Description of the Project

Release 5.3

Yves Renard, Julien Pommier, Konstantinos Poulios

December 21, 2018

Contents

CHAPTER 1

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 Savannah https://savannah.nongnu.org/task/?group=getfem.

The *GetFEM++* project focuses on the development of an open source 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 ...

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 Python, Scilab and matlab interfaces allows to easily build application coupling the definition of the problem, the finite element methods selection and the graphical post-processing.

Public License and GCC Runtime Library Exception for more details. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

CHAPTER 2

How to contribute / Git repository on Savannah

GetFEM++ is an open source finite element library based on a collaborative development. If you intend to make some contributions, you can ask for membership of the project there. Contributions of all kinds are welcome: documentation, bug reports, constructive comments, changes suggestions, bug fix, new models, etc ...

Contributors are of course required to be careful that their changes do not affect the proper functioning of the library and that these changes follow a principle of backward compatibility.

See here for a list of task and discussions about *GetFEM++* development.

IMPORTANT : a contributor implicitly accepts that his/her contribution will be distributed under the LGPL licence of *GetFEM++*.

The main repository of *GetFEM++* is on Savannah, the software forge of the Free Software Foundation (see Savannah). The page of the project on Savannah is Getfem on Savannah. See also Getfem sources on Savannah.

How to get the sources

If you just want the sources and do not intend to make some contributions, you can just use the command

```
git clone https://git.savannah.nongnu.org/git/getfem.git
```

If you intend to make some contributions, the first step is to ask for the inclusion in the *GetFEM++* project (for this you have to create a Savannah account). You have also to register a ssh key (see git on Savannah) and then use the command

```
git clone ssh://savannah-login@git.sv.gnu.org:/srv/git/getfem.git
```

How to contribute

Before modifying any file, you have to create a *development branch* because it is *not allowed to make a modification directly in the master branch*. It is recommended that the branch name is of the type *devel-name-subject* where name is your name or login and subject the main subject of the changes. For instance, if you chose *devel-me-rewrite-fem-kernel* as the branch name, the creation of the branch reads

```
git branch devel-me-rewrite-fem-kernel
git checkout devel-me-rewrite-fem-kernel
```

The first command create the branch and the second one position you on your branch. After that you are nearly ready to makes some modifications. You can specify your contact name and e-mail with the following commands in order to label your changes

```
git config --global user.name "Your Name Comes Here"
git config --global user.email you@yourdomain.example.com
```

Specific branch for doc improvements and typo-fixes

If you want to contribute to the documentation only, it is not necessary to build a specific branch. You can just checkout to the `fixmisspell` branch which has been created for this purpose with

```
git checkout fixmisspell
```

Locally commit your changes

Once you made some modifications of a file or you added a new file, say *src/toto.cc*, the local commit is done with the commands:

```
git add src/toto.cc
git commit -m "Your extensive commit message here"
```

At this stage the commit is done on your local repository but not in the Savannah one.

Push you changes in the Savannah repository

You can now transfer your modifications to the Savannah repository with

```
git push origin devel-me-rewrite-fem-kernel
```

where of course *devel-me-rewrite-fem-kernel* is still the name of your branch. At this stage your modifications are registered in the branch *devel-me-rewrite-fem-kernel* of Savannah repository. Your role stops here, since you are not allowed to modify the master branch of *GetFEM++*.

Ask for an admin to merge your modifications to the master branch of *GetFEM++*

Once you validated your modifications with sufficient tests, you can ask an admin of *GetFEM++* to merge your modifications. For this, contact one of them directly, or send an e-mail to *getfem-commits@nongnu.org* with the message : "please merge branch devel-me-rewrite-fem-kernel" with eventually a short description of the modifications. IMPORTANT : by default, your branch will be deleted after the merge, unless you express the need to keep it.

Merge modifications done by other contributors

You can run a

```
git pull origin master
git merge master
```

in order to integrate the modifications which has been validated and integrated to the master branch. This is recommended to run this command before any request for integration of a modification in the master branch.

Some useful git commands

```
git status  : status of your repository / branch

git log --follow "filepath"   : Show all the commits modifying the specified file (and
 ↪  follow the eventual change of name of the file).

gitk --follow filename : same as previous but with a graphical interface
```

Contributing to document translation

The recommended way for new contributors to translate document is to join Getfem translation team on Transifex . For contribution, please make account in transifex and click request language and fill form . After translation, pull translated po file from site by using transifex-client. You need api token which you can get in transifex site.

```
cd doc/sphinx
tx pull -l <lang>
```

Set code for your native language to <lang> (see Currently supported languages by Sphinx are).

Warning: **DO NOT** tx push to transifex. It will have some trouble. You can upload file one by one in team page.

After pulling translated po files, set <lang> to LANGUAGE in *doc/sphinx/Makefile.am* .

```
LANGUAGE      = <lang>
SPHINXOPTS    = -D language=$(LANGUAGE)
```

Then, you can run a following commands in order to make html localization document.

```
cd doc/sphinx
make html
```

If you want to make pdf file in your language, you can run a

```
make latex
cd build/latex
make all-pdf-<lang>
```

See details in Sphinx Internationalization .

CHAPTER 3

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 ...).

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::pconv$`
: description of the direct product of `p1` and `p2`.

`bgeot::`**`prism_P1_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->nb_faces()`), the dimension of the convex (`p->dim()`), for the number of vertices (`p->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).

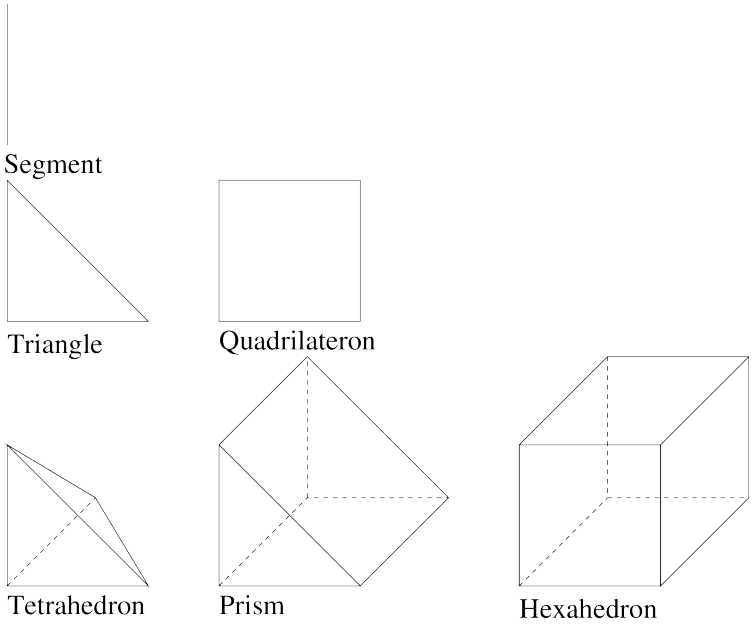


Figure 3.1: usual elements

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`.

