundary to large sliding
contact brick', int indbrick, mesh_im mim, int region, string
```

```
dispname, string lambdaname[, string wname])
```

Adds a slave contact boundary to an existing large sliding contact with friction brick.

```
gf_model_set(model M, 'add master slave contact boundary to large
sliding contact brick', int indbrick, mesh_im mim, int region, string
dispname, string lambdaname[, string wname])
```

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

```
ind = gf_model_set(model M, 'add Nitsche large sliding contact
brick raytracing', 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).

```
gf_model_set(model M, 'add rigid obstacle to Nitsche large sliding
contact brick', int indbrick, string expr, int 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.

```
gf_model_set(model M, 'add master contact boundary to biased Nitsche
large sliding contact brick', int indbrick, mesh_im mim, int region,
string dispname[, string wname])
```

Adds a master contact boundary to an existing biased Nitsche's large sliding contact with friction brick.

```
gf_model_set(model M, 'add slave contact boundary to biased Nitsche
large sliding contact brick', int indbrick, mesh_im mim, int region,
string dispname, string lambdaname[, string wname])
```

Adds a slave contact boundary to an existing biased Nitsche's large sliding contact with friction brick.

```
gf_model_set(model M, 'add contact boundary to unbiased Nitsche large
sliding contact brick', int indbrick, mesh_im mim, int region, string
dispname, string lambdaname[, string wname])
```

Adds a contact boundary to an existing unbiased Nitschelarge sliding contact with friction brick which is both master and slave.

gf_poly

Synopsis

```
gf_poly(poly P, 'print')
gf_poly(poly P, 'product')
```

Description :

Performs various operations on the polynom POLY.

Command list :

`gf_poly(poly P, 'print')`

Prints the content of P.

`gf_poly(poly P, 'product')`

To be done ... !

gf_precond

Synopsis

```
PC = gf_precond('identity')
PC = gf_precond('cidentity')
PC = gf_precond('diagonal', vec D)
PC = gf_precond('ildlt', spmat m)
PC = gf_precond('ilu', spmat m)
PC = gf_precond('ildltt', spmat m[, int fillin[, scalar threshold]])
PC = gf_precond('ilut', spmat m[, int fillin[, scalar threshold]])
PC = gf_precond('superlu', spmat m)
PC = gf_precond('spmat', spmat m)
```

Description :

General constructor for precond objects.

The preconditioners may store REAL or COMPLEX values. They accept getfem sparse matrices and Matlab sparse matrices.

Command list :

`PC = gf_precond('identity')`

Create a REAL identity precondioner.

`PC = gf_precond('cidentity')`

Create a COMPLEX identity precondioner.

`PC = gf_precond('diagonal', vec D)`

Create a diagonal precondioner.

`PC = gf_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 = gf_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 = gf_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 = gf_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 = gf_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 = gf_precond('spmat', spmat m)`

Preconditionner given explicitely by a sparse matrix.

gf_precond_get

Synopsis

```
gf_precond_get(precond P, 'mult', vec V)
gf_precond_get(precond P, 'tmult', vec V)
gf_precond_get(precond P, 'type')
gf_precond_get(precond P, 'size')
gf_precond_get(precond P, 'is_complex')
s = gf_precond_get(precond P, 'char')
gf_precond_get(precond P, 'display')
```

Description :

General function for querying information about precond objects.

Command list :

`gf_precond_get(precond P, 'mult', vec V)`

Apply the preconditioner to the supplied vector.

`gf_precond_get(precond P, 'tmult', vec V)`

Apply the transposed preconditioner to the supplied vector.

`gf_precond_get(precond P, 'type')`

Return a string describing the type of the preconditioner ('ilu', 'ildlt',..).

`gf_precond_get(precond P, 'size')`

Return the dimensions of the preconditioner.

`gf_precond_get(precond P, 'is_complex')`

Return 1 if the preconditioner stores complex values.

`s = gf_precond_get(precond P, '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.

`gf_precond_get(precond P, 'display')`

displays a short summary for a precond object.

gf_slice

Synopsis

