



a Generic Finite Element library in C++

Documentation, part 2

SHORT USER DOCUMENTATION

Yves RENARD, Julien POMMIER ¹

February 26, 2010

Introduction

The GETFEM++ project focuses on the development of a generic and efficient C++ library for finite element methods elementary computations. The goal is to provide a library allowing the computation of any elementary matrix (even for mixed finite element methods) on the largest class of methods and elements, and for arbitrary dimension (i.e. not only 2D and 3D problems).

It offers a complete separation between integration methods (exact or approximated), geometric transformations (linear or not) and finite element methods of arbitrary degrees. It can really relieve a more integrated finite element code of technical difficulties of elementary computations.

Examples of available finite element method are : Pk on simplices in arbitrary degrees and dimensions, Qk on parallelepipeds, P1, P2 with bubble functions, Hermite elements, elements with hierarchic basis (for multigrid methods for instance), discontinuous Pk or Qk, XFem, Argyris, HCT, Raviart-Thomas, ...

The addition of a new finite element method is straightforward. Its description on the reference element must be provided (in most of the cases, this is the description of the basis functions, and nothing more). Extensions are provided for Hermite elements, piecewise polynomial, non-polynomial and vectorial elements, XFem.

The library also includes the usual tools for finite elements such as assembly procedures for classical PDEs, interpolation methods, computation of norms, mesh operations, boundary conditions, post-processing tools such as extraction of slices from a mesh ...

GETFEM++ can be used to build very general finite elements codes, where the finite elements, integration methods, dimension of the meshes, are just some parameters that can be changed very easily, thus allowing a large spectrum of experimentations. Numerous examples are available in the `tests` directory of the distribution.

¹ MIP, INSAT, Complexe scientifique de Rangueil, 31077 Toulouse, France, Yves.Renard@insa-lyon.fr

GETFEM++ has no meshing capabilities (apart regular meshes), hence it is necessary to import meshes. Imports formats currently known by getfem are GiD , GmSH and emc2 mesh files. However, given a mesh, it is possible to refine it automatically.

Copyright (C) 2000-2008 Yves Renard, Julien Pommier.

The program GETFEM++ is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Contents

1	How to install	4
2	Build a mesh	5
2.1	Add an element to a mesh	5
2.2	Remove an element from a mesh	7
2.3	Simple structured meshes	8
2.4	Mesh regions	9
2.5	Methods of the <code>getfem::mesh</code> object	9
2.6	Using <code>dal::bit_vector</code>	11
2.7	Face numbering	11
2.8	Save and load meshes	11
2.8.1	From <code>getfem</code> file format	11
2.8.2	Import a mesh	13
3	Build a finite element method on a mesh	13
3.1	first level: manipulating fems on each elements	14
3.2	Examples	16
3.3	Second level: the optional “vectorization”	16
3.4	Third level: the optional linear transformation (or reduction)	17
3.5	Obtaining generic <code>mesh_fems</code>	18
3.6	The <code>partial_mesh_fem</code> object	18
4	Selecting integration methods	19
4.1	Methods of the <code>mesh_im</code> object	20
5	Mesh refinement	21
6	Linear algebra procedures	22
7	Standard assembly procedures	22
7.1	Laplacian (Poisson) problem	22
7.2	Linear Elasticity problem	24
7.3	Stokes Problem with mixed finite element method	25
7.4	Assembling a mass matrix	25
8	Compute arbitrary elementary matrices - generic assembly procedures	25
9	Incorporate new finite element methods in <code>GETFEM++</code>	28
10	Incorporate new approximated integration methods in <code>GETFEM++</code>	28
11	Level-sets, Xfem, fictitious domains	29
11.1	Representation of level-sets	29
11.2	Mesh cut by level-sets	30
11.3	Adapted integration methods	30
11.4	Discontinuous field across some level-sets	31
11.5	Fictitious domain approach with Xfem	31

12 Support for Xfem methods	31
13 Interpolation of a finite element method on non-matching meshes	32
13.1 mixed methods with different meshes	32
13.2 mortar methods	32
14 Compute L^2 and H^1 norms	33
15 Compute derivatives	33
16 Export and view a solution	33
16.1 Saving mesh and mesh_fem objects for the Matlab interface	34
16.2 Producing mesh slices	34
16.3 Exporting mesh, mesh_fem or slices to VTK	36
16.4 Exporting mesh, mesh_fem or slices to OpenDX	37
17 Interpolation on different meshes	38
18 The model description	38
18.1 The model object	38
18.2 The brick object	42
18.3 How to build a new brick	42
18.4 How to add the brick to a model	46
18.5 Generic elliptic brick	48
18.6 Dirichlet condition brick	49
18.7 Source term bricks (and Neumann condition)	50
18.8 Predefined solvers	50
18.9 Example of a complete Poisson problem	51
18.10 Constraint brick	52
18.11 Other “explicit” bricks	53
18.12 Helmholtz brick	53
18.13 Fourier-Robin brick	53
18.14 Isotropic linearized elasticity brick	54
18.15 linear incompressibility (or nearly incompressibility) brick	54
18.16 Mass brick	55
18.17 The time dispatchers: integration of transient problems	56
18.18 Theta-method dispatcher	56
18.18.1 Basic first order time derivative brick	57
18.18.2 Basic second order time derivative brick	58
18.19 Midpoint dispatcher	58
18.19.1 Basic first order time derivative brick	59
18.19.2 Basic second order time derivative brick	59
18.20 Newmark scheme	59
19 The model bricks (old system)	60
19.1 The model state variable	60
19.2 Basic properties of a brick	60
19.3 Brick parameters	61

19.4	generic elliptic brick	62
19.5	Source term brick	62
19.6	Constraint brick	63
19.7	Dirichlet condition brick	64
19.8	Example of a complete Poisson problem	65
19.9	Predefined solvers	65
19.10	Isotropic linearized elasticity brick	66
19.11	Qu term brick	67
19.12	Helmholtz brick	67
19.13	linear incompressibility (or nearly incompressibility) brick	68
19.14	Small displacement plasticity brick	69
19.15	Contact and friction conditions brick	69
19.16	Linearized plate brick	69
19.17	Large strain elasticity brick	70
19.18	Large strain incompressibility brick	70
20	Parallelization of GETFEM++	71
21	Catch errors	71
22	Example: Laplacian program	72
23	Appendix A. Finite element method list	73
23.1	Dof graphical codification	73
23.2	Classical P_K Lagrange elements on simplices	73
23.3	Classical Lagrange elements on other geometries	76
23.4	Elements with hierarchical basis	80
23.4.1	Hierarchical elements with respect to the degree	80
23.4.2	Composite elements	81
23.4.3	Hierarchical composite elements	82
23.5	Classical vectorial elements	83
23.5.1	Raviart-Thomas of lowest order elements	83
23.5.2	Nedelec (or Whitney) edge elements	84
23.6	Specific elements in dimension 1	84
23.6.1	GaussLobatto element	84
23.6.2	Hermite element	84
23.6.3	Lagrange element with an additional bubble function	85
23.7	Specific elements in dimension 2	86
23.7.1	Elements with additional bubble functions	86
23.7.2	Non-conforming P_1 element	88
23.7.3	Hermite element	88
23.7.4	Morley element	89
23.7.5	Argyris element	90
23.7.6	Hsieh-Clough-Tocher element	91
23.7.7	A composite C^1 element on quadrilaterals	92
23.8	Specific elements in dimension 3	94
23.8.1	Elements with additional bubble functions	94
23.8.2	Hermite element	95

24 Appendix B. Cubature method list	97
24.1 Exact Integration methods	97
24.2 Newton cotes Integration methods	97
24.3 Gauss Integration methods on dimension 1	97
24.4 Gauss Integration methods on dimension 2	98
24.5 Gauss Integration methods on dimension 3	100
24.6 Direct product of integration methods	101
24.7 Composite integration methods	102

1 How to install

Since we used standard GNU tools, the installation of the GETFEM++ library is somewhat standard. If the GETFEM++ archive is in your current directory you can unpack it and enter inside the directory of the distribution with the commands

```
gunzip -c getfem-x.xx.tar.gz | tar xvf -
cd getfem-x.xx
```

Then you have to run the configure script with

```
./configure
```

or if you want to set the prefix directory where to install the library you can use the `--prefix` option (the default prefix directory is `/usr/local`):

```
./configure --prefix=dest_dir
```

Note that there are other options to the configure script. a `./configure --help` will list them. Most important ones are `--enable-matlab`, `--enable-python` and `--enable-scilab` to build the interfaces.

then start the compilation with

```
make
```

(or preferably with `gmake`) and the installation with

```
make install
```

You can also check if the compilation is correct with (this will build all test programs, and run them)

```
make check
```

If you want to use a different compiler than the one chosen automatically by the `./configure` script, just specify its name on the command line:

```
./configure CXX=mycompiler
```

If you want to use a specific BLAS library, you may have to supply the necessary link flags and libs to the configure script with

```
./configure BLAS_LIBS="-L/path/to/lib -lfoo -lbar ....etc"
```

More specific instructions can be found in the `README*` files of the distribution.

If you want to use MUMPS as the default sparse direct solver instead of SuperLU, see the indication of section 6.

2 Build a mesh

As a preliminary, you may want to read this short introduction to the `getfem++` “vocabulary”:

http://download.gna.org/getfem/doc/getfem_reference/index.html.

`GETFEM++` has its own structure to store meshes defined in the files `getfem/bgeot_mesh_structure.h` and `getfem/getfem_mesh.h`. The main structure is defined in `getfem/getfem_mesh.h` by the object `getfem::mesh`.

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

There is no meshing procedures in `GETFEM++` to mesh complex geometries. This is not the goal of this package. But you can easily load a mesh from any format (some procedures are in `getfem/getfem_import.h` to load meshes from some public domain mesh generators).

The structure `getfem::mesh` may also contain a description about a region of the mesh, such as a boundary or a set of elements. This is handled via a container of convexes and convex faces, `getfem::mesh_region`.

2.1 Add an element to a mesh

Suppose the variable `mymesh` has been declared by

```
getfem::mesh mymesh;
```

then you have two ways to insert a new element to this mesh: from a list of points or from a list of indexes of already existing points.

To enter a new point on a mesh use the method

```
i = mymesh.add_point(pt);
```

where `pt` is of type `bgeot::base_node`. The index `i` is the index of this point on the mesh. If the point already exists in the mesh, a new point is not inserted and the index of the already existing point is returned. A mesh has a principal dimension, which is the dimension of its points. It is not possible to have points of different dimensions in a same mesh.

The most basic function to add a new element to a mesh is

```
j = mymesh.add_convex(pgt, it);
```

This is a template function, with `pgt` of type `bgeot::pgeometric_trans` (basically a pointer to an instance of type `bgeot::geometric_trans`) and `it` is an iterator on a list of indexes of already existing points. For instance, if one needs to add a new triangle in a 3D mesh, one needs to define first an array with the indexes of the three points:

```

std::vector<bgeot::size_type> ind(3);
ind[0] = mymesh.add_point(bgeot::base_node(0.0, 0.0, 0.0));
ind[1] = mymesh.add_point(bgeot::base_node(0.0, 1.0, 0.0));
ind[2] = mymesh.add_point(bgeot::base_node(0.0, 0.0, 1.0));

```

then adding the element is done by

```

mymesh.add_convex(bgeot::simplex_trans(2,1), ind.begin());

```

where `bgeot::simplex_trans(N,1)`; denotes the usual linear geometric transformation for simplices of dimension N.

For simplices, a more specialized function exists, which is

```

mymesh.add_simplex(2, ind.begin());

```

It is also possible to give directly the list of points with the function

```

mymesh.add_convex_by_points(pgt, itp);

```

where now `itp` is an iterator on an array of points. For example

```

std::vector<bgeot::base_node> pts(3);
pts[0] = bgeot::base_node(0.0, 0.0, 0.0);
pts[1] = bgeot::base_node(0.0, 1.0, 0.0);
pts[2] = bgeot::base_node(0.0, 0.0, 1.0);
mymesh.add_convex_by_points(bgeot::simplex_trans(2,1), pts.begin());

```

It is possible to use also

```

mymesh.add_simplex_by_points(2, pts.begin());

```

For other elements than simplices, it is still possible to use `mymesh.add_convex_by_points` or `mymesh.add_convex` with the appropriate geometric transformation.

`bgeot::parallelepiped_trans(N, 1)` describes the usual transformation for parallelepipeds of dimension N (quadrilateron for N=2, hexahedron for N=3, ...)

`bgeot::prism_trans(N, 1)` describes the usual transformation for prisms of dimension N (usual prism is for N=3. A generalized prism is the product of a simplex of dimension N-1 with a segment)

Specialized functions exist also:

```

mymesh.add_parallelepiped(N, it);
mymesh.add_parallelepiped_by_points(N, itp);
mymesh.add_prism(N, it);
mymesh.add_prism_by_points(N, itp);

```

The order of the points in the array of points is not important for simplices (except if you care about the orientation of your simplices). For other elements, it is important to respect the order shown in figure 1.

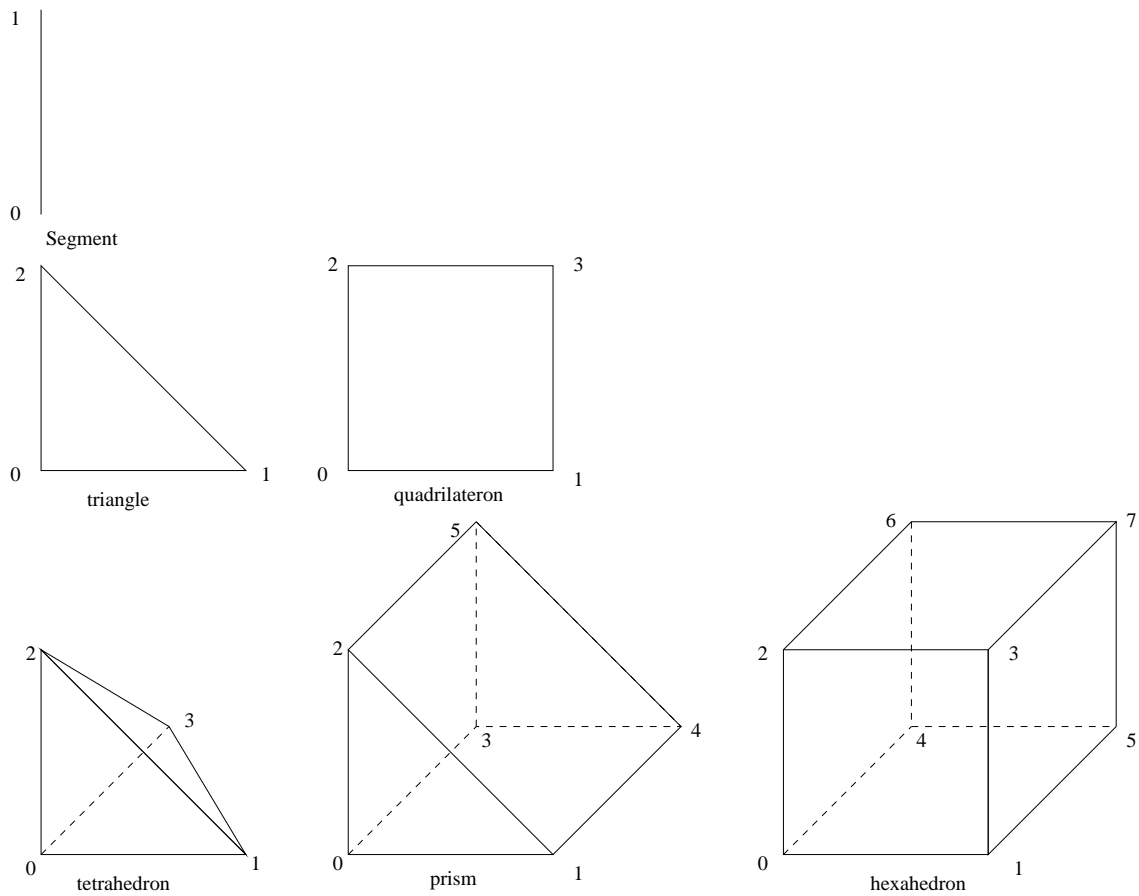


Figure 1: *vertex numeration for usual elements*

2.2 Remove an element from a mesh

To remove an element from a mesh, simply use

```
mymesh.sup_convex(i);
```

where *i* is the index of the element.

2.3 Simple structured meshes

For parallelepiped domains, it is possible to obtain structured meshes with simplices, parallelepipeds or prisms elements from three functions defined in `getfem/getfem_regular_meshes.h`.

The simplest function to use is:

```
void regular_unit_mesh(mesh& m, std::vector<size_type> nsubdiv,
                      bgeot::pgeometric_trans pgt, bool noised = false);
```

which fills the mesh `m` with a regular mesh of simplices/parallelepipeds/prisms (depending on the value of `pgt`). The number of cells in each direction is given by `nsubdiv`. The following example builds a mesh of quadratic triangles on the unit square (the mesh can be scaled and translated afterwards)

```
std::vector<getfem::size_type> nsubdiv(2);
nsubdiv[0] = 10; nsubdiv[1] = 20;
regular_unit_mesh(m, nsubdiv, bgeot::simplex_geotrans(2,2));
```

More specialized regular mesh functions are also available:

```
getfem::parallelepiped_regular_simplex_mesh(mymesh, N, org, ivect, iref);
getfem::parallelepiped_regular_prism_mesh(mymesh, N, org, ivect, iref);
getfem::parallelepiped_regular_mesh(mymesh, N, org, ivect, iref);
```

where `mymesh` is a mesh variable in which the structured mesh will be built, `N` is the dimension (limited to 4 for simplices, 5 for prisms, unlimited for parallelepipeds), `org` is of type `bgeot::base_node` and represents the origin of the mesh, `ivect` is an iterator on an array of `N` vectors to build the parallelepiped domain, `iref` is an iterator on an array of `N` integers representing the number of division on each direction.

For instance, to build a mesh with tetrahedrons for a unit cube with $10 \times 10 \times 10$ cells one can write

```
getfem::mesh mymesh;

bgeot::base_node org(0.0, 0.0, 0.0);

std::vector<bgeot::base_small_vector> vect(3);

vect[0] = bgeot::base_small_vector(1.0, 0.0, 0.0);
vect[1] = bgeot::base_small_vector(0.0, 1.0, 0.0);
vect[2] = bgeot::base_small_vector(0.0, 0.0, 1.0);
```

```

std::vector<int> ref(3);

ref[0] = ref[1] = ref[2] = 10;

getfem::parallelepipede_regular_simplex_mesh(mymesh, 3, org, vect.begin(), ref.begin());

```

Remark: `base_node` and `base_small_vector` are almost identical, they are both “small” vector classes (they cannot store more than 16 elements), used to describe geometrical points, and geometrical vectors. Their memory footprint is lower than a `std::vector`.

2.4 Mesh regions

A mesh object can contain many `getfem::mesh_region` objects (declaration in `getfem/getfem_mesh_region.h`). These objects are containers for a set of convexes and convex faces. They are used to define boundaries, or a partition of the mesh for parallel solvers, etc.

```

mymesh.region(30).add(3); // add convex 3 into region 30
mymesh.region(30).add(4,3); // add face 3 of convex 4 into region 30
mymesh.sup_convex(4); // the corresponding entry will be removed from mesh.region(30)
for (getfem::mr_visitor i(mymesh.region(30)); !i.finished(); ++i) {
    cout << "convex: " << i.cv() << " face:" << i.f() << endl;
}

```

2.5 Methods of the `getfem::mesh` object

The list is not exhaustive.

<code>mymesh.dim()</code>	main dimension of the mesh.
<code>mymesh.points_index()</code>	gives a <code>dal::bit_vector</code> object which represents all the indexes of valid points of a mesh (see below)
<code>mymesh.points()[i]</code>	gives the point of index <code>i</code> (a <code>bgeot::base_node</code>).
<code>mymesh.convex_index()</code>	gives a <code>dal::bit_vector</code> object which represents all the indexes of valid elements of a mesh (see below)
<code>mymesh.structure_of_convex(i)</code>	gives the description of the structure of element of index <code>i</code> . The function return a <code>bgeot::pconvex_structure</code> .
<code>mymesh.structure_of_convex(i)</code> <code>->nb_faces()</code>	number of faces of element of index <code>i</code> .
<code>mymesh.structure_of_convex(i)</code> <code>->nb_points()</code>	number of vertices of element of index <code>i</code> .
<code>mymesh.structure_of_convex(i)</code> <code>->dim()</code>	intrinsic dimension of element of index <code>i</code> .
<code>mymesh.structure_of_convex(i)</code> <code>->nb_points_of_face(f)</code>	number of vertices of the face of local index <code>f</code> of element of index <code>i</code> .
<code>mymesh.structure_of_convex(i)</code> <code>->ind_points_of_face(f)</code>	return a container with the local indexes of all vertices of the face of local index <code>f</code> of element of index <code>i</code> . For instance <code>mesh.structure_of_convex(i) ->ind_points_of_face(f)[0]</code> is the local index of the first vertex.

<code>mymesh.structure_of_convex(i)</code> <code>->face_structure(f)</code>	gives the structure (a <code>bgeot::pconvex_structure</code>) of local index <code>f</code> of element of index <code>i</code> .
<code>mymesh.ind_points_of_convex(i)</code>	gives a container with the global indexes of vertices of element of index <code>i</code> .
<code>mymesh.points_of_convex(i)</code>	gives a container with the vertices of element of index <code>i</code> . This is an array of <code>bgeot::base_node</code> .
<code>mymesh.convex_to_point(ipt)</code>	gives a container with the indexes of all elements attached to the point of global index <code>ipt</code> .
<code>mymesh.neighbours_of_convex(ic, f)</code>	gives a container with the indexes of all elements in <code>mesh</code> having the common face of local index <code>f</code> of element <code>ic</code> except element <code>ic</code> .
<code>mymesh.neighbour_of_convex(ic, f)</code>	gives the index of the first elements in <code>mesh</code> having the common face of local index <code>f</code> of element <code>ic</code> except element <code>ic</code> . return <code>size_type(-1)</code> if none is found.
<code>mymesh.is_convex_having_neighbour(ic, f)</code>	return whether or not the element <code>ic</code> has a neighbour with respect to its face of local index <code>f</code> .
<code>mymesh.clear()</code>	delete all elements and points from the mesh.
<code>mymesh.optimize_structure()</code>	compact the structure (renumbers points and convexes such that there is no hole in their numbering).
<code>mymesh.trans_of_convex(i)</code>	return the geometric transformation of the element of index <code>i</code> (in a <code>bgeot::pgeometric_trans</code>). See [6] for more details about geometric transformations.
<code>mymesh.normal_of_face_of_convex(ic, f, pt)</code>	gives a <code>bgeot::base_small_vector</code> representing an outward normal to the element at the face of local index <code>f</code> at the point of local coordinates (coordinates in the element of reference) <code>pt</code> . The point <code>pt</code> has no influence if the geometric transformation is linear. This is not a unit normal, the norm of the resulting vector is the ratio between the surface of the face of the reference element and the the surface of the face of the real element.
<code>mymesh.convex_area_estimate(ic)</code>	gives an estimate of the area of convex <code>ic</code> .
<code>mymesh.convex_quality_estimate(ic)</code>	gives a rough estimate of the quality of element <code>ic</code> .
<code>mymesh.convex_radius_estimate(ic)</code>	gives an estimate of the radius of element <code>ic</code> .
<code>mymesh.region(irg)</code>	return a <code>getfem::mesh_region</code> . The region is stored in the mesh, and can contain a set of convex numbers and or convex faces.
<code>mymesh.has_region(irg)</code>	returns true if the region of index <code>irg</code> has been created.

The methods of the convexes/convex faces container `getfem::mesh_region` are:

<code>add(ic)</code>	add the convex of index <code>ic</code> to the region
<code>add(ic,f)</code>	add the face number <code>f</code> of the convex <code>ic</code>
<code>sup(ic), sup(ic,f)</code>	remove the convex or the convex face from the region
<code>is_in(ic), is_in(ic,f)</code>	return true if the convex (or convex face) is in the region.
<code>is_only_faces()</code>	return true if the region does not contain any convex.
<code>is_only_convexes()</code>	return true if the region does not contain any convex face.
<code>index()</code>	return a <code>dal::bit_vector</code> containing the list of convexes which are stored (or whose faces are stored) in the region.

Iteration over a `getfem::mesh_region` should be done with `getfem::mr_visitor`:

```
getfem::mesh_region &rg = mymesh.region(2);
for (getfem::mr_visitor i(rg); !i.finished(); ++i) {
    cout << "contains convex " << i.cv();
    if (i.is_face()) cout << "face " << i.f() << endl;
}
```

2.6 Using `dal::bit_vector`

The object `dal::bit_vector` (declared in `getfem/dal_bit_vector.h`) is a structure heavily used in `GETFEM++`. It is very close to `std::bitset` and `std::vector<bool>` but with additional functionalities to represent a set of non negative integers and iterate over them.

If `mn` is declared to be a `dal::bit_vector`, the two instructions `mn.add(6)` or `mn[6] = true` are equivalent and means that integer 6 is added to the set.

In a same way `mn.sup(6)` or `mn[6] = false` remove the integer 6 from the set. The instruction `mn.add(6, 4)` adds 6,7,8,9 to the set.

To iterate on a `dal::bit_vector`, it is possible to use iterators as usual, but, most of the time, as this object represents a set of integers, one just wants to iterate on the integers included into the set. The simplest way to do that is to use the pseudo-iterator `dal::bv_visitor`.

For instance, here is the code to iterate on the points of a mesh and print it to the standard output

```
for (dal::bv_visitor i(mymesh.points_index()); !i.finished(); ++i)
    cout << "Point of index " << i << " of the mesh: " << mymesh.points()[i] << endl;
```

2.7 Face numbering

The numeration of faces on usual elements is given in figure 2.

Note that, while the convexes and the points are globally numbered in a `getfem::mesh` object, there is no global numbering of the faces, so the only way to refer to a given face, is to give the convex number, and the local face number in the convex.

2.8 Save and load meshes

2.8.1 From `getfem` file format

In `getfem/getfem_mesh.h`, two methods are defined to load meshes from file and write meshes to a file.

<code>mymesh.write_to_file(const std::string &name)</code>	save the mesh into a file.
<code>mymesh.read_from_file(const std::string &name)</code>	load the mesh from a file.

The following is an example of how to load a mesh and extract information on it.

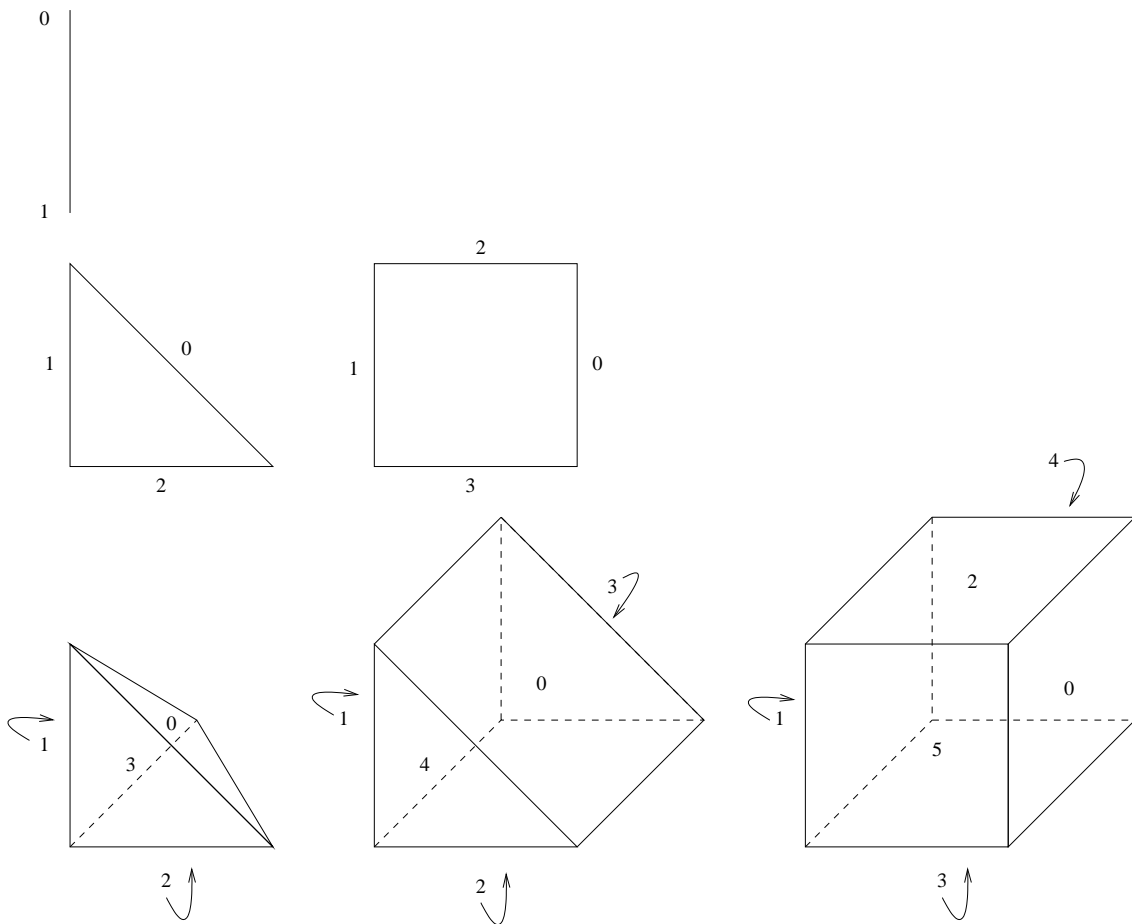


Figure 2: *faces numeration for usual elements*

```

#include <getfem/getfem_mesh.h>

getfem::mesh mymesh;

int main(int argc, char *argv[]) {
  try {

    // read the mesh from the file name given by the first argument
    mymesh.read_from_file(std::string(argv[1]));

    // List all the convexes
    dal::bit_vector nn = mymesh.convex_index();
    bgeot::size_type i;
    for (i << nn; i != bgeot::size_type(-1); i << nn) {
      cout << "Convex of index " << i << endl;
      bgeot::pconvex_structure cvs = mymesh.structure_of_convex(i);
    }
  }
}

```

```

    cout << "Number of vertices: " << cvs->nb_points() << endl;
    cout << "Number of faces: " << cvs->nb_faces() << endl;
    for (bgeot::size_type f = 0; f < cvs->nb_faces(); ++f) {
        cout << "face " << f << " has " << cvs->nb_points_of_face(f);
        cout << " vertices with local indexes: ";
        for (bgeot::size_type k = 0; k < cvs->nb_points_of_face(f); ++k)
            cout << cvs->ind_points_of_face(f)[k] << " ";
        cout << " and global indexes: ";
        for (bgeot::size_type k = 0; k < cvs->nb_points_of_face(f); ++k)
            cout << mymesh.ind_points_of_convex(i)[cvs->ind_points_of_face(f)[k]] << " ";
    }
}

} GMM_STANDARD_CATCH_ERROR; // catches standard errors
}

```

2.8.2 Import a mesh

The file `getfem/getfem_import.h` provides the function:

```
void import_mesh(const std::string& fmtfilename, mesh& m);
```

Here the string `fmtfilename` must contain a descriptor of the file format ("`gid`", "`gmsh`", "`am.fmt`", "`emc2_mesh`", or "`structured`") , followed by a colon and the file name (if there is not format descriptor, it is assumed that the file is a native getfem mesh and the `mesh::read_from_file()` method is used).

