



Description of the Project

Release 4.1.1

Yves Renard, Julien Pommier

February 07, 2012

CONTENTS

1	Introduction	1
2	Introduction to the FEM description in <i>GetFEM++</i>	5
2.1	Convex structures	5
2.2	Convexes of reference	6
2.3	Shape function type	7
2.4	Geometric transformations	7
2.5	Finite element methods description	8
3	Description of the different parts of the library	11
3.1	gmm library	11
3.2	MESH module	12
3.3	FEM module	13
3.4	CUBATURE module	14
3.5	MESHFEM module	14
3.6	MESHIM module	15
3.7	INTEGELEM module	15
3.8	ASSEMBLE module	15
3.9	BRICK module	16
3.10	Events management	16
3.11	Python, Scilab and Matlab interfaces	17
4	Global perspectives of structuration, consolidation and growth	23
4.1	Namespace changes	23
4.2	Basic types used	24
5	Appendix A. Some basic computations between reference and real elements	25
5.1	Volume integral	25
5.2	Surface integral	25
5.3	Derivative computation	26
5.4	Second derivative computation	26
5.5	Example of elementary matrix	26
6	References	29
	Bibliography	31
	Index	33

INTRODUCTION

The aim of this document is to report details of the internal of *GetFEM++* useful for developers that have no place in the user documentation. It is also to outline the main prospects for the future development of *GetFEM++*. A list of modifications to be done and main tasks is updated on the Gna! site <https://gna.org/task/?group=getfem>.

The *GetFEM++* project focuses on the development of a generic finite element library. The goal is to provide a finite element framework which allows to easily build numerical code for the modelisation of system described by partial differential equations (p.d.e.). A special attention is paid to the flexibility of the use of the library in the sense that the switch from a method offered by the library to another is made as easy as possible.

The major point allowing this, compared to traditional finite element codes, is the complete separation between the description of p.d.e. models and finite element methods. Moreover, a separation is made between integration methods (exact or approximated), geometric transformations (linear or not) and finite element methods of arbitrary degrees described on a reference element. *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.

The goal is also to make the addition of new finite element method as simple as possible. For standard method, a description of the finite element shape functions and the type of connection of degrees of freedom on the reference element is sufficient. Extensions are provided for Hermite elements, piecewise polynomial, non-polynomial, vectorial elements and XFem. Examples of predefined available methods are P_k on simplices in arbitrary degrees and dimensions, Q_k on parallelepipeds, P_1 , P_2 with bubble functions, Hermite elements, elements with hierarchic basis (for multigrid methods for instance), discontinuous P_k or Q_k , XFem, Argyris, HCT, Raviart-Thomas.

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++ has no meshing capabilities (apart regular meshes, and a not exploitable attempt), hence, in many situations, 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.

The aim of the *GetFEM++* project is not to provide a ready to use finite element code allowing for instance structural mechanics computations with a graphic interface. It is basically a library allowing the build of C++ finite element codes. However, the matlab and python interfaces allows to easily build application coupling the definition of the problem, the finite element methods selection and the graphical post-processing.

The future of the project is to continue to develop the finite element framework, focusing on the following points.

- Background consolidation of the existing modules (with a reflection on the optimal representation of meshes, degrees of freedom, finite element methods ...).
- Developpement of innovating methods.

- Reflection on the optimal way to represent complex p.d.e. models with the maximum of flexibility and reusability. The brick system is a first step in this direction.

The vocation of *GetFEM++* is to remain a free open source project. The advantage given by the fact to be an open source project is that by proposing a free use, one profits from the experiments of the users who by their tests and the difficulties or bug which they meet make progress the robustness of the algorithms. One also profits from the possible contributions of the users who can find interest to develop new functionalities within the proposed framework. That allows constructive exchanges which clarify the weak points and the strong points of the project.

Figure *Diagram of GetFEM++ library* describes the diagram of the different modules of the *GetFEM++* library. The current state and perspective for each module is described in section *Description of the different parts of the library*.