```
sl = gf_slice(sliceop, {slice sl|{mesh m| mesh_fem mf, vec U}, int refine}[, mat CVfids])
sl = gf_slice('streamlines', mesh_fem mf, mat U, mat S)
sl = gf_slice('points', mesh m, mat Pts)
sl = gf_slice('load', string filename[, mesh m])
```

Description :

General constructor for slice objects.

Creation of a mesh slice. Mesh slices are very similar to a P1-discontinuous mesh_fem on which interpolation is very fast. The slice is built from a mesh object, and a description of the slicing operation, for example:

```
sl = gf_slice({'planar',+1,[0;0],[1;0]}, 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 mesh_fem *mf* instead of a mesh, and if it is followed by a *mf*-field *u* (with size(u,1) == gf_mesh_fem_get(mesh_fem MF, 'nbdof')), then the deformation *u* will be applied to the mesh before the slicing operation.

The first argument can also be a slice.

Command list :

`sl = gf_slice(sliceop, {slice sl|{mesh m| mesh_fem mf, vec U}, int refine}[, mat CVfids])`

Create a slice using *sliceop* operation.

sliceop operation is specified with Matlab CELL arrays (i.e. with braces) . 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, mesh_fem mf, vec U, scalar s} : Cut using the isosurface of the field *U* (defined on the mesh_fem *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 = gf_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 = gf_slice('streamlines', mesh_fem mf, mat U, mat S)`

Compute streamlines of the (vector) field *U*, with seed points given by the columns of *S*.

`sl = gf_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 = gf_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.

gf_slice_get

Synopsis

```
d = gf_slice_get(slice S, 'dim')
a = gf_slice_get(slice S, 'area')
CVids = gf_slice_get(slice S, 'cvs')
n = gf_slice_get(slice S, 'nbpts')
ns = gf_slice_get(slice S, 'nbsplxs'[, int dim])
P = gf_slice_get(slice S, 'pts')
{S, CV2S} = gf_slice_get(slice S, 'splxs',int dim)
{P, E1, E2} = gf_slice_get(slice S, 'edges')
Usl = gf_slice_get(slice S, 'interpolate_convex_data', mat Ucv)
m = gf_slice_get(slice S, 'linked mesh')
m = gf_slice_get(slice S, 'mesh')
z = gf_slice_get(slice S, 'memsize')
```

```
gf_slice_get(slice S, 'export to vtk', string filename, ...)
gf_slice_get(slice S, 'export to pov', string filename)
gf_slice_get(slice S, 'export to dx', string filename, ...)
gf_slice_get(slice S, 'export to pos', string filename[, string name][[,mesh_fem mf1], mat U1, string
s = gf_slice_get(slice S, 'char')
gf_slice_get(slice S, 'display')
```

Description :

General function for querying information about slice objects.

Command list :

`d = gf_slice_get(slice S, 'dim')`

Return the dimension of the slice (2 for a 2D mesh, etc..).

`a = gf_slice_get(slice S, 'area')`

Return the area of the slice.

`CVids = gf_slice_get(slice S, 'cvs')`

Return the list of convexes of the original mesh contained in the slice.

`n = gf_slice_get(slice S, 'nbpts')`

Return the number of points in the slice.

`ns = gf_slice_get(slice S, 'nbsplxs'[, int dim])`

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 gf_slice_get(slice S, 'dim')+1, except if the optional argument *dim* is used.

`P = gf_slice_get(slice S, 'pts')`

Return the list of point coordinates.

`{S, CV2S} = gf_slice_get(slice S, 'splxs',int 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)-1)' gives the list of simplexes for the fourth convex.

`{P, E1, E2} = gf_slice_get(slice S, '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.

`Usl = gf_slice_get(slice S, 'interpolate_convex_data', mat 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 gf_mesh_get(mesh M, 'max cvid').

Example of use: gf_slice_get(slice S, 'interpolate_convex_data', gf_mesh_get(mesh M, 'quality')).

`m = gf_slice_get(slice S, 'linked mesh')`

Return the mesh on which the slice was taken.

