User Documentation

Release 5.3

Yves Renard, Julien Pommier, Konstantinos Poulilos

Dec 01, 2019
10.3 Some basic examples ........................................... 38
10.4 Derivation order and symbolic differentiation ................. 39
10.5 C++ Call of the assembly .................................... 39
10.6 C++ assembly examples ....................................... 41
10.7 Script languages call of the assembly .......................... 43
10.8 The tensors ....................................................... 44
10.9 The variables ..................................................... 44
10.10 The constants or data ......................................... 44
10.11 Test functions ................................................... 44
10.12 Gradient ........................................................ 45
10.13 Hessian .......................................................... 45
10.14 Predefined scalar functions ................................... 45
10.15 User defined scalar functions ................................ 46
10.16 Derivatives of defined scalar functions ....................... 46
10.17 Binary operations .............................................. 46
10.18 Unary operators ................................................ 47
10.19 Parentheses ..................................................... 47
10.20 Explicit vectors ............................................... 47
10.21 Explicit matrices .............................................. 48
10.22 Explicit tensors ................................................. 48
10.23 Access to tensor components .................................. 48
10.24 Constant expressions ......................................... 48
10.25 Special expressions linked to the current position ........... 49
10.26 Print command ................................................ 49
10.27 Reshape a tensor .............................................. 49
10.28 Trace, Deviator, Sym and Skew operators ...................... 49
10.29 Nonlinear operators .......................................... 50
10.30 Macro definition .............................................. 51
10.31 Explicit Differentiation ....................................... 52
10.32 Explicit Gradient ............................................. 52
10.33 Interpolate transformations ................................... 53
10.34 Element extrapolation transformation ......................... 54
10.35 Evaluating discontinuities across inter-element edges/faces . 55
10.36 Double domain integrals or terms (convolution - Kernel - Exchange integrals) 56
10.37 Elementary transformations .................................. 57
10.38 Xfem discontinuity evaluation (with mesh_fem_level_set) .... 58
10.39 Storage of sub-expressions in a getfem::im_data object during assembly ............ 59

11 Compute arbitrary terms - low-level generic assembly procedures 61
   11.1 available operations inside the comp command ............. 63
   11.2 others operations ........................................... 64

12 Some Standard assembly procedures (low-level generic assembly) 65
   12.1 Laplacian (Poisson) problem ................................ 65
   12.2 Linear Elasticity problem ................................... 67
   12.3 Stokes Problem with mixed finite element method .......... 68
   12.4 Assembling a mass matrix ................................... 68

13 Interpolation of arbitrary quantities 69
   13.1 Basic interpolation ........................................... 69
   13.2 Interpolation based on the high-level weak form language .... 70
14 Incorporate new finite element methods in GetFEM++ 73

15 Incorporate new approximated integration methods in GetFEM++ 75

16 Level-sets, Xfem, fictitious domains, Cut-fem 77
   16.1 Representation of level-sets 78
   16.2 Mesh cut by level-sets 78
   16.3 Adapted integration methods 79
   16.4 Cut-fem 80
   16.5 Discontinuous field across some level-sets 80
   16.6 Xfem 80
   16.7 Post treatment 81

17 Tools for HHO (Hybrid High-Order) methods 83
   17.1 HHO elements 83
   17.2 Reconstruction operators 84
   17.3 Stabilization operators 86

18 Interpolation/projection of a finite element method on non-matching meshes 89
   18.1 mixed methods with different meshes 90
   18.2 mortar methods 90

19 Compute $L^2$ and $H^1$ norms 91

20 Compute derivatives 93

21 Export and view a solution 95
   21.1 Saving mesh and mesh_fem objects for the Matlab interface 95
   21.2 Producing mesh slices 96
   21.3 Exporting mesh, mesh_fem or slices to VTK 98
   21.4 Exporting mesh, mesh_fem or slices to OpenDX 98