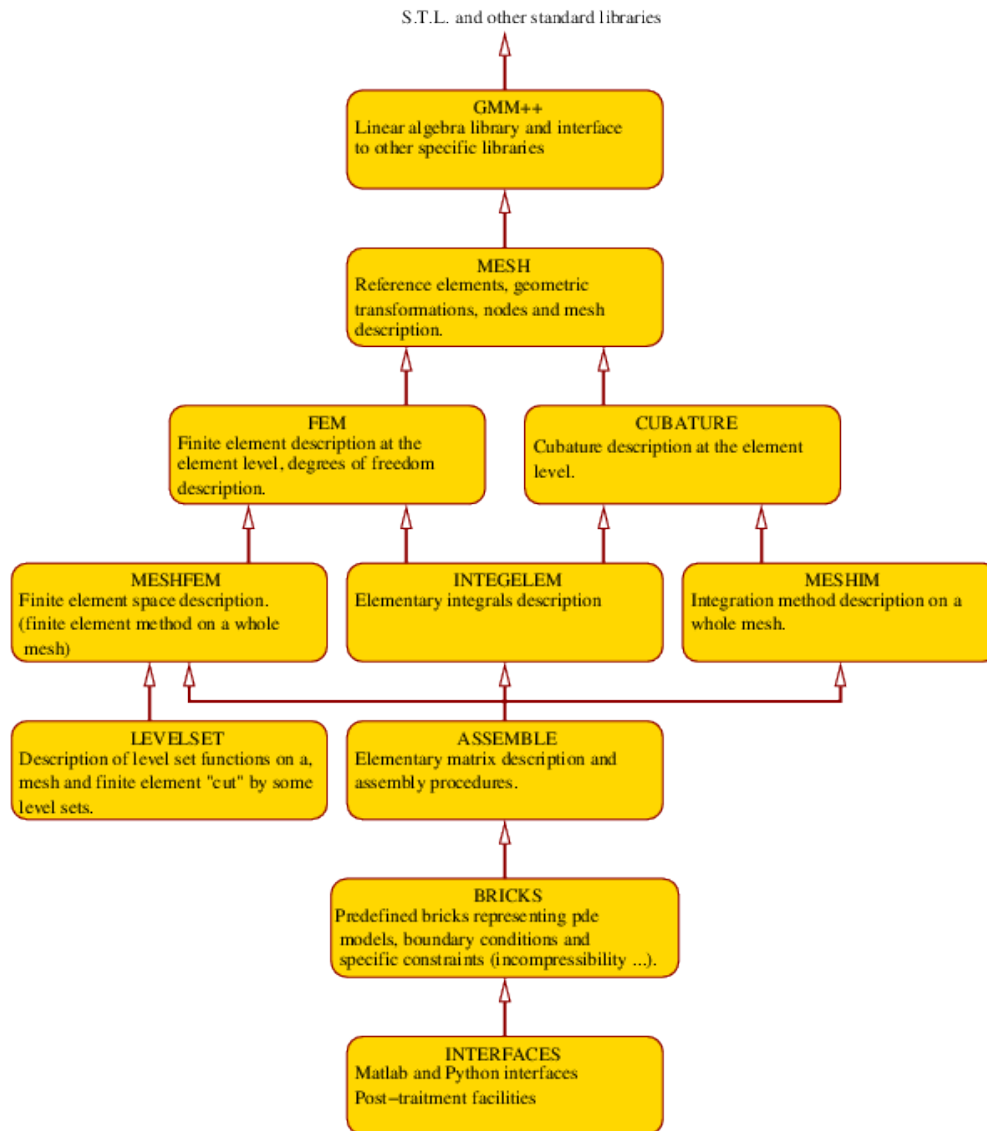


Figure 1.1: Diagram of *GetFEM++* library

Copyright © 2000-2010 Yves Renard, Julien Pommier.

The text of the *GetFEM++* website and the documentations are available for modification and reuse under the terms of the [GNU Free Documentation License](#)

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; version 2.1 of the License. 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., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

INTRODUCTION TO THE FEM DESCRIPTION IN *GETFEM++*

The aim of this section is to briefly introduce the FEM description in *GetFEM++* mainly in order to fix the notation used in the rest of the document (definition of element, reference element, geometric transformation, gradient of the geometric transformation ...).

2.1 Convex structures

Finite element methods are defined on small convex domains called elements. The simplest element on which a finite element method can be defined is a segment (simplex of dimension 1), other possibilities are triangles, tetrahedrons (simplices of dimension 2 and 3), prisms, parallelepiped, etc. In *GetFEM++*, a type of element (for us, a convex) is described by the object `bgeot::convex_structure` defined in the file `bgeot_convex_structure.h`.

It describes only the structure of the convex not the coordinates of the vertices. This structure is not to be manipulated by itself, because it is not necessary that more than one structure of this type describe the same type of convex. What will be manipulated is a pointer on such a descriptor which has to be declared with the type `bgeot::pconvex_structure`

The following functions give a pointer onto the descriptor of the usual type of elements:

`bgeot::simplex_structure` (*dim_type* *d*)
description of a simplex of dimension *d*.

`bgeot::parallelepiped_structure` (*dim_type* *d*)
description of a parallelepiped of dimension *d*.

`bgeot::convex_product_structure` (`bgeot::pconvex_structure` *p1*, `bgeot::pconvex` *p2*)
description of the direct product of *p1* and *p2*.

`bgeot::prism_structure` (*dim_type* *d*)
description of a prism of dimension *d*

For instance if one needs the description of a square, one can call equivalently:

```
p = bgeot::parallelepiped_structure(2);
```

or:

```
p = bgeot::convex_product_structure(bgeot::simplex_structure(1),  
                                   bgeot::simplex_structure(1));
```

The descriptor contains in particular the number of faces ($p \rightarrow nb_faces()$), the dimension of the convex ($p \rightarrow dim()$), for the number of vertices ($p \rightarrow nb_points()$). Other information is the number of vertices of each face, the description of a face and the eventual reference to a more basic description (used for the description of geometric transformations).

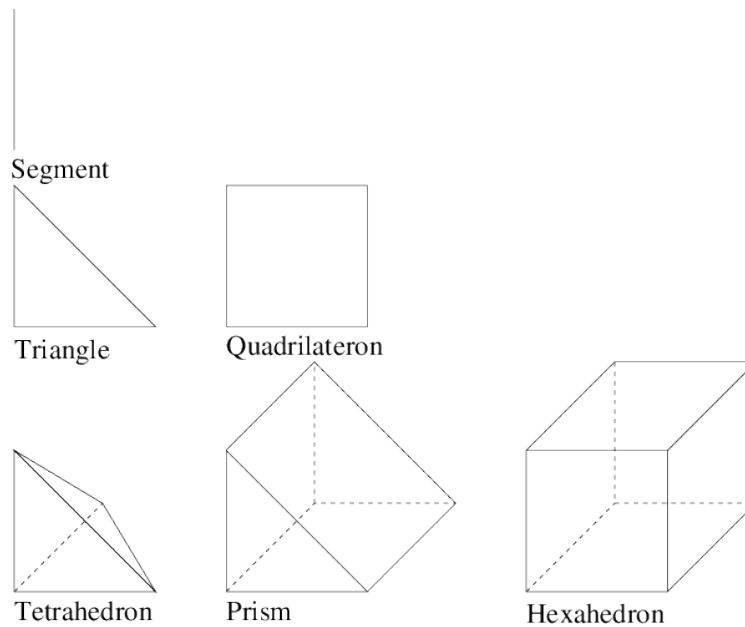


Figure 2.1: usual elements

2.2 Convexes of reference

A convex of reference is a particular real element, i.e. a structure of convex with a list of vertices. It describes the particular element from which a finite element method is defined. In the file `bgeot_convex_ref.h` the object `bgeot::convex_of_reference` makes this description. The library keeps only one description for each type of convex. So what will be manipulated is a pointer of type `bgeot::pconvex_ref` on the descriptor.

The following functions build the descriptions:

- `bgeot::simplex_of_reference`** (*dim_type d*)
description of the simplex of reference of dimension *d*.
- `bgeot::simplex_of_reference`** (*dim_type d, short_type k*)
description of the simplex of reference of dimension *d* with degree *k* Lagrange grid.
- `bgeot::convex_ref_product`** (*pconvex_ref a, pconvex_ref b*)
description of the direct product of two convexes of reference.
- `bgeot::parallelepiped_of_reference`** (*dim_type d*)
description of the parallelepiped of reference of dimension *d*.

The vertices correspond to the classical vertices for such reference element. For instance the vertices for the triangle are $(0, 0)$, $(1, 0)$ and $(0, 1)$. It corresponds to the configuration shown in Figure *usual elements*

If *p* is of type `bgeot::pconvex_ref` then `p->structure()` is the corresponding convex structure. Thus for instance `p->structure()->nb_points()` gives the number of vertices. The function `p->points()` give the array of vertices and `p->points()[0]` is the first vertex. The function `p->is_in(const`

`base_node &pt)` return a real which is negative or null if the point `pt` is in the element. The function `p->is_in_face(short_type f, const base_node &pt)` return a real which is null if the point `pt` is in the face `f` of the element. Other functions can be found in `bgeot_convex_ref.h` and `bgeot_convex.h`.