Example:

```
getfem::mesh m;
getfem::import_mesh("gid:../tests/meshes/tripod.GiD.msh",m);
```

The "`gid`" format is for meshes generated by GiD. The "`gmsh`" is for meshes generated by the open-source mesh generator GMSH, and the "`am.fmt`" and "`emc2_mesh`" are for files built with emc2 (free but 2D only)

The "`structured`" format is just a short specification for regular meshes: the rest of `fmtfilename` in that case is not a filename, but a string whos format is following:

```
getfem::import_mesh("structured:GT='GT_PK(2,1)'; NSUBDIV=[5,5]; ORG=[0,0];"
    "SIZES=[1,1]; NOISED=0", m);
```

where `GT` is the name of the geometric transformation, `NSUBDIV` a vector of the number of subdivisions in each coordinate (default value 2), `ORG` is the origin of the mesh (default value `[0,0,...]`), `SIZES` is a vector of the sizes in each direction (default value `[1, 1, ...]` and if `NOISED=1` the nodes of the interior of the mesh are randomly "shaken"(default value `NOISED=0`). In that string, all the parameters are optional except `GT`.

3 Build a finite element method on a mesh

The object `getfem::mesh_fem` defined in `getfem/getfem_mesh_fem.h` is designed to describe a finite element method on a whole mesh, i.e. to describe the finite element space on which some variables

will be described. This is a rather complex object which is central in GETFEM++. Basically, this structure describes the finite element method on each element of the mesh and some additional optional transformations. It is possible to have an arbitrary number of finite element descriptions for a single mesh. This is particularly necessary for mixed methods, but also to describe different data on the same mesh. One can instantiate a `getfem::mesh_fem` object as follows

```
getfem::mesh_fem mf(mymesh);
```

where `mymesh` is an already existing mesh. The structure will be linked to this mesh and will react when modifications will be done on it.

It is possible to specify element by element the finite element method, so that element of mixed types can be treated, even if the dimensions are different. For usual elements, the connection between two elements is done when the two elements are compatibles (same degrees of freedom on the common face). A numeration of the degrees of freedom is automatically done with a Cuthill Mc Kee like algorithm. You have to keep in mind that there is absolutely no connection between the numeration of vertices of the mesh and the numeration of the degrees of freedom. Every `getfem::mesh_fem` object has its own numeration.

There are three levels in the `getfem::mesh_fem` object:

- The element level: one finite element method per element. It is possible to mix the dimensions of the elements and the property to be vectorial or scalar.
- The optional vectorization (the `qdim` in `getfem` jargon, see [6], http://download.gna.org/getfem/doc/getfem.-project/getfem_project.4.html). For instance to represent a displacement field in continuum mechanics. Scalar elements are used componentwise. Note that you can mix some intrinsic vectorial elements (Raviart-Thomas element for instance) which will not be vectorized and some scalar element which will be.
- (GETFEM++ version 4.0) The optional additional linear transformation (reduction) of the degrees of freedom. It will consist in giving two matrices, the reduction matrix and the extension matrix. The reduction matrix should transform the basic dofs into the reduced dofs (the number of reduced dofs should be less or equal than the number of basic dofs). The extension matrix should describe the inverse transformation. The product of the reduction matrix with the extension matrix should be the identity matrix (ensuring in particular that the two matrices are of maximal rank). This optional transformation can be used to reduce the finite element space to a certain region (typically a boundary) or to prescribe some matching conditions between non naturally compatible fems (for instance fems with different degrees).

One has to keep in mind this construction manipulating the degrees of freedom of a `getfem::mesh_fem` object.

3.1 first level: manipulating fems on each elements

To select a particular finite element method on a given element, use the method

```
mf.set_finite_element(i, pf);
```

where `i` is the index of the element and `pf` is the descriptor (of type `getfem::pfem`, basically a pointer to an object which inherits from `getfem::virtual_fem`) of the finite element method. Alternative forms of this member function are:

```
void mesh_fem::set_finite_element(const dal::bit_vector &cvs,
                                getfem::pfem pf);
void mesh_fem::set_finite_element(getfem::pfem pf);
```

which set the finite elements for either the convexes listed in the `bit_vector` `cvs`, or all the convexes of the mesh. Note that the last method makes a call to the method

```
void mesh_fem::set_auto_add(pfem pf);
```

which defines the default finite element method which will be automatically added on new elements of the mesh (this is very usefull, for instance, when a refinement of the mesh is performed).

Descriptors for finite element methods and integration methods are available thanks to the following function

```
getfem::pfem pf = getfem::fem_descriptor("name of method");
```

where "name of method" is to be chosen among the existing methods. A name of a method can be retrieved thanks to the following functions

```
std::string femname = getfem::name_of_fem(pf);
```

A non exhaustive list (see Appendix A or `getfem/getfem_fem.h` for exhaustive lists) of finite element methods is given by

"FEM_PK(<i>n</i> , <i>k</i>)"	Classical P_K methods on simplexes of dimension <i>n</i> with degree <i>k</i> polynomials.
"FEM_QK(<i>n</i> , <i>k</i>)"	Classical Q_K methods on parallelepiped of dimension <i>n</i> . Tensorial product of degree <i>k</i> P_K method on the segment.
"FEM_PK_PRISM(<i>n</i> , <i>k</i>)"	Classical methods on prism of dimension <i>n</i> . Tensorial product of two degree <i>k</i> P_K method.
"FEM_PRODUCT(<i>a</i> , <i>b</i>)"	Tensorial product of the two polynomial finite element method <i>a</i> and <i>b</i> .
"FEM_PK_DISCONTINUOUS(<i>n</i> , <i>k</i>)"	discontinuous P_K methods on simplexes of dimension <i>n</i> with degree <i>k</i> polynomials.

An alternative way to obtain a Lagrange polynomial fem suitable for a given geometric transformation is to use

```
getfem::pfem getfem::classical_fem(bgeot::pgeometric_trans pg, short_type degree);
getfem::pfem getfem::classical_discontinuous_fem(bgeot::pgeometric_trans pg,
                                                short_type degree);
```

The `mesh_fem` can call directly these functions via:

```

void mesh_fem::set_classical_finite_element(const dal::bit_vector &cvvs,
                                             dim_type fem_degree);
void mesh_fem::set_classical_discontinuous_finite_element
    (const dal::bit_vector &cvvs, dim_type fem_degree);
void mesh_fem::set_classical_finite_element(dim_type fem_degree);
void mesh_fem::set_classical_discontinuous_finite_element(dim_type fem_degree);

```

Some other methods:

<code>mf.convex_index()</code>	Set of indexes (a <code>dal::bit_vector</code>) on which a finite element method is defined.
<code>mf.linked_mesh()</code>	gives a reference to the linked mesh.
<code>mf.fem_of_element(i)</code>	gives a descriptor on the finite element method defined on element of index <code>i</code> (does not take into account the <code>qdim</code> nor the optional reduction).
<code>mf.clear()</code>	Clears the structure, no finite element method is still defined.

3.2 Examples

For instance if one needs to have a description of a P_1 finite element method on a triangle, the way to set it is

```
mf.set_finite_element(i, getfem::fem_descriptor("FEM_PK(2, 1)"));
```

where `i` is still the index of the triangle. It is also possible to select a particular method directly on a set of element, passing to `mf.set_finite_element` a `dal::bit_vector` instead of a single index. For instance

```
mf.set_finite_element(mymesh.convex_index(),
                      getfem::fem_descriptor("FEM_PK(2, 1)"));
```

selects the method on all the elements of the mesh.

3.3 Second level: the optional “vectorization”

If the finite element represents an unknown which is a vector field, one should use `mf.set_qdim(Q)` to set the target dimension for the definition of the target dimension Q .

If the target dimension Q is set to a value different of 1, the scalar FEMs (such as P_k fems etc.) are automatically “vectorized” from the `mesh_fem` object point of vue. I.e. each scalar degree of freedom appears Q times in order to represent the Q components of the vector field. If an intrinsically vectorial element is used, the target dimension of the `fem` and the one of the `mesh_fem` object have to match. To sum it up,

- if the `fem` of the i th element is intrinsically a vector FEM, then


```
mf.get_qdim() == mf.fem_of_element(i)->target_dim()
and
mf.nb_dof_of_element(i) == mf.fem_of_element(i).nb_dof().
```

- if the fem has a `target_dim` equal to 1, then
`mf.nb_dof_of_element(i) == mf.get_qdim()*mf.fem_of_element(i).nb_dof()`.

At this level are defined the basic degrees of freedom. Some methods of the `getfem::mesh_fem` allows to obtain information on the basic dofs:

<code>mf.nb_basic_dof_of_element(i)</code>	gives the number of basic degrees of freedom on the element of index <code>i</code> .
<code>mf.ind_basic_dof_of_element(i)</code>	gives a container (an array) with all the global indexes of the basic degrees of freedom of element of index <code>i</code> .
<code>mf.point_of_basic_dof(i, j)</code>	gives a <code>bgeot::base_node</code> which represents the point associated with the basic dof of local index <code>j</code> on element of index <code>i</code> .
<code>mf.point_of_basic_dof(j)</code>	gives a <code>bgeot::base_node</code> which represents the point associated with the basic dof of global index <code>j</code> .
<code>mf.reference_point_of_basic_dof(i, j)</code>	gives a <code>bgeot::base_node</code> which represents the point associated with the basic dof of local index <code>j</code> on element of index <code>i</code> in the coordinates of the reference element.
<code>mf.first_convex_of_basic_dof(j)</code>	gives the index of the first element on which the basic degree of freedom of global index <code>j</code> is defined.
<code>mf.nb_basic_dof()</code>	gives the total number of different basic degrees of freedom.
<code>mf.get_qdim()</code>	gives the target dimension <code>Q</code> .
<code>mf.basic_dof_on_region(i)</code>	Return a <code>dal::bit_vector</code> which represents the indices of basic dof which are in the set of convexes or the set of faces of index <code>i</code> (see the <code>getfem::mesh</code> object).
<code>mf.dof_on_region(i)</code>	Return a <code>dal::bit_vector</code> which represents the indices of dof which are in the set of convexes or the set of faces of index <code>i</code> (see the <code>getfem::mesh</code> object). 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.

3.4 Third level: the optional linear transformation (or reduction)

As described above, it is possible to provide two matrices, a reduction matrix R and an extension matrix E which will describe a linear transformation of the degrees of freedom. If V is the vector of basic degrees of freedom, then $U = RV$ will be the vector of reduced degrees of freedom. Contrarily, given a vector U of reduced dof, $V = EU$ will correspond to a vector of basic dof. In simple cases, E will be simply the transpose of R . NOTE that every line of the extension matrix should be sparse. Otherwise, each assembled matrix will be plain !

A natural condition is that $RE = I$ where I is the identity matrix.

<code>mf.nb_dof()</code>	gives the total number of different degrees of freedom. If the optional reduction is used, this will be the number of columns of the reduction matrix. Otherwise it will return the number of basic degrees of freedom.
<code>mf.is_reduced()</code>	return a boolean. True if the reduction is used.
<code>mf.reduction_matrix()</code>	return a const reference to the reduction matrix R .
<code>mf.extension_matrix()</code>	return a const reference to the extension matrix E .
<code>mf.set_reduction_matrices(R, E)</code>	Set the reduction and extension matrices to R and E and validate their use.
<code>mf.set_reduction(b)</code>	Where b is a boolean. Cancel the reduction if b is false and validate it if b is true. If b is true, the extension and reduction matrices have to be set previously.
<code>mf.reduce_to_basic_dof(idof)</code>	Set the reduction and extension matrices corresponding to keep only the basic dofs present in <code>idof</code> . The parameter <code>idof</code> is either a <code>dal::bit_vector</code> or a <code>std::set<size_type></code> . This is equivalent to the use of a <code>getfem::partial_mesh_fem</code> object.

3.5 Obtaining generic mesh_fems

It is possible to use the function

```
const mesh_fem &getfem::classical_mesh_fem(const getfem::mesh &mymesh, dim_type K);
```

to get a classical polynomial `mesh_fem` of order K on the given `mymesh`. The returned `mesh_fem` will be destroyed automatically when its linked mesh is destroyed. All the `mesh_fem` built by this function are stored in a cache, which means that calling this function twice with the same arguments will return the same `mesh_fem` object. A consequence is that you should NEVER modify this `mesh_fem`!

3.6 The partial_mesh_fem object

The `getfem::partial_mesh_fem` object defined in the file `getfem_partial_mesh_fem.h` allows to reduce a `getfem::mesh_fem` object to a set of dofs. The interest is this is not a complete description of a finite element method, it refers to the original `getfem::mesh_fem` and just add reduction and extension matrices. For instance, you can reduce a `mesh_fem` obtained by the function `getfem::classical_mesh_fem(mesh, K)` to obtain a finite element method on a mesh region (which can be a boundary). The `getfem::partial_mesh_fem` is in particular used to obtain multiplier description to prescribed boundary conditions.

The declaration of a `getfem::partial_mesh_fem` object is the following:

```
getfem::partial_mesh_fem partial_mf(mf);
```

Then, one has to call the `adapt` method as follows:

```
partial_mf.adapt(kept_dof, rejected_elt = dal::bit_vector());
```

where `kept_dof` and `rejected_elt` are some `dal::bit_vector()`. `kept_dof` is the list of dof indices of the original `mesh_fem` `mf` to be kept. `rejected_elt` is an optional parameter that contains a list of element indices on which the `getfem::partial_mesh_fem` states that there is no finite element method. This is to avoid unnecessary computations during assembly procedures.

4 Selecting integration methods

The description of an integration method on a whole mesh is done thanks to the structure `getfem::mesh_im`, defined in the file `getfem/getfem_mesh_im.h`. Basically, this structure describes the integration method on each element of the mesh. One can instantiate a `getfem::mesh_im` object as follows

```
getfem::mesh_im mim(mymesh);
```

where `mymesh` is an already existing mesh. The structure will be linked to this mesh and will react when modifications will be done on it (for example when the mesh is refined, the integration method will be also refined).

It is possible to specify element by element the integration method, so that element of mixed types can be treated, even if the dimensions are different.

To select a particular integration method on a given element, one can use

```
mf.set_integration_method(i, ppi);
```

where `i` is the index of the element and `ppi` is the descriptor of the integration method. Alternative forms of this member function are:

```
void mesh_fem::set_integration_method(const dal::bit_vector &cvs,  
    getfem::pintegration_method ppi);  
void mesh_fem::set_integration_method(getfem::pintegration_method ppi);
```

which set the integration method for either the convexes listed in the `bit_vector` `cvs`, or all the convexes of the mesh.

The list of all available descriptors of integration methods is in the file `getfem/getfem_integration.h`.

Descriptors for integration methods are available thanks to the following function:

```
getfem::pintegration_method ppi =  
    getfem::int_method_descriptor("name of method");
```

where "name of method" is to be chosen among the existing methods. A name of a method can be retrieved with

```
std::string im_name = getfem::name_of_int_method(ppi);
```

A non exhaustive list (see Appendix B or `getfem/getfem_integration.h` for exhaustive lists) of integration methods is given below.

Examples of exact integration methods:

"IM_NONE()"	Dummy integration method (new in <code>getfem++-1.7</code>).
"IM_EXACT_SIMPLEX(n)"	Description of the exact integration of polynomials on the simplex of reference of dimension <code>n</code> .
"IM_PRODUCT(a, b)"	Description of the exact integration on the convex which is the direct product of the convex in <code>a</code> and in <code>b</code> .

"IM_EXACT_PARALLELEPIPED(<i>n</i>) "	Description of the exact integration of polynomials on the parallelepiped of reference of dimension <i>n</i>
"IM_EXACT_PRISM(<i>n</i>) "	Description of the exact integration of polynomials on the prism of reference of dimension <i>n</i>

Examples of approximated integration methods:

"IM_GAUSS1D(<i>k</i>) "	Description of the Gauss integration on a segment of order <i>k</i> . Available for all odd values of <i>k</i> ≤ 99 .
"IM_NC(<i>n</i> , <i>k</i>) "	Description of the integration on a simplex of reference of dimension <i>n</i> for polynomials of degree <i>k</i> with the Newton Cotes method (based on Lagrange interpolation).
"IM_PRODUCT(<i>a</i> , <i>b</i>) "	Build a method doing the direct product of methods <i>a</i> and <i>b</i> .
"IM_TRIANGLE(2) "	Integration on a triangle of order 2 with 3 points.
"IM_TRIANGLE(7) "	Integration on a triangle of order 7 with 13 points.
"IM_TRIANGLE(19) "	Integration on a triangle of order 19 with 73 points.
"IM_QUAD(2) "	Integration on quadrilaterals of order 2 with 3 points.
"IM_GAUSS_PARALLELEPIPED(2,3) "	Integration on quadrilaterals of order 3 with 4 points (shortcut for "IM_PRODUCT(IM_GAUSS1D(3), IM_GAUSS1D(3))").
"IM_TETRAHEDRON(5) "	Integration on a tetrahedron of order 5 with 15 points.

Remark: note that IM_QUAD(3) is not able to integrate exactly the base functions of the FEM_QK(2,3) finite element! Since its base function are tensorial product of 1D polynomials of degree 3, one would need to use IM_QUAD(7) (6 is not available). Hence IM_GAUSS_PARALLELEPIPED(2,*k*) should always be preferred over IM_QUAD(2**k*) since it has less integration points.

An alternative way to obtain integration methods:

```
getfem::pintegration_method
  getfem::classical_exact_im(bgeot::pgeometric_trans pgt);
getfem::pintegration_method
  getfem::classical_approx_im(bgeot::pgeometric_trans pgt, dim_type d);
```

These functions return an exact (i.e. analytical) integration method, or select an approximate integration method which is able to integrate exactly polynomials of degree $\leq d$ (at least) for convexes defined with the specified geometric transformation.

4.1 Methods of the mesh_im object

Once an integration method is defined on a mesh, it is possible to obtain information on it with the following methods (the list is not exhaustive).

<code>mim.convex_index()</code>	Set of indexes (a <code>dal::bit_vector</code>) on which an integration method is defined.
<code>mim.linked_mesh()</code>	gives a reference to the linked mesh.
<code>mim.int_method_of_element(i)</code>	gives a descriptor on the integration method defined on element of index <code>i</code> .
<code>mim.clear()</code>	Clear the structure. There are no further integration method defined on the mesh.

5 Mesh refinement

Mesh refinement with the Bank et al method (see [2]) is available in dimension 1, 2 or 3 for simplex meshes (segments, triangles and tetrahedrons). For a given object `mymesh` of type `getfem::mesh`, the method

```
mymesh.Bank_refine(bv);
```

refines the elements whose indices are stored in `bv` (a `dal::bit_vectorobject`). The conformity of the mesh is kept thanks to additional refinement (the so called green triangles). Information about green triangles is stored on the mesh object to gather them for further refinements (see [2]).

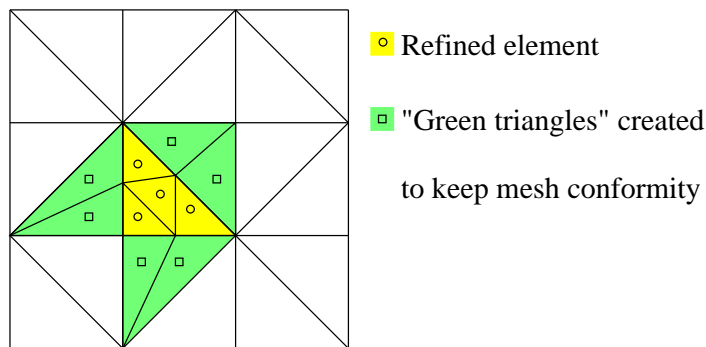


Figure 3: *Exemple of Bank refinement in 2D.*

Mesh refinement is most of the time coupled with an *a posteriori* error estimate. A basic error estimate is available in the file `getfem/getfem_error_estimate.h`.

```
error_estimate(mim, mf, U, err, cvlist);
```

where `mim` is the integration method (a `getfem::mesh_im` object), `mf` is the finite element method on which the unknown has been computed (a `getfem::mesh_fem` object), `U` is the vector of degrees of freedom of the unknown, `err` is a sufficiently large vector in which the error estimate is computed for each element of the mesh, and `cvlist` is a list of indices of element on which the error estimate should be computed (a `dal::bit_vectorobject`).

This basic error estimate is only valid for order two problems and just compute the sum of the jump in normal derivative across the elements on each edge (for two-dimensional problems) or each face (for three-dimensional problems). This means that for each face e of the mesh the following quantity is computed:

$$\int_e |[[\partial_n u]]|^2 d\Gamma,$$

where $[[\partial_n u]]$ is the jump of the normal derivative. Then, for each element the mean value is computed with respect to its faces and stored in the vector `err`. This basic error estimate can be taken as a model for more elaborated ones.

6 Linear algebra procedures

The linear algebra library used by GETFEM++ is Gmm++ which is now a separate library. Please see the GMM++ user documentation.

Note that Getfem++ includes (since release 1.7) its own version of SuperLU 3.0 (see SuperLU web site), hence a direct sparse solver is available out of the box. Note that an option of the `./configure` file allow to disable the included version of SuperLU in order to use a pre-installed version.

A small interface to MUMPS is also provided (see MUMPS web site <http://graal.ens-lyon.fr/MUMPS/> or <http://www.enseiht.fr/apo/MUMPS/>). See the file `gmm_MUMPS_interface.h`. In order to use MUMPS, you have to indicates some options to the configure shell:

```
MUMPS_CFLAGS=" -I /path/to/MUMPS/include "
MUMPS_LIBS=" F90 libraries and libs of MUMPS to be linked "
```

For instance if you want to use the sequential version of MUMPS with double and complex double:

```
MUMPS_CFLAGS=" -I /path/to/MUMPS/include "
MUMPS_LIBS=" ...F90libs... -L /path/to/MUMPS/lib -ldmumps -lzmumps -lpord
-L /path/to/MUMPS/libseq -lmpiseq "
```

where `...F90libs...` are the libraries of the fortran compiler used to compile MUMPS (these are highly dependant on the fortran 90 compiler used, the `./configure` script should detect the options relative to the default f90 compiler on your machine and display it – for example, with the intel `ifort` compiler, it is “`-L/opt/icc8.0/lib -lifport -lifcoremt -limf -lm -lcxa -lunwind -lpthread`”)

7 Standard assembly procedures

Procedures defined in the file `getfem/getfem_assembling.h` allow the assembly of stiffness matrices, mass matrices and boundary conditions for a few amount of classical partial differential equation problems. All the procedures have vectors and matrices template parameters in order to be used with any matrix library.

7.1 Laplacian (Poisson) problem

An assembling procedure is defined to solve the problem

$$\begin{aligned} \operatorname{div}(a(x) \operatorname{grad} u(x)) &= f(x), \quad \text{in } \Omega, \\ u(x) &= U(x), \quad \text{on } \Gamma_D, \\ \frac{\partial u}{\partial \mathbf{n}} &= F(x), \quad \text{on } \Gamma_N, \end{aligned}$$

where Ω is an open domain of arbitrary dimension, Γ_D and Γ_N are parts of the boundary of Ω , $u(x)$ is the unknown, $a(x)$ is a given coefficient, $f(x)$ is a given source term, $U(x)$ the prescribed value of $u(x)$

on Γ_D and $F(x)$ is the prescribed normal derivative of $u(x)$ on Γ_N . The function to be called to assemble the stiffness matrix is

```
getfem::asm_stiffness_matrix_for_laplacian(SM, mim, mfu, mfd, A);
```

where

- `SM` is a matrix of any type having the right dimension (i.e. `me1.nb_dof()`),
- `mim` is a variable of type `getfem::mesh_im` defining the integration method used,
- `mfu` is a variable of type `getfem::mesh_fem` and should define the finite element method for the solution,
- `mfd` is a variable of type `getfem::mesh_fem` (possibly equal to `mfu`) describing the finite element method on which the coefficient $a(x)$ is defined,
- `A` is the (real or complex) vector of the values of this coefficient on each degree of freedom of `mfd`.

Both `mesh_fem` should use the same mesh (i.e. `&mfu.linked_mesh() == &mfd.linked_mesh()`).

It is important to pay attention to the fact that the integration methods stored in `mim`, used to compute the elementary matrices, have to be chosen of sufficient order. The order has to be determined considering the polynomial degrees of element in `mfu`, in `mfd` and the geometric transformations for non-linear cases. For example, with linear geometric transformations, if `mfu` is a P_K FEM, and `mfd` is a P_L FEM, the integration will have to be chosen of order $\geq 2(K - 1) + L$, since the elementary integrals computed during the assembly of `SM` are $\int \nabla \varphi_i \nabla \varphi_j \psi_k$ (with φ_i the basis functions for `mfu` and ψ_i the basis functions for `mfd`).

To assemble the source term, the function to be called is

```
getfem::asm_source_term(B, mim, mfu, mfd, V);
```

where `B` is a vector of any type having the correct dimension (still `mfu.nb_dof()`), `mim` is a variable of type `getfem::mesh_im` defining the integration method used, `mfd` is a variable of type `getfem::mesh_fem` (possibly equal to `mfu`) describing the finite element method on which $f(x)$ is defined, and `V` is the vector of the values of $f(x)$ on each degree of freedom of `mfd`.

The function `asm_source_term` also has an optional argument, which is a reference to a `getfem::mesh_region` (or just an integer `i`, in which case `mim.linked_mesh().region(i)` will be considered). Hence for the Neumann condition on Γ_N , the same function

```
getfem::asm_source_term(B, mim, mfu, mfd, V, nbound);
```

is used again, with `nbound` is the index of the boundary Γ_N in the linked mesh of `mim`, `mfu` and `mfd`.

There is two manner (well not really, since it is also possible to use Lagrange multipliers, or to use penalization) to take into account the Dirichlet condition on Γ_D , changing the linear system or explicitly reduce to the kernel of the Dirichlet condition. For the first manner, the following function is defined

```
getfem::assembling_Dirichlet_condition(SM, B, mfu, nbound, R);
```

where `nbound` is the index of the boundary Γ_D where the Dirichlet condition is applied, `R` is the vector of the values of $R(x)$ on each degree of freedom of `mfu`. This operation should be the last one because it transforms the stiffness matrix `SM`. It works only for Lagrange elements. At the end, one obtains the discrete system

$$[SM]U = B,$$

where U is the discrete unknown.

For the second manner, one should use the more general

```
getfem::asm_dirichlet_constraints(H, R, mim, mf_u, mf_mult, mf_rh, $$$, nbound).
```

See the Dirichlet condition as a general linear constraint that must satisfy the solution u . This function does the assembly of Dirichlet conditions of type $\int_{\Gamma} u(x)v(x) = \int_{\Gamma} r(x)v(x)$ for all v in the space of multiplier defined by `mf_mult`. The fem `mf_mult` could be often chosen equal to `mf_u` except when `mf_u` is too “complex”.

This function just assemble these constraints into a new linear system $Hu = R$, doing some additional simplification in order to obtain a “simple” constraints matrix.

Then, one should call

```
ncols = getfem::Dirichlet_nullspace(H, N, R, Ud);
```

which will return a vector U_d which satisfies the Dirichlet condition, and an orthogonal basis N of the kernel of H . Hence, the discrete system that must be solved is

$$(N'[SM]N)U_{int} = N'(B - [SM]U_d),$$

and the solution is $U = NU_{int} + U_d$. The output matrix N should be a $nbdoof \times nbdoof$ (sparse) matrix but should be resized to `ncols` columns. The output vector U_d should be a $nbdoof$ vector. A big advantage of this approach is to be generic, and do not prescribed for the finite element method `mf_u` to be of Lagrange type. If `mf_u` and `mf_d` are different, there is implicitly a projection (with respect to the L^2 norm) of the data on the finite element `mf_u`.

If you want to treat the more general scalar elliptic equation $\text{div}A(x)\nabla u$, where $A(x)$ is square matrix, you should use

```
getfem::asm_stiffness_matrix_for_scalar_elliptic(M, mim, mfu, mfd, A);
```

The matrix data `A` should be defined on `mfd`. It is expected as a vector representing a $n \times n \times nbdoof$ tensor (in Fortran order), where n is the mesh dimension of `mfu`, and $nbdoof$ is the number of dof of `mfd`.

7.2 Linear Elasticity problem

The following function assembles the stiffness matrix for linear elasticity

```
getfem::asm_stiffness_matrix_for_linear_elasticity(SM, mim, mfu, mfd, LAMBDA, MU);
```

where `SM` is a matrix of any type having the right dimension (i.e. here `me1.nb_dof()`), `mim` is a variable of type `getfem::mesh_im` defining the integration method used, `mfu` is a variable of type `getfem::mesh_fem` and should define the finite element method for the solution, `mfd` is a variable of type `getfem::mesh_fem` (possibly equal to `mfu`) describing the finite element method on which the Lamé coefficients are defined, `LAMBDA` and `MU` are vectors of the values of Lamé coefficients on each degree of freedom of `mfd`.

CAUTION: Linear elasticity problem is a vectorial problem, so the target dimension of `mfu` (see `mf.set_qdim(Q)`) should be the same as the dimension of the mesh.

In order to assemble source term, Neumann and Dirichlet conditions, same functions as in previous section can be used.

7.3 Stokes Problem with mixed finite element method

The assembly of the mixed term $B = -\int p \nabla \cdot v$ is done with:

```
getfem::asm_stokes_B(MATRIX &B, const mesh_im &mim,
                    const mesh_fem &mf_u, const mesh_fem &mf_p);
```

7.4 Assembling a mass matrix

Assembly of a mass matrix between two finite elements:

```
getfem::asm_mass_matrix(M, mim, mf1, mf2);
```

It is also possible to obtain mass matrix on a boundary with the same function:

```
getfem::asm_mass_matrix(M, mim, mf1, mf2, nbound);
```

where `nbound` is the region index in `mim.linked_mesh()`, or a `mesh_region` object.