`m = gf_slice_get(slice S, 'mesh')`

Return the mesh on which the slice was taken (identical to 'linked mesh')

`z = gf_slice_get(slice S, 'memsize')`

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

`gf_slice_get(slice S, 'export to vtk', 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:

- gf_slice_get(slice S, 'export to vtk', 'test.vtk', Usl, 'first_dataset', mf, U2, 'second_dataset')
- gf_slice_get(slice S, 'export to vtk', 'test.vtk', 'ascii', mf,U2)
- gf_slice_get(slice S, 'export to vtk', 'test.vtk', 'edges', 'ascii', Uslice)

`gf_slice_get(slice S, 'export to pov', string filename)`

Export a the triangles of the slice to POV-RAY.

`gf_slice_get(slice S, 'export to dx', 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.

```
gf_slice_get(slice S, 'export to pos', string filename[, string
name][[,mesh_fem mf1], mat U1, string nameU1[[,mesh_fem 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.

`s = gf_slice_get(slice S, '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.

`gf_slice_get(slice S, 'display')`

displays a short summary for a slice object.

gf_slice_set

Synopsis

```
gf_slice_set(slice S, 'pts', mat P)
```

Description :

Edition of mesh slices.

Command list :

`gf_slice_set(slice S, 'pts', mat 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 gf_slice_get(slice S, 'dim')).

gf_spmat

Synopsis

```
SM = gf_spmat('empty', int m [, int n])
SM = gf_spmat('copy', mat K [,  I [,  J]])
SM = gf_spmat('identity', int n)
SM = gf_spmat('mult', spmat A, spmat B)
SM = gf_spmat('add', spmat A, spmat B)
SM = gf_spmat('diag', mat D [, ivec E [, int n [,int m]]])
SM = gf_spmat('load','hb'|'harwell-boeing'|'mm'|'matrix-market', string filename)
```

Description :

General constructor for spmat objects.

Create a new sparse matrix in getfem++ format(, i.e. sparse matrices which are stored in the getfem workspace, not the matlab sparse matrices). 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 `gf_spmat_set(spmat S, '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.

Command list :

`SM = gf_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 = gf_spmat('copy', mat K [, I [, J]])`

Duplicate a matrix *K* (which might be a spmat or a native matlab sparse matrix). If index *I* and/or *J* are given, the matrix will be a submatrix of *K*. For example:

```
m = gf_spmat('copy', sprand(50,50,.1), 1:40, [6 7 8 3 10])
```

will return a 40x5 matrix.

`SM = gf_spmat('identity', int n)`

Create a *n x n* identity matrix.

`SM = gf_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 `gf_spmat_set(spmat S, 'to_complex')`

`SM = gf_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 = gf_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 = gf_spmat('load','hb'|'harwell-boeing'|'mm'|'matrix-market', string filename)`

Read a sparse matrix from an Harwell-Boeing or a Matrix-Market file See also `gf_util('load matrix')`.

gf_spmat_get

Synopsis

```
n = gf_spmat_get(spmat S, 'nnz')
Sm = gf_spmat_get(spmat S, 'full'[, list I[, list J]])
MV = gf_spmat_get(spmat S, 'mult', vec V)
MtV = gf_spmat_get(spmat S, 'tmult', vec V)
D = gf_spmat_get(spmat S, 'diag'[, list E])
s = gf_spmat_get(spmat S, 'storage')
{ni,nj} = gf_spmat_get(spmat S, 'size')
b = gf_spmat_get(spmat S, 'is_complex')
{JC, IR} = gf_spmat_get(spmat S, 'csc_ind')
V = gf_spmat_get(spmat S, 'csc_val')
```