2.3 Shape function type

Most of the time the shape functions of finite element methods are polynomials, at least on the convex of reference. But, the possibility is given to have other types of elements. It is possible to define other kind of base functions such as piecewise polynomials, interpolant wavelets, etc.

To be used by the finite element description, a shape function type must be able to be evaluated on a point (`a = F.eval(pt)`, where `pt` is a `base_node`) and must have a method to compute the derivative with respect to the `i`th variable (`F.derivative(i)`).

For the moment, only polynomials and piecewise polynomials are defined in the files `bgeot_poly.h` and `bgeot_poly_composite.h`.

2.4 Geometric transformations

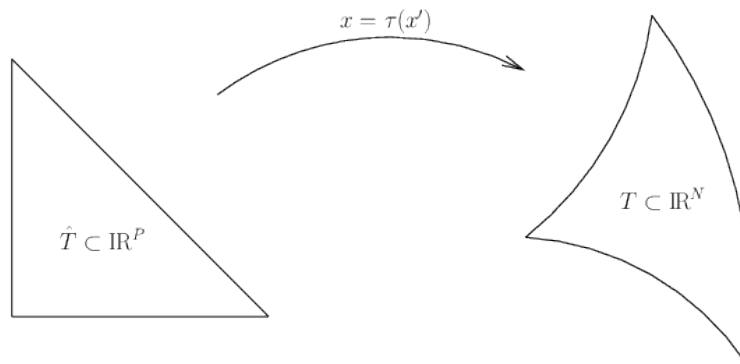


Figure 2.2: geometric transformation

A geometric transformation is a polynomial application:

$$\tau : \hat{T} \subset \mathbb{R}^P \longrightarrow T \subset \mathbb{R}^N,$$

which maps the reference element \hat{T} to the real element T . The geometric nodes are denoted:

$$g^i, i = 0, \dots, n_g - 1.$$

The geometric transformation is described thanks to a n_g components polynomial vector (In fact, as an extension, non polynomial geometric transformation can also be supported by *GetFEM++*, but this is very rarely used)

$$\mathcal{N}(\hat{x}),$$

such that

$$\tau(\hat{x}) = \sum_{i=0}^{n_g-1} \mathcal{N}_i(\hat{x}) g^i.$$

Denoting

$$G = (g^0; g^1; \dots; g^{n_g-1}),$$

the $N \times n_g$ matrix containing of all the geometric nodes, one has

$$\tau(\hat{x}) = G \cdot \mathcal{N}(\hat{x}).$$

The derivative of τ is then

$$K(\hat{x}) := \nabla \tau(\hat{x}) = G \cdot \nabla \mathcal{N}(\hat{x}),$$

where $K(\hat{x}) = \nabla \tau(\hat{x})$ is a $N \times P$ matrix and $\nabla \mathcal{N}(\hat{x})$ a $n_g \times P$ matrix. The (transposed) pseudo-inverse of $\nabla \tau(\hat{x})$ is a $N \times P$ matrix denoted $B(\hat{x})$:

$$B(\hat{x}) := K(\hat{x})(K(\hat{x})^T K(\hat{x}))^{-1},$$

Of course, when $P = N$, one has $B(\hat{x}) = K(\hat{x})^{-T}$.

Pointers on a descriptor of a geometric transformation can be obtained by the following function defined in the file `bgeot_geometric_trans.h`:

```
bgeot::pgeometric_trans pgt = bgeot::geometric_trans_descriptor("name of trans");
```

where "name of trans" can be chosen among the following list.

- "GT_PK (n, k) "
Description of the simplex transformation of dimension n and degree k (Most of the time, the degree 1 is used).
- "GT_QK (n, k) "
Description of the parallelepiped transformation of dimension n and degree k.
- "GT_PRISM (n, k) "
Description of the prism transformation of dimension n and degree k.
- "GT_PRODUCT (a, b) "
Description of the direct product of the two transformations a and b.
- "GT_LINEAR_PRODUCT (a, b) "
Description of the direct product of the two transformations a and b keeping a linear transformation (this is a restriction of the previous function). This allows, for instance, to use exact integrations on regular meshes with parallelograms.

2.5 Finite element methods description

A finite element method is defined on a reference element $\hat{T} \subset \mathbb{R}^P$ by a set of n_d nodes a^i and corresponding base functions

$$(\hat{\varphi})^i : \hat{T} \subset \mathbb{R}^P \longrightarrow \mathbb{R}^Q$$

Denoting

$$\psi^i(x) = (\hat{\varphi})^i(\hat{x}) = (\hat{\varphi})^i(\tau^{-1}(x)),$$

a supplementary linear transformation is allowed for the real base function

$$\varphi^i(x) = \sum_{j=0}^{n_d-1} M_{ij} \psi^j(x),$$

where M is a $n_d \times n_d$ matrix possibly depending on the geometric transformation (i.e. on the real element). For basic elements as Lagrange elements this matrix is the identity matrix (it is simply ignored). In this case, we will say that the element is τ -equivalent.

This approach allows to define hermite elements (Argyris for instance) in a generic way, even with non linear transformations (i.e. mainly for curved boundaries). We denote $[\hat{\varphi}(\hat{x})]$ the $n_d \times Q$ matrix whose i th line is $(\hat{\varphi})^i(\hat{x})$. With this notation, for a function is defined by

$$f(x) = \sum_{i=0}^{n_d-1} \alpha_i \varphi^i(x),$$

one has

$$f(\tau(\hat{x})) = \alpha^T M [\hat{\varphi}(\hat{x})],$$

where α is the vector whose i th component is α_i .

A certain number of description of classical finite element method are defined in the file `getfem_fem.h`. See *Appendix A. Finite element method list* (in *Short User Documentation*) for an exhaustive list of available finite element methods.

A pointer to the finite element descriptor of a method is obtained using the function:

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

We refer to the file `getfem_fem.cc` for how to define a new finite element method.

DESCRIPTION OF THE DIFFERENT PARTS OF THE LIBRARY

3.1 gmm library

3.1.1 Description

Gmm++ is a linear algebra library which was originally designed to make an interface between the need in linear algebra procedures of *GetFEM++* and existing free linear algebra libraries (MTL, Superlu, Blas, Lapack originally). It rapidly evolves to an independent self-consistent library with its own vector and matrix types. It is now used as a base linear algebra library by several other projects (projet [KDE](#), for instance).

However, it preserves the characteristic to be a potential interface for more specific packages. Any vector or matrix type having the minimum of compatibility can be used by generic algorithms of *Gmm++* writing a `linalg_traits` structure.

A *Gmm++* standalone version is distributed since release 1.5 of *GetFEM++*. It is however developed inside the *GetFEM++* project even though since release 3.0 it is completely independent of any *GetFEM++* file.