8 Compute arbitrary elementary matrices - generic assembly procedures

As it can be seen in the file `getfem/getfem_assembling.h`, all the previous assembly procedures use a `getfem::generic_assembly` object and provide it an adequate description of what must be done. For example, the assembly of a volumic source term for a scalar FEM is done with the following excerpt of code:

```
getfem::generic_assembly assem;
assem.push_im(mim);
assem.push_mf(mf);
assem.push_mf(mfdata);
assem.push_data(F);
assem.push_vec(B);
assem.set("Z=data(#2); V(#1)+=comp(Base(#1).Base(#2))(:,j).Z(j);");
assem.assembly();
```

The first instructions declare the object, and set the data that it will use: a `mesh_im` object which holds the integration methods, two `mesh_fem` objects, the input data `F`, and the destination vector `B`.

The input data is the vector F , defined on `mfdata`. One wants to evaluate $\sum_j f_j(\int_{\Omega} \phi^i \psi^j)$. The instruction must be seen as something that will be executed for each convex `cv` of the mesh. The terms `#1` and `#2` refer to the first `mesh_fem` and the second one (i.e. `mf` and `mfdata`). The instruction `Z=data(#2)`; means that for each convex, the “tensor” `Z` will receive the values of the first data argument provided with `push_data`, at indexes corresponding to the degrees of freedom attached to the convex of the second (`#2`) `mesh_fem` (here, `Z = F[mfdata.ind_dof_of_element(cv)]`).

The part `V(#1)+=...` means that the result of the next expression will be accumulated into the output vector (provided with `push_vec`). Here again, `#1` means that we will write the result at indexes corresponding to the degrees of freedom of the current convex with respect to the first (`#1`) `mesh_fem`.

The right hand side `comp(Base(#1).Base(#2))(:,j).Z(j)` contains two operations. The first one is a computation of a tensor on the convex: `comp(Base(#1).Base(#2))` is evaluated as a 2-dimensions tensor, $\int \phi^i \psi^j$, for all degrees of freedom i of `mf` and j of `mfdata` attached to the current convex. The next part is a reduction operation, `C(:,j).Z(j)`: each named index (here j) is summed, i.e. the result is $\sum_j c_{i,j} z_j$.

The integration method used inside `comp(Base(#1).Base(#2))` is taken from `mim`. If you need to use integration methods from another `mesh_im` object, you can specify it as the first argument of `comp`, for example `comp(%2, Base(#1).Grad(#2))` will use the second `mesh_im` object (New in `getfem++-2.0`).

An other example is the assembly of the stiffness matrix for a vector Laplacian:

```
getfem::generic_assembly assem;
assem.push_im(mim);
assem.push_mf(mf);
assem.push_mf(mfdata);
assem.push_data(A);
assem.push_mat(SM);
assem.set("a=data$1(#2);"
         "M$1(#1,#1)+=sym(comp(vGrad(#1).vGrad(#1).Base(#2))(:,j,k, :,j,k,p).a(p))");
assem.assembly();
```

Now the output is written in a sparse matrix, inserted with `assem.push_mat(SM)`. The `$1` in `M$1(#1,#1)` just indicates that we refer to the first matrix “pushed” (it is optional, but if the assembly builds two matrices, the second one must be referred this way). The `sym` function ensure that the result is symmetric (if this is not done, some round-off errors may cancel the symmetricity, and the assembly will be a little bit slower). Next, the `comp` part evaluates a 7D tensor,

$$\int \partial_k \varphi_j^i \partial_l \varphi_n^m \psi^p,$$

where φ_j^i is a j th component of the i th base function of `mf` and ψ^p is a (scalar) base function of the second `mesh_fem`. Since we want to assemble

$$\int a(x) \cdot \nabla \phi^i \cdot \nabla \phi^j, \quad \text{with} \quad a(x) = \sum_p a^p \psi^p(x),$$

the reduction is:

$$\sum_{j,k,p} \left(\int \partial_k \varphi_j^i \partial_k \varphi_j^m \psi^p \right) a^p$$

In the `comp` function, `vGrad` was used instead of `Grad` since we said that we were assembling a *vector* Laplacian: that is why each `vGrad` part has three dimensions (dof number, component number, and derivative

number). For a scalar Laplacian, we could have used `comp(Grad(#1).Grad(#1).Base(#2))(:,k,:,k,p).a(p)`. But the vector form has the advantage to work in both vector and scalar case.

The last instruction, `assem.assembly()`, does evaluate the expression on each convex. For an assembly over a boundary just call `assem.assembly(rg)`, where `rg` is a `getfem::mesh_region` object. `rg` might also be a number, in that case the mesh region taken into account is `mim.linked_mesh().region(rg)`.

The third example shows how to compute the L^2 norm of a scalar or vector field on a mesh boundary:

```
assem.push_im(mim);
assem.push_mf(mf);
assem.push_data(U);
std::vector<scalar_type> v(1);
assem.push_vec(v);
assem.set("u=data(#1); V(+=u(i).u(j).comp(vBase(#1).vBase(#1))(i,k,j,k)");
assem.assembly(boundary_number);
```

This one is easy to read. When `assembly` returns, `v[0]` will contain

$$\sum_{i,j,k} \left(\int_{boundary} u_i \varphi_k^i u_j \varphi_k^j \right)$$

The fourth and last example shows an (sub-optimal) assembly of the linear elasticity problem with a complete Hooke tensor:

```
assem.set("h=data$1(qdim(#1),qdim(#1),qdim(#1),qdim(#1),#2);"
         "t=comp(vGrad(#1).vGrad(#1).Base(#2));"
         "e=(t{: ,2,3, ,5,6, :}+t{: ,3,2, ,5,6, :}+t{: ,2,3, ,6,5, :}+t{: ,3,2, ,6,5, :})/4;"
         "M(#1,#1)+= sym(e(:,j,k, ,m,n,p).h(j,k,m,n,p))");
```

The original equations are:

$$\int \varepsilon(\varphi^i) : \sigma(\phi^j), \quad \text{with} \quad \sigma(u)_{ij} = \sum_{kl} h_{ijkl}(x) \varepsilon_{kl}(u)$$

where h is the Hooke tensor, and $\cdot :$ means the scalar product between matrices. Since we assume it is not constant, h is given on the second `mesh_fem`: $h_{ijkl}(x) = \sum_p h_{ijkl}^p \psi^p$. Hence the first line declares that the first data “pushed” is indeed a five-dimensions tensor, the first fourth ones being all equal to the target dimension of the first `mesh_fem`, and the last one being equal to the number of degrees of freedom of the second `mesh_fem`. The `comp` part still computes the same 7D tensor than for the vector Laplacian case. From this tensor, one evaluates $\varepsilon(\varphi^i)_{jk} \varepsilon(\phi^l)_{mn} \psi^p$ via permutations, and finally the expression is reduced against the hook tensor.

available operations inside the comp command

- `Base(#i)`: evaluate the value of the base functions of the *ith* `mesh_fem`
- `Grad(#i)`: evaluate the value of the gradient of the base functions of the *ith* `mesh_fem`
- `Hess(#i)`: evaluate the value of the Hessian of the base functions of the *ith* `mesh_fem`
- `Normal()`: evaluate the unit normal (should not be used for volumic integrations !)

- `NonLinear$x(#mf1,...#mfn)`: evaluate the x th non-linear term (inserted with `push_nonlinear_term(pnonlinear_elem_term)`) using the listed `mesh_fem` objects.
- you may reference any data object inside the `comp` command, and perform reductions inside the `comp()`. This feature is mostly interesting for speeding up assembly of nonlinear terms (see the file `getfem/getfem_nonlinear_elasticity.h` for an example of use).
- `GradGT()`, `GradGTInv`: evaluate the gradient (and its inverse) of the geometric transformation of the current convex.

others operations

Slices may be mixed with reduction operations `t(:,4,i,i)` takes a slice at index 4 of the second dimension, and reduces the diagonal of dimension 3 and 4. *Please note that index numbers for slices start at 1 and not 0 !!*

`mdim(#2)` is evaluated as the mesh dimension associated to the second `mesh_fem`, while `qdim(#2)` is the target dimension of the `mesh_fem`.

The diagonal of a tensor can be obtained with `t{:, :, 3, 3}` (which is strictly equivalent to `t{1, 2, 3, 3}`: the colon is just here to improve the readability). This is the same operator than for permutation operations. Note that `t{:, :, 1, 1}` or `t{:, :, 4, 4}` are not valid operations.

The `print` command can be used to see the tensor: `"print comp(Base(#1));"` will print the integrals of the base functions for each convex.

If there is more than one data array, output array or output sparse matrix, one can use `data$2`, `data$3`, `V$2`, `M$2`,...

9 Incorporate new finite element methods in GETFEM++

Basically, It is sufficient to describe an element on the reference element, i.e. to describe each base function of each degree of freedom. Intrinsically vectorial elements are supported (see for instance Nedelec and Raviart-Thomas elements). Finite element methods that are not equivalent via the geometric transformation (not τ -equivalent in GETFEM++ jargon, such as vectorial elements, hermite elements ...) an additional linear transformation of the degrees of freedom depending on the real element should be described (see the implementation of Argyris element for instance).

Please read [6] for more details and see the files `getfem/getfem_fem.h`, `getfem/getfem_fem.cc` for practical implementation.

10 Incorporate new approximated integration methods in GETFEM++

A perl script automatically incorporates new cubature methods from a description file. You can see in the directory `cubature` such description files (with extension `.IM`). For instance for `IM_TETRAHEDRON(5)` the following file describes the method:

```
NAME = IM_TETRAHEDRON(5)
N = 3
GEOTRANS = GT_PK(3,1)
NBPT = 4
0, 0.25, 0.25, 0.25, 0.008818342151675485
1, 0.31979362782962991, 0.31979362782962991, 0.31979362782962991, 0.011511367871045398
```

```

1, 0.091971078052723033, 0.091971078052723033, 0.091971078052723033, 0.01198951396316977
1, 0.056350832689629156, 0.056350832689629156, 0.44364916731037084, 0.008818342151675485
NBF = 4
IM_TRIANGLE(5)
IM_TRIANGLE(5)
IM_TRIANGLE(5)
IM_TRIANGLE(5)

```

where `NAME` is the name of the method in `GETFEM++` (constant integer parameter are allowed), `N` is the dimension, `GEOTRANS` describes a valid geometric transformation of `GETFEM++`. This geometric transformation just defines the reference element on which the integration method is described. `NBPT` is the number of integration node definitions. Integration node definitions include a symmetry definition such that the total number of integration nodes would be greater than `NBPT`.

Composition of the integration node definition:

- an integer: 0 = no symmetry, 1 = full symmetric (x6 for a triangle, x4 for a quadrangle, x24 for a tetrahedron ...),
- the `N` coordinates of the integration node,
- the load.

`NBF` is the number of faces of the reference element (should correspond to `GEOTRANS`). Then follows an already existing integration method for each face (each on a line). This is necessary to make integrations on boundaries.

The file format is inspired from [3].

11 Level-sets, Xfem, fictitious domains

`GETFEM++` offers (since v2.0) a certain number of functionalities concerning level-sets, support for Xfem and fictitious domain methods and discontinuous field across a level-set.

Important: All the tools listed below needs the package `qhull` installed on your system. This package is widely available. It computes convex hull and delaunay triangulations in arbitrary dimension. Everything here is considered “work in progress”, it is still subject to major changes if needed.

The program `crack.cc` on the `tests` directory of the distribution is a good example of use of these tools.

11.1 Representation of level-sets

`GETFEM++` deals with level-set defined by piecewise polynomial function on a mesh. It will be defined as the zero of this function. In the file `getfem/getfem_level_set.h` a level-set is represented by a function defined on a lagrange fem of a certain degree on a mesh. The constructor to define a new `getfem::level_set` is the following:

```
getfem::level_set ls(mesh, degree = 1, with_secondary = false);
```

where `mesh` is a valid mesh of type `getfem::mesh`, `degree` is the degree of the polynomials (1 is the default value), and `with_secondary` is a boolean whose default value is false. The secondary level-set is used to represent fractures (if $p(x)$ is the primary levelset function and $s(x)$ is the secondary levelset function, the crack is defined by $p(x) = 0$ and $s(x) \leq 0$: the role of the secondary is to stop the crack).

Each level-set function is defined by a mesh_fem `mf` and the dof values over this mesh_fem, in a vector. The object `getfem::level_set` contains a mesh_fem and the vectors of dof for the corresponding function(s). The method `ls.value(0)` returns the vector of dof for the primary level-set function, so that these values can be set. The method `ls.value(1)` returns the dof vector for the secondary level-set function if any. The method `ls.get_mesh_fem()` returns a reference on the `getfem::mesh_fem` object.

11.2 Mesh cut by level-sets

In order to compute adapted integration methods and finite element methods to represent a field which is discontinuous across a level-set, a certain number of pre-computations have to be done at the mesh level. The file `getfem/getfem_mesh_level_set.h` defines the object `getfem::mesh_level_set` which handles these pre-computations. The constructor of this object is the following:

```
getfem::mesh_level_set mls(mesh);
```

where `mesh` is a valid mesh of type `getfem::mesh`. In order to indicate that the mesh is cut by a level-set, one has to call the method `mls.add_level_set(ls)`, where `ls` is an object of type `getfem::level_set`. An arbitrary number of level-sets can be added. To initialize the object or to actualize it when the value of the level-set function is modified, one has to call the method `mls.adapt()`.

In particular a subdivision of each element cut by the level-set is made with simplices.

11.3 Adapted integration methods

For fields which are discontinuous across a level-set, integration methods have to be adapted. The object `getfem::mesh_im_level_set` defined in the file `getfem/getfem_mesh_im_level_set.h` defines a composite integration method for the elements cut by the level-set. The constructor of this object is the following:

```
getfem::mesh_im_level_set mim(mls, where, regular_im = 0, singular_im = 0);
```

where `mls` is an object of type `getfem::mesh_level_set`, `where` is an enum for which possible values are

- `getfem::mesh_im_level_set::INTEGRATE_INSIDE` (integrate over $p(x) < 0$),
- `getfem::mesh_im_level_set::INTEGRATE_OUTSIDE` (integrate over $p(x) > 0$),
- `getfem::mesh_im_level_set::INTEGRATE_ALL`,
- `getfem::mesh_im_level_set::INTEGRATE_BOUNDARY` (integrate over $p(x) = 0$ and $s(x) \leq 0$)

The argument `regular_im` should be of type `pintegration_method`, and will be the integration method applied on each sub-simplex of the composite integration for convexes cut by the levelset. The optional `singular_im` should be also of type `pintegration_method` and is used for crack singular functions: it is applied to sub-simplices which share a vertex with the crack tip (the specific integration method `IM_QUASI_POLAR(...)` is well suited for this purpose).

The object `getfem::mesh_im_level_set` can be used as a classical `getfem::mesh_im` object (for instance the method `mim.set_integration_method(...)` allows to set the integration methods for the elements which are not cut by the level-set).

To initialize the object or to actualize it when the value of the level-set function is modified, one has to call the method `mim.adapt()`.

11.4 Discontinuous field across some level-sets

The object `getfem::mesh_fem_level_set` is defined in the file `getfem/getfem_mesh_fem_level_set.h`. It is derivated from `getfem::mesh_fem` object and can be used in the same way. It defines a finite element method with discontinuity across the level-sets (it can deal with an arbitrary number of level-sets). The constructor is the following:

```
getfem::mesh_fem_level_set mfls(mesh, mf);
```

where `mesh` is a valid mesh of type `getfem::mesh` and `mf` is the an object of type `getfem::mesh_fem` which defines the finite element method used for elements which are not cut by the level-sets.

To initialize the object or to actualize it when the value of the level-set function is modified, one has to call the method `mfls.adapt()`.

To represent discontinuous fields, the finite element method is enriched with discontinuous functions which are the product of a Heaviside function by the base functions of the finite element method represented by `mf` (see [4] for more details).

11.5 Fictitious domain approach with Xfem

An example of a Poisson problem with a Dirichlet condition posed on a boundary independant of the mesh is present on the `tests` directory of the distribution. See `xfem_dirichlet.cc` file.

12 Support for Xfem methods

(outdated, to be done again ...)

Xfem are finite element method with a particular enrichment with non-polynomials functions (see [4] for instance). The file `getfem/getfem_Xfem.h` gives a support for this kind of method. Any (τ -equivalent) valid finite element method can be extended. If `pf` is a valid descriptor of a finite element method, one can build a Xfem with the declaration

```
Xfem xf(pf);
```

then one adds a global function with

```
xf.add_function(pXf, pXg, pXh, ind);
```

where `pXf` should be a pointer on a type derived from the object `virtual_Xfem_func` representing the global function, `pXg` should be a pointer on a type derived from the object `virtual_Xfem_grad` representing the global function gradient, `pXh` is an optional parameter (only for fourth order derivative problems) which should be a pointer on a type derived from the object `virtual_Xfem_hess` representing the global

function Hessian and `ind` is an index which should correspond to this function to identify the degrees of freedom (this should be different for each function added). It is possible to add an arbitrary number of global functions. The total number of degrees of freedom of the Xfem is the number of degrees of freedom of the initial fem times $N_f + 1$ where N_f is the number of global functions added.

If φ_i for $i = 1..N_d$ are the basis functions of `pf` and f_j for $j = 1..N_f$ the additional global functions, the basis functions of the Xfem are the basis function φ_i for $i = 1..N_d$ and the basis functions $f_j\varphi_i$ for $i = 1..N_d$ and $j = 1..N_f$. From an element to another and for each function f_j , the corresponding degrees of freedom are connected in a same manner as the corresponding degrees of freedom of `pf`.

Most of the time, one only needs an enrichment on a subset of node of the original element. If it is so, one should a posteriori eliminate the unwanted extra-dofs.

13 Interpolation of a finite element method on non-matching meshes

A special finite element method is defined in `getfem/getfem_interpolated_fem.h` which is not a real finite element method, but a pseudo-fem which interpolates a finite element method defined on another mesh. If you need to assemble a matrix with finite element methods defined on different meshes, you may use the “interpolated fem” for that purpose:

```
getfem::new_interpolated_fem(getfem::mesh_fem mf, getfem::mesh_im mim)
```

Because each base function of the finite element method has to be interpolated, such a computation can be a heavy procedure. By default, the interpolated fem object store the interpolation data.

The interpolation is made on each Gauss point of the integration methods of `mim`, so that you have to use these integration methods in the assembling procedures.

For instance if you need to compute the mass matrix between to different finite element methods defined on two different meshes, this is an example of code which interpolate the second f.e.m. on the mesh of the first f.e.m., assuming that `mf` describes the finite element method and `mim` is the chosen integration method.

```
getfem::mesh_fem mf_interpole(mfu.linked_mesh());
pfem ifem = getfem::new_interpolated_fem(mf, mim);
dal::bit_vector nn = mfu.convex_index();
mf_interpole.set_finite_element(nn, ifem);
getfem::asm_mass_matrix(SM1, mim, mfu, mf_interpole);
del_interpolated_fem(ifem);
```

The object pointed by `ifem` contains all the information concerning the interpolation. It could use a lot of memory. As `pfem` is a smart pointer (a boost intrusive_ptr), the interpolated fem will be automatically destroyed when the last pointer on it is destroyed. To obtain a better accuracy, it is better to refine the integration method (with `IM_STRUCTURED_COMPOSITE` for instance) rather than increase its order.

13.1 mixed methods with different meshes

to be described ...

13.2 mortar methods

to be described ...

14 Compute L^2 and H^1 norms

The file `getfem/getfem_assembling.h` defines the functions to compute L^2 and H^1 norms of a solution. The following functions compute the different norms

```
getfem::asm_L2_norm(mim, mf, U);
getfem::asm_H1_semi_norm(mim, mf, U);
getfem::asm_H1_norm(mim, mf, U);
```

where `mim` is a `getfem::mesh_im` used for the integration, `mf` is a `getfem::mesh_fem` and describes the finite element method on which the solution is defined, `U` is the vector of values of the solution on each degree of freedom of `mf`. The size of `U` should be `mf.nb_dof()`.

In order to compare two solutions, it is often simpler and faster to use the following function than to interpolate one `mesh_fem` on another:

```
getfem::asm_L2_dist(mim, mf1, U1, mf2, U2);
getfem::asm_H1_dist(mim, mf1, U1, mf2, U2);
```

These functions return the L^2 and H^1 norms of $u_1 - u_2$.

15 Compute derivatives

The file `getfem/getfem_derivatives.h` defines the following function to compute the gradient of a solution

```
getfem::compute_gradient(mf1, mf2, U, V);
```

where `mf1` is a variable of type `getfem::mesh_fem` and describes the finite element method on which the solution is defined, `mf2` describes the finite element method to compute the gradient, `U` is a vector representing the solution and should be of size `mf1.nb_dof()`, `V` is the vector on which the gradient will be computed and should be of size `N * mf2.nb_dof()`, with `N` the dimension of the domain. IMPORTANT: This function only works when `mf2` is a Lagrange element. This element should be, most of the time, a discontinuous Lagrangian element, because for usual element (for instance `getfem::FEM_PK_DISCONTINUOUS(n, k)`), the gradient is not continuous.

16 Export and view a solution

There are essentially three ways to view the result of `getfem` computations:

- Matlab, with the `matlab-interface`.
- the open-source Mayavi or any other VTK files viewer.
- the open-source OpenDX program.

The objects that can be exported are, meshes, `mesh_fem` objects, and mesh slices.

16.1 Saving mesh and mesh_fem objects for the Matlab interface

If you have installed the Matlab interface, you can simply use `mesh_fem::write_to_file` and save the solution as a plain text file, and then, load them into Matlab. For example, supposing you have a solution `U` on a `mesh_fem` `mf`,

```
std::fstream f("solution.U",std::ios::out);
for (unsigned i=0; i < gmm::vect_size(U); ++i)
    f << U[i] << "\n";

// when the 2nd arg is true, the mesh is saved with the mesh_fem
mf.write_to_file("solution.mf", true);
```

and then, under matlab:

```
>> U=load('solution.U');
>> mf=gfMeshFem('load','solution.mf');
>> gf_plot(mf,U,'mesh','on');
```

See the `getfem-matlab` interface documentation for more details.

Two other file formats are supported for export: the VTK file format and the OpenDX file format. Both can export either a `getfem::mesh` or `getfem::mesh_fem`, but also the more versatile `getfem::stored_mesh_slice`.

Examples of use can be found in the examples of the tests directory.

16.2 Producing mesh slices

`Getfem++` provides “slicers” objects which are dedicated to generating post-treatment data from meshes and solutions. These slicers, defined in the file `getfem/getfem_mesh_slicers.h` take a mesh (and sometimes a `mesh_fem` with a solution field) on input, and produce a set of simplices after applying some operations such as “intersection with a plane”, “extraction of the mesh boundary”, “refinement of each convex”, “extraction of isosurfaces”, etc. The output of these slicers can be stored in a `getfem::stored_mesh_slice` object (see the file `getfem/getfem_mesh_slice.h`). A `stored_mesh_slice` object may be considered as a P1 discontinuous FEM on a non-conformal mesh with fast interpolation ability. Slices are made of segments, triangles and tetrahedrons, so the convexes of the original mesh are always simplexified.

All slicer operation inherit from `getfem::slicer_action`, it is very easy to create a new slicer. Example of slicers are (some of them use a `getfem::mesh_slice_cv_dof_data_base` which is just a reference to a `mesh_fem` `mf` and a field `U` on this `mesh_fem`).

<code>getfem::slicer_none</code>	empty slicer.
<code>getfem::slicer_boundary</code> (const mesh &m, ...)	extract the boundary of a mesh.
<code>getfem::slicer_apply_deformation</code> (mesh_slice_cv_dof_data_base &)	apply a deformation to the mesh, the deformation field is defined on a <code>mesh_fem</code> .
<code>getfem::slicer_half_space</code> (base_node x0, base_node n, int orient)	cut the mesh with a half space (if <code>orient = -1</code> or <code>+1</code>), or a plane (if <code>orient = 0</code>), <code>x0</code> being a node of the plane, and <code>n</code> being a normal of the plane.

<code>getfem::slicer_sphere (base_node x0, scalar_type R, int orient)</code>	cut with the interior (<code>orient=-1</code>), boundary (<code>orient=0</code>) or exterior (<code>orient=+1</code>) or a sphere of center <code>x0</code> and radius <code>R</code> .
<code>getfem::slicer_cylinder (base_node x0, base_node x1, scalar_type R, int orient)</code>	slice with the interior/boundary/exterior of a cylinder of axis <code>(x0,x1)</code> and radius <code>R</code> .
<code>getfem::slicer_isovalues (const mesh_slice_cv_dof_data_base& mfU, scalar_type val, int orient)</code>	cut with the isosurface defined by the scalar field <code>mfU</code> and <code>val</code> . Keep only simplices where $u(x)_i \text{val}$ (<code>orient=-1</code>), $u(x)_i = \text{cppval}$ (<code>orient=0</code> or $u(x)_i > \text{val}$).
<code>getfem::slicer_mesh_with_-mesh (const mesh& m2)</code>	cut the convexes with the convexes of the mesh <code>m2</code> .
<code>getfem::slicer_union (const slicer_action &sA, const slicer_action &sB)</code>	merges the output of two slicer operations.
<code>getfem::slicer_intersect (slicer_action &sA, slicer_action &sB)</code>	intersect the output of two slicer operations.
<code>getfem::slicer_-complementary (slicer_action &s)</code>	return the complementary of a slicer operation.
<code>getfem::slicer_build_edges_-mesh (mesh& edges_m)</code>	slicer whose side-effect is to build the mesh <code>edges_m</code> with the edges of the sliced mesh. in some (rare) occasions , it might be useful to build a mesh from a slice. Note however that there is absolutely no guaranty that the mesh will be conformal (although it is often the case).
<code>getfem::slicer_build_mesh (mesh &m)</code>	record the output of the slicing operation into a <code>stored_mesh_slice</code> object. Note that it is often more convenient to use the <code>stored_mesh_slice::build(...)</code> method to achieve the same result.
<code>getfem::slicer_build_stored_mesh_slice (stored_mesh_slice& sl)</code>	shrink or expand each convex with respect to its gravity center.
<code>getfem::slicer_explode (c)</code>	

In order to apply these slicers, a `getfem::mesh_slicer (mesh&m)` object should be created, and the `getfem::slicer_action` are then stacked with `mesh_slicer::push_back_action(slicer_action&)` and `mesh_slicer::push_front_action(slicer_action&)`. The slicing operation is finally executed with `mesh_slicer::exec(int nrefine)` (or `mesh_slicer::exec(int nrefine, const mesh_region &cvlst)` to apply the operation to a subset of the mesh, or its boundary etc.).

The `nrefine` parameter is very important, as the “precision” of the final result will depend on it: if the data that is represented on the final slice is just P1 data on convexes with a linear geometric transformation, `nrefine = 1` is the right choice, but for P2, P3, non linear transformation etc, it is better to refine each convex of the original mesh during the slicing operation. This allows an accurate representation of any finite element field onto a very simple structure (linear segment/triangles/tetrahedrons with P1 discontinuous data on them) which is what most visualization programs (mayavi, opendx, matlab, etc.) expect.

Example of use (cut the boundary of a mesh `m` with a half-space, and save the result into a `stored_mesh_slice`):

```
getfem::slicer_boundary a0(m);
getfem::slicer_half_space a1(base_node(0,0), base_node(1, 0), -1);
getfem::stored_mesh_slice sl;
getfem::slicer_build_stored_mesh_slice a2(sl);
```

```

getfem::mesh_slicer slicer(m);
slicer.push_back_action(a1);
slicer.push_back_action(a2);
int nrefine = 3;
slicer.exec(nrefine);

```

In order to build a `getfem::stored_mesh_slice` object during the slicing operation, the `stored_mesh_slice::build()` method is often more convenient than using explicitly the `slicer_build_stored_mesh_slice` slicer:

```

getfem::stored_mesh_slice sl;
sl.build(m, getfem::slicer_boundary(m),
         getfem::slicer_half_space(base_node(0,0), base_node(1, 0), -1),
         nrefine);

```

The simplest way to use these slices is to export them to VTK or opendx. The file `getfem/getfem_export.h` contains two classes: `getfem::vtk_export` and `getfem::dx_export`.

16.3 Exporting mesh, mesh_fem or slices to VTK

First, it is important to know the limitation of VTK data files: each file can contain only one mesh, with at most one scalar field and one vector field and one tensor field on this mesh (in that order). VTK files can handle data on segment, triangles, quadrangles, tetrahedrons and hexahedrons. Although quadratic triangles, segments etc are said to be supported, it is just equivalent to using `nrefine=2` when building a slice. VTK data file do support meshes with more than one type of element (i.e. meshes with triangles and quadrangles, for example).

For example, supposing that a `stored_mesh_slice sl` has already been built:

```

// an optional the 2nd argument can be set to true to produce
// a text file instead of a binary file
vtk_export exp("output.vtk");
exp.exporting(sl); // will save the geometrical structure of the slice
exp.write_point_data(mfp, P, "pressure"); // write a scalar field
exp.write_point_data(mfu, U, "displacement"); // write a vector field

```

In this example, the fields P and U are interpolated on the slice nodes, and then written into the VTK field. The vector fields should always be written after the scalar fields (and the tensor fields should be written last).

It is also possible to export a `mesh_fem` without having to build a slice:

```

// an optional the 2nd argument can be set to true to produce
// a text file instead of a binary file
vtk_export exp("output.vtk");
exp.exporting(mfu);
exp.write_point_data(mfp, P, "pressure"); // write a scalar field
exp.write_point_data(mfu, U, "displacement"); // write a vector field

```

Note however that with this approach, the `vtk_export` will map each convex/fem of `mfu` to a VTK element type. As VTK does not handle elements of degree greater than 2, there will be a loss of precision for higher degree FEMs.

16.4 Exporting mesh, mesh_fem or slices to OpenDX

The `opendx` data file is more versatile than the VTK one. It is able to store more than one mesh, any number of fields on these meshes etc. However, it does only handle elements of degree 1 and 0 (segments, triangles, tetrahedrons, quadrangles etc.). And each mesh can only be made of one type of element, it cannot mix triangles and quadrangles in a same object. For that reason, it is generally preferable to export `getfem::stored_mesh_slice` objects (in which non simplex elements are simplexified, and which allows refinement of elements) than `getfem::mesh_fem` and `getfem::mesh` objects.