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 ith 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`.

Geometric transformations

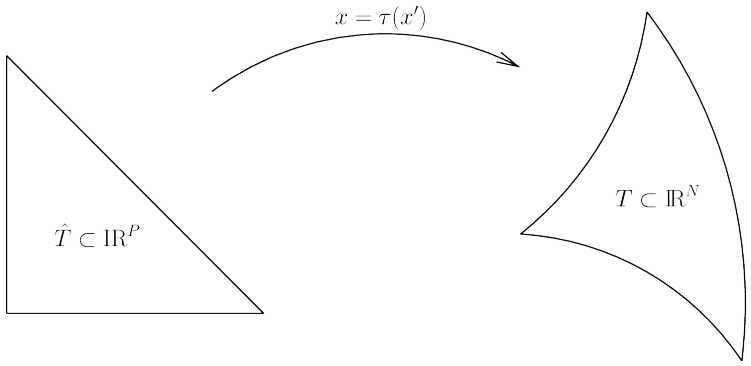


Figure 3.2: geometric transformation

A geometric transformation is a polynomial application:

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

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

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

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

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

such that

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

Denoting

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

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

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

The derivative of τ is then

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

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

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

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

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.

Finite element methods description

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

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

Denoting

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

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 $[\widehat{\varphi}(\widehat{x})]$ the $n_d \times Q$ matrix whose ith line is $(\widehat{\varphi})^i(\widehat{x})$. Whis this notation, for a function is defined by

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

one has

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

where α is the vector whose ith component is α_i.

A certain number of description of classical finite element method are defined in the file `getfem_fem.h`. See *ud-appendixa* 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.

CHAPTER 4

Description of the different parts of the library

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*.

Gmm library

Description

Gmm++ is a template 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.

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`.

See *gmm* documenation for more details.

Files

All files in src/gmm

State

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

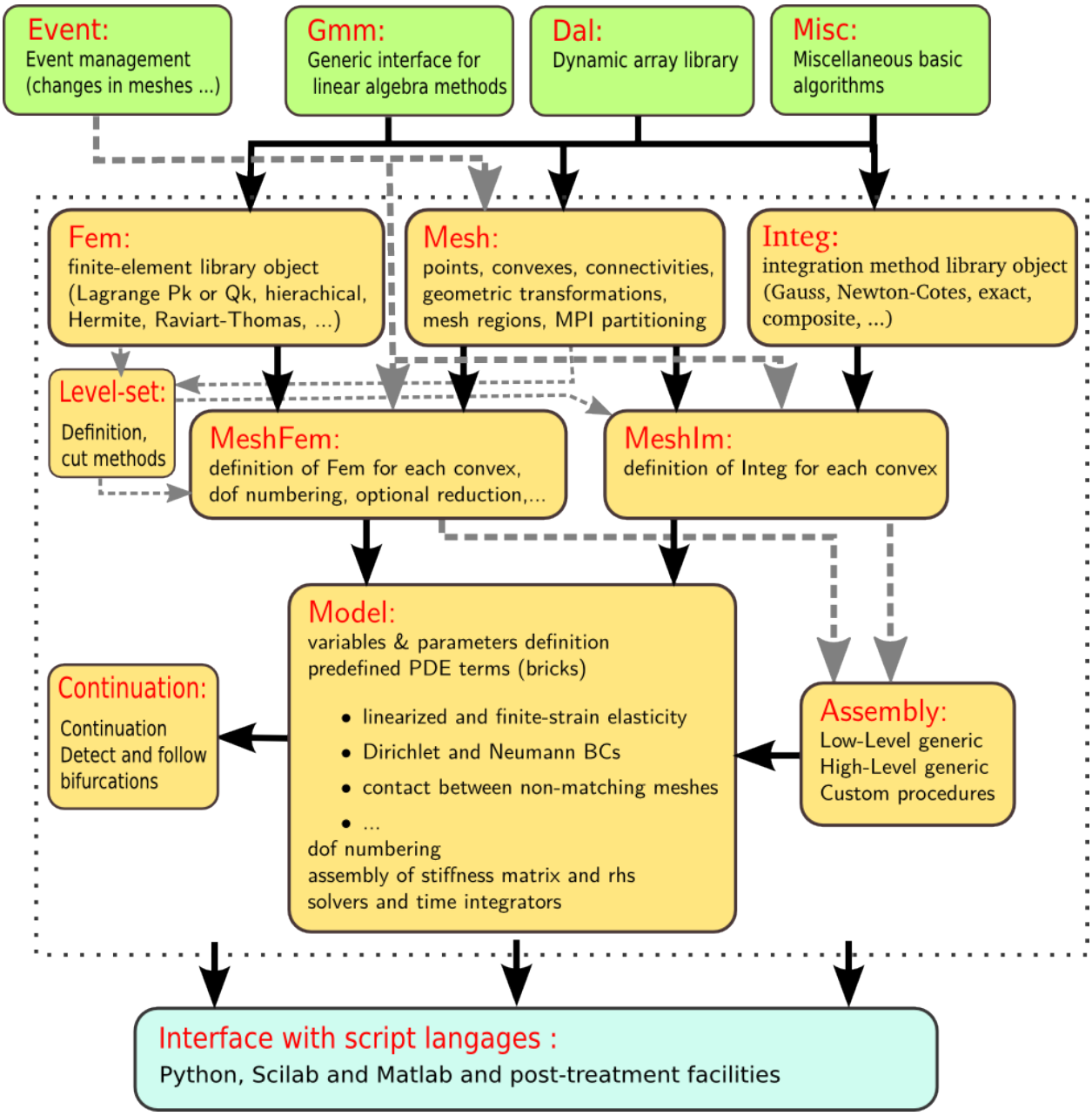


Figure 4.1: Diagram of *GetFEM++* library

Perspectives

There is potentially several points to be improved in *Gmm++* (partial introduction of expression template for some base types of matrix and vectors, think about 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 change seem to be unnecessary. 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 (DIFFPACK for instance) 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.