In addition to the linear algebra procedures, it furnishes also the following utilities to *GetFEM++*.

- Fix some eventual compatibility problems in `gmm_std.h`.
- Error, warning and trace management in `gmm_except.h`.
- Some extended math definitions in `gmm_def.h`.

3.1.2 State

For the moment, *Gmm++* cover the needs of *GetFEM++* concerning the basic linear algebra procedures.

3.1.3 Perspectives

There is potentially several points to be improved in *Gmm++* (partial introduction of expression template for some base types of matrix and vectors, reflection on the way to represent in a more coherent manner sparse sub-vectors and sub-matrices, introduction of C++ concepts, etc.). However, since *Gmm++* globally cover the needs of *GetFEM++* and since there exists some other project like [Glas](#) to build a reference C++ library for linear algebra, a global reflection seems not necessary for the moment. This part is considered to be stabilized.

The current vocation of *Gmm++* is to continue to collect generic algorithms and interfaces to some other packages in order to cover new needs of the whole project. The library is now frequently used as a separate package and has

also the vocation to collect the contribution of any person who propose some improvements, new algorithms or new interfaces.

3.2 MESH module

3.2.1 Description

This part of the library has the role to store and manage the meshes, i.e. a collection of elements (real elements) connected to each other by some of their faces. For that, it develops concepts of elements, elements of reference, structure of meshes, collection of nodes, geometric transformations, subpart of the boundary or subzone of the mesh.

There is no really effective meshing capabilities available for the moment in *GetFEM++*. The meshes of complex objects must be imported from existing meshers such as *Gmsh* or *GiD*. Some importing functions of meshes have been written and can be easily extended for other formats.

The object which represents a mesh declared in the file `getfem_mesh.h` and which is used as a basis for handling of the meshes in *GetFEM++* manages also the possibility for the structures depending on a mesh (see *MESHFEM* and *MESHIM* modules) to react to the evolution of the mesh (addition or removal of elements, etc.).

3.2.2 State

The main C++ header files are

- `bgeot_convex_structure.h`
Describes the structure of an element disregarding the coordinates of its vertices.
- `bgeot_mesh_structure.h`
Describes the structure of a mesh disregarding the coordinates of the nodes.
- `bgeot_node_tab.h`
A node container allowing the fast search of a node.
- `bgeot_convex.h`
Describes an element with its vertices.
- `bgeot_convex_ref.h`
Describe reference elements.
- `bgeot_mesh.h`
Describes a mesh with the collection of node (but without the description of geometric transformations).
- `bgeot_geometric_trans.h`
Describes geometric transformations.
- `bgeot_geotrans_inv.h`
A tool to invert geometric transformations.
- `getfem_mesh.h`
Fully describes a mesh (with the geometric transformations, subparts of the mesh, support for parallelization). Includes the Bank algorithm to refine a mesh.

- `getfem_mesher.h`

An attempt to develop a mesher. To be use with care.

A prototype of mesher is in the files `getfem_mesher.h` and `getfem_mesher.cc` which makes it possible to mesh geometries defined by some level sets. However, the continuation of the development of this mesher is not planned for the moment because the project *GetFEM++* has vocation to focus on the finite element methods themselves.

3.2.3 Perspectives

For the moment, the module is split into two parts which lie into two different namespaces. Of course, It would be more coherent to gather the module in only one namespace (`getfem`).

Note: The file `bgeot_mesh.h` could be renamed `getfem_basic_mesh.h`.

A possible work to do on this part would be to examine the manner of storing the meshes and possibly to make a bibliographical study on the manner of storing a mesh (for instance see [remacle2002]). It would be necessary to supplement documentation and to examine also the management of the events and the way in which the structures which depend on the mesh react to these events.

3.3 FEM module

3.3.1 Description

The FEM module is the part of *GetFEM++* which describes the finite elements at the element level and the degrees of freedom. Finite element methods can be of different types. They could be scalar or vectorial, polynomial, piecewise polynomial or non-polynomial, equivalent via the geometric transformation or not. Moreover, the description of the degrees of freedom have to be such that it is possible to gather the compatible degrees of freedom between two neighbor elements in a generic way (for instance connecting a Lagrange 2D element to another Lagrange 1D element).

3.3.2 State

The main files of the module are

- `getfem_fem.h`
Abstract definition of a finite element and a degree of freedom. Interface for the exported functions of `getfem_fem.cc` and `getfem_fem_composite.cc`.
- `getfem_fem.cc`
Definition of the polynomial finite elements and interface to get the descriptor on these elements (function `pfem_fem_descriptor(std::string name)`).
- `getfem_fem_composite.cc`
Definition of the piecewise polynomial finite elements.

The two files `getfem_fem.cc` and `getfem_fem_composite.cc` mainly contains all the finite element description for basic elements. A exhaustive list of the defined finite elements is given in *Appendix A. Finite element method list* (in *Short User Documentation*).

Some other files define some specific finite element such as `getfem_fem_level_set.h` which is a complex construction which allows to “cut” a existing element by one or several level sets (see the `LEVELSET` module).

The manner to describe the degrees of freedom globally satisfies the needing (connecting dof from an element to another in a generic way) but is a little bit obscure and too much complicated.

Conversely, the way to represent non-equivalent elements with the supplementary matrix M has proven its efficiency on several elements (Hermite elements, Argyris, etc.).

3.3.3 Perspectives

The principal dissatisfaction of this module is that description of the degrees of freedom is not completely satisfactory. It is the principal reason why one documentation on how to build an element from A to Z was not made for the moment because description of the degrees of freedom was conceived to be temporary. An effort of design is thus to be provided to completely stabilize this module mainly thus with regard to the description of degrees of freedom but also perhaps the description of finite elements which could be partially externalized in a similar way to the cubature methods, at least for the simplest finite elements (equivalent and polynomial finite elements).

3.4 CUBATURE module

3.4.1 Description

The CUBATURE module gives access to the numerical integration methods on reference elements. In fact it does not only contain some cubature formulas because it also give access to some exact integration methods. However, the exact integration methods are only usable for polynomial element and affine geometric transformations. This explain why exact integration methods are not widely used. The description of cubature formulas is done either directly in the file `getfem_integration.h` or via a description file in the directory `cubature` of *GetFEM++*. The addition of new cubature formulas is then very simple, it suffices to reference the element on which it is defined and the list of Gauss points in a file and add it to this directory. Additionally, In order to integrate terms defined on a boundary of a domain, the description should also contains the reference to a method of same order on each face of the element.

3.4.2 State

This module meets the present needs for the project and is considered as stabilized. The list of available cubature formulas is given in *Appendix B. Cubature method list* (in *Short User Documentation*).