The basic usage is very similar to `getfem::vtk_export`:

```
getfem::dx_export exp("output.dx");
exp.exporting(sl);
exp.write_point_data(mfu, U, "displacement");
```

Moreover, `getfem::dx_export` is able to reopen a '.dx' file and append new data into it. Hence it is possible, if many time-steps are to be saved, to view intermediate results in OpenDX during the computations. The prototype of the constructor is:

```
dx_export(const std::string& filename, bool ascii = false, bool append = false);
dx_export(std::ostream &os_, bool ascii = false);
```

An example of use, with multiple time steps (taken from `tests/dynamic_friction.cc`):

```
getfem::stored_mesh_slice sl;
getfem::dx_export exp("output.dx", false);
if (N <= 2) sl.build(mesh, getfem::slicer_none(),4);
else      sl.build(mesh, getfem::slicer_boundary(mesh),4);
exp.exporting(sl,true);

// for each mesh object, a corresponding ‘‘mesh’’ object will be
// created in the data file for the edges of the original mesh
exp.exporting_mesh_edges();

while (t <= T) {
  ...
  exp.write_point_data(mf_u, U);
  exp.serie_add_object("deformation");
  exp.write_point_data(mf_vm, VM);
  exp.serie_add_object("von_mises_stress");
}
```

In this example, an OpenDX “time series” is created, for each time step, two data fields are saved: a vector field called “deformation”, and a scalar field called “von_mises_stress”.

Note also that the `dx_export::exporting_mesh_edges()` function has been called. It implies that for each mesh exported, the edges of the original mesh are also exported (into another `opendx` mesh). In this example, you have access in OpenDX to 4 data fields: “deformation”, “deformation_edges”, “von_mises_stress” and “von_mises_stress_edges”.

The `tests/dynamic_friction.net` is an example of `opendx` program for these data (run it with `cd tests; dx -edit dynamic_friction.net , menu “Execute/sequencer”`).

17 Interpolation on different meshes

The file `getfem/getfem_interpolation.h` defines the function `getfem::interpolation(...)` to interpolate a solution from a given mesh/finite element method on another mesh and/or another Lagrange finite element method.

```
getfem::interpolation(mf1, mf2, U, V, extrapolation = 0);
```

where `mf1` is a variable of type `getfem::mesh_fem` and describes the finite element method on which the source field `U` is defined, `mf2` is the finite element method on which `U` will be interpolated. `extrapolation` is a optional parameter. The values are 0 not to allow the extrapolation, 1 for an extrapolation of the exterior points near the boundary and 1 for the extrapolation of all exterior points (could be expensive).

The dimension of `U` should be a multiple of `mf1.nb_dof()`, and the interpolated data `V` should be correctly sized (multiple of `mf2.nb_dof()`).

IMPORTANT: `mf2` should be of Lagrange type for the interpolation to make sense but the meshes linked to `mf1` and `mf2` may be different (and this is the interest of this function). There is no restriction for the dimension of the domain (you can interpolate a 2D mesh on a line etc.).

If you need to perform more than one interpolation between the same finite element methods, it might be more efficient to use the function

```
getfem::interpolation(mf1, mf2, M, extrapolation = 0);
```

where `M` is a row matrix which will be filled with the linear map representing the interpolation (i.e. such that $V = MU$). The matrix should have the correct dimensions (i.e. `mf2.nb_dof() x mf1.nb_dof()`). Once this matrix is built, the interpolation is done with a simple matrix multiplication: `gmm::mult(M, U, V)`;

18 The model description

This part is a work in progress for GETFEM++ 4.0. The model description of GETFEM++ allows to quickly build some fem applications on complex linear or nonlinear PDE coupled models. The principle is to propose predefined bricks which can be assembled to describe a complex situation. A brick can describe either an equation (Poisson equation, linear elasticity ...) or a boundary condition (Dirichlet, Neumann ...) or any relation between two variables. Once a brick is written, it is possible to use it in very different situations. This allows a reusability of the produced code and the possibility of a growing library of bricks. An effort as been made in order to facilitate as much as possible the definition of a new brick. A brick is mainly defined by its contribution in the tangent linear system to be solved.

This model description is an evolution of the model bricks of previous versions of GETFEM++. Compared to the old system, it is more flexible, more general, allows the coupling of model (multiphysics) in a easier way and facilitate the writing of new components. It also facilitate the write of time integration schemes for evolving PDEs.

The kernel of the model description is contained in the file `getfem_models.h`. The two main objects are the `model` and the `bricks`.

18.1 The model object

The aim of the `model` object, defined in file `getfem_models.h`, is to globally describe a PDE model. It mainly contains two lists: a list of variables (related or not to the `mesh_fem` objects) and data (also related or not to the `mesh_fem` objects) and a list of bricks. The role of the `model` object is to coordinate the

module and make them produce a linear system of equations. If the model is linear, this will simply be the linear system of equation on the corresponding dofs. If the model is nonlinear, this will be the tangent linear system. There is two versions of the `model` object: a real one and complex one.

The declaration of a model object is done by

```
getfem::model md(complex_version = false);
```

The parameter of the constructor is a boolean which sets if the model deals with complex number or real numbers. The default is false for a model dealing with real numbers.

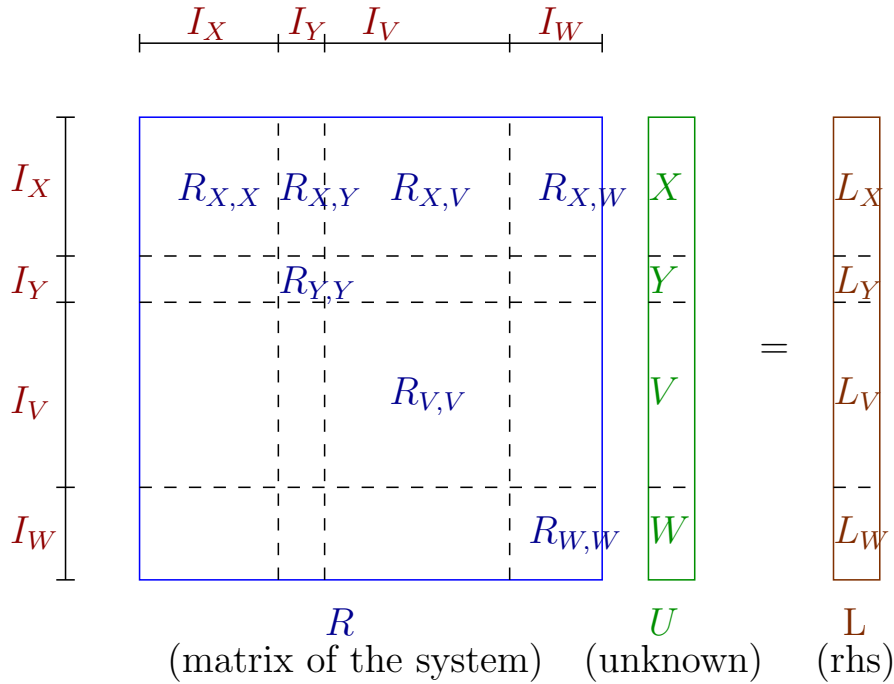


Figure 4: *The (tangent) linear system*

There are different kinds of variables/data in the model. The variables are the unknown of the model. They will be (generally) computed by solving the (tangent) linear system build by the model. Generally, the model will have several variables. Each variable as a certain size (number of degrees of freedom) and the different variables are sorted in alphanumeric order to form the global unknown(U in Fig. 4). Each variable will be associated to an interval $I = [n_1, n_2]$ which will represent the degrees of freedom indices correspondig to this variable in the global system. The model stores also some data (in the same format than the variables). The difference between data and variables is that a data is not an unknown of the model. The value of the data should be provided. In some cases (nonlinear models) some variables can be considered as some data for certain terms. Variables and data are of two kind. They can have a fixed size, or they can depend on a finite element method (be the d.o.f. of a finite element method).

For instance, in the situation described in Fig. 4, there is four variables in the model, namely X, Y, V and W . The role of the model object will be to assemble the linear system, i.e. to fill the sub matrices corresponding to each variable ($R_{X,X}, R_{Y,Y}, R_{V,V}$, and $R_{W,W}$) and the coupling terms between two variables ($R_{X,Y}, R_{X,V}, R_{W,V}, \dots$). This different contributions will be given by the different bricks added to the model.

The main usefull methods on a `model` object are

<code>m.is_complex()</code>	A boolean which says if the model deals with real or complex unknowns and data.
<code>add_fixed_size_variable(name, size, niter=1)</code>	Add a variable of fixed size. <code>name</code> is a string which designate the variable. <code>niter</code> is the number of copy of the variable (used for time integration schemes).
<code>add_fixed_size_data(name, size, niter=1)</code>	Add a data of fixed size. <code>name</code> is a string which designate the data. <code>niter</code> is the number of copy of the data (used for time integration schemes).
<code>add_initialized_fixed_size_data(name, V)</code>	Add a data of fixed size initialized with the given vector <code>V</code> . <code>name</code> is a string which designate the data.
<code>add_initialized_scalar_data(name, e)</code>	Add a data of size 1 initialized with the given scalar value <code>e</code> . <code>name</code> is a string which designate the data.
<code>add_fem_variable(name, mf, niter=1)</code>	Add a variable being the dofs of a finite element method <code>mf</code> . <code>name</code> is a string which designate the variable. <code>niter</code> is the number of copy of the variable (used for time integration schemes).
<code>add_fem_data(name, mf, niter=1)</code>	Add a data being the dofs of a finite element method <code>mf</code> . <code>name</code> is a string which designate the data. <code>niter</code> is the number of copy of the data (used for time integration schemes).
<code>add_initialized_fem_data(name, mf, V, niter=1)</code>	Add a data being the dofs of a finite element method <code>mf</code> initialized with the given vector <code>V</code> . <code>name</code> is a string which designate the data. <code>niter</code> is the number of copy of the data (used for time integration schemes).
<code>add_multiplier(name, mf, primal_name, niter=1)</code>	Add a special variable linked to the finite element method <code>mf</code> and being a multiplier for certain constraints (Dirichlet condition for instance) on a primal variable <code>primal_name</code> . The most important is that the degrees of freedom will be filtered thanks to a <code>partial_mesh_fem</code> object in order to retain only a set of linearly independent constraints. To ensure this, a call to the bricks having a term linking the multiplier and the primal variable is done and a special algorithm is called to extract independent constraints. This algorithm is optimized for boundary multipliers (see <code>gmm::range.basis</code>). Use it with care for volumic multipliers. <code>niter</code> is the number of copy of the variable (used for time integration schemes). Note that for complex terms, only the real part is considered to filter the multiplier.

<code>real_variable(name, niter=1)</code>	Gives the access to the vector value of a variable or data. Real version.
<code>complex_variable(name, niter=1)</code>	Gives the access to the vector value of a variable or data. Complex version.
<code>mesh_fem_of_variable(name)</code>	Gives a reference on the mesh_fem on which the variable is defined. Throw an exception if this is not a fem variable.
<code>real_tangent_matrix()</code>	Gives the access to tangent matrix. Real version. A computation of the tangent system have to be done first.
<code>complex_tangent_matrix()</code>	Gives the access to tangent matrix. Complex version. A computation of the tangent system have to be done first.
<code>real_rhs()</code>	Gives the access to right hand side vector of the linear system. real version. A computation of the tangent system have to be done first.
<code>complex_rhs()</code>	Gives the access to right hand side vector of the linear system. Complex version. A computation of the tangent system have to be done first.

18.2 The brick object

A model brick is an object which is supposed to represent a part of a model. It aims to represent some integral terms in a weak formulation of a pde model. The model object will contain a list of brick. All the terms described by the brick will be finally assembled to build the linear system to be solved (the tangent linear system for a nonlinear problem). For instance if a term Δu is present on the pde model (Laplacian of u) then the weak formulation will contain the term $\int_{\Omega} \nabla u \cdot \nabla v dx$, where v is the test function corresponding to u . Then the role of the corresponding brick is to assemble the term $\int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j dx$, where φ_i and φ_j are the shape functions of the finite element method describing u . This term will be added by the model object to the global linear system on a diagonal block corresponding to the variable u . The only role of the brick is thus to call the corresponding assembly procedure when the model object ask for it. The construction of a brick for such a linear term is thus very simple.

Basically, the brick object will derive from the object `virtual_brick` defined in `getfem/getfem_model.h` and should redefine the method `asm_real_tangent_terms` or `asm_complex_tangent_terms` depending on whether it is a real term or an intrinsic complex term.

18.3 How to build a new brick

According to the spirit in which the brick has been designed, a brick should avoid as much as possible to store additional data. The parameter of a brick should be contained in the variable and data of the model. For instance, the parameter of a linear elasticity brick are the elasticity coefficient. This coefficients have to be some data of the model. When the brick is called by the model object, a list of variables and data is given to the brick. The great majority of the predefined bricks do not store any data. This allows to instantiate such a bricks only once.

An example of a brick corresponding to the laplacian term is the following (other examples can be found in the file `src/getfem_models.cc` which contains the very standards bricks):

```
struct my_Laplacian_brick: public getfem::virtual_brick {

    void asm_real_tangent_terms(const getfem::model &md, size_type ib,
                               const getfem::model::varnamelist &varl,
                               const getfem::model::varnamelist &datal,
```

```

        const getfem::model::mimlist &mims,
        getfem::model::real_matlist &matl,
        getfem::model::real_veclist &vecl,
        getfem::model::real_veclist &vecl_sym,
        size_type region, build_version nl) const {
    GMM_ASSERT1(matl.size() == 1,
    "My Laplacian brick has one and only one term");
    GMM_ASSERT1(mims.size() == 1,
    "My Laplacian brick need one and only one mesh_im");
    GMM_ASSERT1(varl.size() == 1 && datal.size() == 0,
    "Wrong number of variables for my Laplacian brick");

    const getfem::mesh_fem &mf_u = md.mesh_fem_of_variable(varl[0]);
    const getfem::mesh_im &mim = *mims[0];

    gmm::clear(matl[0]);
    getfem::asm_stiffness_matrix_for_homogeneous_laplacian
    (matl[0], mim, mf_u, region);
    }

    my_Laplacian_brick(void)
    { set_flags("My Laplacian brick", true /* linear      */,
              true /* symmetric    */,
              true /* coercivity   */,
              true /* real version defined */,
              false /* no complex version */); }
};

```

The constructor of a brick should call the method `set_flags`. The first parameter of this method is a name for the brick (this allows to list the bricks of a model and facilitate their identification). The other parameters are some flags, respectively

- if the brick terms are all linear or not
- if the brick terms are globally symmetric (conjugated in the complex version) or at least do not affect the symmetry. The terms corresponding to two different variables and declared symmetric are added twice in the global linear system (the term and the transpose of the term).
- if the terms do not affect the coercivity.
- if the terms have a real version or not. If yes, the method `asm_real_tangent_terms` should be redefined.
- if the terms have a complex version or not. If yes, the method `asm_complex_tangent_terms` should be redefined.

The method `asm_real_tangent_terms` will be called by the model object for the assembly of the tangent system. The model object gives the whole framework to the brick to build its terms. The parameter `md` of the `asm_real_tangent_terms` method is the model that called the brick, `ib` being the

brick number in the model. The parameter `var1` is an array of variable/data names defined in this model and needed in the brick. `mims` is an array of `mesh_im` pointers. It corresponds to the integration methods needed to assemble the terms. `mat1` is an array of matrices to be computed. `vec1` is an array of vectors to be computed (rhs or residual vectors). `vec1_sym` is an array of vectors to be computed only for symmetric terms and corresponding to the rhs of the second variable. A brick can have an arbitrary number of terms. For each term, at least the corresponding matrix or the corresponding vector as to be filled (or both the two). `region` is a mesh region number indicated that the terms have to be assembled on a certain region. `nl` is for nonlinear bricks only. It says if the tangent matrix or the residual or both the two are to be computed (for linear bricks, all is to be computed at each call).

For the very simple Laplacian brick defined above, only one variable is used and no data and there is only one term. The lines

```
GMM_ASSERT1(mat1.size() == 1,
"My Laplacian brick has one and only one term");
GMM_ASSERT1(mims.size() == 1,
"My Laplacian brick need one and only one mesh_im");
GMM_ASSERT1(var1.size() == 1 && data1.size() == 0,
"Wrong number of variables for my Laplacian brick");
```

are not mandatory and just verify that the good number of terms (1), integration methods (1), variables(1), data(0) are passed to the `asm_real_tangent_terms` method.

The lines

```
const getfem::mesh_fem &mf_u = md.mesh_fem_of_variable(var1[0]);
const getfem::mesh_im &mim = *mims[0];
```

takes the `mesh_fem` object from the variable on which the Laplacian term will be added and the `mesh_im` object in the list of integrations methods. Finally, the lines

```
gmm::clear(mat1[0]);
getfem::asm_stiffness_matrix_for_homogeneous_laplacian
(mat1[0], mim, mf_u, region);
```

call a standard assembly procedure for the Laplacian term defined in the file `getfem/getfem_assembling.h`. The clear method is necessary because although it is guaranteed that the matrices in `mat1` have the good sizes they maybe not cleared before the call of `asm_real_tangent_terms`.

Note that this simple brick have only one term and is linear. In the case of a linear birck, either the matrix or the right hand side vector have to be filled but not both the two. Depending on the declaration of the term. See below the integration of the birck to the model.

Le us see now a second example of a simple brick which prescribe a Dirichlet condition thanks to the use of a Lagrange multiplier. The Dirichlet condition is of the form

$$u = u_D \text{ on } \Gamma,$$

where u is the variable, u_D is a given value and Γ is a part on the boundary of the considered domain. The weak terms corresponding to this condition prescribed with a Lagrange multiplier are

$$\int_{\Gamma} u \mu d\Gamma = \int_{\Gamma} u_D \mu d\Gamma, \quad \forall \mu \in M,$$

where M is a appropriate multiplier space. The contributions to the global linear system can be viewed in Fig. 5. The matrix B is the “mass matrix” between the finite element space of the variable u and the finite element space of the multiplier μ . L_u is the right end side corresponding to the data u_D .


```

    getfem::asm_source_term(vecl[0], mim, mf_mult, *mf_data, A, region);
else
    getfem::asm_homogeneous_source_term(vecl[0], mim, mf_mult, A, region);

gmm::clear(matl[0]);
getfem::asm_mass_matrix(matl[0], mim, mf_mult, mf_u, region);
}

my_Dirichlet_brick(void)
{ set_flags("My Dirichlet brick", true /* linear      */,
            true /* symmetric    */,
            false /* coercivity  */,
            true /* real version defined */,
            false /* no complex version */); }
};

```

This brick has again only one term but define both the matrix and the right hand side parts. Two variables are concerned, the primal variable on which the Dirichlet condition is prescribed, and the multiplier variable which should be defined on a mesh region corresponding to a boundary (it should be added to the model with the method `add_multiplier`). The term of the brick will be declared symmetric (see the next section).

The lines

```

const getfem::model_real_plain_vector &A = md.real_variable(datal[ind]);
const getfem::mesh_fem *mf_data = md.pmesh_fem_of_variable(datal[ind]);

```

allow to have the access to the value of the data corresponding to the right hand side of the Dirichlet condition and to the `mesh_fem` on which this data is defined. If the data is constant (no describe on a fem) then `mf_data` is a null pointer. The lines

```

if (mf_data)
    getfem::asm_source_term(vecl[0], mim, mf_mult, *mf_data, A, rg);
else
    getfem::asm_homogeneous_source_term(vecl[0], mim, mf_mult, A, rg);

```

make the assembly of the right hand side. The two versions correspond to a data defined on a finite element method or constant size data.

(+ some example with a nonlinear term ...)

18.4 How to add the brick to a model

In order to add a brick to a model, a certain information have to be passed to the model:

- A pointer to the brick itself.
- The set of variable names concerned with the terms of the brick.
- The set of data names concerned with the terms of the brick.
- A list of terms description.

- A list of integration methods.
- Eventually the concerned mesh region.

This is done by the call of the `getfem::model` object method

```
md.add_brick(pbr, const getfem::model::varnamelist &varnames,
             const getfem::model::varnamelist &datanames,
             const getfem::model::termlist &terms,
             const getfem::model::mimlist &mims,
             size_t region);
```

The method return the index of the brick in the model. The call of this method is rather complex because it can be adapted to many situations. The construction of a new brick should be accompanied to the definition of a function that add the new brick to the model calling this method and more simple to use.

For instance, for the simple Laplacian brick described above, this function can be defined as follows:

```
size_t add_my_Laplacian_brick(getfem::model &md, const getfem::mesh_im &mim,
                              const std::string &varname,
                              size_t region = size_t(-1)) {
    getfem::pbrick pbr = new my_Laplacian_brick;
    getfem::model::termlist tl;
    tl.push_back(getfem::model::term_description(varname, varname, true));
    return md.add_brick(pbr, getfem::model::varnamelist(1, varname),
                       getfem::model::varnamelist(), tl,
                       getfem::model::mimlist(1, &mim), region);
}
```

This function will be called by the user of your brick. The type `getfem::model::varnamelist` is a `std::vector<std::string>` and represent an array of variable names. The type `getfem::model::mimlist` is a `std::vector<const getfem::mesh_im *>` and represent an array of pointers to integration methods. The type `getfem::model::termlist` is an array of terms description. There is two kind of terms. The terms adding only a right hand side to the linear (tangent) system which have to be added to the list by

```
tl.push_back(getfem::model::term_description(varname));
```

and the terms having a contribution to the matrix of the linear system which have to be added to the list by

```
tl.push_back(getfem::model::term_description(varname1, varname2, true/false));
```

In this case, the matrix term is added in the rows corresponding to the variable `varname1` and the columns corresponding to the variable `varname2`. The boolean being the third parameter is to declare if the term is symmetric or not. If it is symmetric and if the two variables are different then the assembly procedure add the corresponding term AND its transpose. The number of terms is arbitrary. For each term declared, the brick have to fill the corresponding right hand side vector (parameter `vec1` of `asm_real_tangent_terms` above) or/and the matrix term (parameter `mat1` of `asm_real_tangent_terms`) depending on the declaration of the term. Note that for nonlinear bricks, both the matrix and the right hand side vectors have to be filled.

The variable names and the data names are given in two separate arrays because the dependence of the brick is not the same in both cases. A linear term have to be recomputed if the value of a data is changed but not if the value of a variable is changed.

The function allowing to add the simple Dirichlet brick described above can be defined as follows:

```

size_t add_my_Dirichlet_condition_brick
(model &md, const mesh_im &mim, const std::string &varname,
 const std::string &multname, size_t region, const std::string &dataname) {
    pbrick pbr = new my_Dirichlet_brick;
    model::termlist tl;
    tl.push_back(model::term_description(multname, varname, true));
    model::varnamelist vl(1, varname);
    vl.push_back(multname);
    model::varnamelist dl;
    if (dataname.size()) dl.push_back(dataname);
    return md.add_brick(pbr, vl, dl, tl, model::mimlist(1, &mim), region);
}

```

Again, here, the term is declared symmetric and then the matrix term and its transpose will be added.

18.5 Generic elliptic brick

This brick add an elliptic term on a variable of a model. The shape of the elliptic term depends both on the variable and a given coefficient. This corresponds to a term

$$-\text{div}(a\nabla u),$$

where a is the coefficient and u the variable. The coefficient can be a scalar, a matrix or an order four tensor. The variable can be vector valued or not. This means that the brick treats several different situations. If the coefficient is a scalar or a matrix and the variable is vector valued then the term is added componentwise. An order four tensor coefficient is allowed for vector valued variable only. The coefficient can be constant or described on a fem. Of course, when the coefficient is a tensor described on a finite element method (a tensor field) the corresponding 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 corresponding to the coefficient (compatibility with blas). The symmetry and coercivity of the given matrix/tensor is not verified (but assumed).

This brick can be added to a model `md` thanks to two functions. The first one is

```
size_type getfem::add_Laplacian_brick(md, mim, varname, region = -1);
```

that adds an elliptic term relatively to the variable `varname` of the model with a constant coefficient equal to 1 (a Laplacian term). This corresponds to the Laplace operator. `mim` is the integration method which will be used to compute the term. `region` is an optional region number. If it is omitted, it is assumed that the term will be computed on the whole mesh. The result of the function is the brick index in the model.

The second function is

```
size_type getfem::add_generic_elliptic_brick(md, mim, varname, dataname,
      region = -1);
```

It adds a term with an arbitrary coefficient given by the data `dataname` of the model. This data have to be defined first in the model.

Note that very general equations can be obtained with this brick. For instance, linear anisotropic elasticity can be obtained with a tensor data. When an order four tensor is used, the corresponding weak term is the following

$$\int_{\Omega} \sum_{i,j,k,l} a_{i,j,k,l} \partial_i u_j \partial_k v_l dx$$

where $a_{i,j,k,l}$ is the order four tensor and $\partial_i u_j$ is the partial derivative with respect to the i^{th} variable of the component j of the unknown k . v is the test function. However, for linear isotropic elasticity, a more adapted brick is available (see below).

The brick has a working complex version.

18.6 Dirichlet condition brick

The aim of the Dirichlet condition brick is to prescribe a Dirichlet condition on a part of the boundary of the domain for a variable of the model. This means that the value of this variable is prescribed on the boundary. There is two versions of this brick. The first version prescribe the Dirichlet thank to a multiplier. The associated weak form of the term is the following:

$$\int_{\Gamma} u \mu d\Gamma = \int_{\Gamma} u_D \mu d\Gamma, \forall \mu \in M.$$

where u is the variable, M is the space of multipliers, u is the variable and Γ the Dirichlet boundary. For this version, an additional variable have to be added to represent the multiplier. It can be done directly to the model or thanks to the functions below. There are three functions allowing to add a Dirichlet condition prescribed with a multiplier. The first one is

```
add_Dirichlet_condition_with_multipliers(md, mim, varname,
    multname, region, dataname = std::string());
```

adding a Dirichlet condition on `varname` thanks to a multiplier variable `multname` on the mesh region `region` (which should be a boundary). The value of the variable on that boundary is described by the data `dataname` which should be previously defined in the model. If the data is omitted, the Dirichlet condition is assumed to be an homogeneous one (vanishing variable on the boundary). The data can be constant or described on a fem. It can also be scalar or vector valued, depending on the variable. The variable `multname` should be added to the model by the method `add_multiplier`. The function returns the brick index in the model. The second function is

```
add_Dirichlet_condition_with_multipliers(md, mim, varname,
    mf_mult, region, dataname = std::string());
```

The only difference is that `multname` is replaced by `mf_mult` which means that only the finite element on which the multiplier will be built is given. The function adds itself the multiplier variable to the model. The third function is very similar

```
add_Dirichlet_condition_with_multipliers(md, mim, varname,
    degree, region, dataname = std::string());
```

The parameter `mf_mult` is replaced by a integer `degree` indicating that the multiplier will be build on a classical finite element method of that degree.

Note, that in all the cases, when a variable is added by the method `add_multiplier` of the model object, the mesh_fem will be filtered (thank to a `partial_mesh_fem_object` in order to retain only the degrees of freedom having a non vanishing contribution on the considered boundary.

Finally, the variable name of the multiplier can be obtained thank to the function

```
mult_varname_Dirichlet(md, ind_brick);
```

where `ind_brick` is the brick index in the model. This function has an undefined behavior if it applied to another kind of brick.

The second version of the Dirichlet condition brick is the one with penalization. The function allowing to add this brick is

```
add_Dirichlet_condition_with_penalization(md, mim, varname,
    penalization_coeff, region, dataname = std::string());
```

The penalization consists in computing the mass matrix of the variable and add it multiplied by the penalization coefficient to the stiffness matrix. The penalization coefficient is added as a data of the model and can be changed thanks to the function

```
change_penalization_coeff(md, ind_brick, penalisation_coeff);
```

18.7 Source term bricks (and Neumann condition)

This brick add a source term, i.e. a term which occurs only in the right hand side of the linear (tangent) system build by the model. If f denotes the value of the source term, the weak form of such a term is

$$\int_{\Omega} f v dx$$

where v is the test function. The value f can be constant or described on a finite element method.

It can also represent a Neumann condition if it is applied on a boundary of the domain.

The function to add a source term to a model is

```
add_source_term_brick(md, mim, varname, dataname, region = -1,
    directdataname = std::string());
```

where `md` is the model object, `mim` is the integration method, `varname` is the variable of the model for which the source term is added, `dataname` is the name of the data in the model which represents the source term. It has to be scalar or vector valued depending on the fact that the variable is scalar or vector valued itself. `region` is a mesh region on which the term is added. If the region corresponds to a boundary, the source term will represent a Neumann condition. `directdataname` is an optional additional data which will directly be added to the right hand side without assembly.

The brick has a working complex version.

A slightly different brick, especially dedicated to deal with a Neumann condition, is added by the following function

```
add_normal_source_term_brick(md, mim, varname, dataname, region);
```

The difference compared to the basic source term brick is that the data should be a vector field (a matrix field if the variable `varname` is itself vector valued) and a scalar product with the outward unit normal is performed on it.

18.8 Predefined solvers

Of course, for many problems, it will be more convenient to make a specific solver. Even so, one generic solver is available to test your models quickly. It can also be taken as an example to build your own solvers. It is defined in `getfem/getfem_model_solvers.h` and the call is

```
getfem::standard_solve(md, iter);
```

where `md` is the model object and `iter` is an iteration object from Gmm++. See also the next section for an example of use.

Note that SuperLu is used by default on “small” problems. You can also link MUMPS with Getfem (see section 6) and used the parallele version.