Dal library

Description

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.

Furthermore, `dal` contains a certain number of basic algorithms to deal with static stored objects (description of finite element methods, intermediary structures for auxiliary computations ...).

Files

File(s)	Description
`dal_config.h`	Mainly load *Gmm++* header files
`dal_basic.h`	A variable size array container, dal::dynamic_array<T>.
`dal_bit_vector.h` and `dal_bit_vector.cc`	A improved bit vector container based on dal::dynamic_array<T>.
`dal_tas.h`	A heap container based on dal::dynamic_array<T>.
`dal_tree_sorted.h`	A balanced tree stored array based on dal::dynamic_array<T>.
`dal_static_stored_objects.h` and `dal_static_stored_objects.cc`	Allows to store some objects and dependencies between some objects. Used to store many things in *GetFEM++* (finite element methods, integration methods, pre-computations, ...).
`dal_naming_system.h`	A generic object to associate a name to a method descriptor and store the method descriptor. Used for finite element methods, integration methods and geometric transformations. Uses dal::static_stored_object.
`dal_shared_ptr.h`	A simplified version of boost::shared_ptr.
`dal_singleton.h` and `dal_singleton.cc`	A simple singleton implementation which has been made thread safe for OpenMP (singletons are replicated n each thread).
`dal_backtrace.h` and `dal_backtrace.cc`	For debugging, dump glibc backtrace.

State

Stable, not evolving too much.

Perspectives

No plan.

Miscellaneous algorithms

Description

A set of miscellaneous basic algorithms and definitions used in *GetFEM++*.

Files

File(s)	Description
`bgeot_comma_init.h`	Allow to init container with a list of values, from boost init.hpp.
`bgeot_ftool.h` and `bgeot_ftool.cc`	Small language allowing to read a parameter file with a Matlab syntax like. Used also for structured meshes.
`bgeot_kdtree.h` and `bgeot_kdtree.cc`	Balanced N-dimensional tree. Store a list of points and allows a quick search of points lying in a given box.
`bgeot_rtree.h` and `bgeot_rtree.cc`	Rectangle tree. Store a list of N-dimensional rectangles and allows a quick search of rectangles containing a given point.
`permutations.h`	Allows to iterate on permutations. Only used in `getfem_integration.cc`.
`bgeot_small_vector.h` and `bgeot_small_vector.cc`	Defines a vector of low dimension mainly used to represent mesh nodes. Optimized operations.
`bgeot_tensor.h`	Arbitrary order tensor. Used in assembly.
`bgeot_sparse_tensors.h` and `bgeot_sparse_tensors.cc`	Arbitrary order sparse tensor. Used in the low-level generic assembly.
`getfem_omp.h` and `getfem_omp.cc`	Tools for multithreaded, OpenMP and Boost based parallelization.
`getfem_export.h` and `getfem_export.cc`	Export in pos and vtk formats
`getfem_superlu.h` and `getfem_superlu.cc`	Interface with Superlu (the included version or an external one)

State

Perspectives

Events management

Description

The `mesh`, *mesh_fem*, *mesh_im* and *model* description are linkedtogether 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.

Files

File(s)	Description
`getfem_context.h` and `getfem_context.cc`	Define a class *context_dependencies* from which all object has to derive in order to manage events.

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.

Perspectives

The event management of some objects should be analysed with care This is the case for instance of|mls|, *mesh_fem_level_set*, *partial_mesh_fem*, etc.

The event management still have to be improved to be a fully reactive system.

Mesh module

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.).

Files

File(s)	Description
`bgeot_convex_structure.h` and `bgeot_convex_structure.cc`	Describes the structure of an element disregarding the coordinates of its vertices.
`bgeot_mesh_structure.h` and `bgeot_mesh_structure.cc`	Describes the structure of a mesh disregarding the coordinates of the nodes.
`bgeot_node_tab.h` and `bgeot_node_tab.cc`	A node container allowing the fast search of a node and store nodes identifying the too much close nodes.
`bgeot_convex.h`	Describes an element with its vertices.
`bgeot_convex_ref.h` and `bgeot_convex_ref.cc` and `bgeot_convex_structure.cc`	Describe reference elements.
`bgeot_mesh.h`	Describes a mesh with the collection of node (but without the description of geometric transformations).
`getfem_mesh_region.h` and `getfem_mesh_region.cc`	Object representing a mesh region (boundary or part of a mesh).
`bgeot_geometric_trans.h` and `bgeot_geometric_trans.cc`	Describes geometric transformations.
`bgeot_geotrans_inv.h` and `bgeot_geotrans_inv.cc`	A tool to invert geometric transformations.
`getfem_mesh.h` and `getfem_mesh.cc`	Fully describes a mesh (with the geometric transformations, subparts of the mesh, support for parallelization). Includes the Bank algorithm to refine a mesh.
`getfem_deformable_mesh.h`	defines an object capable to deform a mesh with respect to a displacement field and capable to restore it
`getfem_mesher.h` and `getfem_mesher.cc`	An experimental mesher, in arbitrary dimension. To be used with care and quite slow (because of node optimization). It meshes geometries defined by some level sets.
`getfem_import.h` and `getfem_import.cc`	Import mesh files in various formats
`getfem_regular_meshes.h` and `getfem_regular_meshes.cc`	Produces structured meshes
`getfem_mesh_slicers.h` and `getfem_mesh_slicers.cc`	A slice is built from a mesh, by applying some slicing operations (cut the mesh with a plane, intersect with a sphere, take the boundary faces, etc..). They are used for post-treatment (exportation of results to VTK or OpenDX, etc.).
`getfem_mesh_slice.h` and `getfem_mesh_slice.cc`	Store mesh slices.

State

Stable and not evolving so much.

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 bibliographical review on how to efficiently store a mesh and implement the main operations (add a node, an element, deal with faces, find the neighbour elements, the isolated faces ...) would be interesting to make the mesh structure evolve.

A senstive algorithm is the one (in bgeot_node_tab.cc) which identify the too much close nodes. More investigations (and documentation) are probably necessary.

Fem module

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 neighbour elements in a generic way (for instance connecting a Lagrange 2D element to another Lagrange 1D element).