22 A pure convection method 101

23 The model description and basic model bricks 103
   23.1 The model object 103
   23.2 The brick object 106
   23.3 How to build a new brick 106
   23.4 How to add the brick to a model 110
   23.5 Generic assembly bricks 112
   23.6 Generic elliptic brick 113
   23.7 Dirichlet condition brick 114
   23.8 Generalized Dirichlet condition brick 115
   23.9 Pointwise constraints brick 116
   23.10 Source term bricks (and Neumann condition) 117
   23.11 Predefined solvers 117
   23.12 Example of a complete Poisson problem 118
   23.13 Nitsche’s method for dirichlet and contact boundary conditions 120
   23.14 Constraint brick 123
   23.15 Other “explicit” bricks 124
   23.16 Helmholtz brick 124
   23.17 Fourier-Robin brick 124
   23.18 Isotropic linearized elasticity brick 125
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.19</td>
<td>Linear incompressibility (or nearly incompressibility) brick</td>
<td>126</td>
<td></td>
</tr>
<tr>
<td>23.20</td>
<td>Mass brick</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>23.21</td>
<td>Bilaplacian and Kirchhoff-Love plate bricks</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>23.22</td>
<td>Mindlin-Reissner plate model</td>
<td>129</td>
<td></td>
</tr>
<tr>
<td>23.23</td>
<td>The model tools for the integration of transient problems</td>
<td>131</td>
<td></td>
</tr>
<tr>
<td>23.24</td>
<td>Small sliding contact with friction bricks</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>23.25</td>
<td>Large sliding/large deformation contact with friction bricks</td>
<td>151</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Numerical continuation and bifurcation</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>24.1</td>
<td>Numerical continuation</td>
<td>161</td>
<td></td>
</tr>
<tr>
<td>24.2</td>
<td>Detection of limit points</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>24.3</td>
<td>Numerical bifurcation</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>24.4</td>
<td>Approximation of solution curves of a model</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Finite strain Elasticity bricks</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>25.1</td>
<td>Some recalls on finite strain elasticity</td>
<td>169</td>
<td></td>
</tr>
<tr>
<td>25.2</td>
<td>Add an nonlinear elasticity brick to a model</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>25.3</td>
<td>Add a large strain incompressibility brick to a model</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>25.4</td>
<td>High-level generic assembly versions</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Small strain plasticity</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>26.1</td>
<td>Theoretical background</td>
<td>179</td>
<td></td>
</tr>
<tr>
<td>26.2</td>
<td>Flow rule integration</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>26.3</td>
<td>Some classical laws</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>26.4</td>
<td>Elasto-plasticity bricks</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>ALE Support for object having a large rigid body motion</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>27.1</td>
<td>ALE terms for rotating objects</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>27.2</td>
<td>ALE terms for a uniformly translated part of an object</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Appendix A. Finite element method list</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>28.1</td>
<td>Classical $P_K$ Lagrange elements on simplices</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>28.2</td>
<td>Classical Lagrange elements on other geometries</td>
<td>205</td>
<td></td>
</tr>
<tr>
<td>28.3</td>
<td>Elements with hierarchical basis</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>28.4</td>
<td>Classical vector elements</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>28.5</td>
<td>Specific elements in dimension 1</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>28.6</td>
<td>Specific elements in dimension 2</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>28.7</td>
<td>Specific elements in dimension 3</td>
<td>230</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Appendix B. Cubature method list</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>29.1</td>
<td>Exact Integration methods</td>
<td>237</td>
<td></td>
</tr>
<tr>
<td>29.2</td>
<td>Newton cotes Integration methods</td>
<td>238</td>
<td></td>
</tr>
<tr>
<td>29.3</td>
<td>Gauss Integration methods on dimension 1</td>
<td>238</td>
<td></td>
</tr>
<tr>
<td>29.4</td>
<td>Gauss Integration methods on dimension 2</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>29.5</td>
<td>Gauss Integration methods on dimension 3</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>29.6</td>
<td>Direct product of integration methods</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>29.7</td>
<td>Specific integration methods</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>29.8</td>
<td>Composite integration methods</td>
<td>245</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>References</td>
<td>247</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>249</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

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

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

Examples of available finite element method are: $P_k$ 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, etc.