3.4.3 Perspectives

No change needed for the moment. An effort could be done on the documentation to describe completely how to add a new cubature formula (format off description files).

3.5 MESHFEM module

to be done

3.5.1 Description

3.5.2 State

3.5.3 Perspectives

Parallelisation of dof numbering to be done. An optimal (an simple) algorithm exists.

3.5.4 LEVELSET module

to be done

3.5.5 Description

3.5.6 State

3.5.7 Perspectives

3.6 MESHIM module

to be done

3.6.1 Description

3.6.2 State

3.6.3 Perspectives

3.7 INTEGELEM module

to be done

3.7.1 Description

3.7.2 State

3.7.3 Perspectives

3.8 ASSEMBLE module

to be done

3.8.1 Description

3.8.2 State

3.8.3 Perspectives

3.9 BRICK module

to be done

3.9.1 Description

3.9.2 State

3.9.3 Perspectives

3.10 Events management

3.10.1 Description

The *mesh*, *mesh_fem*, *mesh_im* and *model* description are linked together in the sense that there is some dependencies between them. For instance, when an element is suppressed to a mesh, the *mesh_fem* object has to react.

3.10.2 State

The main tool to deal with simple dependence of object is in `getfem_context.h`. An object `context_dependencies` is defined there. In order to deal with the dependencies of an object, the object `context_dependencies` needs to be a parent class of this object. It adds the following methods to the object:

add_dependency (*ct*)

Add an object (which has to have `context_dependencies` as a parent class) to the list of objects from which the current object depend.

touch ()

Indicates to the dependent objects that something has change in the object.

context_check ()

Check if the object has to be updated. if it is the case it makes first a check to the dependency list and call the update function of the object. (the update function of the dependencies are called before the update function of the current object).

context_valid ()

Says if the object has still a valid context, i.e. if the object in the dependency list still exist.

Moreover, the object has to define a method:

```
``void update_from_context(void) const``
```

which is called after a `context_check()` if the context has changed.

An additional system is present in the object *mesh*. Each individual element has a version number in order for the objects *mesh_fem* and *mesh_im* to detect which element has changed between two calls.

3.10.3 Perspectives

Some object do not manage satisfactorily events. This is the case for instance of *mesh_level_set*, *mesh_fem_level_set*, *partial_mesh_fem*, etc.

This is clear that the event management still have to be tested and improved to have a fully reactive system.

3.11 Python, Scilab and Matlab interfaces

A simplified interface of *GetFEM++* is provided, so that it is possible to use *getfem* in other languages.

3.11.1 Description

All sources are located in the `interface/src` directory. The interface is composed of one large library `getfemint` (which stands for *getfem* interaction), which is acts as a layer above the *GetFEM++* library, and is used by both the python and matlab interfaces.

This interface is not something that is generated automatically from `c++` sources (as that could be the case with tools such as `swig`). It is something that has been designed as a simplified and consistent interface to *getfem*. Adding a new language should be quite easy (assuming the language provides some structures for dense arrays manipulations).

3.11.2 State

Here is a list of the various files, with a short description:

- `getfem_interface.cc`.

This is the bridge between the script language and the *getfem* interface. The function `getfem_interface_main` is exported as an `extern "C"` function, so this is a sort of `c++` barrier between the script language and the *getfem* interface (exporting only a C interface avoids many compilation problems).

- `matlab/gfm_mex.c`.

The matlab interface. The only thing it knows about *getfem* is in `getfem_interface.h`.

- `python/getfem_python.c`.

The python interface. The only thing it knows about *getfem* is in `getfem_interface.h`.

- `gfi_array.h`, `gfi_array.c`.

Both `gfm_mex.c` and `getfem_python.c` need a simple convention on how to send and receive arrays, and object handles, from `getfem_interface_main()`. This file provide such fonctionnality.

- `getfemint_object.h`.

Not all *getfem* objects are exported, only a selected subset, mostly *mesh*, *mesh_im*, *mesh_fem*, *slice*, *brick*, etc. They are all wrapped in a common interface, which is `getfemint::getfem_object`.

- `getfemint_mesh.h`, `getfemint_mesh_fem.h`, etc.

All the wrapped *GetFEM++* objects. Some of them are quite complicated (`getfemint_gsparse` which export some kind of mutable sparse matrix that can switch between different storage types, and real of complex elements).

- `gf_workspace.cc`, `gf_delete.cc`.

Memory management for `getfem` objects. There is a layer in `getfemint::getfem_object` which handles the dependency between for example a `getfemint_mesh` and a `getfemint_mesh_fem`. It makes sure that no object will be destroyed while there is still another `getfem_object` using it. The goal is to make sure that under no circumstances the user is able to crash `getfem` (and the host program, `matlab` or `python`) by passing incorrect argument to the `getfem` interface.

It also provides a kind of workspace stack, which was designed to simplify handling and cleaning of many `getfem` objects in `matlab` (since `matlab` does not have “object destructors”).

- `getfemint.h`, `getfemint.cc`.

Define the `mexarg_in`, `mexarg_out` classes, which are used to parse the list of input and output arguments to the `getfem` interface functions. The name is not adequate anymore since any reference to “`mex`” has been moved into `gfm_mex.c`.

- `gf_mesh.cc`, `gf_mesh_get.cc`, `gf_mesh_set.cc`, `gf_fem.cc`, etc.

All the functions exported by the `getfem` interfaces, sorted by object type (`gf_mesh*`, `gf_mesh_fem*`, `gf_fem*`), and then organized as one for the object construction (`gf_mesh`), one for the object modification (`gf_mesh_set`), and one for the object inquiry (`gf_mesh_get`). Each of these files contain one main function, that receives a `mexargs_in` and `mexargs_out` stack of arguments. It parses then, and usually interprets the first argument as the name of a subfunction (`gf_mesh_get('nbpts')` in `matlab`, or `Mesh.nbpts()` in `python`).

- `matlab/gfm_rpx_mexint.c`.

An alternative to `gfm_mex.c` which is used when the `--enable-matlab-rpc` is passed to the `./configure` script. The main use for that is debugging the interface, since in that case, the `matlab` interface communicates via sockets with a “`getfem_server`” program, so it is possible to debug that server program, and identify memory leaks or anything else without having to mess with `matlab` (it is pain to debug).

- `python/getfem.py`.

The `python` interface is available as a “`getfem.py`” file which is produced during compilation by the `python` script “`binextract_doc.py`”.

3.11.3 Objects, methods and functions of the interface

The main concepts manipulated by the interface are a limited number of objects (`Fem`, `Mesh`, `MeshFem`, `Model` ...), the associated methods and some functions defined on these objects.