Files

File(s)	Description
`bgeot_poly.h` and `bgeot_poly_composite.h` and `bgeot_poly.cc` and `bgeot_poly_composite.cc`	Some classes to represent polynomials and piecewise polynomials in order to describe shape functions of a finite element method on the reference element.
`getfem_fem.h` and `getfem_fem.cc` and `getfem_fem_composite.cc`	Descriptors for finite element and a degree of freedom. Polynomial finite elements are defined in `getfem_fem.cc` and piecewise polynomial finite elements in `getfem_fem_composite.cc`
`getfem_fem_global_function.h` and `getfem_fem_global_function.cc`	Defines a fem with base functions defined as global functions given by the user. Useful for enrichment with singular functions and for implementation of meshless methods.
`getfem_projected_fem.h` and `getfem_projected_fem.cc`	Defines a fem which is the projection of a finite element space (represented by a mesh_fem) on a different mesh. Note that the high-generic assembly language offers also this functionality by means of the interpolated transformations.
`getfem_interpolated_fem.h` and `getfem_interpolated_fem.cc`	Dsfines a fem which is the interpolation of a finite element space (represented by a mesh_fem) on a different mesh. Note that the high-generic assembly language offers also this functionality by means of the interpolated transformations.

State

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 *ud-appendixa*.

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.

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 (Hermites elements, Argyris, etc.).

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).

Integ module

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.

Files

File(s)	Description
`getfem_integration.h` and `getfem_integration.cc` and `getfem_integration_composite.cc`	Structure of integration methods, basic integration methods, product of integration method and composite methods.
`getfem_im_list.h`	file generated by `cubature/make_getfem_list` with the integration methods defined in cubature directory. This gives the possibility to define a new integration method just listing the Gauss points and weigth in a text file.

State

This module meets the present needs for the project and is considered as stabilized. The list of available cubature formulas is given in *ud-appendixb*.

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 of descritption files).

MeshFem module

Description

The MeshFem module aims to represent a finite element method (space) with respect to a given mesh. The mesh_fem object will be permanently linked to the given mesh and will be able to react to changes in the mesh (addition or deletion of elements, in particular). A mesh_fem object may associate a different finite element method on each element of the mesh even though of course, the most common case it that all the element share the same finite element method.

Files

File(s)	Description
`getfem_mesh_fem.h` and `getfem_mesh_fem.cc`	Defines the structure representing a finite element on a whole mesh. Each element of the mesh is associated with a finite element method. This is a quite complex structure which perform dof identification and numbering, allows a global linear reduction.
`getfem_mesh_fem_global_function.h` and `getfem_mesh_fem_global_function.cc`	Defines mesh_fem with fem defined as a fem_global_function. It provides convenience methods for updating the list of base functions in the linked fem_global_function.
`getfem_mesh_fem_product.h` and `getfem_mesh_fem_product.cc`	Produces a mesh_fem object which is a kind of direct product of two finite element method. Useful for Xfem enrichment.
`getfem_mesh_fem_sum.h` and `getfem_mesh_fem_sum.cc`	Produces a mesh_fem object which is a kind of direct sum of two finite element method. Useful for Xfem enrichment.
`getfem_partial_mesh_fem.h` and `getfem_partial_mesh_fem.cc`	Produces a mesh_fem with a reduced number of dofs
`getfem_interpolation.h` and `getfem_interpolation.cc`	Interpolation between two finite element methods, possibly between different meshes. The interpolation facilities of the high-level generic assembly can be used instead.
`getfem_derivatives.h`	Interpolation of some derivatives of a finite element field on a (discontinuous) Lagrange finite element. The interpolation facilities of the high-level generic assembly can be used instead.
`getfem_inter_element.h` and `getfem_inter_element.cc`	An attempt to make framework for inter-element computations (jump in normal derivative for instance). To be continuated and perhaps integrated into the generic assembly language.
`getfem_error_estimate.h` and `getfem_error_estimate.cc`	An attempt to make framework for computation of error estimates. To be continuated and perhaps integrated into the generic assembly language.
`getfem_crack_sif.h`	Crack support functions for computation of SIF(stress intensity factors).
`getfem_torus.h` and `getfem_torus.cc`	Adapt a mesh_fem object which extends a 2D dimensional structure with a radial dimension.

State

Stable. Not evolving so much.

Perspectives

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

MeshIm module

Description

Defines an integration method on a whole mesh.

Files

File(s)	Description
`getfem_mesh_im.h` and `getfem_mesh_im.cc`	Object which defines an integration method on each element of the mesh. Reacts to the main mesh changes (add or deletion of elements).
`getfem_im_data.h` and `getfem_im_data.cc`	Define an object representing a scalar, a vector or a tensor on each Gauss point of a mesh_im object. Used for instance in plasticity approximation. Interpolation of arbitrary expressions can be made thanks to the weak form language.

State

Stable, not evolving so much.

Perspectives

Level-set module

Description

Define level-set objects and cut meshes, integration method and finite element method with respect to one or several level-set.

Files

File(s)	Description
`getfem_level_set.h` and `getfem_level_set.cc`	Define a level-set function (scalar field defined on a Lagrange fem) with an optional secondary level-set function.
`getfem_mesh_level_set.h` and `getfem_mesh_level_set.cc`	Cut a mesh with respect to one or several level-sets.
`getfem_fem_level_set.h` and `getfem_fem_level_set.cc`	Define a special finite element method which depends on the element and which is cut by one or several level-sets.
`getfem_mesh_fem_level_set.h` and `getfem_mesh_fem_level_set.cc`	Produces a mesh_fem object with shape functions cut by one or several level-sets.
`getfem_mesh_im_level_set.h` and `getfem_mesh_im_level_set.cc`	Produce a mesh_im representing an intergation method cut by the level set and being on on side of level-set, the oter side, both or only on the levelset itself.
`getfem_level_set_contact.h` and `getfem_level_set_contact.cc`	A level set based large sliding contact algorithm for an easy analysis of implant positioning.
`getfem_convect.h`	Compute the convection of a quantity with respect to a vector field. Used to computate the evolution of a level-set function for instance. Galerkin characteristic method.

State

Stable.

Perspectives

Clarify the alorithm computing the different zones.

The high-level generic assembly module in *GetFEM++*

Description

The high level generic assembly module of *GetFEM++* and its weak form language is a key module which allows to describe weak formulation of partial differential equation problems. See the description of the language in the user documentation section *ud-gasm-high*.