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

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

*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.

*GetFEM++* has only a (very) experimental meshing procedure (and produces regular meshes), hence it is generally necessary to import meshes. Imports formats currently known by *GetFEM++* are GiD, Gmsh and emc2 mesh files. However, given a mesh, it is possible to refine it automatically.

Copyright © 2004-2018 *GetFEM++* project.

The text of the *GetFEM++* website and the documentations are available for modification and reuse under the terms of the GNU Free Documentation License
GetFEM++ is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version along with the GCC Runtime Library Exception either version 3.1 or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License and GCC Runtime Library Exception for more details. You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.
CHAPTER 2

How to install

Since we use standard GNU tools, the installation of the GetFEM++ library is somewhat standard. See the download and install page for more details for the installations on the different plateforms.
CHAPTER 3

Linear algebra procedures

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

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

A small interface to MUMPS is also provided (see MUMPS web1 or MUMPS web2). See the file gmm/gmm_MUMPS_interface.h. In order to use MUMPS, you have to indicates some options to the configure shell:

```
--with-mumps-include-dir=" -I /path/to/MUMPS/include "
--with-mumps=" F90 libraries and libs of MUMPS to be linked "
```

alternatively, the option --enable-mumps will search for an installed MUMPS library. Note that if both the sequential and the parallel version is installed on your system (especially on Debian and Ubuntu), the default version will be the parallel one. To select the sequential one it is necessary to add the option --with-mumps="-lsmumps_seq -ldmumps_seq -lcumps_seq -lzmumps_seq".

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

```
--with-mumps-include-dir=" -I /path/to/MUMPS/include "
--with-mumps=" ...F90libs... -L /path/to/MUMPS/lib -ldmumps -lzmumps -lpord
-L /path/to/MUMPS/libseq -lmpiseq "
```

where ...F90libs... are the libraries of the fortran compiler used to compile MUMPS (these are highly dependant on the fortran 90 compiler used, the ./configure script should detect the options relative to the default fortran 90 compiler on your machine and display it – for example, with the intel ifort compiler, it is -L/opt/icc8.0/lib -lifport -lifcoremt -limf -lm -lcxa -lunwind -lpthread)
Of course, each different problem should require a different parallelization adapted to its specificities in order to obtain a good load balancing. You may build your own parallelization using the mesh regions to parallelize assembly procedures.

Nevertheless, the brick system offers a generic parallelization based on MPI (communication between processes), METIS (partition of the mesh) and MUMPS (parallel sparse direct solver). It is available with the compiler option \texttt{-D GETFEM\_PARA\_LEVEL=2} and the library itself has to be compiled with the option \texttt{--enable-paralevel=2} of the configure script. Initial MPI parallelization of \textit{GetFEM++} has been designed with the help of Nicolas Renon from CALMIP, Toulouse.

When the configure script is run with the option \texttt{--enable-paralevel=2}, it searches for MPI, METIS and parallel MUMPS libraries. If the python interface is built, it searches also for MPI4PY library. In that case, the python interface can be used to drive the parallel version of getfem (the other interfaces has not been parallelized for the moment). See \texttt{demo\_parallel\_laplacian.py} in the interface/test/python directory.

With the option \texttt{-D GETFEM\_PARA\_LEVEL=2}, each mesh used is implicitly partitioned (using METIS) into a number of regions corresponding to the number of processors and the assembly procedures are parallelized. This means that the tangent matrix and the constraint matrix assembled in the \texttt{model\_state} variable are distributed. The choice made (for the moment) is not to distribute the vectors. So that the right hand side vectors in the \texttt{model\_state} variable are communicated to each processor (the sum of each contribution is made at the end of the assembly and each processor has the complete vector). Note that you have to think to the fact that the matrices stored by the bricks are all distributed.