18.9 Example of a complete Poisson problem

The following example is a part of the test program `tests/laplacian_with_bricks.cc`. Construction of the mesh and finite element methods are omitted. It is assumed that a mesh is build and two finite element methods `mf_u` and `mf_rhs` are build on this mesh. It is also assumed that `NEUMANN_BOUNDARY_NUM` and `DIRICHLET_BOUNDARY_NUM` are two valid boundary indices on that mesh. The code begins by the definition of three functions which are interpolated on `mf_rhs` in order to build the data for the source term, the Neumann condition and the Dirichlet condition. Follows the declaration of the model object, the addition of the bricks and the solving of the problem.

```
using bgeot::base_small_vector;
// Exact solution. Allows an interpolation for the Dirichlet condition.
scalar_type sol_u(const base_node &x) { return sin(x[0]+x[1]); }
// Right hand side. Allows an interpolation for the source term.
scalar_type sol_f(const base_node &x) { return 2*sin(x[0]+x[1]); }
// Gradient of the solution. Allows an interpolation for the Neumann term.
base_small_vector sol_grad(const base_node &x)
{ return base_small_vector(cos(x[0]+x[1]), cos(x[0]+x[1])); }

int main(void) {

    // ... definition of a mesh
    // ... definition of a finite element method mf_u
    // ... definition of a finite element method mf_rhs
    // ... definition of a integration method mim
    // ... definition of boundaries NEUMANN_BOUNDARY_NUM
    //                and DIRICHLET_BOUNDARY_NUM

    // Model object
    getfem::model laplacian_model;

    // Main unknown of the problem
    laplacian_model.add_fem_variable("u", mf_u);

    // Laplacian term on u.
    getfem::add_Laplacian_brick(laplacian_model, mim, "u");

    // Volumic source term.
    std::vector<scalar_type> F(mf_rhs.nb_dof());
    getfem::interpolation_function(mf_rhs, F, sol_f);
    laplacian_model.add_initialized_fem_data("VolumicData", mf_rhs, F);
    getfem::add_source_term_brick(laplacian_model, mim, "u", "VolumicData");

    // Neumann condition.
    gmm::resize(F, mf_rhs.nb_dof()*N);
    getfem::interpolation_function(mf_rhs, F, sol_grad);
    laplacian_model.add_initialized_fem_data("NeumannData", mf_rhs, F);
    getfem::add_normal_source_term_brick
```

```

(laplacian_model, mim, "u", "NeumannData", NEUMANN_BOUNDARY_NUM);

// Dirichlet condition.
gmm::resize(F, mf_rhs.nb_dof());
getfem::interpolation_function(mf_rhs, F, sol_u);
laplacian_model.add_initialized_fem_data("DirichletData", mf_rhs, F);
getfem::add_Dirichlet_condition_with_multipliers
(laplacian_model, mim, "u", mf_u,
 DIRICHLET_BOUNDARY_NUM, "DirichletData");

gmm::iteration iter(residual, 1, 40000);
getfem::standard_solve(laplacian_model, iter);

std::vector<scalar_type> U(mf_u.nb_dof());
gmm::copy(laplacian_model.real_variable("u"), U);

// ... doing something with the solution ...

return 0;
}

```

Note that the brick can be added in an arbitrary order.

18.10 Constraint brick

The constraint brick allows to add an explicit constraint on a variable. Explicit means that no integration is done. if U is a variable then a constraint of the type

$$BU = L,$$

can be added with the two following functions:

```

indbrick = getfem::add_constraint_with_penalization(md, varname, penalisation_coeff, B, L);
indbrick = getfem::add_constraint_with_multipliers(md, varname, multname, B, L);

```

In the second case, a (fixed size) variable which will serve as a multiplier should be first added to the model.

For the penalized version 'B' should not contain a plain row, otherwise the whole tangent matrix will be plain. The penalization parameter can be changed thanks to the function

```

change_penalization_coeff(md, ind_brick, penalisation_coeff);

```

It is possible to change the constraints at any time thanks to the two following functions:

```

getfem::set_private_data_matrix(md, indbrick, B)
getfem::set_private_data_rhs(md, indbrick, L)

```

where `indbrick` is the index of the brick in the model.

18.11 Other “explicit” bricks

Two (very simple) bricks allow to add some explicit terms to the tangent system.

The function

```
indbrick = getfem::add_explicit_matrix(md, varname1, varname2, B, issymmetric = false,
                                       iscoercive = false)
```

adds a brick which just adds the matrix `B` to the tangent system relatively to the variables `varname1` and `varname2`. The given matrix should have as 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 true then the transpose of the matrix is also added to the tangent system (default is false). set `iscoercive` to true if the term does not affect the coercivity of the tangent system (default is false). The matrix can be changed by the command

```
getfem::set_private_data_matrix(md, indbrick, B)
```

The function

```
getfem::add_explicit_rhs(md, varname, L);
```

add a brick which just add the vector `L` to the right hand side of the tangent system relatively to the variable `varname`. The given vector should have the same size as the variable `varname`. The value of the vector can be changed by the command

```
getfem::set_private_data_rhs(md, indbrick, L)
```

18.12 Helmholtz brick

This brick represents the complex or real Helmholtz problem

$$\Delta u + k^2 u = \dots$$

where k the wave number is a real or complex value. For a complex version, a complex model has to be used (see `helmholtz.cc` in the tests directory)

The function adding a Helmholtz brick to a model is

```
getfem::add_Helmholtz_brick(md, mim, varname, dataname, region);
```

where `varname` is the variable on which the Helmholtz term is added and `dataname` should contain the wave number.

18.13 Fourier-Robin brick

This brick can be used to add boundary conditions of Fourier-Robin type like

$$\frac{\partial u}{\partial n} = Qu$$

for scalar problems, or

$$\sigma n = Qu$$

for linearized elasticity problems. Q is a scalar field in the scalar case or a matrix field in the vectorial case. This brick works for both real or complex terms in scalar or vectorial problems.

The function adding this brick to a model is:

```
add_Fourier_Robin_brick(md, mim, varname, dataname, region);
```

where `dataname` is the pdata of the model which represents the coefficient Q .

Note that an additional right hand side can be added with a source term brick.

18.14 Isotropic linearized elasticity brick

This brick represents a term

$$-div(\sigma) = \dots;$$

with

$$\sigma = \lambda \text{tr}(\varepsilon(u))I + 2\mu\varepsilon(u); \quad \varepsilon(u) = (\nabla u + \nabla u^T)/2.$$

$\varepsilon(u)$ is the small strain tensor, σ is the stress tensor, λ and μ are the Lam coefficients. This represents the system of linearized isotropic elasticity. It can also be used with $\lambda = 0$ together with the linear incompressible brick to build the Stokes problem.

The function which adds this brick to a model is

```
ind_brick = getfem::add_isotropic_linearized_elasticity_brick
            (md, mim, varname, dataname_lambda, dataname_mu,
             region = size_type(-1));
```

where `dataname_lambda` and `dataname_mu` are the data of the model representing the lam coefficients (constant or described on a finite element method).

The function

```
getfem::compute_isotropic_linearized_Von_Mises_or_Tresca
(md, varname, dataname_lambda, dataname_mu, mf_vm, VM, tresca_flag = false);
```

compute the Von Mises criterion (or Tresca if `tresca_flag` is set to true) on the displacement field stored in `varname`. The stress is evaluated on the mesh fem `mf_vm` and stored in the vector `VM`.

The program `elastostatic.cc` in the tests directory of Getfem++ distribution can be taken as a model of use of this brick.

18.15 linear incompressibility (or nearly incompressibility) brick

This brick adds a linear incompressibility condition (or a nearly incompressible condition) in a problem of type

$$\text{div}(u) = 0, \quad (\text{ or } \text{div}(u) = \varepsilon p)$$

This constraint is enforced with Lagrange multipliers representing the pressure, introduced in a mixed formulation.

The function adding this incompressibility condition is:

```
ind_brick = getfem::add_linear_incompressibility
            (md, mim, varname, multname_pressure, region = size_type(-1),
             dataname_penal_coeff = std::string());
```

where `varname` is the variable on which the incompressibility condition is prescribed, `multname_pressure` is a variable which should be described on a scalar fem representing the multiplier (the pressure) and `dataname_penal_coeff` is an optional penalization coefficient (constant or described on a finite element method) for the nearly incompressible condition.

In nearly incompressible homogeneous linearized elasticity, one has $\varepsilon = 1/\lambda$ where λ is one of the Lam coefficient and ε the penalization coefficient.

For instance, the following program defines a Stokes problem with a source term and an homogeneous Dirichlet condition on boundary 0. `mf_u`, `mf_data` and `mf_p` have to be valid finite element description on the same mesh. `mim` should be a valid integration method on the same mesh.

```

typedef std::vector<getfem::scalar_type> plain_vector;
size_type N = mf_u.linked_mesh().dim();

getfem::model Stokes_model;

laplacian_model.add_fem_variable("u", mf_u);

getfem::scalar_type mu = 1.0;
Stokes_model.add_initialized_data("lambda", plain_vector(1, 0.0));
Stokes_model.add_initialized_data("mu", plain_vector(1, mu));
getfem::add_isotropic_linearized_elasticity_brick(Stokes_model, mim,
                                                  "u", "lambda", "mu");

laplacian_model.add_fem_variable("p", mf_p);
getfem::add_linear_incompressibility(Stokes_model, mim, "u", "p");

plain_vector F(mf_data.nb_dof()*N);
for (int i = 0; i < mf_data.nb_dof()*N; ++i) F(i) = ...;
Stokes_model.add_initialized_fem_data("VolumicData", mf_data, F);
getfem::add_source_term_brick(Stokes_model, mim, "u", "VolumicData");

getfem::add_Dirichlet_condition_with_multipliers(Stokes_model, mim,
                                                  "u", mf_u, 1);

gmm::iteration iter(residual, 1, 40000);
getfem::standard_solve(Stokes_model, iter);

plain_vector U(mf_u.nb_dof());
gmm::copy(Stokes_model.real_variable("u"), U);

```

An example for a nearly incompressibility condition can be found in the program `tests/elastostatic.cc`.

18.16 Mass brick

This brick represents a weak term of the form

$$\int_{\Omega} \rho u.v dx + \dots;$$

It mainly represents a mass term for transient problems but can also be used for other applications (it can be used on a boundary). Basically, this brick adds a mass matrix on the tangent linear system with respect to a certain variable.

The function which adds this brick to a model is

```

ind_brick = getfem::add_mass_brick
            (md, mim, varname, dataname_rho="", region = size_type(-1));

```

where `dataname_rho` is an optional data of the model representing the density ρ . If it is omitted, the density is assumed to be equal to one.

Note that for time integrations scheme, there exist specific bricks for the discretisation of time derivatives.

18.17 The time dispatchers: integration of transient problems

The role of time dispatchers is to allow the integration of transient problems with some pre-defined time integration schemes. The principle of the time dispatchers is to dispatch the terms of a brick on the different time steps of the considered time integration scheme. When time derivative terms are present in the model (this is generally the case except for quasistatic models), the time dispatcher will be associated to a specific brick representing this time derivative term ($\partial u/\partial t$ or $\partial^2 u/\partial t^2$ for instance). For this, a number of tools are available in GETFEM++ to help the construction of a time dispatcher. Mainly they are the two following:

- The variables can be duplicated to take into account the different versions corresponding to each time iteration. For instance, for simplest time integration scheme, two versions U^n and U^{n+1} of a variable U are stored. The addition of a variable u with two versions can be done with the method of the model object

```
model.add_fem_variable("u", mf_u, 2);
```

where 2 is here the number of versions. The variable which is actually computed have always the index 0 and will be accessed with `model.real_variable("u", 0)` or simply with `model.real_variable("u")`. It will generally represent the version U^{n+1} . The version U^n (corresponding to the previous time step) will be accessed with `model.real_variable("u", 1)`. Generally, it will be necessary to set this version with `model.set_real_variable("u", 1)` to define the initial condition of the model. At the end of each iteration, the different versions of a variable are automatically shifted (version 0 \rightarrow version 1 ...).

- The right hand side of a brick is dispatched into several right hand sides for each time iteration which are stored. To avoid unnecessary computation, the time dispatcher can shift these extra right hand sides at the end of each time iteration.

18.18 Theta-method dispatcher

This is the simplest time dispatcher. The use of this dispatcher will be described in details. Since the use of the other dispatchers is similar, only their specificities will be described later on.

The principle of the θ -method is to dispatch the term F into

$$(\theta)F^{n+1} + (1 - \theta)F^n,$$

For specific values of θ one obtains some classical schemes: backward Euler for $\theta = 1$, forward Euler for $\theta = 0$ and Crank-Nicholson scheme for $\theta = 1/2$ (which is an order two scheme).

For instance, if the dispatcher is applied to a brick representing a linear elliptic term KU where K is the stiffness matrix and U the unknown, it will be transformed into

$$(\theta)KU^{n+1} + (1 - \theta)KU^n.$$

Since U^{n+1} is the real unknown, the effect will be to multiply by θ the stiffness matrix and to add to the right hand side the term $(1 - \theta)KU^n$. This means also that U^n have to be initialized (with something like `gmm::copy(U0, model.real_variable("u", 1))`). It represents an initial data for the problem. Remember this principle: each time you apply a time dispatcher to a brick, the corresponding variables have to have the right number of versions (see above) and should be initialized before the first time iteration.

You can apply the dispatcher to a brick having only a right hand side (a source term for instance). It is not necessary if the term is constant in time.

When a brick represents a constraint (Dirichlet condition, incompressibility ...) this is not mandatory to apply the dispatcher. Of course, the result will not be exactly the same if you apply or not the dispatcher. If you do not apply it, the constraint will be applied to the current variable (U^{n+1} for the θ -method). If you apply it, the constraint will be in a sense applied to $(\theta)U^{n+1} + (1 - \theta)U^n$. If the constraint is applied thanks to a multiplier, this multiplier will need to have different versions and will need to have an initial value.

In order to apply the θ -method dispatcher to a set of brick you must execute

```
model.add_initialized_scalar_data("theta", theta);
getfem::add_theta_method_dispatcher(model, transient_bricks, "theta");
```

where `transient_bricks` is a `dal::bit_vector` containing the indices of the corresponding bricks. The value of θ can be modified from an iteration to another.

The global structure of the loop solving the different time steps should be the following

```
gmm::iteration solver_iter(residual, 0, 40000);

// Set here the initial values.

model.first_iter(); // initialize the iterations.

for (scalar_type t = 0; t < T; t += dt)

    solver_iter.init();
    getfem::standard_solve(model, solver_iter); // solve an iteration.

    model.next_iter(); // shift the variables and additional right hand sides.
```

where `model.first_iter()` should be called before the first iteration to initialize the right hand side of the time dispatchers. The initial data should be set before the call to `model.first_iter()`. The method `model.next_iter()` is to be called at the end of each iteration. It calls the dispatcher to shift there additional right hand side and it shifts the version of the variables.

18.18.1 Basic first order time derivative brick

A term like $\rho \partial u / \partial t$ will be represented in the model by $(MU^{n+1} - MU^n) / dt$,

where M is the mass matrix and dt is the time step. The θ -method is compatible with this. A brick is dedicated to represent this term. It can be added to the model by the function

```
getfem::add_basic_d_on_dt_brick(model, mim, varname, dataname_dt,
                               dataname_rho = std::string(), region = size_type(-1));
```

where `varname` is the name of the variable on which the time derivative is applied (should have at least two versions), `dataname_dt` is the name of the data corresponding to the time step (added by `model.add_initialized_scalar_data("dt", dt)` for instance) which could be modified from an iteration to another and `dataname_rho` is an optional parameter (whose default value is 1) corresponding to the term ρ in $\rho \partial u / \partial t$.

NOTE that the time dispatcher should not be applied to this brick !

A good model of the use of this brick and the θ -method time dispatcher can be found in the test program `tests/heat_equation.cc`.

18.18.2 Basic second order time derivative brick

This brick represents a second order time derivative like $\rho \partial^2 u / \partial t^2$. The problem with such a term is that the θ -method should be applied both on u and $\partial u / \partial t$ which means that $\partial u / \partial t$ is a natural unknown of the problem. The easiest way is then to add the time derivative of the variable u has a independent variables of the model (a drawback, of course, is that one has twice as much unknowns). This Basic second order time derivative brick does not apply this strategy. The time derivative $\partial u / \partial t$ is considered as a data which is updated at a post-treatment stage (in some cases, this strategy cannot be applied if the time derivative appears to be a required unknown of the model).

The term $\rho \partial^2 u / \partial t^2$ will be represented by

$$(MU^{n+1} - MU^n) / (\alpha dt^2) - MV^n / (\alpha dt), \quad (*)$$

where M is the mass matrix, dt is the time step, α is a parameter which is equal to θ for the θ -method and V^n the time derivative at the previous time step. This means in particular that V should be added as a data on the model with (at least) two versions.

The function adding the brick is

```
getfem::add_basic_d2_on_dt2_brick(model, mim, varname, dataname_V,
    dataname_dt, dataname_alpha, dataname_rho = std::string(),
    region = size_type(-1));
```

where `varname` is the name of the variable on which the second order time derivative is applied, `dataname_V` is the data representing the time derivative, `dataname_dt` is the name of the data corresponding to the time step (added by `model.add_initialized_scalar_data("dt", dt)` for instance) which could be modified from an iteration to another, `dataname_alpha` is the name of the data containing the parameter α in (*) and `dataname_rho` is an optional parameter (whose default value is 1) corresponding to the term ρ in $\rho \partial^2 u / \partial t^2$.

At the end of each iteration, the data `dataname_V` should be updated (before the call to `model.next_iter()` by the call to

```
getfem::velocity_update_for_order_two_theta_method
(model, varname, dataname_V, dataname_dt, dataname_alpha);
```

A good model of the use of this brick and the θ -method time dispatcher can be found in the test program `tests/wave_equation.cc`.

18.19 Midpoint dispatcher

The principle of the midpoint scheme is to dispatch a term $F(U)$ into

$$F((U^{n+1} - U^n)/2),$$

It is different from the Crank-Nicholson scheme (θ -method for $\theta = 1/2$) only for nonlinear terms.

The real unknown remains U^{n+1} . the effect will be to multiply by 1/2 the stiffness (or tangent) matrix and to add to a right hand side the term $(KU^n)/2$ for a linear matrix term K . As for the θ -method, the variables have to have two version and the second version have to be initialized.

You can apply the dispatcher to a brick having only a right hand side (a source term for instance). It is not necessary if the term is constant in time.

NOTE that if the brick depend on a data which is not constant in time, the data either have to have two versions (and the mean of the two versions are taken into account) or evaluated at the middle of the time step.

When a brick represents a constraint (Dirichlet condition, incompressibility ...) this is not mandatory to apply the dispatcher. Of course, the result will not be exactly the same if you apply or not the dispatcher. If you do not apply it, the constraint will be applied to the current variable U^{n+1} . If you apply it, the constraint will be applied to $(U^{n+1}+U^n)/2$. If the constraint is applied thanks to a multiplier, this multiplier will need to have different versions and will need to have an initial value.

In order to apply the midpoint dispatcher to a set of brick you must execute

```
getfem::add_midpoint_dispatcher(model, transient_bricks);
```

where `transient_bricks` is a `dal::bit_vector` containing the indices of the corresponding bricks.

18.19.1 Basic first order time derivative brick

The same brick as for the θ -method can be used to represent a first order time derivative.

18.19.2 Basic second order time derivative brick

The same brick as for the θ -method can be used to represent a second order time derivative. The value of α should be $1/2$.

18.20 Newmark scheme

For a system

$$M\ddot{U} + K(U) = F,$$

the Newmark scheme of parameter β and γ is defined by

$$M(U^{n+1} - U^n) = dtMV^n + dt^2/2(2\beta(F^{n+1} - K(U^{n+1})) + (1 - 2\beta)(F^n - K(U^n))),$$

$$M(V^{n+1} - V^n) = dt(2\gamma(F^{n+1} - K(U^{n+1})) + (1 - 2\gamma)(F^n - K(U^n))),$$

where V represents the time derivative of U .

The implementation of the Newmark scheme proposed is not optimal and should be adapted. It can be obtained using the basic second order time derivative brick (see θ -method) and the θ -method time dispatcher used with $\theta = 2\beta$. Additionally, one has to use the following function which compute the time derivative of the variable as a post-computation:

```
getfem::velocity_update_for_Newmark_scheme
(model, id2dt2, varname, dataname_V, dataname_dt, dataname_alpha);
```

where `id2dt2` is the index of the basic second order time derivative brick (see the section on the θ -method for more details and the implementation in the test program `tests/wave_equation.cc`).

This implementation of the Newmark-scheme is not optimal since the latter function inverts the mass matrix to compute the time derivative using a conjugate gradient. This linear system solve could be avoided by keeping the multiplication of the mass matrix with the time derivative as a data, with an adaptation of the time derivative brick.

19 The model bricks (old system)

The brick system of GETFEM++ 3.x described in this section evaluated on GETFEM++ 4.x. The new system is described in previous section 18. The system of GETFEM++ 3.x is kept for compatibility reasons but is somehow deprecated.

It is possible to use predefined bricks to build up very quickly a certain number of models. Most of the bricks are defined in `getfem/getfem_modeling.h`.

A model brick is basically an object which modifies a global tangent matrix and its associated right hand side. Typical modifications are insertion of the stiffness matrix for the problem considered (linear elasticity, laplacian, ...), handling of a set of constraints, Dirichlet condition, addition of a source term to the right hand side etc. The global tangent matrix and its right hand side are stored in a `model_state` structure.

19.1 The model state variable

The `getfem::model_state` object is an object which stores the state of the system and the tangent system with eventual constraints. There are two predefined `model_state` types:

```
getfem::standard_model_state
getfem::standard_complex_model_state
```

The second one is for models with complex degrees of freedom like Helmholtz problem. These two predefined `model_state` type are built with the following predefined sparse matrices and plain vectors:

```
getfem::modeling_standard_sparse_matrix (gmm::col_matrix<gmm::rsvector<double> >>)
getfem::modeling_standard_plain_vector (std::vector<double>)
getfem::modeling_standard_complex_sparse_matrix
    (gmm::col_matrix<gmm::rsvector<std::complex<double> >>)
getfem::modeling_standard_complex_plain_vector (std::vector<std::complex<double> >>)
```

But you can define your own model state type with arbitrary types of sparse matrices and plain vectors (see the file `getfem/getfem_modeling.h`)

19.2 Basic properties of a brick

A brick represents a basic problem (elasticity, Helmholtz, Poisson problems ...) or a modifier of such problems (addition of a Dirichlet or Neumann condition, source term, incompressibility term ...). Each brick will participate on the global linear system to be solved (the tangent system for non linear problem). A brick is an object which derives from `getfem::mbrick_abstract <MODEL_STATE>` (which itself derive from the non-template class `getfem::mbrick_abstract_common_base`) with the following virtual methods to be defined:

```
brick.proper_update()
```

called each time the brick should update itself. In particular, this function is expected to assign the correct values to `proper_nb_dof` (the nb of new dof introduced by this brick), `proper_nb_constraints` and `proper_mixed_variables`. It may also precompute certain components (like stiffness matrices for linear problems).

<code>brick.do_compute_tangent_matrix(MS, i0, j0)</code>	the brick has to compute its own part of the tangent and constraint matrices (i0 and j0 are optional arguments representing the shifts in the matrices defined in MS).
<code>brick.do_compute_residual(MS, i0, j0)</code>	the brick has to compute its own part of the residual of the linear system and of the constraint system (i0 and j0 are the shifts in the residual vectors defined in MS).

Of course, each specific brick may have additional methods to build the brick, define some parameters and extract the solution from the model state variable. The brick may also do some extra efforts in order to avoid unnecessary recomputations (see for example the `K_uptodate` flag of the `getfem::mbrick_abstract_linear_pde` brick).

19.3 Brick parameters

Many bricks depend on one or more parameter fields. For example, the linear elasticity brick uses the two Lam coefficients λ and μ . These Lam coefficients are described as a field (i.e. with a `mesh_fem` and a vector of dof values), in a template structure `getfem::mbrick_parameter <VECTOR_TYPE>`.

Some problems require a matrix or a tensor field, instead of a scalar field. For example, the brick responsible for the Dirichlet condition is used to impose $h(x)u(x) = r(x)$ on a region of the mesh. When the `mesh_fem` is a vector one ($Q \geq 1$), $h(x)$ is a QQ matrix field, and $r(x)$ is a vector field of dimension Q . That case is also handled by the `getfem::mbrick_parameter` structure.

Basically, this structure contains

- a `getfem::mesh_fem` (whose `Qdim` is always equal to one!).
- a description of the field dimensions (scalar, matrix, ..)
- a vector, whose length is the field number of elements times the `nb_dof()` of the `mesh_fem`. For a matrix field $h(x)$, the order is the fortran one, with the dof number as the slowest varying index:

$$[h_{11}^1, h_{21}^1, h_{12}^1, h_{22}^1, h_{11}^2, \dots, h_{11}^n, h_{21}^n, h_{12}^n, h_{22}^n]$$

This structure provides these methods:

<code>field.mf()</code>	return the <code>mesh_fem</code> on which the field is defined.
<code>field.get()</code>	return the current dof data of the field.
<code>field.fsizes()</code>	return the field size, as a vector. For a scalar field, this is an empty vector, for a np matrix field, this is the vector $[n, p]$, etc.
<code>field.fsize()</code>	return the product of the elements of the vector <code>fsizes()</code> .
<code>field.set([mf,] V)</code>	change the field value. The value <code>V</code> can be a scalar value (constant field), a vector of length <code>fsize()</code> to set a constant non-scalar field, or a large vector of length <code>fsize()*mf().nb_dof()</code> to set a non-constant field. The <code>mesh_fem</code> <code>mf</code> is an optional parameter, hence it is possible to change the <code>mesh_fem</code> associated to the parameter (typically it is a polynomial <code>mesh_fem</code> of degree 0).

<code>set_diagonal(V)</code>	can be used with matrix fields, to set only the diagonal elements (V length should be <code>fsize()[0]</code> or <code>fsize()[0]*mf().nb_dof()</code>).
------------------------------	---

19.4 generic elliptic brick

The generic elliptic brick is a basic brick representing a term such as

$$-div(k\nabla u) = \dots$$

where u is a scalar field and the coefficient k is a positive scalar or a symmetric positive definite order two tensor field, or a symmetric positive definite order four tensor (and u a vector field). The constructor initializes the brick for a scalar constant coefficient k :

```
getfem::mbrick_generic_elliptic<MODEL_STATE> brick(mim, mf_u, k = 1.0);
```

where `mim` is a variable of type `getfem::mesh_im` defining the integration method used, and `mf_u` is a `getfem::mesh_fem` on the same mesh. `mf_u` describes the finite element method used for the unknown.

A local copy of the stiffness matrix K is stored in the brick, this obviously has a memory cost but allows not to recompute it each time when `compute_tangent_matrix(...)` is called.

In fact this bricks cover several situations. When k is a scalar coefficient, the brick represents a laplace operator which is componentwise if the `mf_u` represent a vector field (`mf_u.get_qdim()`>1). When k is an order two tensor coefficient, the brick represent a scalar generic elliptic operator which is componentwise if `mf_u` is a vector field. And finally, When k is an order four tensor coefficient, the brick represents a vectorial generic elliptic operator (for example linear elasticity with a generic Hooke tensor).

A general tensor field k can be set thanks to the two functions: `brick.set_coeff_dimension(d)` (with $d = 0, 2$ or 4) sets the tensor dimension, and `brick.coeff().set(mf_data, new_k)` sets the value of the tensor field (`mf_data` could be omitted, it is an order 0 element by default).

The following additional methods are available on this brick:

<code>brick.coeff()</code>	gives the access to the parameter k (see section on bricks parameters).
<code>brick.set_coeff_dimension(d)</code>	Set the tensor dimension of k . $d = 1$ for laplacian operator, $d = 2$ for generic scalar elliptic operator and $d = 4$ for generic vectorial elliptic operator.
<code>brick.get_solution(MS, V)</code>	After a solve, extract the solution of the model state variable MS and put it in the vector V.

19.5 Source term brick

The brick `getfem::mbrick_source_term` represents either a volumic source term or a Neumann condition, i.e. a term $\int_{\Omega} f.vdx$ or $\int_{\Gamma} f.vdx$ in the weak formulation, with Γ a part of $\partial\Omega$. This brick works for both real or complex terms in scalar or vectorial problems. The constructor of this brick is:

```
getfem::mbrick_source_term<MODEL_STATE> brick(problem, mf_data, F,
                                             bound=-1, num_fem=0);
getfem::mbrick_source_term<MODEL_STATE> brick(problem,
                                             bound=-1, num_fem=0);
```

where `problem` is the problem on which the source term will be added (a scalar elliptic brick for instance), `mf_data` is the finite element description for the source term f , `F` is a vector of type `MODEL_STATE::vector_type` which contains the values of the source term on each degree of freedom of `mf_data`, `bound` is an optional parameter specifying on which boundary of the main mesh fem of `problem` the Neumann condition is applied. If this parameter is omitted, a volumic source term will be taken into account. `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems (for example, in a mixed problem the `num_fem` 0 may correspond to the mesh_fem used for the velocity, and the `num_fem` 1 may correspond to the mesh_fem used for the pressure).

The following additional methods are available on this brick:

<code>brick.source_term()</code>	give the access to the parameter <code>F</code> defining the source term (see section on bricks parameters for how to change the value of the parameter)
----------------------------------	--

19.6 Constraint brick

The constraint brick `getfem::mbrick_constraint` adds constraints on the degrees of freedom of a mesh_fem. This brick is a base class for the Dirichlet condition bricks for instance. It can also be used on its own to add constraints on an unknown when this unknown is not fully determinated (for instance, when only a Neumann condition is present on the boundary of the domain). This brick deals directly with the vectorial format of the unknown, i.e. with a system representing the constraints $BU = R$, where B is a $n_c n_d$ matrix, U is the vector of unknown corresponding to a finite element method `mf_u` having n_d dofs and R is a vector of size n_c corresponding to the right hand side of the constraints. This corresponds to add n_c constraints to the system. These constraints have to be independant, which means that the matrix B has to be of maximal rank.

The brick offers three different manners to take the constraints into account. This is represented by an enum in `getfem/getfem_modeling.h`:

- `getfem::AUGMENTED_CONSTRAINTS` consists in the addition of Lagrange multipliers (one multiplier for each constraint) so that the final linear system is augmented with the corresponding number of multipliers.

The inconvenient of this approach is that the final linear system looses is larger, and is not positive definite.

- `getfem::PENALIZED_CONSTRAINTS`: add a penalization $\frac{1}{\varepsilon}(B^T BU - B^T R)$ to the linear system. The penalization parameter ε has a default value equal to 10^{-9} .

This method is simple and robust, and does not increase the linear system. However the condition number of the final system is worse.

- `getfem::ELIMINATED_CONSTRAINTS` consists in collecting all the constraints on the system in a global constraint system (in the `getfem::model_state` variable) and to “eliminate” the dofs concerned by the constraints before solving the linear system. This is done computing the kernel of the constraints and projecting the linear system on this kernel.

This method is efficient on “simple cases”, but it may not be very robust with high degree FEMs, it is not parallelisable, and the computation of the kernel takes a non-negligible amount of time.

The constructor of this brick is

```
getfem::mbrick_constraint<MODEL_STATE> brick(problem, num_fem);
```

where `problem` is the problem on which the constraints will be added (a generic elliptic brick for instance) and `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems (0 is the default).