```
{N, U0} = gf_spmat_get(spmat S, 'dirichlet nullspace', vec R)
gf_spmat_get(spmat S, 'save', string format, string filename)
s = gf_spmat_get(spmat S, 'char')
gf_spmat_get(spmat S, 'display')
{mantissa_r, mantissa_i, exponent} = gf_spmat_get(spmat S, 'determinant')
```

Description :

Command list :

`n = gf_spmat_get(spmat S, 'nnz')`

Return the number of non-null values stored in the sparse matrix.

`Sm = gf_spmat_get(spmat S, 'full'[, 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.

`MV = gf_spmat_get(spmat S, 'mult', vec V)`

Product of the sparse matrix *M* with a vector *V*.

For matrix-matrix multiplications, see gf_spmat('mult').

`MtV = gf_spmat_get(spmat S, 'tmult', vec V)`

Product of *M* transposed (conjugated if *M* is complex) with the vector *V*.

`D = gf_spmat_get(spmat S, 'diag'[, list E])`

Return the diagonal of *M* as a vector.

If *E* is used, return the sub-diagonals whose ranks are given in E.

`s = gf_spmat_get(spmat S, 'storage')`

Return the storage type currently used for the matrix.

The storage is returned as a string, either 'CSC' or 'WSC'.

`{ni,nj} = gf_spmat_get(spmat S, 'size')`

Return a vector where *ni* and *nj* are the dimensions of the matrix.

`b = gf_spmat_get(spmat S, 'is_complex')`

Return 1 if the matrix contains complex values.

`{JC, IR} = gf_spmat_get(spmat S, '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.

`V = gf_spmat_get(spmat S, '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.

`{N, U0} = gf_spmat_get(spmat S, 'dirichlet nullspace', vec 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*

`gf_spmat_get(spmat S, 'save', string format, string filename)`

Export the sparse matrix.

the format of the file may be 'hb' for Harwell-Boeing, or 'mm' for Matrix-Market.

`s = gf_spmat_get(spmat S, '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.

`gf_spmat_get(spmat S, 'display')`

displays a short summary for a spmat object.

`{mantissa_r, mantissa_i, exponent} = gf_spmat_get(spmat S, 'determinant')`

returns the matrix determinant calculated using MUMPS.

gf_spmat_set

Synopsis

```
gf_spmat_set(spmat S, 'clear'[, list I[, list J]])
gf_spmat_set(spmat S, 'scale', scalar v)
gf_spmat_set(spmat S, 'transpose')
gf_spmat_set(spmat S, 'conjugate')
gf_spmat_set(spmat S, 'transconj')
gf_spmat_set(spmat S, 'to_csc')
gf_spmat_set(spmat S, 'to_wsc')
gf_spmat_set(spmat S, 'to_complex')
gf_spmat_set(spmat S, 'diag', mat D [, ivec E])
gf_spmat_set(spmat S, 'assign', ivec I, ivec J, mat V)
gf_spmat_set(spmat S, 'add', ivec I, ivec J, mat V)
```

Description :

Modification of the content of a getfem sparse matrix.

Command list :

`gf_spmat_set(spmat S, 'clear'[, 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.

`gf_spmat_set(spmat S, 'scale', scalar v)`

Multiplies the matrix by a scalar value *v*.

`gf_spmat_set(spmat S, 'transpose')`

Transpose the matrix.

`gf_spmat_set(spmat S, 'conjugate')`

Conjugate each element of the matrix.

`gf_spmat_set(spmat S, 'transconj')`

Transpose and conjugate the matrix.

`gf_spmat_set(spmat S, 'to_csc')`

Convert the matrix to CSC storage.

CSC storage is recommended for matrix-vector multiplications.

`gf_spmat_set(spmat S, 'to_wsc')`

Convert the matrix to WSC storage.

Read and write operation are quite fast with WSC storage.

`gf_spmat_set(spmat S, 'to_complex')`

Store complex numbers.

`gf_spmat_set(spmat S, 'diag', mat D [, ivec E])`

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*.

`gf_spmat_set(spmat S, 'assign', ivec I, ivec J, mat V)`

Copy V into the sub-matrix 'M(I,J)'.

V might be a sparse matrix or a full matrix.

`gf_spmat_set(spmat S, 'add', ivec I, ivec J, mat V)`

Add *V* to the sub-matrix 'M(I,J)'.

V might be a sparse matrix or a full matrix.

gf_util

Synopsis