A model of C++ parallelized program is \texttt{tests/elastostatic.cc}. To run it in parallel you have to launch for instance:

\begin{verbatim}
mpirun -n 4 elastostatic elastostatic.param
\end{verbatim}

For a python interfaced program, the call reads:

\begin{verbatim}
mpirun -n 4 python demo_parallel_laplacian.py
\end{verbatim}
If you do not perform a make install, do not forget to first set the shell variable PYTHONPATH to the python-getfem library with for instance:

\texttt{export PYTHONPATH=my_getfem_directory/interface/src/python}

\section{State of progress of GetFEM++ MPI parallelization}

Parallelization of getfem is still considered a “work in progress”. A certain number of procedure are still remaining sequential. Of course, a good test to see if the parallelization of your program is correct is to verify that the result of the computation is indeed independent of the number of process.

\begin{itemize}
  \item \textbf{Assembly procedures}
    
    Most of assembly procedures (in getfem/getfem_assembling.h) have a parameter corresponding to the region in which the assembly is to be computed. They are not parallelized themselves but aimed to be called with a different region in each process to distribute the job. Note that the file getfem/getfem_config.h contains a procedures called MPI\_SUM\_SPARSE\_MATRIX allowing to gather the contributions of a distributed sparse matrix.
    
    The following assembly procedures are implicitly parallelized using the option \texttt{-D GETFEM\_PARA\_LEVEL=2}:
    
    \begin{itemize}
      \item computation of norms (asm\_L2\_norm, asm\_H1\_norm, asm\_H2\_norm ..., in getfem/getfem_assembling.h),
      \item asm\_mean\_value (in getfem/getfem_assembling.h),
      \item error\_estimate (in getfem/getfem_error\_estimate.h).
    \end{itemize}
    
    This means in particular that these functions have to be called on each processor.
  \item \textbf{Mesh\_fem object}
    
    The dof numbering of the getfem::mesh\_fem object remains sequential and is executed on each process. The parallelization is to be done. This could affect the efficiency of the parallelization for very large and/or evolving meshes.
  \item \textbf{Model object and bricks}
    
    The model system is globally parallelized, which mainly means that the assembly procedures of standard bricks use a METIS partition of the meshes to distribute the assembly. The tangent/stiffness matrices remain distributed and the standard solve call the parallel version of MUMPS (which accept distributed matrices).
    
    For the moment, the procedure actualize\_sizes() of the model object remains sequential and is executed on each process. The parallelization is to be done.
    
    Some specificities:
    
    \begin{itemize}
      \item The explicit matrix brick: the given matrix is considered to be distributed. If it is not, only add it on the master process (otherwise, the contribution will be multiplied by the number of processes).
      \item The explicit rhs brick: the given vector is not considered to be distributed. Only the given vector on the master process is taken into account.
    \end{itemize}
\end{itemize}
– Constraint brick: The given matrix and rhs are not considered to be distributed. Only the given matrix and vector on the master process are taken into account.

– Concerning contact bricks, only integral contact bricks are fully parallelized for the moment. Nodal contact bricks work in parallel but all the computation is done on the master process.
Errors used in GetFEM++ are defined in the file gmm/gmm_except.h. In order to make easier the error catching all errors derive from the type `std::logic_error` defined in the file `stdexcept` of the S.T.L.

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

```c
int main(void) {
    try {
        ... main program ...
    } GMM_STANDARD_CATCH_ERROR;
}
```
CHAPTER 6

Build a mesh

As a preliminary, you may want to read this short introduction to the GetFEM++ vocabulary.

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

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

There is only a (very) experimental meshing procedure in GetFEM++ to mesh complex geometries. But you can easily load a mesh from any format (some procedures are in getfem/getfem_import.h to load meshes from some public domain mesh generators).

The structure getfem::mesh may also contain a description about a region of the mesh, such as a boundary or a set of elements. This is handled via a container of convexes and convex faces, getfem::mesh_region. We refer to [remacle2002] for a discussion on mesh representation.