The specification of the constraints is done using the method

```
brick.set_constraints(B, R);
```

The following additional methods are available on this brick:

<code>brick.set_constraints_rhs(R)</code>	changes only the right hand side of the constraints.
<code>brick.set_constraints_type(c)</code>	set the method to take into account the constraints. The parameter <code>c</code> is either <code>getfem::AUGMENTED_CONSTRAINTS</code> , <code>getfem::PENALIZED_CONSTRAINTS</code> , or <code>getfem::ELIMINATED_CONSTRAINTS</code> .
<code>brick.set_penalization_parameter(eps)</code>	set the penalization parameter for the method <code>getfem::PENALIZED_CONSTRAINTS</code> .

19.7 Dirichlet condition brick

The `getfem::mbrick_Dirichlet` brick allows to define a Dirichlet condition on a part Γ of the boundary of the domain (i.e. set the value of the unknown on this part of the boundary $u = r$). This brick is derived from the constraint brick and automatically set the constraints system $BU = R$. As a consequence, the methods of the constraint brick are available (`brick.set_constraints_type(c)` and `brick.set_penalization_parameter(eps)`).

In order to be able to treat arbitrariable finite element methods, the Dirichlet condition is considered in the weak form $\int_{\Gamma} u(x)v(x)d\Gamma = \int_{\Gamma} r(x)v(x)d\Gamma$ for all v taken in a space of convenient multipliers. This allows to describe the data $r(x)$ on a different fem than the unknown $u(x)$ and allows also to have a more stable condition when the unknown $u(x)$ is described on a complex fem like an Xfem using a standard lagrangian fem for the multipliers.

```
getfem::mbrick_Dirichlet<MODEL_STATE> brick(problem, bound, mf_mult, num_fem);
```

where `problem` is the problem on which the Dirichlet condition will be added, `bound` specifies on which boundary of the main mesh of `problem` the Dirichlet condition is applied, `mf_mult` is an optional parameter representing the fem for the multipliers (the default value is to take the same fem as the unknown) and `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems (0 is the default).

The fem `mf_mult` has to be chosen in order to satisfy the Babuska-Brezzi inf-sup condition (i.e. the fact that the matrix B , which represent here a mass matrix on the boundary Γ , is of maximal rank). It is satisfied when `mf_mult` is the same fem as the one for the unknown and generally when `mf_mult` is “less rich” than this fem.

By default, the prescribed value is zero. For non-homogenous Dirichlet condition $u = r$, the parameter `rhs` of the brick has to be set by the command:

```
brick.rhs().set(mf_data, R);
```

where `mf_data` is the finite element description for the data and `F` is a vector of type `MODEL_STATE::vector_type` which contains the values of the data on each degree of freedom of `mf_data`.

The following additional methods are available on this brick:

<code>brick.rhs()</code>	gives the access to the parameter representing the value of the Dirichlet condition.
<code>brick.set_constraints_type(c)</code>	set the method to take into account the constraints. The parameter <code>c</code> is either <code>getfem::AUGMENTED_CONSTRAINTS</code> , <code>getfem::PENALIZED_CONSTRAINTS</code> , or <code>getfem::ELIMINATED_CONSTRAINTS</code> .
<code>brick.set_penalization_parameter(eps)</code>	set the penalization parameter for the method <code>getfem::PENALIZED_CONSTRAINTS</code> .

Remark: except for the `getfem::AUGMENTED_CONSTRAINTS` option, an algorithm of simplification tries when it is possible to have a matrix B with only one element per line. This is possible when the mass matrix on Γ of `mf_u` the fem for the unknown and `mf_mult` is invertible.

19.8 Example of a complete Poisson problem

If `mf_u` and `mf_data` are valid finite element descriptions on a mesh representing the domain on which the problem is defined, the following sequence will define a Poisson (laplacian) problem with a Dirichlet condition on boundary 5 of `mf_u` and a Neumann condition on boundary 7 of `mf_u`:

```
typedef getfem::modeling_standard_plain_vector plain_vector;

plain_vector U(mf_u.nb_dof()), F(mf_data.nb_dof());

getfem::mbrick_generic_elliptic<> laplacian(mim, mf_u);

for (int i = 0; i < mf_data.nb_dof(); ++i) F(i) = ...;
getfem::mbrick_source_term<> volumic_source_term(laplacian, mf_data, F);

for (int i = 0; i < mf_data.nb_dof(); ++i) F(i) = ...;
getfem::mbrick_source_term<> neumann_condition(volumic_source_term, mf_data, F, 7);

for (int i = 0; i < mf_data.nb_dof(); ++i) F(i) = ...;
getfem::mbrick_Dirichlet<> final_model(neumann_condition, 5);
final_model.rhs().set(mf_data, F);

getfem::standard_model_state MS(final_model);
gmm::iteration iter(residual, 1, 40000);

getfem::standard_solve(MS, final_model, iter);

laplacian.get_solution(MS, U);
```

Remark how the bricks are linked, each condition is applied to the brick defined with the previous condition. The order of the conditions is of course arbitrary, you can define the Dirichlet condition before the source term for instance.

19.9 Predefined solvers

Of course, for many problems, it will be more convenient to make a specific solver. Even so, one generic solver is at the moment available to test your models quickly. It can also be taken as a model to build

your own solvers. It is defined in `getfem/getfem_model_solvers.h` and the call is

```
getfem::standard_solve(MS, problem, iter);
```

where `MS` is a model state variable, `problem` is the brick that represent your global problem and `iter` is an iteration object from Gmm++. See also the previous section for an example of use.

Note that SuperLu is used by default on “small” problems. You can also link MUMPS with Getfem (see section 6) and used the parallel version.

19.10 Isotropic linearized elasticity brick

The `getfem::mbrick_isotropic_linearized_elasticity` is a basic brick representing a term such as

$$-div(\sigma) = \dots;$$

with

$$\sigma = \lambda \text{tr}(\varepsilon(u))I + 2\mu\varepsilon(u); \quad \varepsilon(u) = (\nabla u + \nabla u^T)/2.$$

$\varepsilon(u)$ is the small strain tensor, σ is the stress tensor, λ and μ are the Lam coefficients. This represents the system of linearized isotropic elasticity. It can also be used with $\lambda = 0$ together with the linear incompressible brick to build the Stokes problem.

The constructors build the brick for constant Lam coefficients:

```
getfem::mbrick_isotropic_linearized_elasticity<MODEL_STATE>
brick(mim, mf_u);
```

where `mim` is a variable of type `getfem::mesh_im` defining the integration method used, `mf_u` is a valid fem descriptor. `mf_u` describe the finite element method used for the unknown.

The brick has two parameters, `lambda()` and `mu()` for the usual Lam coefficients. As they are `getfem::mbrick_parameter`, it is possible to use either a constant value (defined with for example `brick.lambda().set(100.)`), or a non constant value (for example `brick.lambda().set(mf_lambda, lambdav)` with `lambdav` of type `MODEL_STATE::vector_type`).

The stiffness matrix is “cached” in the brick (and available with `brick.get_K()`), it has a memory cost but avoids unnecessary recomputations each time `compute_tangent_matrix(...)` is called.

The following additional methods are available on this brick:

<code>brick.lambda()</code>	gives access to the brick parameter <code>lambda</code> .
<code>brick.mu()</code>	gives access to the brick parameter <code>mu</code> .
<code>brick.get_solution(MS, V)</code>	After a solve, extract the solution of the model state variable <code>MS</code> and put it in the vector <code>V</code> .
<code>brick.compute_Von_Mises_or_Tresca(MS, mf_vm, VM, tresca_flag)</code>	Compute the Von Mises criterion (or Tresca is <code>tresca_flag</code> is set to true) on the displacement field stored in <code>MS</code> . The stress is evaluated on the mesh.fem <code>mf_vm</code> and stored in the vector <code>VM</code> .

The program `elastostatic.cc` in the tests directory of Getfem++ distribution can be taken as a model of use of this brick.

19.11 Qu term brick

The `getfem::mbrick_QU_term` brick can be used to add boundary conditions of Fourier-Robin type like

$$\frac{\partial u}{\partial n} = Qu$$

for scalar problems, or

$$\sigma n = Qu$$

for linearized elasticity problems. Q is a scalar field in the scalar case or a matrix field in the vectorial case. This brick works for both real or complex terms in scalar or vectorial problems.

The constructor is the following:

```
getfem::mbrick_QU_term<MODEL_STATE> brick(problem, Q_diag=0.0, bound=-1, numfem=0);
```

where `problem` is the problem on which the condition will be added. `Q_diag` is a real for the homogeneous and diagonal case ($Q = Q_{diag} * I$). `bound` is the number of the boundary of `main_mesh_fem()` on which the condition will be applied. `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems.

The following additional methods are available on this brick:

<code>brick.Q()</code>	gives access to the brick parameter Q .
------------------------	---

19.12 Helmholtz brick

This brick represents the complex or real Helmholtz problem

$$\Delta u + k^2 u = \dots$$

where k the wave number is a real or complex value. For a complex version, a complex model state variable has to be used (for example `getfem::standard_complex_model_state`, see `helmholtz.cc` in the tests directory)

The constructor is

```
getfem::mbrick_Helmholtz<MODEL_STATE> brick(mim, mf_u, k);
```

where `mim` is a variable of type `getfem::mesh_im` defining the integration method used, `mf_u` describes the finite element method used for the unknown. `k` is the (homogeneous) wave number. It can be changed to a non-homogeneous wave number afterwards, with `brick.wave_number().set(mf_k, k)` etc.

The following additional methods are available on this brick:

<code>brick.wave_number()</code>	gives access to the brick parameter k .
<code>brick.get_solution(MS, V)</code>	After a solve, extract the solution of the model state variable MS and put it in the vector V .

19.13 linear incompressibility (or nearly incompressibility) brick

The `getfem::mbrick_linear_incomp` brick adds a linear incompressibility condition (or a nearly incompressible condition) in a problem of type

$$\operatorname{div}(u) = 0, \quad (\text{or } \operatorname{div}(u) = \varepsilon p)$$

This constraint is enforced with Lagrange multipliers representing the pressure, introduced in a mixed formulation.

The constructor for the incompressibility condition is:

```
getfem::mbrick_linear_incomp<MODEL_STATE> brick(problem, mf_p, numfem);
```

where `problem` is the problem on which the incompressibility condition is applied, `mf_p` is the finite element description for the pressure (be aware that the LBB inf-sup condition has to be satisfied between the `main_mesh_fem()` and `mf_p`. `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems.

The nearly incompressibility condition is used when it is switched on with `brick.set_penalized(true)`. The penalization parameter ε is accessed with `brick.penalization_coeff()`.

In nearly incompressible homogeneous linearized elasticity, one has $\varepsilon = 1/\lambda$ where λ is one of the Lamé coefficients.

For instance, the following program defines a Stokes problem with a source term and an homogeneous Dirichlet condition on boundary 0. `mf_u`, `mf_data` and `mf_p` have to be valid finite element descriptions on the same mesh.

```
typedef getfem::modeling_standard_plain_vector plain_vector;

plain_vector U(mf_u.nb_dof()), F(mf_data.nb_dof());

double mu = 1.0;
getfem::mbrick_isotropic_linearized_elasticity<>
  stokes(mim, mf_u, 0.0, mu);

getfem::mbrick_linear_incomp<> incomp(stokes, mf_p);

plain_vector F(mf_data.nb_dof());
for (int i = 0; i < mf_data.nb_dof(); ++i) F(i) = ...;
getfem::mbrick_source_term<> volumic_source_term(incomp, mf_data, F);

gmm::clear(F);
getfem::mbrick_Dirichlet<> final_model(volumic_source_term, 0);
final_model.rhs().set(mf_data, F);

getfem::standard_model_state MS(final_model);
gmm::iteration iter(residual, 1, 40000);
getfem::standard_solve(MS, final_model, iter);

stokes.get_solution(MS, U);
```

An example for a nearly incompressibility condition can be found in the program `tests/elastostatic.cc`.

19.14 Small displacement plasticity brick

The `getfem::mbrick_plasticity` brick modelizes small-displacement quasi-static plasticity problems. It is defined in `getfem/getfem_plasticity.h`

Plasticity happens when you stress an object too much, so that even when you remove the charge, constraints remain 'trapped' into the object. When a stress is applied to an object, if the stress is small enough, the displacement stays elastic. If that stress overrides a constant value called stress threshold (intrinsic to the object), then the displacements becomes plastic.

Quasi-static means that we do not take inertia into account, however the algorithm used needs some kind of a time representation. It is not possible to put the charge on the model at once, as it will render false results. Instead, we have to put the charge only a bit at a time, and calculate the deformation each time, hence the 'quasi-static' name. For instance, if you wish to put a 100N/m charge on your object, you should put it by small steps (20,40,60,80,100 is ok). We have to use that method to keep the consistency of the problem.

The constructor is

```
getfem::mbrick_plasticity(mesh_im &mim_, mesh_fem &mf_u_,
                          value_type lambdai, value_type mui,
                          value_type stress_threshold,
                          const abstract_constraints_project &t_proj_);
```

The `stress_threshold` is the 'elasticity limit': if constraints are below that limit, the problem is an elasticity problem. Otherwise, it is a plasticity one. `t_proj_` is an instance of a constraints projection object, such as `getfem::VM_projection`.

Using that constructor, you can build the problem like with any other bricks, adding the Volumic, Neumann and Dirichlet bricks as usual (look at `elastostatic.cc` and `plasticity.cc` in tests directory for examples), and call the solver with `getfem::standard_solve(MS, final_model, iter);`.

Once it's done, you need to know what the results are. You can know the displacement using `brick.get_solution(MS, U)`; where `MS` is defined by `getfem::standard_model_state MS(final_model);`, and `U` the displacement vector.

The Von Mises constraints can be obtained with `brick.compute_Von_Mises_or_Tresca(mf_vm, VM, tresca_flag)` (see the description in the linearized isotropic elasticity brick).

19.15 Contact and friction conditions brick

(to be documented, see the test program `tests/dynamic_friction.cc`, and the source file `getfem/getfem_Coulomb_friction.h`)

19.16 Linearized plate brick

(to be documented, see the test program `tests/plate.cc`)

- u_t is the membrane displacement.
- u_3 is the transverse displacement.
- θ is the rotation of the normal (section rotation).

Many specialized bricks are defined in `getfem/getfem_linearized_plates.h`:

- `getfem::mbrick_isotropic_linearized_plate`: linear plate model brick (for moderately thick plates, using the Reissner-Mindlin model).
- `getfem::mbrick_mixed_isotropic_linearized_plate`: mixed linear plate model brick (for thin plates, using Kirchhoff-Love model). The `getfem::mbrick_plate_closing` has to be used in conjunction with this one.
- `getfem::mbrick_plate_source_term`: apply a classical source term on the u_t , u_3 , and θ fields.
- `getfem::mbrick_plate_simple_support`: Dirichlet condition on u_t and u_3 , free rotation.
- `getfem::mbrick_plate_clamped_support`: Dirichlet condition on the displacement and the rotation.
- `getfem::mbrick_plate_closing`: free edges condition for mixed plate model brick. This brick has to be added for the mixed linearized plate brick after all other boundary conditions.

19.17 Large strain elasticity brick

The `getfem::mbrick_nonlinear_elasticity` brick represents a large strain elasticity problem. It is defined in `getfem/getfem_nonlinear_elasticity.h`

The constructor is:

```
getfem::mbrick_nonlinear_elasticity<MODEL_STATE> brick
  (Hyperelastic_Law, mim, mf_u, lawparams);
```

where `Hyperelastic_Law` is an object of type `getfem::abstract_hyperelastic_law` representing the considered hyperelastic law. It has to be chosen between:

```
getfem::SaintVenant_Kirchhoff_hyperelastic_law ~Hyperelastic_Law;
getfem::Ciarlet_Geymonat_hyperelastic_law ~Hyperelastic_Law;
getfem::Mooney_Rivlin_hyperelastic_law ~Hyperelastic_Law;
```

The Saint-Venant Kirchhoff law is a linearized law defined with the two Lam coefficients, Ciarlet Geymonat law is defined with the two Lam coefficients and an additional coefficient and the Mooney-Rivlin law is defined with two coefficients and is to be used with the large strain incompressibility condition.

`mf_u` describes the finite element method used for the unknown and `mf_data` the finite element method used for the parameters of the selected hyperelastic law. The parameters of the hyperelastic law are supplied in the vector `lawparams` (by default they are constant over the mesh, but you can change that later with `brick.params().set(mf, V)`).

The program `nonlinear_elastostatic.cc` in tests directory is an example of use of this brick with or without an incompressibility condition.

19.18 Large strain incompressibility brick

The `getfem::mbrick_nonlinear_incomp` brick adds an incompressibility condition in a large strain problem of type

$$\det(I + \nabla u) = 1,$$

For this, Lagrange multipliers representing the pressure are introduced in a mixed formulation.

The constructor is:

```
getfem::mbrick_nonlinear_incomp<MODEL_STATE> brick(problem, mf_p, numfem);
```

where `problem` is the problem on which the incompressibility condition is applied, `mf_p` is the finite element description for the pressure (be aware that the LBB (Ladyzhenskaja-Babuska-Brezzi) inf-sup condition has to be satisfied between the `main_mesh_fem()` and `mf_p`. `num_fem` is an optional parameter allowing to choose a fem if the problem has several fems.

The program `nonlinear_elastostatic.cc` in `tests` directory is an example of use of this brick.

20 Parallelization of GETFEM++

Of course, each different problem should require a different parallelization adapted to its specificities. You may build your own parallelization using the mesh regions to parallelize assembly procedures.

Nevertheless, the brick system offers a generic parallelization based on MPI (communication between processes), METIS (partition of the mesh) and MUMPS (parallel sparse direct solver). One has to compile GETFEM++ with the option `"-D GETFEM_PARA_LEVEL=2"` to use it. With this option, each mesh used is implicitly partitionned (using METIS) into a number of regions corresponding to the number of processors and the assembly procedures are parallelized. This means that the tangent matrix and the constraint matrix assembled in the `model_state` variable are distributed. The choice made (for the moment) is not to distribute the vectors. So that the right hand side vectors in the `model_state` variable are communicated to each processor (the sum of each contribution is made at the end of the assembly and each processor has the complete vector). Note that you have to think to the fact that the matrices stored by the bricks are all distributed.

Concerning the constraints, it is preferable to avoid the `getfem::ELIMINATED_CONSTRAINTS` option for a better parallelization (i.e. not to use the constraint matrix).

A model of parallelized program is `elastostatic.cc` in the directory `tests` of the distribution.

The following functions are also implicitly parallelized using the option `"-D GETFEM_PARA_LEVEL=2"`:

- `computation of norms` (`asm_L2_norm`, `asm_H1_norm`, `asm_H2_norm` ..., in `getfem/getfem_assembling.h`),
- `asm_mean_value` (in `getfem/getfem_assembling.h`),
- `error_estimate` (in `getfem/getfem_error_estimate.h`).

This means that these functions have to be called on each processor.

Parallelization of `getfem` is still considered a "work in progress"..

21 Catch errors

Errors used in GETFEM++ are defined in the file `gmm/gmm_except.h`. In order to make easier the error catching all errors derive from the type `std::logic_error` defined in the file `stdexcept` of the S.T.L.

A standard procedure, `GMM_STANDARD_CATCH_ERROR`, is defined in `gmm/gmm_except.h`. This procedure catches all errors and prints the error message when an error occurs. It can be used in the main procedure of the program as follows

```

int main(void) {
    try {
        ... main program ...
    }
    GMM_STANDARD_CATCH_ERROR;
}

```

22 Example: Laplacian program

The program `laplacian` is provided in the directory `tests` of GETFEM++ distribution. This program computes the solution of the Poisson problem in a parallelepiped domain in any dimension with various finite element methods and elements. This program can be used as a model to build application programs. It is built when a `gmake check` is done on the root directory of GETFEM++ (or just with `cd tests; make laplacian`).

Once the program is compiled you can test it executing the command

```
cd tests; ./laplacian laplacian.param
```

The file `laplacian.param` is the parameter file. You can edit it and test various situation. The program prints the L^2 and H^1 error from an exact solution.

The program `elastostatic` is built in a same way and compute the solution of linear elasticity problem. Many more examples can be found in the `tests` directory.

23 Appendix A. Finite element method list

Let us recall that all finite element methods defined in GETFEM++ are declared in the file `getfem_fem.h` and that a descriptor on a finite element method is obtained thanks to the function

```
getfem::pfem pf = getfem::fem_descriptor("name of method");
```

where "name of method" is a string to be chosen among the existing methods.

23.1 Dof graphical codification

•	Value of the function at the node
→ ←	Value of the gradient along the first coordinate
↑ ↓	Value of the gradient along the second coordinate
↗ ↘	Value of the gradient along the third coordinate for 3D elements
⊙	Value of the whole gradient at the node
⊥→	Value of the normal derivative to a face
⇨ ⇩	Value of the second derivative along the first coordinate (twice)
⇩ ⇨	Value of the second derivative along the second coordinate (twice)
↗↗ ↘↘	Value of the second cross derivative in 2D or second derivative along the third coordinate (twice) in 3D.
⊙⊙	Value of the whole second derivative (hessian) at the node
→	Scalar product with a certain vector (for instance an edge) for a vectorial element
⊥→	Scalar product with the normal to a face for a vectorial element
⊙	Bubble function on an element or a face, to be specified.
×	Lagrange hierarchical d.o.f. Value at the node in a space of details.

Figure 6: *Symbols representing degree of freedom types*

23.2 Classical P_K Lagrange elements on simplices

It is possible to define a classical P_K Lagrange element of arbitrary dimension and arbitrary degree. Each degree of freedom of such an element corresponds to the value of the function on a corresponding node. The grid of node is the so-called Lagrange grid. Figures 7, 8 and 9 show examples in dimension 1, 2 and 3.

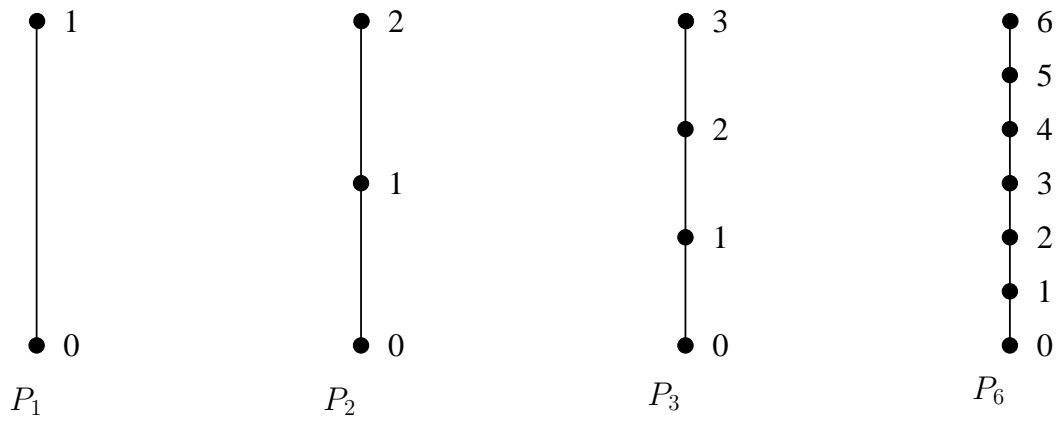


Figure 7: *Examples of classical P_K Lagrange elements on a segment.*

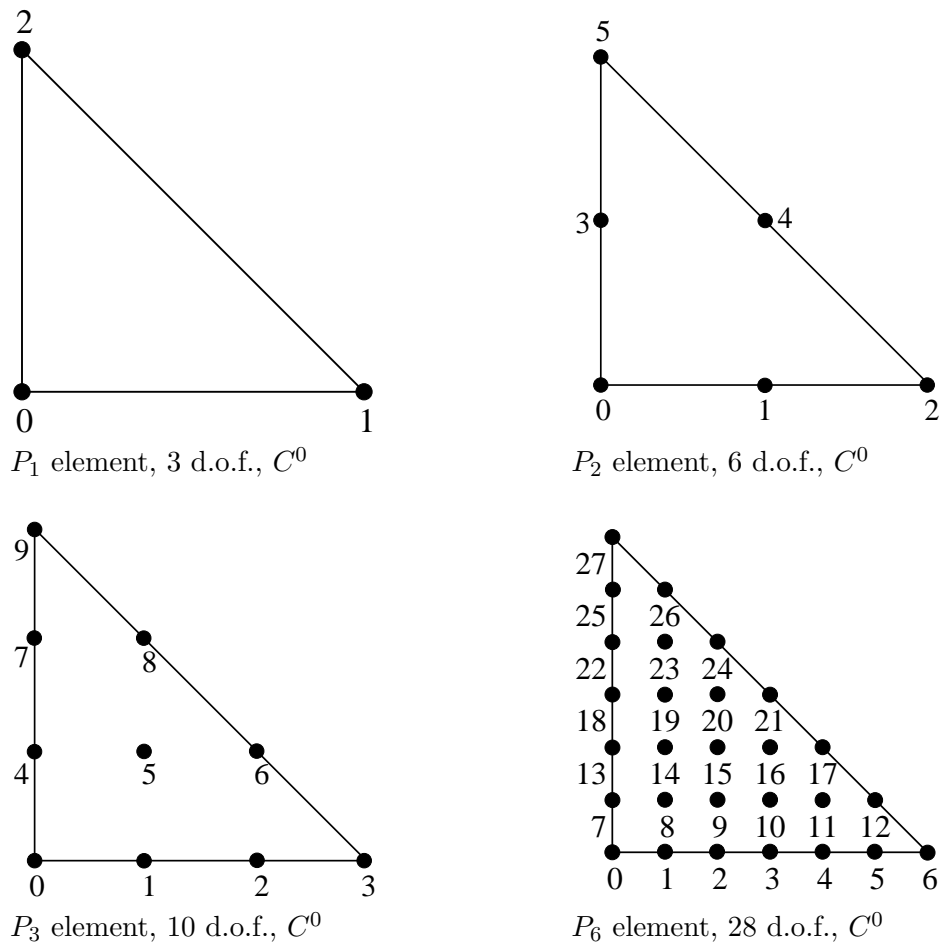


Figure 8: *Examples of classical P_K Lagrange elements on a triangle.*

The number of degrees of freedom for a classical P_K Lagrange element of dimension P and degree

K is $\frac{(P+K)!}{P!K!}$. For instance, in dimension 2 ($P=2$), this value is $\frac{(K+1)(K+2)}{2}$ and in dimension 3 ($P=3$), it is $\frac{(K+1)(K+2)(K+3)}{6}$.

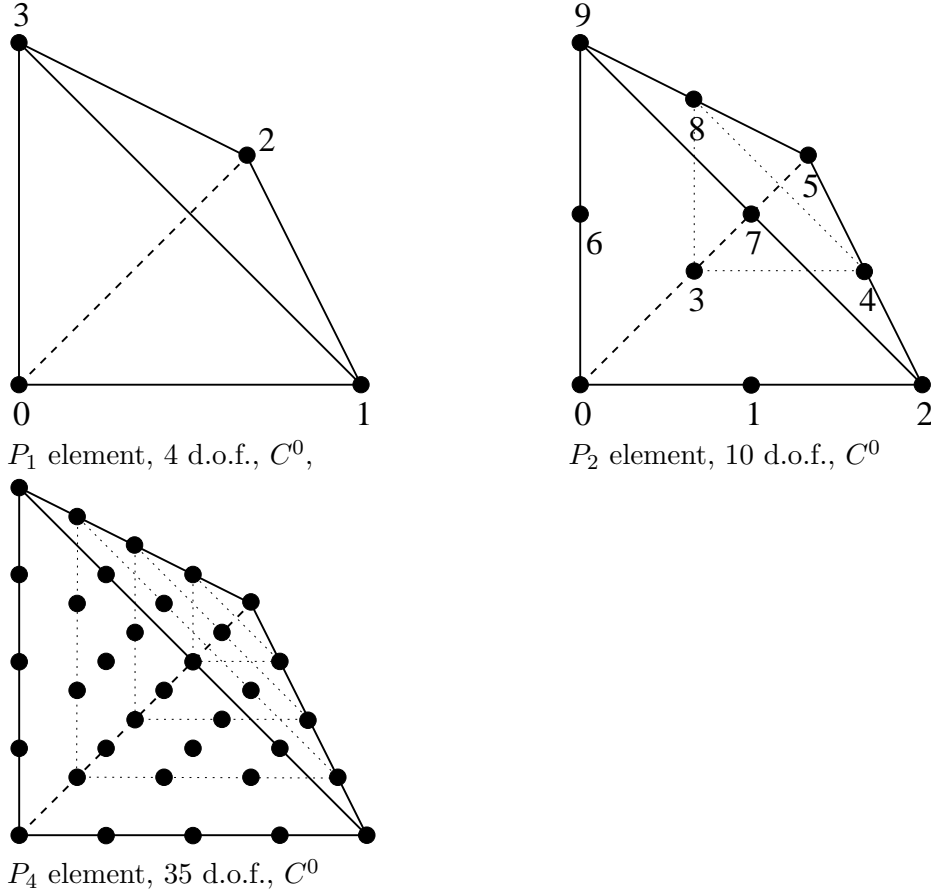


Figure 9: *Examples of classical P_K Lagrange elements on a tetrahedron.*

The particular way used in GETFEM++ to numerate the nodes are also shown in figures 7, 8 and 9. Using another numeration, let

$$i_0, i_1, \dots, i_P,$$

be some indices such that

$$0 \leq i_0, i_1, \dots, i_P \leq K, \quad \text{and} \quad \sum_{n=0}^P i_n = K.$$

Then, the coordinate of a node can be computed as

$$a_{i_0, i_1, \dots, i_P} = \sum_{n=0}^P \frac{i_n}{K} S_n, \quad \text{for } K \neq 0,$$

where S_0, S_1, \dots, S_N are the vertices of the simplex (for $K=0$ the particular choice $a_{0,0,\dots,0} = \sum_{n=0}^P \frac{1}{P+1} S_n$

has been chosen). Then each base function, corresponding of each node a_{i_0, i_1, \dots, i_P} is defined by

$$\phi_{i_0, i_1, \dots, i_P} = \prod_{n=0}^P \prod_{j=0}^{i_n-1} \left(\frac{K\lambda_n - j}{j+1} \right).$$

where λ_n are the barycentric coordinates, i.e. the polynomials of degree 1 whose value is 1 on the vertex S_n and whose value is 0 on other vertices. On the reference element, one has

$$\lambda_n = x_n, \quad 0 \leq n < P,$$

$$\lambda_P = 1 - x_0 - x_1 - \dots - x_{P-1}.$$

When between two elements of the same degrees (even with different dimensions), the d.o.f. of a common face are linked, the element is of class C^0 . This means that the global polynomial is continuous. If you try to link elements of different degrees, you will get some trouble with the unlinked d.o.f. This is not automatically supported by GETFEM++, so you will have to support it (add constraints on these d.o.f.).