Files

File(s)	Description
`getfem_generic_assembly.h`	Main header for exported definitions. Only this header has to be included to use the generic assembly. Other headers of the module are for internal use only.
`getfem_generic_assembly_tree.h` and `getfem_generic_assembly_tree.cc`	Definition of the tree structure and basic operations on it, including reading an assembly string and transform it in a syntax tree and make the invert transformation of a tree into a string.
`getfem_generic_assembly_function_and_operators.h` and `getfem_generic_assembly_function_and_operators.cc`	Definition of predefined function and nonlinear operator of the weak form language.
`getfem_generic_assembly_semantic.h` and `getfem_generic_assembly_semantic.cc`	Semantic analysis and enrichment of the syntax tree. Include some operations such as making the derivation of a tree with respect to a variable or computing the tree corresponding to the gradient of an expression.
`getfem_generic_assembly_workspace.cc`	Methodes of the workspace object (defined in `getfem_generic_assembly.h`).
`getfem_generic_assembly_compile_and_exec.h` and `getfem_generic_assembly_compile_and_exec.cc`	Definition of the optimized instructions, compilation into a sequel of optimize instructions and execution of the instructions on Gauss point/interpolation points.
`getfem_generic_assembly_interpolation.cc`	Interpolation operations and interpolate transformations.

A few implementation details

The assembly string is transformed in an assembly tree by a set of function in `src/getfem_generic_assembly.cc`. The process has 6 steps:

- Lexical analysis with the procedure `ga_get_token(...)`.
- Syntax analysis and transformation into a syntax tree by `ga_read_string(...)`.
- Semantic analysis, simplification (pre-computation) of constant expressions and enrichment of the tree by `ga_semantic_analysis(...)`.
- Symbolic (automatic) differentiation of an assembly tree by `ga_derivative(...)`
- Symbolic (automatic) gradient computation of an assembly tree by `ga_gradient(...)`
- Compilation in a sequence of instructions with optimizations by `ga_compile(...)`.
- Execution of the sequence of instructions and assembly by `ga_exec(...)`.

These steps are performed only once at the beginning of the assembly. The final tree is compiled in a sequence of optimized instructions which are executed on each Gauss point of each element. The compilation performed some optimizations : repeated terms are automatically detected and evaluated only once, simplifications if the mesh has uniform type of elements, simplifications for vectorized fnite element methods.

Moreover, there is specifics function for interpolation operations (ga_interpolation(...), ga_interpolation_exec(...), ga_interpolation_Lagrange_fem, ga_interpolation_mti, ga_interpolation_im_data, ...)

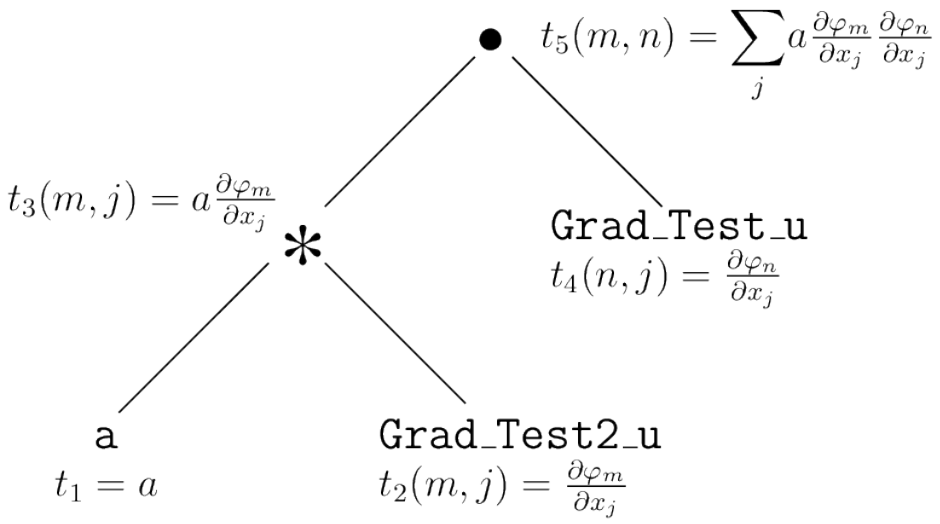
Assembly tree

Assembly strings are transformed into assembly trees by `ga_read_string(...)`. Assembly trees are syntax trees that are progressively enriched in information in the differents steps (semantic analysis, derivation, compilation).

The object `ga_tree` represents an assembly tree. It is a copyable object that only contains a pointer to the root of the tree. Each tree node is an object `ga_tree_node` that contains the main following information:

- node_type (function, variable value, variable gradient, operation ...)
- operation type for operation nodes.
- assembly tensor: used at execution time by optimized instructions to compute the intermediary results. The final result is in the assembly string of the root node at the end of the execution (for each Gauss point).
- term type: value, order one term (ith order one test functions), order two term (with order two test functions) or with both order one and order two test functions (tangent term).
- variable name of tests functions for order 1 or 2 terms.
- pointer to the parent node.
- pointers to the children nodes.

For example, the assembly tree for the assembly string "a*Grad_Test2_u.Grad_Test_u" for the stiffness matrix of a Laplacian problem can be represented as follows with its assembly tensors at each node:



Assembly tensors

Assembly tensors are represented on each node by a `bgeot::tensor<double>` object. However, there is a specific structure in `src/getfem_generic_assembly.cc` for assembly tensors because there is several format of assembly tensors :

- Normal tensor. The first and second indices may represent the test function local indices if the node represent a first or second order term. Remember that in *GetFEM++* all tensors are stored with a Fortran order. This means that for instance a for a $N \times P \times Q$ tensor one has `t(i, j, k) = t[i + j*N + k*N*P]`.
- Copied tensor. When a node is detected to have exactly the same expression compared to an already compiled one, the assembly tensor will contain a pointer to the assembly tensor of the already compiled node. The consequence is that no unnecessary copy is made.
- Sparse tensor with a listed sparsity. When working with a vector field, the finite element method is applied on each component. This results on vector base functions having only one nonzero component and some components are duplicated. The tensor are fully represented because it would be difficul to gain in efficiency with that kind of small sparse tensor format. However, some operation can be optimized with the knoledge of a certain

sparsity (and duplication). This can change the order of complexity of a reduction. In order to allows this gain in efficiency, the tensor are labellised with some known sparsity format (vectorisation and format coming from operation applied on vectorized tensors). This results in a certain number of sparsity formats that are listed below:

- 1: Vectorized base sparsity format: The tensor represent a vectorized value. Each value of the condensed tensor is repeated on Q components of the vectorized tensor. The mesh dimensions is denoted N. For instance if φ_i are the M local base functions on an element and the evaluation is on a Gauss point x, then the non vectorized tensor is $\bar{t}(i) = \varphi_i(x)$ and the vectorized one is $t(j,k) = \varphi_{j/Q}(x)\delta_{k,j \bmod Q}$ where j/M is the integer division. For $M = 2$, $Q = 2$ and $N = 3$ the components of the two tensors are represented in the following table

Scalar tensor	Vectorized tensor
$\bar{t}(i) = \varphi_i(x)$	$t(j,k) = \varphi_{j/Q}(x)\delta_{k,(j \bmod Q)}$
$[\varphi_0(x), \varphi_1(x)]$	$[\varphi_0(x), 0, \varphi_1(x), 0, 0, \varphi_0(x), 0, \varphi_1(x)]$

- 2: Grad condensed format

Scalar tensor	Vectorized tensor
$\bar{t}(i,j) = \partial_j\varphi_i(x)$	$t(k,l,m) = \partial_m\varphi_{k/Q}(x)\delta_{l,(m \bmod Q)}$
$[\partial_0\varphi_0(x), \partial_0\varphi_1(x), \partial_1\varphi_0(x), \partial_1\varphi_1(x), \partial_2\varphi_0(x), \partial_2\varphi_1(x)]$	

- 3: Hessian condensed format
- 10: Vectorized mass: the tensor represent a scalar product of two vectorised base functions. This means a tensor $t(\cdot,\cdot)$ where $t(i*Q+k, j*Q+l) = 0$ for $k \neq l$ and $t(i*Q+k, j*Q+k)$ are equals for $0 \leq k < Q$.

Optimized instructions

Optimized instructions for variable evaluation, operations, vector and matrix assembly ... to be described.

Predefined functions

Some predefined scalar functions are available in *GetFEM++* weak form language in order to describe a weak formulation (or a

- A C++ function which computes the value given the argument(s).
- The support of the function in the first each argument in term of a (possibly infinite) interval (this is for simplification of expressions).
- The string corresponding of the derivative in terms of already known functions

A new predefined function is easy to add. See init_predefined_functions() in file `src/getfem_generic_assembly.cc`. + describe how to give the derivative ...

Predefined nonlinear operators

to be described ...

State

Stable.

Perspectives

- Is a certain extension to complex data possible ?
- More simplifications : study the possibility to automatically factorize some terms (for instance scalar ones) to reduce the number of operations.

The low-level generic assembly module in *GetFEM++*

Description

First version of the generic assembly. Base on tensor reduction. Not very convenient for nonlinear terms. The high-level generic assembly have to be prefered now.

Files

File(s)	Description
`getfem_mat_elem_type.h` and:file:‘ getfem_mat_elem_type.cc	Defines bes type for components of an elementary matrix.
`getfem_mat_elem.h` and:file:‘ getfem_mat_elem.cc	Describes an compute elementary matrices.
`getfem_assembling_tensors.h` and:file:*getfem_assembling_tensors.cc*	Performs the assembly.
`getfem_assembling.h`	Various assembly terms (linear elasticity, generix elliptic term, Dirichlet condition ...

State

Stable.

Perspectives

Will not evolve since the efforts are now focused on the high-level generic assembly.

Model module

Description

Describe a model (variable, data and equation terms linking the variables).

Files

File(s)	Description
getfem_models.h and getfem_models.cc	Defines the object models, its internal and the standard bricks (linear elasticity, generic elliptic brick, Dirichlet boundary conditions ...).
getfem_model_solvers.h and getfem_model_solvers.cc	Defines the the standard solvers for the model object.
getfem_contact_and_friction_common.h and getfem_contact_and_friction_common.cc	Common algorithms for contact/friction conditions on deformable bodies
getfem_contact_and_friction_integral.h and getfem_contact_and_friction_integral.cc	Small sliding Contact/friction bricks of integral type.
getfem_contact_and_friction_large_sliding.h and getfem_contact_and_friction_large_sliding.cc	Large sliding contact/friction bricks.
getfem_contact_and_friction_nodal.h and getfem_contact_and_friction_nodal.cc	Small sliding nodal Contact/friction bricks.
getfem_Navier_Stokes.h	An attempt for Navier-Stokes bricks. To be improved.
getfem_fourth_order.h and getfem_fourth_order.cc	Bilaplacian and Kirchhoff-Love plate bricks
getfem_linearized_plates.h and getfem_linearized_plates.cc	Mindlin-Reissner plate brick
getfem_nonlinear_elasticity.h and getfem_nonlinear_elasticity.cc	Large deformation elasticity bricks.
getfem_plasticity.h and getfem_plasticity.cc	Plasticity bricks.

State

Constant evolution to includes nex models.

Perspectives

More plate, road and shell bricks, plasticity in large deformation, ...

Continuation module

Description

Allows to follow a solution with respect to a parameter (continuation method), detect a bifurcation and allow branching. Work for low regularity problems (Lipschitz regularity). Use an adapted Moore-Penrose continuation method.

Files

File(s)	Description
getfem_continuation.h and getfem_continuation.cc	The generic continuation and branching method

State

Have already generic and advanced functionalities.

Perspectives

Still in developpement.

Interface with scripts languages (Python, Scilab and Matlab)

A simplified (but rather complete) interface of *GetFEM++* is provided, so that it is possible to use getfem in some script languages.

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 acts as a layer above the *GetFEM++* library, and is used by the python, matlab and scilab 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).

Files

All the files in the directory `interfacesrc`. A short description of main files:

- `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 functionnality.

- `getfemint_gsparse.h`, `getfemint_precond.h`, etc.

  Files specific to an interfaced object if needed. (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 which handles the dependency between for example a `mesh` and a `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, scilab 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 be 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 "`bin/extract_doc.py`".

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 separetaly 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 things work 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-comands.
- 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, ...)
   Documention 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, Matlab and Scilab.

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 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 begins with a dot; this means that it corresponds 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 are some authorized abreviations:

- `@tcs` for `@tcont_struct`
- `@tmf` for `@tmesh_fem`
- `@tgt` for `@tgeotrans`
- `@tgf` for `@tglobal_function`
- `@tmo` for `@tmesher_object`
- `@tmls` for `@tmesh_levelset`
- `@tmim` for `@tmesh_im`
- `@tls` for `@tlevelset`
- `@tsl` for `@tslice`
- `@tsp` for `@tspmat`
- `@tpre` for `@tprecond`

Three dots at the end of the parameter list (`...`) mean that additional parameters are possible. Optional parameters can be described with brackets. For instance `/*@SET v = ('name'[, @int i])`. But be carreful 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 language (Matlab, Scilab or Python).