6.1 Add an element to a mesh

Suppose the variable mymesh has been declared by:

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

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

To enter a new point on a mesh use the method:

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

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

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

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

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

then adding the element is done by:

```cpp
mymesh.add_convex(bgeot::simplex_geotrans(2,1), ind.begin());
```

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

For simplices, a more specialized function exists, which is:

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

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

```cpp
mymesh.add_convex_by_points(pgt, itp);
```

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

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

It is possible to use also:

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

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

- `bgeot::parallelepiped_geotrans(N, 1)` describes the usual transformation for parallelepipeds of dimension `N` (quadrilateron for `N=2`, hexahedron for `N=3`,...)
- `bgeot::prism_geotrans(N, 1)` describes the usual transformation for prisms of dimension `N` (usual prism is for `N=3`. A generalized prism is the product of a simplex of dimension `N-1` with a segment)

Specialized functions exist also:

```cpp
mymesh.add_parallelepiped(N, it);
mymesh.add_parallelepiped_by_points(N, itp);
mymesh.add_prism(N, it);
mymesh.add_prism_by_points(N, itp);
```
The order of the points in the array of points is not important for simplices (except if you care about the orientation of your simplices). For other elements, it is important to respect the vertex order shown in **Vertex numeration for usual first order elements** (first order elements).

![Vertex numeration for usual first order elements](image)

Fig. 1: Vertex numeration for usual first order elements

Note that a general rule, including for higher order transformations, is that the vertex numeration follows the one of the corresponding Lagrange finite element method (see Appendix A. Finite element method list).

### 6.2 Remove an element from a mesh

To remove an element from a mesh, simply use:

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

where `i` is the index of the element.

### 6.3 Simple structured meshes

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

The simplest function to use is:

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

which fills the mesh `m` with a regular mesh of simplices/parallelepipeds/prisms (depending on the value of `pgt`). The number of cells in each direction is given by `nsubdiv`. The following example builds a mesh of quadratic triangles on the unit square (the mesh can be scaled and translated afterwards):
std::vector<getfem::size_type> nsubdiv(2);
nsubdiv[0] = 10; nsubdiv[1] = 20;
regular_unit_mesh(m, nsubdiv, bgeot::simplex_geotrans(2,2));

More specialized regular mesh functions are also available:

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

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

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

getfem::mesh mymesh;
bgeot::base_node org(0.0, 0.0, 0.0);
std::vector<bgeot::base_small_vector> vect(3);
vect[0] = bgeot::base_small_vector(0.1, 0.0, 0.0);
vect[1] = bgeot::base_small_vector(0.0, 0.1, 0.0);
vect[2] = bgeot::base_small_vector(0.0, 0.0, 0.1);
std::vector<int> ref(3);
getfem::parallelepiped_regular_simplex_mesh(mymesh, 3, org, vect.begin(), ref.begin());

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

### 6.4 Mesh regions

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

mymesh.region(30).add(2); // adds convex 2 into region 30
mymesh.region(30).add(3); // adds convex 3 into region 30
mymesh.region(30).add(4,3); // adds face 3 of convex 4 into region 30
mymesh.region(30).sup(3); // Removes convex 3 from region 30
mymesh.sup_convex(4); // Removes convex 4 from both the mesh and all
                       // the regions
for (getfem::mr_visitor i(mymesh.region(30)); !i.finished(); ++i) {
  cout << "convex: " << i.cv() << " face:" << i.f() << endl;
}
6.5 Methods of the `getfem::mesh` object

The list is not exhaustive.

- `mymesh.dim()`
  main dimension of the mesh.

- `mymesh.points_index()`
  gives a `dal::bit_vector` object which represents all the indexes of valid points of a mesh (see below).

- `mymesh.points()[i]`
  gives the point of index `i` (`bgeot::base_node`).

- `mymesh.convex_index()`
  gives a `dal::bit_vector` object which represents all the indexes of valid elements of a mesh (see below).

- `mymesh.structure_of_convex(i)`
  gives the description of the structure of element of index `i`. The function return a `bgeot::pconvex_structure`.

- `mymesh.structure_of_convex(i)->nb_faces()`
  number of faces of element of index `i`.