A special effort has been done to facilitate the addition of new objects, methods and functions to the interface without doing it separately for each partsupported script language (`Python`, `Scilab`, `Matlab`).

All the information needed to build the interface for the different objects, methods and functions is contained in the files `interface/src/gf*.cc`. A `python` script (`bin/extract_doc`) produces all the necessary files from the information it takes there. In particular, it produces the `python` file `getfem.py`, the `matlab` `m`-files for the different functions and objects (including subdirectories) and it also produces the automatic documentations.

To make all the thing works automatically, a certain number of rules have to be respected:

- An object have to be defined by three files on the interface
 - `gf_objectname.cc` : contains the constructors of the object
 - `gf_objectname_get.cc` : contains the methods which only get some information about the object (if any).
 - `gf_objectname_set.cc` : contains the methods which transform the object (if any).

- A list of function is defined by only one file `gf_commandname.cc` it contains a list of sub-commands.
- For each file, the main commentary on the list of functions or methods is delimited by the tags `'/@GFDOC'` and `'@/'`. For a file corresponding to the constructors of an object, the commentary should correspond to the description of the object.
- Each non trivial file `gf_*.cc` contains a macro allowing to define the methods of the object or the sub-commands. In particular, this system allows to have a efficient search of the called method/function. This macro allows to declare a new method/function with the following syntax:

```

/*@GET val = ('method-name', params, ...)
   Documentation of the method/function.
@*/
sub_command
("method-name", 0, 0, 0, 1,
  ...
  body of the method/function
  ...
);

```

The first three line are a c++ commentary which describes the call of the method/function with a special syntax and also gives a description of the method/function which will be included in the documentations. The first line of this commentary is important since it will be analyzed to produce the right interface for Python and Matlab.

The syntax for the description of the call of a method/function is the following: After `/*@` a special keyword should be present. It is either `INIT`, `GET`, `SET`, `RDATTR` or `FUNC`. The keyword `INIT` means that this is the description of a constructor of an object. `RDATTR` is for a short method allowing to get an attribut of an object. `GET` is for a method of an object which does not modify it. `SET` is for a method which modifies an object and `FUNC` is for the sub-command of a function list.

If the method/function returns a value, then a name for the return value is given (which is arbitrary) followed by `=`.

The the parameters of the method/function are described. For a method, the object itself is not mentionned. The first parameter should be the method or sub-command name between single quotes (a speical case is when this name begin with a dot this means that it correspond to a method/function where the command name is not required).

The other parameters, if any, should be declared with a type. predefined types are the following:

- `@CELL` : a cell array,
- `@imat` : matrix of integers,
- `@ivec` : vector of integers,
- `@cvec` : vector of complex values,
- `@dcvec` : vector of complex values,
- `@dvec` : vector of real values,
- `@vec` : vector of real or complex values,
- `@dmat` : matrix of real values,
- `@mat` : matrix of real or complex values,
- `@str` : a string,
- `@int` : an integer,
- `@bool` : a boolean,

- @real : a real value,
- @scalar : a real or complex value,
- @list : a list.

Moreover, @tobj refers to an object defined by the interface. For instance, ou can refer to @tmesh, @tmesh_fem, @tfem, etc. There is some authorized abbreviations:

- @tmf for ``@tmesh_fem
- @tbrick for @tmdbrick
- @tstate for @tmdstate
- @tgt for @tgeotrans
- @tgf for @tglobal_function
- @tmls for @tmesh_levelset
- @tls for @tlevelset
- @tsl for @tslice
- @tsp for @tspmat
- @tpre for @tprecond

Three dots at the end of the parameter list (...) means that the additional parameters are possible. Optional parameters can be described with brackets. For instance /*@SET v = ('name'[, @int i]). But be careful how it is interpreted by the extract_doc script to build the python interface.

The second to fifth parameters of the macro correspond respectively to the minimum number of input arguments, the maximum one, the minimum number of output arguments and the maximum number of output arguments. It is dynamically verified.

Additional parameters for the function lists

For unknown reasons, the body of the function cannot contain multiple declarations such as int a, b; (c++ believes that it is an additional parameter of the macro).

- The parts of documentation included in the c++ commentaries should be in **reStructuredText** format. In particular, math formulas can be included with :math:'f(x) = 3x^2+2x+4' or with:

```
.. math::  
  
f(x) = 3x^2+2x+4
```

It is possible to refer to another method or function of the interface with the syntax
INIT::OBJNAME('method-name', ...), GET::OBJNAME('method-name', ...),
SET::OBJNAME('method-name', ...), FUNC::FUNCNAME('subcommand-name', ...).
This will be replaced with the right syntax depending on the langage (Matlab, Scilab or Python).

- Still in the documentations, parts for a specific langage can be added by @MATLAB{specific part ...}, @SCILAB{specific part ...} and @PYTHON{specific part ...}. If a method/sub-command is specific to an interface, it cen be added, for instance for Matlab, replacing *GET* by *MATLABGET*, *FUNC* by *MATLABFUNC*, etc. If a specific code is needed for this additional function, it can be added with the tags /*@MATLABEXT, /*@SCILABEXT, /*@PYTHONEXT. See for instance the file gf_mesh_fem_get.cc.
- For Python and the Matlab object, if a *SET* method has the same name as a *GET* method, the *SET* method is prefixed by *set_*.

3.11.4 Adding a new function or object method to the getfem interface

If one want to add a new function `gf_mesh_get(m, "foobar", .)`, then the main file to modify is `gf_mesh_get.cc`. Remember to check every argument passed to the function in order to make sure that the user cannot crash scilab, matlab or python when using that function. Use the macro defined in `gf_mesh_get.cc` to add your function.

Do not forget to add documentation for that function: in `gf_mesh_get.cc`, this is the documentation that appears in the matlab/scilab/python help files (that is when on type “`help gf_mesh_get`” at the matlab prompt), and in the `getfem_python` autogenerated documentation.

IMPORTANT. Note that the array indices start at 0 in Python and 1 in Matlab and Scilab. A specific function:

```
config::base_index()
```

whose value is 0 in python and 1 in Matlab and Scilab have to be used to exchange indices and array of indices. Take care not to make the correction twice. Some Array of indices are automatically shifted.

3.11.5 Adding a new object to the getfem interface

In order to add a new object to the interface, you have to build the new corresponding sources `gf_obj.cc`, `gf_obj_get.cc` and `gf_obj_set.cc`. Of course you can take the existing ones as a model.

A structure name *getfemint_object_name* has to be defined (see `getfemint_mesh.h` for instance). Moreover, for the management of the object, you have to declare the class in `getfemint.cc` and `getfemint.h` and add the methods *is_object()*, *to_const_object()*, *to_object()* and *to_getfemint_global_function()*. You have to set its `class_id` in `gfi_array.h` (with respect to the alphabetic order of its name).