- Still in the documentations, parts for a specific language can be added by `@MATLAB{specific part ...}`, `@SCILAB{specific part ...}` and `@PYTHON{specific part ...}`. If a method/subcommand is specific to an interface, it can 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_*.

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 has 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.

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.

For the management of the object, you have to declare the class at the begining of `getfemint.h` (with respect to the alphabetic order), and declare three functions:

```
bool is_"name"_object(const mexarg_in &p);
id_type store_"name"_object(const std::shared_ptr<object_class> &shp);
object_class *to_"name"_object(const mexarg_in &p);
```

where "name" is the name of the object in the interface and `object_class` is the class name in getfem (for instance `getfem::mesh` for the mesh object). Alternatively, for the object that are manipulated by a shared pointer in *GetFEM++*, the third function can return a shared pointer.

IMPORTANT: In order to be interfaced, a *GetFEM++* object has to derive from `dal::static_stored_object`. However, if it is not the case, a wrapper class can be defined such as the one for `bgeot::base_poly` (see the end of `getfemint.h`).

The previous three functions have to be implemented at the end of `getfemint.cc`.It is possible to use one of the two macros defined in `getfemint.cc`. The firs macro is for a standard object and the second one for an object which is manipulated in *GetFEM++* with a shared pointer.

You have also to complete functions `name_of_getfemint_class_id` and `class_id_of_object` at the end of `getfemint.cc`.

You have to add the call of the interface function in `getfem_interface.cc` and modifiy the file `bin/extract_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 the object to be saved in a file and the second one is to give some information about the object. Additionaly, a constructor from a string is necessary to load the object from a file.

For the Scilab interface the file `sci_gateway/c/builder_gateway_c.sce.in` has to be modified and the files in the directory `macros/overload`.

State

Perspectives

The interface grows in conjunction with *GetFEM++*. The main *GetFEM++* functionalities are interfaced.

Appendix A. Some basic computations between reference and real elements

Volume integral

One has

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

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

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

one finally has

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

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

Surface integral

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

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

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

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

For n the unit normal to T on Γ.

Derivative computation

One has

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

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

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

the $P \times P$ matrix, then

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

and thus

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

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

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

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

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

one has

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

Example of elementary matrix

Assume one needs to compute the elementary "matrix":

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

The computations to be made on the reference elements are

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

and

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

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 ($\partial^2_{i_7/P,i_7}$ 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_3B_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.

CHAPTER 6

References

[AL-CU1991] P. Alart, A. Curnier. *A mixed formulation for frictional contact problems prone to newton like solution methods*. Comput. Methods Appl. Mech. Engrg. 92, 353–375, 1991.

[Al-Ge1997] E.L. Allgower and K. Georg. *Numerical Path Following*, Handbook of Numerical Analysis, Vol. V (P.G. Ciarlet and J.L. Lions, eds.). Elsevier, pp. 3-207, 1997.

[AM-MO-RE2014] S. Amdouni, M. Moakher, Y. Renard, *A local projection stabilization of fictitious domain method for elliptic boundary value problems*. Appl. Numer. Math., 76:60-75, 2014.

[AM-MO-RE2014b] S. Amdouni, M. Moakher, Y. Renard. *A stabilized Lagrange multiplier method for the enriched finite element approximation of Tresca contact problems of cracked elastic bodies*. Comput. Methods Appl. Mech. Engrg., 270:178-200, 2014.

[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.

[ba-dv1985] K.J. Bathe, E.N. Dvorkin, *A four-node plate bending element based on Mindlin-Reissner plate theory and a mixed interpolation*. Internat. J. Numer. Methods Engrg., 21, 367-383, 1985.

[Be-Mi-Mo-Bu2005] Bechet E, Minnebo H, Moës N, Burgardt B. *Improved implementation and robustness study of the X-FEM for stress analysis around cracks*. Internat. J. Numer. Methods Engrg., 64, 1033-1056, 2005.

[BE-CO-DU2010] M. Bergot, G. Cohen, M. Duruflé. *Higher-order finite elements for hybrid meshes using new nodal pyramidal elements* J. Sci. Comput., 42, 345-381, 2010.

[br-ba-fo1989] F. Brezzi, K.J. Bathe, M. Fortin. *Mixed-interpolated element for Reissner-Mindlin plates*. Internat. J. Numer. Methods Engrg., 28, 1787-1801, 1989.

[bu-ha2010] E. Burman, P. Hansbo. *Fictitious domain finite element methods using cut elements: I. A stabilized Lagrange multiplier method*. Computer Methods in Applied Mechanics, 199:41-44, 2680-2686, 2010.

[ca-re-so1994] D. Calvetti, L. Reichel and D.C. Sorensen. *An implicitely restarted Lanczos method for large symmetric eigenvalue problems*. Electronic Transaction on Numerical Analysis}. 2:1-21, 1994.

[CH-LA-RE2008] E. Chahine, P. Laborde, Y. Renard. *Crack-tip enrichment in the Xfem method using a cut-off function*. Int. J. Numer. Meth. Engng., 75(6):629-646, 2008.

[CH-LA-RE2011] E. Chahine, P. Laborde, Y. Renard. *A non-conformal eXtended Finite Element approach: Integral matching Xfem*. Applied Numerical Mathematics, 61:322-343, 2011.

[ciarlet1978] P.G. Ciarlet. *The finite element method for elliptic problems*. Studies in Mathematics and its Applications vol. 4, North-Holland, 1978.

[ciarlet1988] P.G. Ciarlet. *Mathematical Elasticity*. Volume 1: Three-Dimensional Elasticity. North-Holland, 1988.

[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, 141-164, 2003.

[Duan2014] H. Duan. *A finite element method for Reissner-Mindlin plates*. Math. Comp., 83:286, 701-733, 2014.

[Dr-La-Ek2014] A. Draganis, F. Larsson, A. Ekberg. *Finite element analysis of transient thermomechanical rolling contact using an efficient arbitrary Lagrangian-Eulerian description*. Comput. Mech., 54, 389-405, 2014.