- `mymesh.structure_of_convex(i)->nb_points()`
  number of vertices of element of index `i`.

- `mymesh.structure_of_convex(i)->dim()`
  intrinsic dimension of element of index `i`.

- `mymesh.structure_of_convex(i)->nb_points_of_face(f)`
  number of vertices of the face of local index `f` of element of index `i`.

- `mymesh.structure_of_convex(i)->ind_points_of_face(f)`
  return a container with the local indexes of all vertices of the face of local index `f` of element of index `i`. For instance `mesh.structure_of_convex(i)->ind_points_of_face(f)[0]` is the local index of the first vertex.

- `mymesh.structure_of_convex(i)->face_structure(f)`
  gives the structure (`bgeot::pconvex_structure`) of local index `f` of element of index `i`.

- `mymesh.ind_points_of_convex(i)`
  gives a container with the global indexes of vertices of element of index `i`.

- `mymesh.points_of_convex(i)`
  gives a container with the vertices of element of index `i`. This is an array of `bgeot::base_node`.

- `mymesh.convex_to_point(ipt)`
  gives a container with the indexes of all elements attached to the point of global index `ipt`.

- `mymesh.neighbours_of_convex(ic,f)`
  gives a container with the indexes of all elements in `mesh` having the common face of local index `f` of element `ic` except element `ic`. 
mymesh.neighbour_of_convex(ic, f)
gives the index of the first elements in mesh having the common face of local index \( f \) of element ic except element ic. return size_type(-1) if none is found.

mymesh.is_convex_having_neighbour(ic, f)
return whether or not the element ic has a neighbour with respect to its face of local index \( f \).

mymesh.clear()
delete all elements and points from the mesh.

mymesh.optimize_structure()
compact the structure (renumbers points and convexes such that there is no hole in their numbering).

mymesh.trans_of_convex(i)
return the geometric transformation of the element of index \( i \) (in a bgeot::pgeometric_trans). See dp for more details about geometric transformations.

mymesh.normal_of_face_of_convex(ic, f, pt)
gives a bgeot::base_small_vector representing an outward normal to the element at the face of local index \( f \) at the point of local coordinates (coordinates in the element of reference) pt. The point pt has no influence if the geometric transformation is linear. This is not a unit normal, the norm of the resulting vector is the ratio between the surface of the face of the reference element and the surface of the face of the real element.

mymesh.convex_area_estimate(ic)
gives an estimate of the area of convex ic.

mymesh.convex_quality_estimate(ic)
gives a rough estimate of the quality of element ic.

mymesh.convex_radius_estimate(ic)
gives an estimate of the radius of element ic.

mymesh.region(irg)
return a getfem::mesh_region. The region is stored in the mesh, and can contain a set of convex numbers and or convex faces.

mymesh.has_region(irg)
returns true if the region of index irg has been created.

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

add(ic)
add the convex of index ic to the region.

add(ic, f)
add the face number f of the convex ic.

sup(ic)
sup(ic, f)
remove the convex or the convex face from the region.

is_in(ic)
is_in(ic, f)
return true if the convex (or convex face) is in the region.
is_only_faces()
return true if the region does not contain any convex.

is_only_convexes()
return true if the region does not contain any convex face.

index()
return a dal::bit_vector containing the list of convexes which are stored (or whose faces are stored) in the region.

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

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

## 6.6 Using dal::bit_vector

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

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

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

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

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

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

## 6.7 Face numbering

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

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

6.8.1 From GetFEM++ file format

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

```cpp
mymesh.write_to_file(const std::string &name)
```

save the mesh into a file.

```cpp
mymesh.read_from_file(const std::string &name)
```

load the mesh from a file.