For some applications (computation of a gradient for instance) one may not want the d.o.f. of a common face to be linked. This is why there are two versions of the classical P_K Lagrange element.

Classical P_K Lagrange element "FEM_PK(P, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$K,$ $0 \leq K \leq 255$	$P,$ $1 \leq P \leq 255$	$\frac{(K+P)!}{K!P!}$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

Discontinuous P_K Lagrange element "FEM_PK_DISCONTINUOUS(P, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$K,$ $0 \leq K \leq 255$	$P,$ $1 \leq P \leq 255$	$\frac{(K+P)!}{K!P!}$	discon- tinuous	No ($Q = 1$)	Yes ($M = Id$)	Yes

Even though Lagrange elements are defined for arbitrary degrees, to choose a high degree can be problematic for a large number of applications due to the "noisy" characteristic of the lagrange basis. These elements are recommended for the basic interpolation but for p.d.e. applications elements with hierarchical basis are preferable (see the corresponding section).

23.3 Classical Lagrange elements on other geometries

Classical Lagrange elements on parallelepipeds or prisms are obtained as tensor product of Lagrange elements on simplices. When two elements are defined, one on a dimension P^1 and the other in dimension P^2 , one obtains the base functions of the tensorial product (on the reference element) as

$$\phi'_{ij}(x, y) = \phi_i^1(x)\phi_j^2(y), \quad x \in \mathbb{R}^{P^1}, y \in \mathbb{R}^{P^2},$$

where ϕ_i^1 and ϕ_j^2 are respectively the base functions of the first and second element.

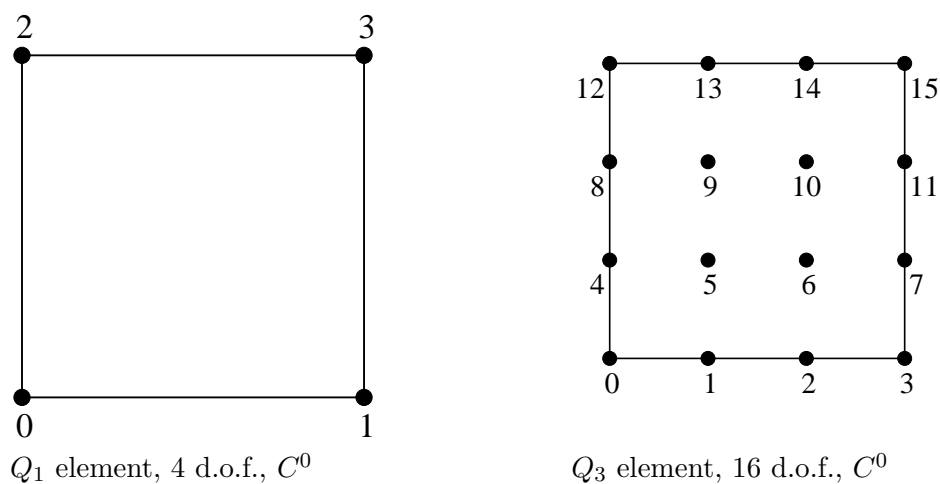


Figure 10: *Examples of classical Q_K Lagrange elements in dimension 2*

The Q_K element on a parallelepiped of dimension P is obtained as the tensorial product of P classical P_K elements on the segment. Examples in dimension 2 are shown in figure 10 and in dimension 3 in figure 11.

A prism in dimension $P > 1$ is the direct product of a simplex of dimension $P - 1$ with a segment. The $P_K \otimes P_K$ element on this prism is the tensorial product of the classical P_K element on a simplex of dimension $P - 1$ with the classical P_K element on a segment. For $P = 2$ this coincide with a parallelepiped. Examples in dimension 3 are shown in figure 11. This is also possible not to have the same degree on each dimension. An example is shown on figure 12.

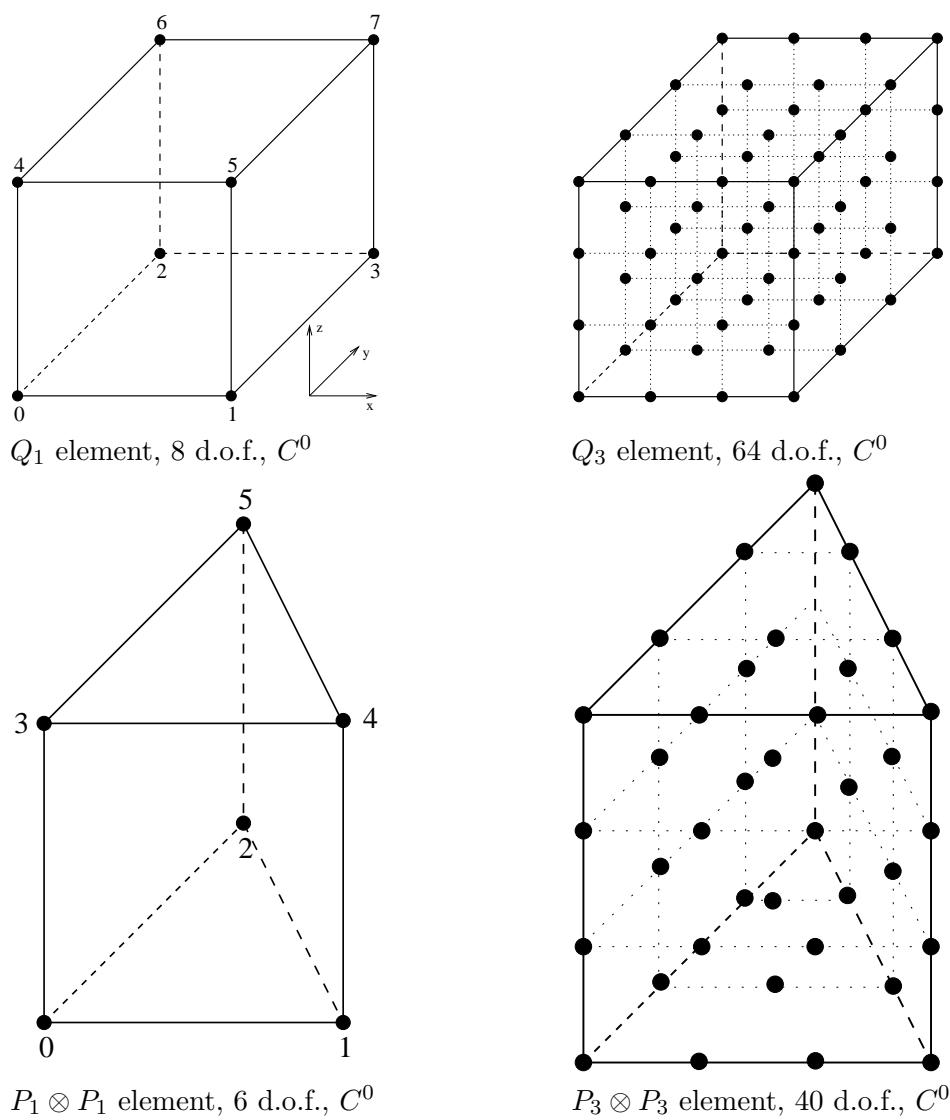


Figure 11: *Examples of classical Lagrange elements in dimension 3*

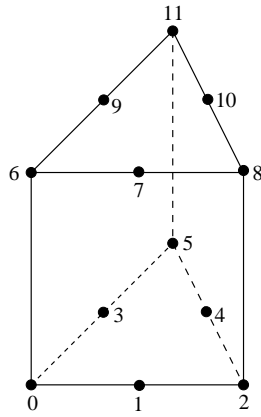
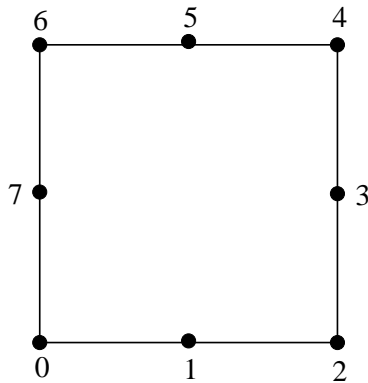


Figure 12: $P_2 \otimes P_1$ Lagrange element on a prism, 12 d.o.f., C^0

Q_K Lagrange element on parallelepipeds "FEM_QK(P, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$KP,$ $0 \leq K \leq 255$	$P,$ $2 \leq P \leq 255$	$(K + 1)^P$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

$P_K \otimes P_K$ Lagrange element on prisms "FEM_PK_PRISM(P, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$2K,$ $0 \leq K \leq 255$	$P,$ $2 \leq P \leq 255$	$(K + 1) \times \frac{(K + P - 1)!}{K!(P - 1)!}$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

$P_{K_1} \otimes P_{K_2}$ Lagrange element on prisms "FEM_PRODUCT(FEM_PK(P-1, K ₁), FEM_PK(1, K ₂))"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$K_1 + K_2,$ $0 \leq K_1, K_2 \leq 255$	$P,$ $2 \leq P \leq 255$	$(K_2 + 1) \times \frac{(K_1 + P - 1)!}{K_1!(P - 1)!}$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes



Incomplete Q_2 element, 8 d.o.f., C^0

Incomplete Q_2 Lagrange element on quadrilateral (Quad 8 serendipity element) "FEM_INCOMPLETE_Q2"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	8	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

23.4 Elements with hierarchical basis

The idea behind hierarchical basis is the description of the solution at different level: a rough level, a more refined level ... In the same discretisation some degrees of freedom represent the rough description, some other the more refined and so on. This corresponds to imbricated spaces of discretisation. The hierarchical basis contains a basis of each of these spaces (this is not the case in classical Lagrange elements when the mesh is refined).

Among the advantages, the condition number of rigidity matrices can be greatly improved, it allows local raffinement and a resolution with a multigrid approach.

23.4.1 Hierarchical elements with respect to the degree

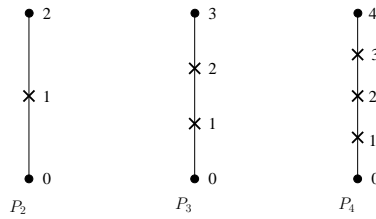


Figure 13: P_K Hierarchical element on a segment, C^0

P_K Classical Lagrange element on simplices but with a hierarchical basis with respect to the degree "FEM_PK_HIERARCHICAL(P,K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
K , $0 \leq K \leq 255$	P , $1 \leq P \leq 255$	$\frac{(K+P)!}{K!P!}$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

Q_K Classical Lagrange element on parallelepipeds but with a hierarchical basis with respect to the degree "FEM_QK_HIERARCHICAL(P,K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
K , $0 \leq K \leq 255$	P , $2 \leq P \leq 255$	$(K+1)^P$	C^0	No ($Q = 1$)	Yes ($M = Id$)	Yes

P_K Classical Lagrange element on prisms but with a hierarchical basis with respect to the degree "FEM_PK_PRISM_HIERARCHICAL(P,K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
$K,$ $0 \leq K \leq 255$	$P,$ $2 \leq P \leq 255$	$(K+1) \times \frac{(K+P-1)!}{K!(P-1)!}$	C^0	No ($Q=1$)	Yes ($M=Id$)	Yes

some particular choices: P_4 will be built with the basis of the P_1 , the additional basis of the P_2 then the additional basis of the P_4 .

P_6 will be built with the basis of the P_1 , the additional basis of the P_2 then the additional basis of the P_6 (not with the basis of the P_1 , the additional basis of the P_3 then the additional basis of the P_6 , it is possible to build the latter with "FEM_GEN_HIERARCHICAL(a,b)")

23.4.2 Composite elements

The principal interest of the composite elements is to build hierarchical elements. But this tool can also be used to build piecewise polynomial elements.

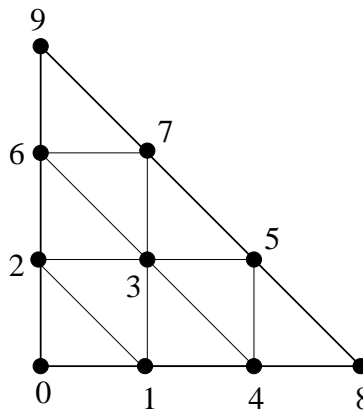


Figure 14: *composite element* "FEM_STRUCTURED_COMPOSITE(FEM_PK(2,1), 3)"

Composition of a finite element method on a element with S subdivisions "FEM_STRUCTURED_COMPOSITE(FEM1, S)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
degree of FEM1	dimension of FEM1	variable	variable	No ($Q=1$)	If FEM1 is	piecewise

It is important to use a corresponding composite integration method.

23.4.3 Hierarchical composite elements

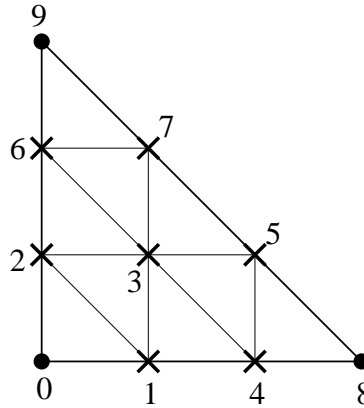


Figure 15: *hierarchical composite element "FEM_PK_HIERARCHICAL_COMPOSITE(2,1,3)"*

Hierarchical composition of a P_K finite element method on a simplex with S subdivisions "FEM_PK_HIERARCHICAL_COMPOSITE(P,K,S)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
K	P	$\frac{(SK + P)!}{(SK)!P!}$	variable	No ($Q = 1$)	Yes	piecewise

hierarchical composition of a hierarchical P_K finite element method on a simplex with S subdivisions "FEM_PK_FULL_HIERARCHICAL_COMPOSITE(P,K,S)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
K	P	$\frac{(SK + P)!}{(SK)!P!}$	variable	No ($Q = 1$)	Yes	piecewise

Other constructions are possible thanks to "FEM_GEN_HIERARCHICAL(FEM1, FEM2)" and "FEM_STRUCTURED_COMPOSITE(FEM1, S)"

It is important to use a corresponding composite integration method.

23.5 Classical vectorial elements

23.5.1 Raviart-Thomas of lowest order elements

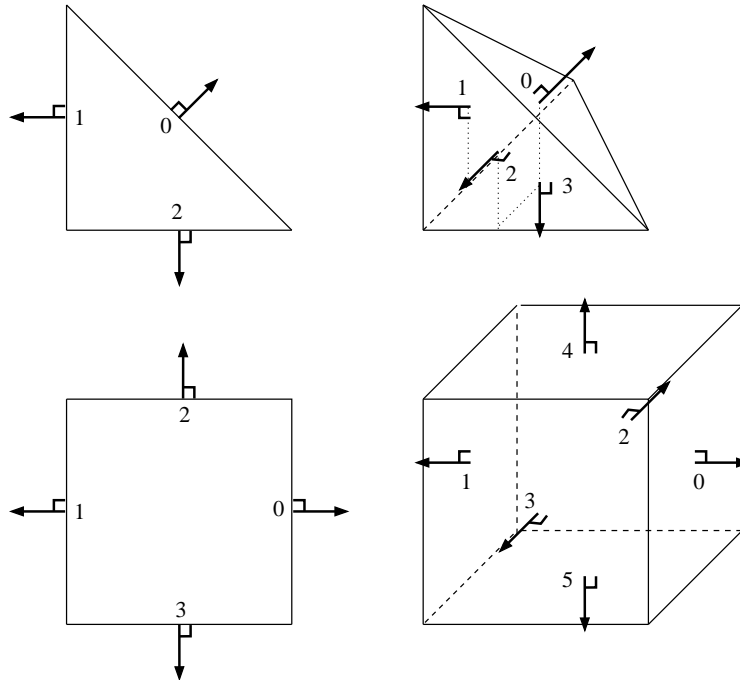


Figure 16: *RTO elements in dimension two and three. ($P+1$ dof, $H(div)$)*

Raviart-Thomas of lowest order element on simplices "FEM_RT0(P)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
1	P	$P + 1$	$H(div)$	Yes ($Q = P$)	No	Yes

Raviart-Thomas of lowest order element on parallelepipeds (quadrilaterals, hexahedrals) "FEM_RTOQ(P)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
1	P	$2P$	$H(div)$	Yes ($Q = P$)	No	Yes

23.5.2 Nedelec (or Whitney) edge elements

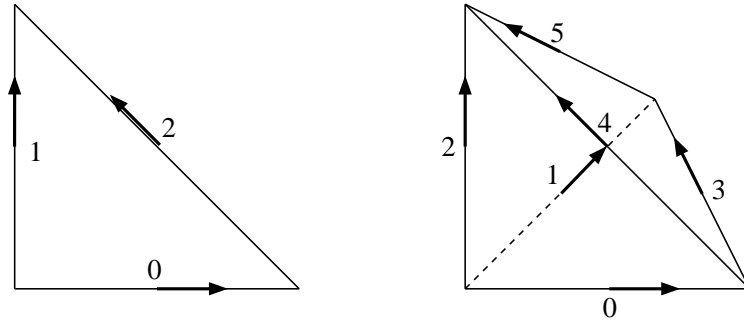


Figure 17: Nedelec edge elements in dimension two and three. ($P(P+1)/2$ dof, $H(rot)$)

Nedelec (or Whitney) edge element "FEM_NEDELEC(P)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
1	P	$P(P+1)/2$	$H(rot)$	Yes ($Q = P$)	No	Yes

23.6 Specific elements in dimension 1

23.6.1 GaussLobatto element

The 1D GaussLobatto P_K element is similar to the classical P_K fem on the segment, but the nodes are given by the Gauss-Lobatto-Legendre quadrature rule of order $2K - 1$. This FEM is known to lead to better conditioned linear systems, and can be used with the corresponding quadrature to perform mass-lumping (on segments or parallelepipeds).

The polynomials coefficients have been pre-computed with Maple (they require the inversion of an ill-conditioned system), hence they are only available for the following values

of K : 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 24, 32. Note that for $K = 1$ and $K = 2$, this is the classical $P1$ and $P2$ fem.

GaussLobatto P_K element on the segment "FEM_PK_GAUSSLOBATTO1D(K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
K	1	$K + 1$	C^0	No ($Q = 1$)	Yes	Yes

23.6.2 Hermite element

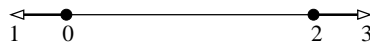


Figure 18: P_3 Hermite element on a segment, 4 d.o.f., C^1

Base functions on the reference element

$$\begin{aligned} \varphi'_0 &= (2x + 1)(x - 1)^2, & \varphi'_1 &= x(x - 1)^2, \\ \varphi'_2 &= x^2(3 - 2x), & \varphi'_3 &= x^2(x - 1). \end{aligned}$$

This element is close to be τ -equivalent but it is not. On the real element the value of the gradient on vertices will be multiplied by the gradient of the geometric transformation. The matrix M is not equal to identity but is still diagonal.

Hermite element on the segment "FEM_HERMITE(1)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	1	4	C^1	No ($Q = 1$)	No	Yes

23.6.3 Lagrange element with an additional bubble function

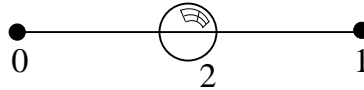


Figure 19: P_1 Lagrange element on a segment with additional internal bubble function, 3 d.o.f., C^0

Lagrange P_1 element with an additional internal bubble function "FEM_PK_WITH_CUBIC_BUBBLE(1, 1)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
2	1	3	C^0	No ($Q = 1$)	Yes	Yes

23.7 Specific elements in dimension 2

23.7.1 Elements with additional bubble functions

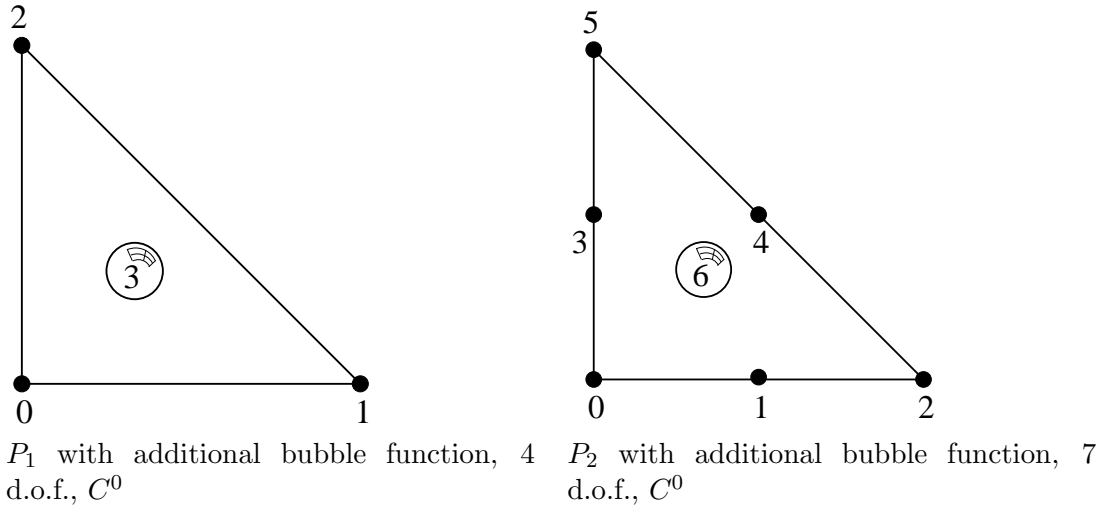


Figure 20: *Lagrange element on a triangle with additional internal bubble function*

Lagrange P_1 or P_2 element with an additional internal bubble function						
"FEM_PK_WITH_CUBIC_BUBBLE(2, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	4 or 7	C^0	No ($Q = 1$)	Yes	Yes

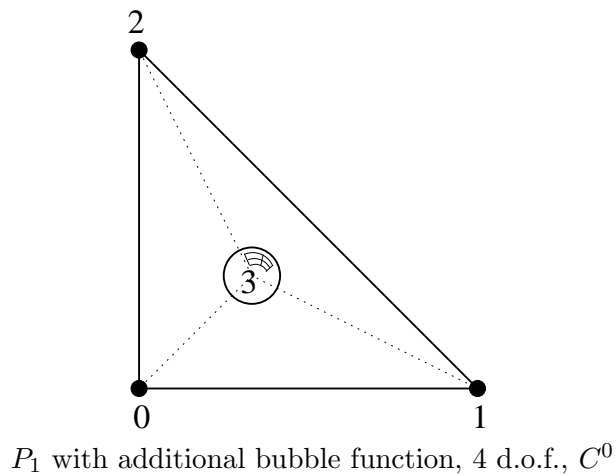


Figure 21: *P_1 Lagrange element on a triangle with additional internal piecewise linear bubble function*

Lagrange P_1 with an additional internal piecewise linear bubble function "FEM_P1_PIECEWISE_LINEAR_BUBBLE"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
1	2	4	C^0	No ($Q = 1$)	Yes	Piecewise

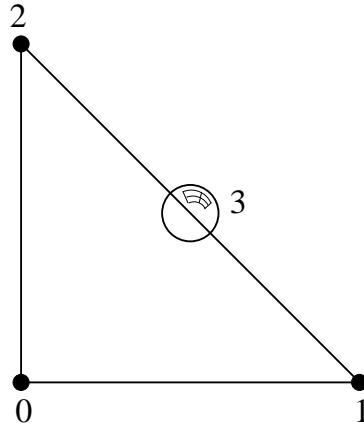


Figure 22: P_1 Lagrange element on a triangle with additional bubble function on face 0, 4 d.o.f., C^0

Lagrange P_1 element with an additional bubble function on face 0 "FEM_P1_BUBBLE_FACE(2)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
2	2	4	C^0	No ($Q = 1$)	Yes	Yes

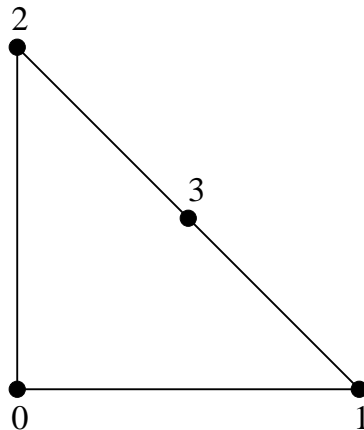


Figure 23: P_1 Lagrange element on a triangle with additional d.o.f on face 0, 4 d.o.f., C^0

P_1 Lagrange element on a triangle with additional d.o.f on face 0 "FEM_P1_BUBBLE_FACE_LAG"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
2	2	4	C^0	No ($Q = 1$)	Yes	Yes

23.7.2 Non-conforming P_1 element

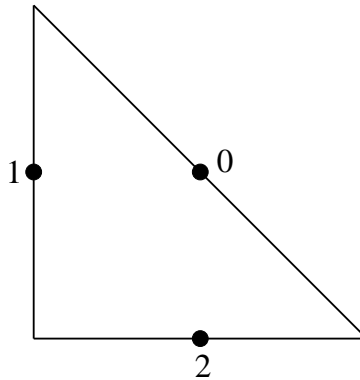


Figure 24: P_1 non-conforming element on a triangle, 3 d.o.f., discontinuous

P_1 non-conforming element on a triangle "FEM_P1_NONCONFORMING"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
1	2	3	discon- tinuous	No ($Q = 1$)	Yes	Yes

23.7.3 Hermite element

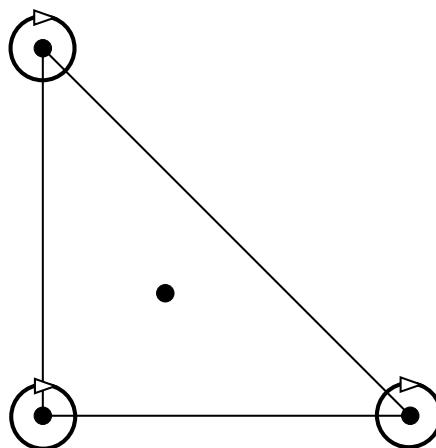


Figure 25: Hermite element on a triangle, P_3 , 10 d.o.f., C^0

Base functions on the reference element:

$$\begin{aligned}
 \varphi'_0 &= (1-x-y)(1+x+y-2x^2-2y^2-11xy), & (\varphi'_0(0,0) &= 1), \\
 \varphi'_1 &= x(1-x-y)(1-x-2y), & (\partial_x \varphi'_1(0,0) &= 1), \\
 \varphi'_2 &= y(1-x-y)(1-2x-y), & (\partial_y \varphi'_2(0,0) &= 1), \\
 \varphi'_3 &= -2x^3+7x^2y+7xy^2+3x^2-7xy, & (\varphi'_3(1,0) &= 1), \\
 \varphi'_4 &= x^3-2x^2y-2xy^2-x^2+2xy, & (\partial_x \varphi'_4(1,0) &= 1), \\
 \varphi'_5 &= xy(y+2x-1), & (\partial_y \varphi'_5(1,0) &= 1), \\
 \varphi'_6 &= 7x^2y+7xy^2-2y^3+3y^2-7xy, & (\varphi'_6(0,1) &= 1), \\
 \varphi'_7 &= xy(x+2y-1), & (\partial_x \varphi'_7(0,1) &= 1), \\
 \varphi'_8 &= y^3-2x^2y-2xy^2-y^2+2xy, & (\partial_y \varphi'_8(0,1) &= 1), \\
 \varphi'_9 &= 27xy(1-x-y), & (\varphi'_9(1/3,1/3) &= 1),
 \end{aligned}$$

This element is not τ -equivalent (The matrix M is not equal to identity). On the real element linear combinations of φ'_4 and φ'_7 are used to match the gradient on the corresponding vertex. Idem for the two couples (φ'_5, φ'_8) and (φ'_6, φ'_9) for the two other vertices.

Hermite element on a triangle "FEM_HERMITE(2)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	10	C^0	No ($Q = 1$)	No	Yes

23.7.4 Morley element

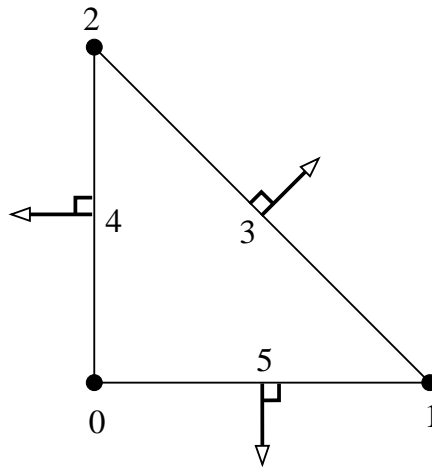


Figure 26: *triangle Morley element, P_2 , 6 d.o.f., C^0*

This element is not τ -equivalent (The matrix M is not equal to identity). In particular, it can be used for non-conforming discretization of fourth order problems, despite the fact that it is not C^0 .