```
gf_util('save matrix', string FMT, string FILENAME, mat A)
A = gf_util('load matrix', string FMT, string FILENAME)
tl = gf_util('trace level' [, int level])
tl = gf_util('warning level', int level)
```

Description :

Performs various operations which do not fit elsewhere.

Command list :

`gf_util('save matrix', string FMT, string FILENAME, mat A)`

Exports a sparse matrix into the file named FILENAME, using Harwell-Boeing (FMT='hb') or Matrix-Market (FMT='mm') formatting.

`A = gf_util('load matrix', string FMT, string FILENAME)`

Imports a sparse matrix from a file.

`tl = gf_util('trace level' [, int level])`

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.

`tl = gf_util('warning level', int level)`

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.

gf_workspace

Synopsis

```
gf_workspace('push')
gf_workspace('pop',  [,i,j, ...])
gf_workspace('stat')
gf_workspace('stats')
gf_workspace('keep', i[,j,k...])
gf_workspace('keep all')
gf_workspace('clear')
gf_workspace('clear all')
gf_workspace('class name', i)
```

Description :

Getfem workspace management function.

Getfem uses its own workspaces in Matlab, independently of the matlab workspaces (this is due to some limitations in the memory management of matlab objects). By default, all getfem variables belong to the root getfem workspace. A function can create its own workspace by invoking gf_workspace('push') at its beginning. When exiting, this function MUST invoke gf_workspace('pop') (you can use matlab exceptions handling to do this cleanly when the function exits on an error).

Command list :

`gf_workspace('push')`

Create a new temporary workspace on the workspace stack.

`gf_workspace('pop', [,i,j, ...])`

Leave the current workspace, destroying all getfem objects belonging to it, except the one listed after 'pop', and the ones moved to parent workspace by gf_workspace('keep').

`gf_workspace('stat')`

Print informations about variables in current workspace.

`gf_workspace('stats')`

Print informations about all getfem variables.

`gf_workspace('keep', i[,j,k...])`

prevent the listed variables from being deleted when gf_workspace("pop") will be called by moving these variables in the parent workspace.

`gf_workspace('keep all')`

prevent all variables from being deleted when gf_workspace("pop") will be called by moving the variables in the parent workspace.

`gf_workspace('clear')`

Clear the current workspace.

`gf_workspace('clear all')`

Clear every workspace, and returns to the main workspace (you should not need this command).

`gf_workspace('class name', i)`

Return the class name of object i (if I is a mesh handle, it return gfMesh etc..)

CHAPTER

EIGHT

GETFEM++ OO-COMMANDS

The toolbox comes with a set of *MatLab* objects mathworks-oo, (look at the `@gf*` sub-directories in the toolbox directory). These object are no more than the getfem object handles, which are flagged by *MatLab* as objects.

In order to use these objects, you have to call their constructors: `gfMesh`, `gfMeshFem`, `gfGeoTrans`, `gfFem`, `gfInteg`. These constructor just call the corresponding *GetFEM++* function (i.e. `gf_mesh`, `gf_mesh_fem`, ...), and convert the structure returned by these function into a *MatLab* object. There is also a `gfObject` function which converts any getfem handle into the corresponding *MatLab* object.

With such object, the most interesting feature is that you do not have to call the "long" functions names `gf_mesh_fem_get(obj,...)`, `gf_slice_set(obj,...)` etc., instead you just call the shorter `get(obj,...)` or `set(obj,...)` whatever the type of `obj` is.

A small number of "pseudo-properties" are also defined on these objects, for example if `m` is a `gfMesh` object, you can use directly `m.nbpts` instead of `get(m, 'nbpts')`.

As an example:

```
% classical creation of a mesh object
>> m=gf_mesh('load', 'many_element.mesh_fem')
m =
     id: 2
    cid: 0
% conversion to a matlab object. the display function is overloaded for gfMesh.
>> mm=gfMesh(m)
gfMesh object ID=2 [11544 bytes], dim=3, nbpts=40, nbcvs=7
% direct creation of a gfMesh object. Arguments are the same than those of gf_mesh
>> m=gfMesh('load', 'many_element.mesh_fem')
gfMesh object ID=3 [11544 bytes], dim=3, nbpts=40, nbcvs=7
% get(m, 'pid_from_cvid') is redirected to gf_mesh_get(m,'pid from cvid')
>> get(m, 'pid_from_cvid', 3)
ans =
     8     9    11    15    17    16    18    10    12
% m.nbpts is directly translated into gf_mesh_get(m,'nbpts')
>> m.nbpts
ans =
    40