You have also to add the call of the interface function in `getfem_interface.cc` and modify the file `binextract_doc` and run the configure file.

The methods `get('char')` and `get('display')` should be defined for each object. The first one should give a string allowing te object to be saved in a file and the second one is to give some informations about the object. Additionnaly, a constructor from a string is necessary to load the object from a file.

For the Scilab interface the file `sci_gatewaycbuilder_gateway_c.sce.in` has to be modified and the file in the directory `macrosoverload`.

3.11.6 Perspectives

The interface grows in conjunction with *GetFEM++*. The objective is to interface the maximum of the *GetFEM++* functionalities.

GLOBAL PERSPECTIVES OF STRUCTURATION, CONSOLIDATION AND GROWTH

intro to the main modifications to be done ...

Modifications to be done are of three kind:

- Background consolidation of the existing modules (with a reflection on the optimal representation of meshes, degrees of freedom, finite element methods, etc.).
- Developpement of innovating methods.
- Reflection on the optimal way to represent complex p.d.e. models with the maximum of flexibility and reusability. The brick system is a first step in this direction. It should be replaced soon by a more elaborated system.

4.1 Namespace changes

After the elimination of the small namespaces `linkmsg` and `ftool` in release 3.0, it remains now four namespaces in the *GetFEM++* project.

- `gmm` (Generic Matrix Methods) : for the linear algebra procedures.
- `dal` (Dynamic Array Library) : some basic algorithms including the definition of some containers (`dal::dynamic_array`, `dal::dynamic_tas`, `dal::tree_sorted_array`, `dal::bit_vector`).
- `bgeot` (Basic GEometric Tool) : some basic algorithms including the definition of geometric objects (convex structure, convex, convex of reference, basic mesh).
- `getfem` : the main namespace of *GetFEM++*.

It is clear that the separation into these remaining four namespaces is mainly historical. The separate `gmm` namespace for *Gmm++* is clearly justified. The contour of namespaces `dal` and `bgeot` is more vague. Historically, those two namespaces had their own justifications.

In the very begining of *GetFEM++* (the first files was written in 1995) the S.T.L. was not available and the containers defined in the `dal` namespace was used everywhere. Now, in *GetFEM++*, the S.T.L. containers are mainly used. The remaining uses of `dal` containers are eather historical or due to the specificities of these containers. It is however clear that this is not the aim of the *GetFEM++* project to developp new container concept. So, the use of the `dal` containers has to be as much as possible reduced.

Now, concerning `bgeot`, it was containing some other geometrical object at the beginning and was originally designed to be a self-consistent library of geometric concepts. It slowly derived to be like it is now, a collection of algorithms and object definition more or less related to geometry (`rtree`, `kdtree`, `ftool`, `polynomials` ...).

The conclusion of this is that `dal` and `bgeot` namespaces can be advantageously merged to the `get fem` namespace, reducing to the minimum the use of the `dal` containers. This should be done preserving the backward compatibility. An intermediary study would be to see if the `dal` cannot be directly derived from S.T.L. containers preserving the used specificities.

4.2 Basic types used

Basic type of integer, real ... used. to be done.

APPENDIX A. SOME BASIC COMPUTATIONS BETWEEN REFERENCE AND REAL ELEMENTS

5.1 Volume integral

One has

$$\int_T f(x) dx = \int_{\hat{T}} \hat{f}(\hat{x}) |\text{vol} \left(\frac{\partial \tau(\hat{x})}{\partial \hat{x}_0}; \frac{\partial \tau(\hat{x})}{\partial \hat{x}_1}; \dots; \frac{\partial \tau(\hat{x})}{\partial \hat{x}_{P-1}} \right)| d\hat{x}.$$

Denoting $J_\tau(\hat{x})$ the jacobian

$$J_\tau(\hat{x}) := |\text{vol} \left(\frac{\partial \tau(\hat{x})}{\partial \hat{x}_0}; \frac{\partial \tau(\hat{x})}{\partial \hat{x}_1}; \dots; \frac{\partial \tau(\hat{x})}{\partial \hat{x}_{P-1}} \right)| = (\det(K(\hat{x})^T K(\hat{x})))^{1/2},$$

one finally has

$$\int_T f(x) dx = \int_{\hat{T}} \hat{f}(\hat{x}) J_\tau(\hat{x}) d\hat{x}.$$

When $P = N$, the expression of the jacobian reduces to $J_\tau(\hat{x}) = |\det(K(\hat{x}))|$.

5.2 Surface integral

With Γ a part of the boundary of T a real element and $\hat{\Gamma}$ the corresponding boundary on the reference element \hat{T} , one has

$$\int_\Gamma f(x) d\sigma = \int_{\hat{\Gamma}} \hat{f}(\hat{x}) \|B(\hat{x})\hat{n}\| J_\tau(\hat{x}) d\hat{\sigma},$$

where \hat{n} is the unit normal to \hat{T} on $\hat{\Gamma}$. In a same way

$$\int_\Gamma F(x) \cdot n d\sigma = \int_{\hat{\Gamma}} \hat{F}(\hat{x}) \cdot (B(\hat{x}) \cdot \hat{n}) J_\tau(\hat{x}) d\hat{\sigma},$$

For n the unit normal to T on Γ .

5.3 Derivative computation

One has

$$\nabla f(x) = B(\hat{x})\widehat{\nabla}f(\hat{x}).$$

5.4 Second derivative computation

Denoting

$$\nabla^2 f = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right]_{ij},$$

the $N \times N$ matrix and

$$\widehat{X}(\hat{x}) = \sum_{k=0}^{N-1} \widehat{\nabla}^2 \tau_k(\hat{x}) \frac{\partial f}{\partial x_k}(x) = \sum_{k=0}^{N-1} \sum_{i=0}^{P-1} \widehat{\nabla}^2 \tau_k(\hat{x}) B_{ki} \frac{\partial f}{\partial \hat{x}_i}(\hat{x}),$$

the $P \times P$ matrix, then

$$\widehat{\nabla}^2 f(\hat{x}) = \widehat{X}(\hat{x}) + K(\hat{x})^T \nabla^2 f(x) K(\hat{x}),$$

and thus

$$\nabla^2 f(x) = B(\hat{x})(\widehat{\nabla}^2 f(\hat{x}) - \widehat{X}(\hat{x}))B(\hat{x})^T.$$