Morley element on a triangle "FEM_MORLEY"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
2	2	6		No ($Q = 1$)	No	Yes

23.7.5 Argyris element

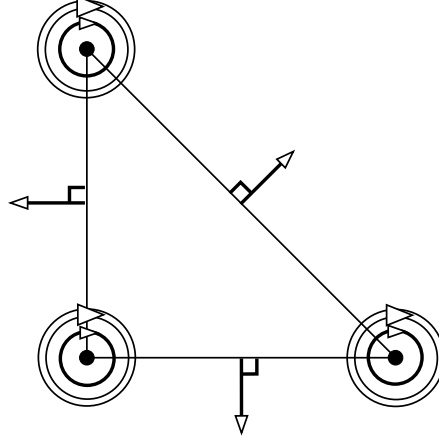


Figure 27: *Argyris element, P_5 , 21 d.o.f., C^1*

The base functions on the reference element are:

$$\begin{aligned}
 \varphi'_0(x, y) &= 1 - 10x^3 - 10y^3 + 15x^4 - 30x^2y^2 + 15y^4 - 6x^5 + 30x^3y^2 + 30x^2y^3 - 6y^5, & (\varphi'_0(0, 0) &= 1), \\
 \varphi'_1(x, y) &= x - 6x^3 - 11xy^2 + 8x^4 + 10x^2y^2 + 18xy^3 - 3x^5 + x^3y^2 - 10x^2y^3 - 8xy^4, & (\partial_x \varphi'_1(0, 0) &= 1), \\
 \varphi'_2(x, y) &= y - 11x^2y - 6y^3 + 18x^3y + 10x^2y^2 + 8y^4 - 8x^4y - 10x^3y^2 + x^2y^3 - 3y^5, & (\partial_y \varphi'_2(0, 0) &= 1), \\
 \varphi'_3(x, y) &= 0.5x^2 - 1.5x^3 + 1.5x^4 - 1.5x^2y^2 - 0.5x^5 + 1.5x^3y^2 + x^2y^3, & (\partial_{xx}^2 \varphi'_3(0, 0) &= 1), \\
 \varphi'_4(x, y) &= xy - 4x^2y - 4xy^2 + 5x^3y + 10x^2y^2 + 5xy^3 - 2x^4y - 6x^3y^2 - 6x^2y^3 - 2xy^4, & (\partial_{xy}^2 \varphi'_4(0, 0) &= 1), \\
 \varphi'_5(x, y) &= 0.5y^2 - 1.5y^3 - 1.5x^2y^2 + 1.5y^4 + x^3y^2 + 1.5x^2y^3 - 0.5y^5, & (\partial_{yy}^2 \varphi'_5(0, 0) &= 1), \\
 \varphi'_6(x, y) &= 10x^3 - 15x^4 + 15x^2y^2 + 6x^5 - 15x^3y^2 - 15x^2y^3, & (\varphi'_6(1, 0) &= 1), \\
 \varphi'_7(x, y) &= -4x^3 + 7x^4 - 3.5x^2y^2 - 3x^5 + 3.5x^3y^2 + 3.5x^2y^3, & (\partial_x \varphi'_7(1, 0) &= 1), \\
 \varphi'_8(x, y) &= -5x^2y + 14x^3y + 18.5x^2y^2 - 8x^4y - 18.5x^3y^2 - 13.5x^2y^3, & (\partial_y \varphi'_8(1, 0) &= 1), \\
 \varphi'_9(x, y) &= 0.5x^3 - x^4 + 0.25x^2y^2 + 0.5x^5 - 0.25x^3y^2 - 0.25x^2y^3, & (\partial_{xx}^2 \varphi'_9(1, 0) &= 1), \\
 \varphi'_{10}(x, y) &= x^2y - 3x^3y - 3.5x^2y^2 + 2x^4y + 3.5x^3y^2 + 2.5x^2y^3, & (\partial_{xy}^2 \varphi'_{10}(1, 0) &= 1), \\
 \varphi'_{11}(x, y) &= 1.25x^2y^2 - 0.75x^3y^2 - 1.25x^2y^3, & (\partial_{yy}^2 \varphi'_{11}(1, 0) &= 1), \\
 \varphi'_{12}(x, y) &= 10y^3 + 15x^2y^2 - 15y^4 - 15x^3y^2 - 15x^2y^3 + 6y^5, & (\varphi'_{12}(0, 1) &= 1), \\
 \varphi'_{13}(x, y) &= -5xy^2 + 18.5x^2y^2 + 14xy^3 - 13.5x^3y^2 - 18.5x^2y^3 - 8xy^4, & (\partial_x \varphi'_{13}(0, 1) &= 1), \\
 \varphi'_{14}(x, y) &= -4y^3 - 3.5x^2y^2 + 7y^4 + 3.5x^3y^2 + 3.5x^2y^3 - 3y^5, & (\partial_y \varphi'_{14}(0, 0) &= 1), \\
 \varphi'_{15}(x, y) &= 1.25x^2y^2 - 1.25x^3y^2 - 0.75x^2y^3, & (\partial_{xx}^2 \varphi'_{15}(0, 1) &= 1), \\
 \varphi'_{16}(x, y) &= xy^2 - 3.5x^2y^2 - 3xy^3 + 2.5x^3y^2 + 3.5x^2y^3 + 2xy^4, & (\partial_{xy}^2 \varphi'_{16}(0, 1) &= 1), \\
 \varphi'_{17}(x, y) &= 0.5y^3 + 0.25x^2y^2 - y^4 - 0.25x^3y^2 - 0.25x^2y^3 + 0.5y^5, & (\partial_{yy}^2 \varphi'_{17}(0, 1) &= 1), \\
 \varphi'_{18}(x, y) &= \sqrt{2}(-8x^2y^2 + 8x^3y^2 + 8x^2y^3), & (\sqrt{0.5}(\partial_x \varphi'_{18}(0.5, 0.5) + \partial_y \varphi'_{18}(0.5, 0.5)) &= 1), \\
 \varphi'_{19}(x, y) &= -16xy^2 + 32x^2y^2 + 32xy^3 - 16x^3y^2 - 32x^2y^3 - 16xy^4, & (-\partial_x \varphi'_{19}(0, 0.5) &= 1), \\
 \varphi'_{20}(x, y) &= -16x^2y + 32x^3y + 32x^2y^2 - 16x^4y - 32x^3y^2 - 16x^2y^3, & (-\partial_y \varphi'_{20}(0.5, 0) &= 1),
 \end{aligned}$$

This element is not τ -equivalent (The matrix M is not equal to identity). On the real element linear combinations of the transformed base functions φ'_i are used to match the gradient, the second derivatives and the normal derivatives on the faces. Note that the use of the matrix M allows to define Argyris element even with nonlinear geometric transformations (for instance to treat curved boundaries).

Argyris element on a triangle "FEM_ARGYRIS"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
5	2	21	C^1	No ($Q = 1$)	No	Yes

23.7.6 Hsieh-Clough-Tocher element

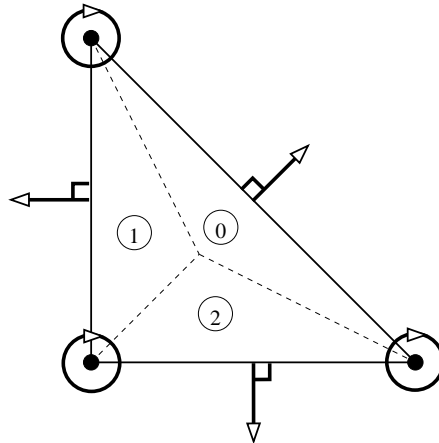


Figure 28: *Hsieh-Clough-Tocher (HCT) element, P_3 , 12 d.o.f., C^1*

This element is not τ -equivalent. This is a composite element. Polynomial of degree 3 on each of the three sub-triangles (see figure 28 and [1]). It is strongly advised to use a `IM_HCT_COMPOSITE` integration method with this finite element. The numeration of the dof is the following: 0, 3 and 6 for the lagrange dof on the first second and third vertex respectively; 1, 4, 7 for the derivative with respects to the first variable; 2, 5, 8 for the derivative with respects to the second variable and 9, 10, 11 for the normal derivatives on face 0, 1, 2 respectively.

HCT element on a triangle "FEM_HCT_TRIANGLE"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	12	C^1	No ($Q = 1$)	No	piecewise

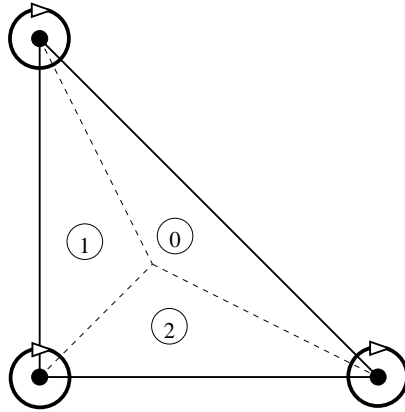


Figure 29: *Reduced Hsieh-Clough-Tocher (reduced HCT) element, P_3 , 9 d.o.f., C^1*

This element exists also in its reduced form, where the normal derivatives are assumed to be polynomial of degree one on each edge (see figure 29)

Reduced HCT element on a triangle						
"FEM_REDUCED_HCT_TRIANGLE"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	9	C^1	No ($Q = 1$)	No	piecewise

23.7.7 A composite C^1 element on quadrilaterals

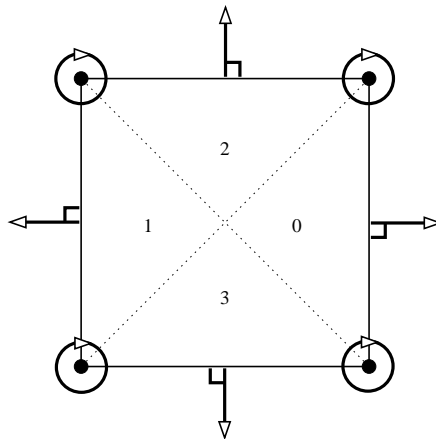


Figure 30: *Composite element on quadrilaterals, piecewise P_3 , 16 d.o.f., C^1*

This element is not τ -equivalent. This is a composite element. Polynomial of degree 3 on each of the four sub-triangles (see figure 30). At least on the reference element it corresponds to the Fraeijns de Veubeke-Sander element (see [1]). It is strongly advised to use a `IM_QUADC1_COMPOSITE` integration method with this finite element.

C^1 composite element on a quadrilateral (FVS) "FEM_QUADC1_COMPOSITE"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	16	C^1	No ($Q = 1$)	No	piecewise

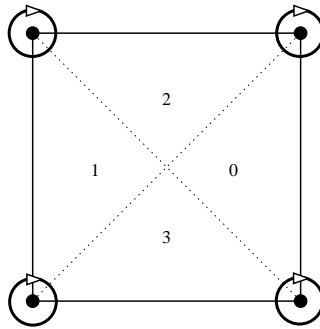


Figure 31: *Reduced composite element on quadrilaterals, piecewise P_3 , 12 d.o.f., C^1*

This element exists also in its reduced form, where the normal derivatives are assumed to be polynomial of degree one on each edge (see figure 31)

Reduced C^1 composite element on a quadrilateral (reduced FVS) "FEM_REDUCED_QUADC1_COMPOSITE"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	2	12	C^1	No ($Q = 1$)	No	piecewise

23.8 Specific elements in dimension 3

23.8.1 Elements with additional bubble functions

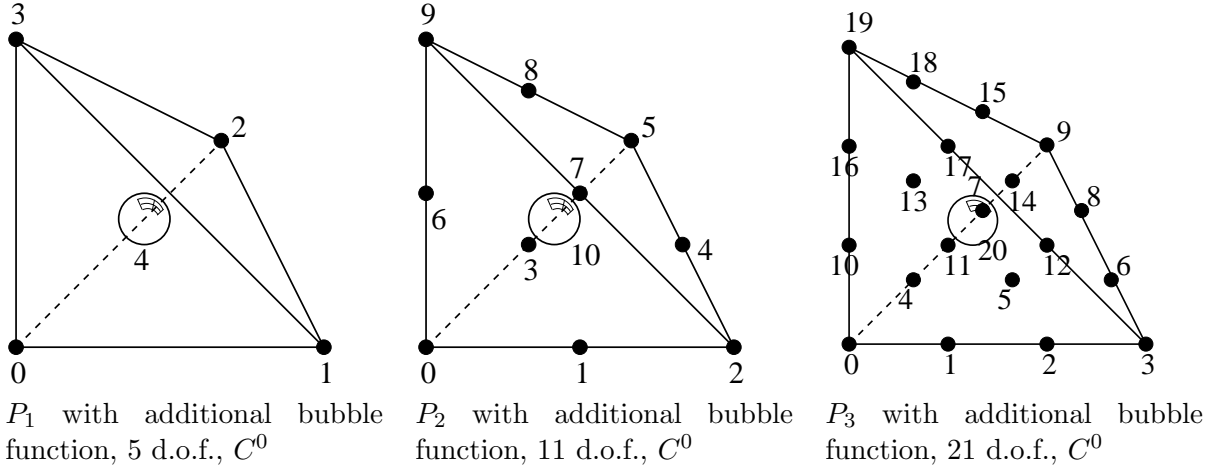


Figure 32: Lagrange element on a tetrahedron with additional internal bubble function.

P_K Lagrange element with an additional internal bubble function "FEM_PK_WITH_CUBIC_BUBBLE(3, K)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
4	3	5, 11 or 21	C^0	No ($Q = 1$)	Yes	Yes

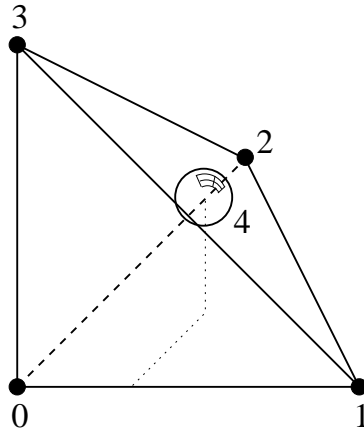


Figure 33: P_1 Lagrange element on a tetrahedron with additional bubble function on face 0, 5 d.o.f., C^0

Lagrange P_1 element with an additional bubble function on face 0 "FEM_P1_BUBBLE_FACE(3)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	3	5	C^0	No ($Q = 1$)	Yes	Yes

23.8.2 Hermite element

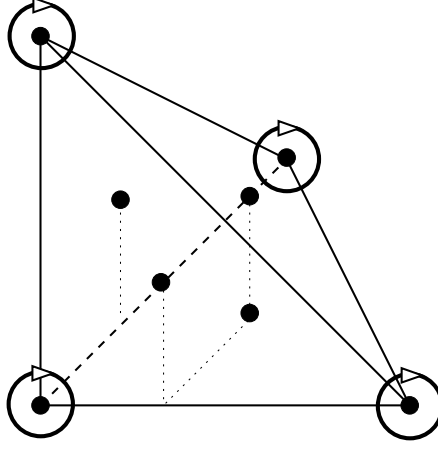


Figure 34: *Hermite element on a tetrahedron, P_3 , 20 d.o.f., C^0*

Base functions on the reference element:

$$\begin{aligned}
 \varphi'_0(x, y) &= 1 - 3x^2 - 13xy - 13xz - 3y^2 - 13yz - 3z^2 + 2x^3 + 13x^2y + 13x^2z \\
 &\quad + 13xy^2 + 33xyz + 13xz^2 + 2y^3 + 13y^2z + 13yz^2 + 2z^3, & (\varphi'_0(0, 0, 0) &= 1), \\
 \varphi'_1(x, y) &= x - 2x^2 - 3xy - 3xz + x^3 + 3x^2y + 3x^2z + 2xy^2 + 4xyz + 2xz^2, & (\partial_x \varphi'_1(0, 0, 0) &= 1), \\
 \varphi'_2(x, y) &= y - 3xy - 2y^2 - 3yz + 2x^2y + 3xy^2 + 4xyz + y^3 + 3y^2z + 2yz^2, & (\partial_y \varphi'_2(0, 0, 0) &= 1), \\
 \varphi'_3(x, y) &= z - 3xz - 3yz - 2z^2 + 2x^2z + 4xyz + 3xz^2 + 2y^2z + 3yz^2 + z^3, & (\partial_z \varphi'_3(0, 0, 0) &= 1), \\
 \varphi'_4(x, y) &= 3x^2 - 7xy - 7xz - 2x^3 + 7x^2y + 7x^2z + 7xy^2 + 7xyz + 7xz^2, & (\varphi'_4(1, 0, 0) &= 1), \\
 \varphi'_5(x, y) &= -x^2 + 2xy + 2xz + x^3 - 2x^2y - 2x^2z - 2xy^2 - 2xyz - 2xz^2, & (\partial_x \varphi'_5(1, 0, 0) &= 1), \\
 \varphi'_6(x, y) &= -xy + 2x^2y + xy^2, & (\partial_y \varphi'_6(1, 0, 0) &= 1), \\
 \varphi'_7(x, y) &= -xz + 2x^2z + xz^2, & (\partial_z \varphi'_7(1, 0, 0) &= 1), \\
 \varphi'_8(x, y) &= -7xy + 3y^2 - 7yz + 7x^2y + 7xy^2 + 7xyz - 2y^3 + 7y^2z + 7yz^2, & (\varphi'_8(0, 1, 0) &= 1), \\
 \varphi'_9(x, y) &= -xy + x^2y + 2xy^2, & (\partial_x \varphi'_9(0, 1, 0) &= 1), \\
 \varphi'_{10}(x, y) &= 2xy - y^2 + 2yz - 2x^2y - 2xy^2 - 2xyz + y^3 - 2y^2z - 2yz^2, & (\partial_y \varphi'_{10}(0, 1, 0) &= 1), \\
 \varphi'_{11}(x, y) &= -yz + 2y^2z + yz^2, & (\partial_z \varphi'_{11}(0, 1, 0) &= 1), \\
 \varphi'_{12}(x, y) &= -7xz - 7yz + 3z^2 + 7x^2z + 7xyz + 7xz^2 + 7y^2z + 7yz^2 - 2z^3, & (\varphi'_{12}(0, 0, 1) &= 1), \\
 \varphi'_{13}(x, y) &= -xz + x^2z + 2xz^2, & (\partial_x \varphi'_{13}(0, 0, 1) &= 1), \\
 \varphi'_{14}(x, y) &= -yz + y^2z + 2yz^2, & (\partial_y \varphi'_{14}(0, 0, 1) &= 1), \\
 \varphi'_{15}(x, y) &= 2xz + 2yz - z^2 - 2x^2z - 2xyz - 2xz^2 - 2y^2z - 2yz^2 + z^3, & (\partial_z \varphi'_{15}(0, 0, 1) &= 1), \\
 \varphi'_{16}(x, y) &= 27xyz, & (\varphi'_{16}(1/3, 1/3, 1/3) &= 1), \\
 \varphi'_{17}(x, y) &= 27yz - 27xyz - 27y^2z - 27yz^2, & (\varphi'_{17}(0, 1/3, 1/3) &= 1), \\
 \varphi'_{18}(x, y) &= 27xz - 27x^2z - 27xyz - 27xz^2, & (\varphi'_{18}(1/3, 0, 1/3) &= 1), \\
 \varphi'_{19}(x, y) &= 27xy - 27x^2y - 27xy^2 - 27xyz, & (\varphi'_{19}(1/3, 1/3, 0) &= 1),
 \end{aligned}$$

This element is not τ -equivalent (The matrix M is not equal to identity). On the real element linear combinations of φ'_8 , φ'_{12} and φ'_{16} are used to match the gradient on the corresponding vertex. Idem on the other vertices.

Hermite element on a tetrahedron						
"FEM_HERMITE(3)"						
Degree	dimension	d.o.f. number	class	vectorial	τ -equivalent	Polynomial
3	3	20	C^0	No ($Q = 1$)	No	Yes

24 Appendix B. Cubature method list

The integration methods are of two kinds. Exact integrations of polynomials and approximated integrations (cubature formulas) of any function. The exact integration can only be used if all the elements are polynomial and if the geometric transformation is linear.

A descriptor on an integration method is given by the function

```
ppi = getfem::int_method_descriptor("name of method");
```

where "name of method" is a string to be chosen among the existing methods.

The program `integration` located in the `tests` directory lists and checks the degree of each integration method.

24.1 Exact Integration methods

The list of available exact integration methods is the following

"IM_NONE()"	Dummy integration method.
"IM_EXACT_SIMPLEX(n) "	Description of the exact integration of polynomials on the simplex of reference of dimension n .
"IM_PRODUCT(a, b) "	Description of the exact integration on the convex which is the direct product of the convex in a and in b .
"IM_EXACT_PARALLELEPIPED(n) "	Description of the exact integration of polynomials on the parallelepiped of reference of dimension n
"IM_EXACT_PRISM(n) "	Description of the exact integration of polynomials on the prism of reference of dimension n

Even though a description of exact integration method exists on parallelepipeds or prisms, most of the time the geometric transformations on such elements are not linear and the exact integration cannot be used.

Beware: In fact a lot of computation cannot be done with exact integration methods. So, it is recommended to use cubature formulas instead.

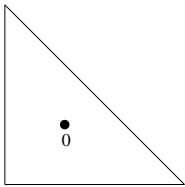
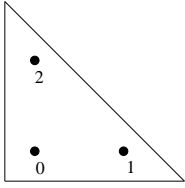
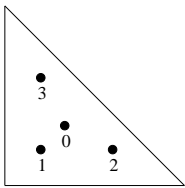
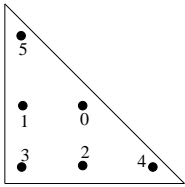
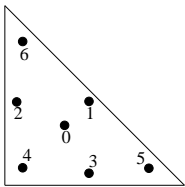
24.2 Newton cotes Integration methods

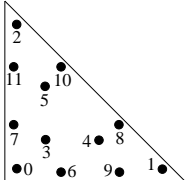
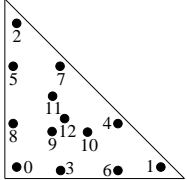
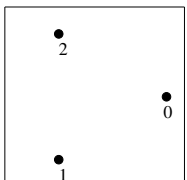
Use "IM_NC(N,K)", "IM_NC_PARALLELEPIPED(N,K)" and "IM_NC_PRISM(N,K)" to have the Newton cotes integration of order K on simplices, parallelepipeds and prisms respectively.

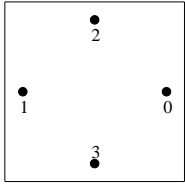
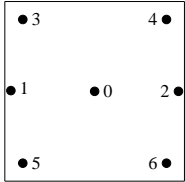
24.3 Gauss Integration methods on dimension 1

Use "IM_GAUSS1D(K)" to have the Gauss-Legendre integration on the segment of order K (with $K/2 + 1$ points), and "IM_GAUSSLOBATTO1D(K)" to have the Gauss-Lobatto-Legendre integration on the segment of order K (with $K/2 + 1$ points). The latter integration method is only available for odd values of K . The Gauss-Lobatto integration method can be used in conjunction with "FEM_PK_GAUSSLOBATTO1D(K/2)" to perform mass-lumping.

24.4 Gauss Integration methods on dimension 2

graphic	coordinates		weights	function to call / order
	x	y		
	1/3	1/3	1/2	"IM_TRIANGLE(1)" 1 point, order 1.
	1/6	1/6	1/6	"IM_TRIANGLE(2)" 3 points, order 2.
	2/3	1/6	1/6	
	1/6	2/3	1/6	
	1/3	1/3	-27/96	"IM_TRIANGLE(3)" 4 points, order 3.
	1/5	1/5	25/96	
	3/5	1/5	25/96	
	1/5	3/5	25/96	
	a	a	c	"IM_TRIANGLE(4)" 6 points, order 4, $a = 0.445948490915965$, $b = 0.091576213509771$, $c = 0.111690794839005$, $d = 0.054975871827661$.
	$1 - 2a$	a	c	
	a	$1 - 2a$	c	
	b	b	d	
	$1 - 2b$	b	d	
	b	$1 - 2b$	d	
	1/3	1/3	9/80	"IM_TRIANGLE(5)" 7 points, order 5, $a = \frac{6 + \sqrt{15}}{21}$, $c = \frac{155 + \sqrt{15}}{2400}$, $d = 31/240 - c$, $b = 4/7 - a$,
	a	a	c	
	$1 - 2a$	a	c	
	a	$1 - 2a$	c	
	b	b	d	
	$1 - 2b$	b	d	
	b	$1 - 2b$	d	

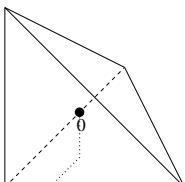
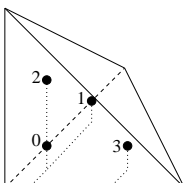
	a $1 - 2a$ a b $1 - 2b$ b $1 - 2b$ c d $1 - c - d$ $1 - c - d$ c d	a a $1 - 2a$ b b $1 - 2b$ d c c d $1 - c - d$ $1 - c - d$	e e e f f f g g g g g g g <p>"IM_TRIANGLE(6)" 12 points, order 6, $a = 0.063089104491502$, $b = 0.249286745170910$, $c = 0.310352451033785$, $d = 0.053145049844816$, $e = 0.025422453185103$, $f = 0.058393137863189$, $g = 0.041425537809187$.</p>
	a b a c d e d c e f g f $1/3$	a a b e c d e d c c f f g $1/3$	h h h i i i i i i j j j k <p>"IM_TRIANGLE(7)" 13 points, order 7, $a = 0.0651301029022$, $b = 0.8697397941956$, $c = 0.3128654960049$, $d = 0.6384441885698$, $e = 0.0486903154253$, $f = 0.2603459660790$, $g = 0.4793080678419$, $h = 0.0266736178044$, $i = 0.0385568804451$, $j = 0.0878076287166$, $k = -0.0747850222338$.</p>
			"IM_TRIANGLE(8)" (see [3]) 16 points, order 8
			"IM_TRIANGLE(9)" (see [3]) 19 points, order 9
			"IM_TRIANGLE(10)" (see [3]) 25 points, order 10
			"IM_TRIANGLE(13)" (see [3]) 37 points, order 13
	$1/2 + \sqrt{1/6}$ $1/2 - \sqrt{1/24}$	$1/2$ $1/2 \pm \sqrt{1/8}$	$1/3$ $1/3$ <p>"IM_QUAD(2)" 3 points, order 2.</p>

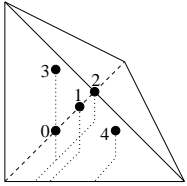
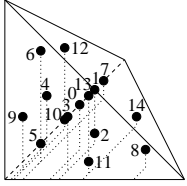
	$1/2 \pm \sqrt{1/6}$	$1/2$	$1/4$	"IM_QUAD(3)" 4 points, order 3.
	$1/2$	$1/2 \pm \sqrt{1/6}$	$1/4$	
	$1/2$	$1/2$	$2/7$	"IM_QUAD(5)" 7 points, order 5.
	$1/2 \pm \sqrt{7/30}$	$1/2$	$5/63$	
	$1/2 \pm \sqrt{1/12}$	$1/2 \pm \sqrt{3/20}$	$5/36$	
				"IM_QUAD(7)" 12 points, order 7
				"IM_QUAD(9)" 20 points, order 9
				"IM_QUAD(17)" 70 points, order 17

There is also the IM_GAUSS_PARALLELEPIPED(n, k) which is a direct product of 1D gauss integrations.

Important note: do not forget that IM_QUAD(k) is exact for polynomials up to degree k , and that a Q_k polynomial has a degree of $2 * k$. For example, IM_QUAD(7) cannot integrate exactly the product of two Q_2 polynomials. On the other hand, IM_GAUSS_PARALLELEPIPED(2,4) can integrate exactly that product...

24.5 Gauss Integration methods on dimension 3

graphic	coordinates			weights	function to call / order
	x	y	z		
	$1/4$	$1/4$	$1/4$	$1/6$	"IM_TETRAHEDRON(1)" 1 point, order 1.
	a	a	a	$1/24$	"IM_TETRAHEDRON(2)" 4 points, order 2 $a = \frac{5 - \sqrt{5}}{20}$, $b = \frac{5 + 3\sqrt{5}}{20}$.
	a	b	a	$1/24$	
	a	a	b	$1/24$	
	b	a	a	$1/24$	

	1/4 1/6 1/6 1/6 1/2	1/4 1/6 1/2 1/6 1/6	1/4 1/6 1/6 1/2 1/6	-2/15 3/40 3/40 3/40 3/40	"IM_TETRAHEDRON(3)" 5 points, order 3
	1/4 a a a c b b b d b e e f f f f	1/4 a a c a b b d b b e f e f f f	1/4 a c a a b d b b f e e f f e	8/405 h h h h i i i i 5/567 5/567 5/567 5/567 5/567	"IM_TETRAHEDRON(5)" 15 points, order 5 $a = \frac{7 + \sqrt{15}}{34}$, $b = \frac{7 - \sqrt{15}}{34}$, $c = \frac{13 + 3\sqrt{15}}{34}$, $d = \frac{13 - 3\sqrt{15}}{34}$, $e = \frac{5 - \sqrt{15}}{20}$, $f = \frac{5 + \sqrt{15}}{20}$, $h = \frac{2665 - 14\sqrt{15}}{226800}$, $i = \frac{2665 + 14\sqrt{15}}{226800}$

Others methods are:

name	convex type	nb of points
IM_TETRAHEDRON(6)	3D simplex	24
IM_TETRAHEDRON(8)	3D simplex	43
IM_SIMPLEX4D(3)	4D simplex	6
IM_HEXAHEDRON(5)	3D parallelepiped	14
IM_HEXAHEDRON(9)	3D parallelepiped	58
IM_HEXAHEDRON(11)	3D parallelepiped	90
IM_CUBE4D(5)	4D parallelepiped	24
IM_CUBE4D(9)	4D parallelepiped	145

24.6 Direct product of integration methods

You can use "IM_PRODUCT(IM1, IM2)" to produce integration methods on quadrilateral or prisms. It gives the direct product of two integration methods. For instance IM_GAUSS_PARALLELEPIPED(2,k) is an alias for IM_PRODUCT(IM_GAUSS1D(2,k),IM_GAUSS1D(2,k)) and can be used instead of the IM_QUAD integrations.

24.7 Composite integration methods

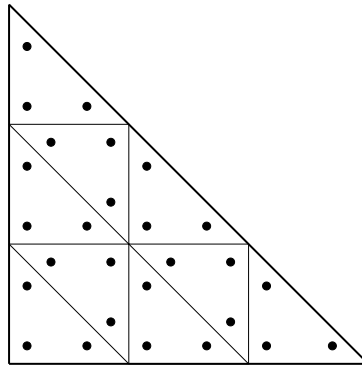


Figure 35: *composite method "IM_STRUCTURED_COMPOSITE(IM_TRIANGLE(2), 3)"*

Use `"IM_STRUCTURED_COMPOSITE(IM1, S)"` to copy `IM1` on an element with `S` subdivisions. The resulting integration method has the same order but with more points. It could be more stable to use a composite method rather than to improve the order of the method. Those methods have to be used also with composite elements. Most of the time for composite element, it is preferable to choose the basic method `IM1` with no points on the boundary (because the gradient could be not defined on the boundary of sub-elements).

For the HCT element, it is advised to use the `IM_HCT_COMPOSITE(im)` composite integration (which split the original triangle into 3 sub-triangles).

References

- [1] P.G.. CIARLET, *The finite element method for elliptic problems*, Studies in Mathematics and its Applications vol. 4, North-Holland, 1978. 91, 92
- [2] R.E. BANK, A.H. SHERMAN, A. WEISER *Refinement algorithms and data structures for regular local mesh refinement*, in Scientific Computing IMACS, Amsterdam, North-Holland, pp 3-17, 1983 21
- [3] R. COOLS *An Encyclopaedia of Cubature Formulas*, J. Complexity, <http://www.cs.kuleuven.ac.be/~ines/research/ecf/ecf.html> 29, 99
- [4] N. MOËS, J. DOLBOW AND T. BELYTSCHKO *A finite element method for crack growth without remeshing*, Int. J. Num. Meth. Engng. 46, 131-150 (1999). 31
- [5] G. DHATT, AND G. TOUZOT *The Finite Element Method Displayed*, J. Wiley & Sons, New York, 1984.
- [6] Y. RENARD, *The GETFEM++ project*, <http://download.gna.org/getfem/doc/getfem-project.pdf> 10, 14, 28