[Fa-Po-Re2015] M. Fabre, J. Pousin, Y. Renard. *A fictitious domain method for frictionless contact problems in elasticity using Nitsche's method*. preprint, https://hal.archives-ouvertes.fr/hal-00960996v1

[Fa-Pa2003] F. Facchinei and J.-S. Pang. *Finite-Dimensional Variational Inequalities and Complementarity Problems, Vol. II*. Springer Series in Operations Research, Springer, New York, 2003.

[Georg2001] K. Georg. *Matrix-free numerical continuation and bifurcation*. Numer. Funct. Anal. Optimization 22, 303-320, 2001.

[GR-GH1999] R.D. Graglia, I.-L. Gheorma. *Higher order interpolatory vector bases on pyramidal elements* IEEE transactions on antennas and propagation, 47:5, 775-782, 1999.

[GR-ST2015] D. Grandi, U. Stefanelli. *The Souza-Auricchio model for shape-memory alloys* Discrete and Continuous Dynamical Systems, Series S, 8(4):723-747, 2015.

[HA-WO2009] C. Hager, B.I. Wohlmuth. *Nonlinear complementarity functions for plasticity problems with frictional contact*. Comput. Methods Appl. Mech. Engrg., 198:3411-3427, 2009

[HA-HA2004] A Hansbo, P Hansbo. *A finite element method for the simulation of strong and weak discontinuities in solid mechanics*. Comput. Methods Appl. Mech. Engrg. 193 (33-35), 3523-3540, 2004.

[HA-RE2009] J. Haslinger, Y. Renard. *A new fictitious domain approach inspired by the extended finite element method*. Siam J. on Numer. Anal., 47(2):1474-1499, 2009.

[HI-RE2010] Hild P., Renard Y. *Stabilized lagrange multiplier method for the finite element approximation of contact problems in elastostatics*. Numer. Math. 15:1, 101–129, 2010.

[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] N. Kikuchi, J.T. Oden. *Contact problems in elasticity*. SIAM, 1988.

[LA-PO-RE-SA2005] Laborde P., Pommier J., Renard Y., Salaun M. *High order extended finite element method for cracked domains*. Int. J. Numer. Meth. Engng., 64:354-381, 2005.

[LA-RE-SA2010] J. Lasry, Y. Renard, M. Salaun. *eXtended Finite Element Method for thin cracked plates with Kirchhoff-Love theory*. Int. J. Numer. Meth. Engng., 84(9):1115-1138, 2010.

[KO-RE2014] K. Poulios, Y. Renard, *An unconstrained integral approximation of large sliding frictional contact between deformable solids*. Computers and Structures, 153:75-90, 2015.

[LA-RE2006] P. Laborde, Y. Renard. *Fixed point strategies for elastostatic frictional contact problems*. Math. Meth. Appl. Sci., 31:415-441, 2008.

[Li-Re2014] T. Ligurský and Y. Renard. *A Continuation Problem for Computing Solutions of Discretised Evolution Problems with Application to Plane Quasi-Static Contact Problems with Friction*. Comput. Methods Appl. Mech. Engrg. 280, 222-262, 2014.

[Li-Re2014hal] T. Ligurský and Y. Renard. *Bifurcations in Piecewise-Smooth Steady-State Problems: Abstract Study and Application to Plane Contact Problems with Friction*. Computational Mechanics, 56:1:39-62, 2015.

[Li-Re2015hal] T. Ligurský and Y. Renard. *A Method of Piecewise-Smooth Numerical Branching*. Z. Angew. Math. Mech., 97:7:815–827, 2017.

[Mi-Zh2002] P. Ming and Z. Shi, *Optimal L2 error bounds for MITC3 type element*. Numer. Math. 91, 77-91, 2002.

[Xfem] N. Moës, J. Dolbow and T. Belytschko, *A finite element method for crack growth without remeshing*. Internat. J. Numer. Methods Engrg., 46, 131-150, 1999.

[Nackenhorst2004] U. Nackenhorst, *The ALE formulation of bodies in rolling contact. Theoretical foundation and finite element approach*. Comput. Methods Appl. Mech. Engrg., 193:4299-4322, 2004.

[nedelec1991] J.-C. Nedelec. *Notions sur les techniques d'elements finis*. Ellipses, SMAI, Mathematiques & Applications no 7, 1991.

[NI-RE-CH2011] S. Nicaise, Y. Renard, E. Chahine, *Optimal convergence analysis for the eXtended Finite Element Method*. Int. J. Numer. Meth. Engng., 86:528-548, 2011.

[Pantz2008] O. Pantz *The Modeling of Deformable Bodies with Frictionless (Self-)Contacts*. Archive for Rational Mechanics and Analysis, Volume 188, Issue 2, pp 183-212, 2008.

[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.

[PO-NI2016] K. Poulios, C.F. Niordson, *Homogenization of long fiber reinforced composites including fiber bending effects*. Journal of the Mechanics and Physics of Solids, 94, pp 433-452, 2016.

[remacle2002] J-F. Remacle, M. Shephard, *An algorithm oriented database*. Internat. J. Numer. Methods Engrg., 58, 349-374, 2003.

[SE-PO-WO2015] A. Seitz, A. Popp, W.A. Wall, *A semi-smooth Newton method for orthotropic plasticity and frictional contact at finite strains*. Comput. Methods Appl. Mech. Engrg. 285:228-254, 2015.

[SI-HU1998] J.C. Simo, T.J.R. Hughes. *Computational Inelasticity*. Interdisciplinary Applied Mathematics, vol 7, Springer, New York 1998.

[so-se-do2004] P. Šolín, K. Segeth, I. Doležel, *Higher-Order Finite Element Methods*. Chapman and Hall/CRC, Studies in advanced mathematics, 2004.

[SO-PE-OW2008] E.A. de Souza Neto, D Perić, D.R.J. Owen. *Computational methods for plasticity*. J. Wiley & Sons, New York, 2008.

[renard2013] Y. Renard, *Generalized Newton's methods for the approximation and resolution of frictional contact problems in elasticity*. Comput. Methods Appl. Mech. Engrg., 256:38-55, 2013.

[SU-CH-MO-BE2001] Sukumar N., Chopp D.L., Moës N., Belytschko T. *Modeling holes and inclusions by level sets in the extended finite-element method*. Comput. Methods Appl. Mech. Engrg., 190:46-47, 2001.

[ZT1989] Zienkiewicz and Taylor. *The finite element method*. 5th edition, volume 3 : Fluids Dynamics.

A

B

C

T