In order to have uniform methods for the computation of elementary matrices, the Hessian is computed as a column vector Hf whose components are $\frac{\partial^2 f}{\partial x_0^2}, \frac{\partial^2 f}{\partial x_1 \partial x_0}, \dots, \frac{\partial^2 f}{\partial x_{N-1}^2}$. Then, with B_2 the $P^2 \times P$ matrix defined as

$$[B_2(\hat{x})]_{ij} = \sum_{k=0}^{N-1} \frac{\partial^2 \tau_k(\hat{x})}{\partial \hat{x}_{i/P} \partial \hat{x}_{j \bmod P}} B_{kj}(\hat{x}),$$

and B_3 the $N^2 \times P^2$ matrix defined as

$$[B_3(\hat{x})]_{ij} = B_{i/N, j/P}(\hat{x}) B_{i \bmod N, j \bmod P}(\hat{x}),$$

one has

$$Hf(x) = B_3(\hat{x}) \left(\widehat{H} f(\hat{x}) - B_2(\hat{x}) \widehat{\nabla} f(\hat{x}) \right).$$

5.5 Example of elementary matrix

Assume one needs to compute the elementary “matrix”:

$$t(i_0, i_1, \dots, i_7) = \int_T \varphi_{i_1}^{i_0} \partial_{i_4} \varphi_{i_3}^{i_2} \partial_{i_7/P, i_7 \bmod P}^2 \varphi_{i_6}^{i_5} dx,$$

The computations to be made on the reference elements are

$$\widehat{t}_0(i_0, i_1, \dots, i_7) = \int_{\widehat{T}} (\widehat{\varphi})_{i_1}^{i_0} \partial_{i_4} (\widehat{\varphi})_{i_3}^{i_2} \partial_{i_7/P, i_7 \bmod P}^2 (\widehat{\varphi})_{i_6}^{i_5} J(\widehat{x}) d\widehat{x},$$

and

$$\widehat{t}_1(i_0, i_1, \dots, i_7) = \int_{\widehat{T}} (\widehat{\varphi})_{i_1}^{i_0} \partial_{i_4} (\widehat{\varphi})_{i_3}^{i_2} \partial_{i_7} (\widehat{\varphi})_{i_6}^{i_5} J(\widehat{x}) d\widehat{x},$$

Those two tensor can be computed once on the whole reference element if the geometric transformation is linear (because $J(\widehat{x})$ is constant). If the geometric transformation is non-linear, what has to be stored is the value on each integration point. To compute the integral on the real element a certain number of reductions have to be made:

- Concerning the first term $(\varphi_{i_1}^{i_0})$ nothing.
- Concerning the second term $(\partial_{i_4} \varphi_{i_3}^{i_2})$ a reduction with respect to i_4 with the matrix B .
- Concerning the third term $(\partial_{i_7/P, i_7}^2 \text{ mod } P \varphi_{i_6}^{i_5})$, a reduction of \widehat{t}_0 with respect to i_7 with the matrix B_3 and a reduction of \widehat{t}_1 with respect also to i_7 with the matrix $B_3 B_2$

The reductions are to be made on each integration point if the geometric transformation is non-linear. Once those reductions are done, an addition of all the tensor resulting of those reductions is made (with a factor equal to the load of each integration point if the geometric transformation is non-linear).

If the finite element is non- τ -equivalent, a supplementary reduction of the resulting tensor with the matrix M has to be made.

REFERENCES

BIBLIOGRAPHY

- [AL-CU1991] P. Alart, A. Curnier. *A mixed formulation for frictional contact problems prone to newton like solution methods*, Computer Methods in Applied Mechanics and Engineering 92, 353–375, (1991).
- [bank1983] 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).
- [ca-re-so1994] D. Calvetti, L. Reichel and D.C. Sorensen. *An implicitly restarted Lanczos method for large symmetric eigenvalue problems*. Electronic Transaction on Numerical Analysis}. 2:1-21, (1994).
- [ciarlet1978] P.G. Ciarlet, *The finite element method for elliptic problems*, Studies in Mathematics and its Applications vol. 4 (1978), North-Holland.
- [EncyclopCubature] R. Cools, [An Encyclopedia of Cubature Formulas](#), J. Complexity.
- [dh-to1984] G. Dhatt, G. Touzot, *The Finite Element Method Displayed*, J. Wiley & Sons, New York, (1984).
- [dh-go-ku2003] A. Dhooge, W. Govaerts and Y. A. Kuznetsov, *MATCONT: A MATLAB Package for Numerical Bifurcation Analysis of ODEs*, ACM Trans. Math. Software 31 (2003), 141-164.
- [LA-RE2006] P. Laborde, Y. Renard. *Fixed point strategies for elastostatic frictional contact problems*, Math. Meth. Appl. Sci., 31:415-441, (2008).
- [Xfem] N. Moës, J. Dolbow and T. Belytschko, *A finite element method for crack growth without remeshing*, Int. J. Num. Meth. Engng. 46 (1999), 131-150.
- [KH-PO-RE2006] Khenous H., Pommier J., Renard Y. *Hybrid discretization of the Signorini problem with Coulomb friction, theoretical aspects and comparison of some numerical solvers*. Applied Numerical Mathematics, 56/2:163-192, 2006.
- [KI-OD1988] Kikuchi N., Oden J.T., *Contact problems in elasticity*, SIAM, 1988.
- [HI-RE2010] Hild P., Renard Y. *Stabilized lagrange multiplier method for the finite element approximation of contact problems in elastostatics*. Numer. Math. 15:1 (2010), 101–129.
- [nedelec1991] J.-C. Nedelec. *Notions sur les techniques d'éléments finis*, Ellipses, SMAI, Mathématiques & Applications n°7, (1991).
- [SCHADD] L.F. Pavarino. *Domain decomposition algorithms for the p-version finite element method for elliptic problems*, Luca F. Pavarino. PhD thesis, Courant Institute of Mathematical Sciences}. 1992.
- [remacle2002] J-F. Remacle, M. Shephard, *An algorithm oriented database*, Int. J. Num. Meth. Engng. 58 (2003), 349-374.
- [so-se-do2004] P. Solin, K. Segeth, I. Dolezel, *Higher-Order Finite Element Methods*, Chapman and Hall/CRC, Studies in advanced mathematics, 2004.
- [ZT1989] Zienkiewicz and Taylor “The finite element method” 5th edition volume 3 : Fluids Dynamics, section 2.6

INDEX

A

add_dependency (C function), 16

B

bgeot::convex_ref_product (C function), 6

bgeot::parallelepiped_of_reference (C function), 6

bgeot::parallelepiped_structure (C function), 5

bgeot::prism_structure (C function), 5

bgeot::simplex_of_reference (C function), 6

bgeot::simplex_structure (C function), 5

C

context_check (C function), 16

context_valid (C function), 16

T

touch (C function), 16