>> mf=gfMeshFem('load','many_element.mesh_fem')
gfMeshFem object: ID=5 [1600 bytes], qdim=1, nbdof=99,
  linked gfMesh object: dim=3, nbpts=40, nbcvs=7
>> mf.mesh
gfMesh object ID=4 [11544 bytes], dim=3, nbpts=40, nbcvs=7
% accessing the linked mesh object
>> mf.mesh.nbpts
ans =
    40
```

```
>> get(mf.mesh, 'pid_from_cvid', 3)
ans =
     8     9    11    15    17    16    18    10    12

>> mf.nbdof
ans =
    99

% access to fem of convex 1
>> mf.fem(2)
gfFem object ID=0 dim=2, target_dim=1, nbdof=9,[EQUIV, POLY, LAGR], est.degree=4
 -> FEM_QK(2,2)
>> mf.mesh.geotrans(1)
gfGeoTrans object ID= 0 dim=2, nbpts= 6 : GT_PK(2,2)
```

Although this interface seems more convenient, you must be aware that this always induce a call to a mex-file, and additional *MatLab* code:

```
>> tic; j=0; for i=1:1000, j=j+mf.nbdof; end; toc
elapsed_time =
    0.6060
>> tic; j=0; for i=1:1000, j=j+gf_mesh_fem_get(mf,'nbdof'); end; toc
elapsed_time =
    0.1698
>> tic; j=0;n=mf.nbdof;  for i=1:1000, j=j+n; end; toc
elapsed_time =
    0.0088
```

Hence you should always try to store data in *MatLab* arrays instead of repetitively calling the getfem functions.

Avalaible object types are `gfCvStruct`, `gfGeoTrans`, `gfEltm`, `gfInteg`, `gfFem`, `gfMesh`, `gfMeshFem`, `gfMeshIm`, `gfMdBrick`, `gfMdState`, `gfModel`, `gfSpmat`, `gfPrecond`, and `gfSlice`.

A

assembly, 5

B

boundary, 12

C

convex id, 6
convexes, 5
cvid, 6

D

degrees of freedom, 5
dof, 5

E

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
- gfCvStruct, 10, 122
- gfEltm, 122
- gfFem, 10, 122
- gfGeoTrans, 8, 122
- gfGlobalFunction, 8
- gfInteg, 10, 122
- gfMdBrick, 10, 122
- gfMdState, 10, 122
- gfMesh, 8, 122
- gfMeshFem, 10, 122
- gfMeshIm, 122
- gfMeshImM, 10
- gfMeshSlice, 10
- gfModel, 10, 122
- gfPrecond, 122
- gfSlice, 122
- gfSpmat, 122
- integration method, 5
- interpolate, 5
- Lagrangian, 5
- Laplacian, 11
- memory management, 10
- mesh, 5
- mesh nodes, 5
- mesh_fem, 5
- mesh_im, 5
- mex-file, 7
- model, 13
- pid, 6
- point id, 6
- quadrature formula, 12
- quadrature formulas, 5
- reference convex, 5
- Von Mises, 17

F

FEM, 5

G

geometric transformation, 5
geometrical nodes, 5
gfCvStruct, 10, 122
gfEltm, 122
gfFem, 10, 122
gfGeoTrans, 8, 122
gfGlobalFunction, 8
gfInteg, 10, 122
gfMdBrick, 10, 122
gfMdState, 10, 122
gfMesh, 8, 122
gfMeshFem, 10, 122
gfMeshIm, 122
gfMeshImM, 10
gfMeshSlice, 10
gfModel, 10, 122
gfPrecond, 122
gfSlice, 122

gfSpmat, 122

I

integration method, 5
interpolate, 5

L

Lagrangian, 5
Laplacian, 11

M

memory management, 10
mesh, 5
mesh nodes, 5
mesh_fem, 5
mesh_im, 5
mex-file, 7
model, 13

P

pid, 6
point id, 6

Q

quadrature formula, 12
quadrature formulas, 5

R

reference convex, 5

V

Von Mises, 17