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

```cpp
#include <getfem/getfem_mesh.h>

getfem::mesh mymesh;

int main(int argc, char *argv[]) {
  try {
    // read the mesh from the file name given by the first argument
    mymesh.read_from_file(std::string(argv[1]));

    // List all the convexes
    dal::bit_vector nn = mymesh.convex_index();
    bgeot::size_type i;
    for (i << nn; i != bgeot::size_type(-1); i << nn) {
      cout << "Convex of index " << i << endl;
    }
  } catch (const getfem::error &e) {
    std::cerr << e.what() << std::endl;
  }
}
```

(continues on next page)
bgeot::pconvex_structure cvs = mymesh.structure_of_convex(i);
cout << "Number of vertices: " << cvs->nb_points() << endl;
cout << "Number of faces: " << cvs->nb_faces() << endl;
for (bgeot::short_type f = 0; f < cvs->nb_faces(); ++f) {
cout << "face " << f << " has " << cvs->nb_points_of_face(f);
cout << " vertices with local indexes: ";
for (bgeot::size_type k = 0; k < cvs->nb_points_of_face(f); ++k)
cout << cvs->ind_points_of_face(f)[k] << " ";
cout << " and global indexes: ";
for (bgeot::size_type k = 0; k < mymesh.ind_points_of_convex(i)[cvs->ind_points_of_face(f)[k]]; ++k)
cout << mymesh.ind_points_of_convex(i)[cvs->ind_points_of_face(f)[k]] << " ";
}
} GMM_STANDARD_CATCH_ERROR; // catches standard errors

6.8.2 Import a mesh

The file getfem/getfem_import.h provides the function:

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

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

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

Alternatively the function:

```cpp
void import_mesh(const std::string& filename, const std::string& fmt, mesh& m);
```

can be used in an equivalent manner with the string `fmt` being one of the aforementioned format specifiers.

The “gid” format specifier is for meshes generated by GiD and “gmsh” is for meshes generated by the open-source mesh generator Gmsh. The “cdb” format specifier is for reading meshes from ANSYS models exported in blocked format with the CDWRITE command. Currently the ANSYS element types 42,45,73,82,87,89,90,92,95,162,182,183,185,186,187 and 191 can be imported, this however does not include any finite element technology linked to these elements but only their geometry. The “noboite” format is for TetMesh-GHS3D, and the “am_fmt” and “emc2_mesh” are for files built with EMC2 (but 2D only).

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

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

(continues on next page)
where \( GT \) is the name of the geometric transformation, \( \text{NSUBDIV} \) a vector of the number of subdivisions in each coordinate (default value 2), \( \text{ORG} \) is the origin of the mesh (default value \([0, 0, \ldots]\)), \( \text{SIZES} \) is a vector of the sizes in each direction (default value \([1, 1, \ldots]\)) and if \( \text{NOISED}=1 \) the nodes of the interior of the mesh are randomly “shaken” (default value \( \text{NOISED}=0 \)). In that string, all the parameters are optional except \( GT \).
The object `getfem::mesh_fem` defined in `getfem/getfem_mesh_fem.h` is designed to describe a finite element method on a whole mesh, i.e. to describe the finite element space on which some variables will be described. This is a rather complex object which is central in GetFEM++. Basically, this structure describes the finite element method on each element of the mesh and some additional optional transformations. It is possible to have an arbitrary number of finite element descriptions for a single mesh. This is particularly necessary for mixed methods, but also to describe different data on the same mesh. One can instantiate a `getfem::mesh_fem` object as follows:

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

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

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

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

- The element level: one finite element method per element. It is possible to mix the dimensions of the elements and the property to be vectorial or scalar.

- The optional vectorization/tensorization (the qdim in getfem jargon, see vocabulary). For instance to represent a displacement or a tensor field in continuum mechanics. Scalar elements are used componentwise. Note that you can mix some intrinsic vectorial elements (Raviart-Thomas element for instance) which will not be vectorized and scalar elements which will be.

- (GetFEM++ version 4.0) The optional additional linear transformation (reduction) of the degrees of freedom. It will consist in giving two matrices, the reduction matrix and the extension matrix. The reduction matrix should transform the basic dofs into the reduced dofs (the number of reduced
dofs should be less or equal than the number of basic dofs). The extension matrix should describe
the inverse transformation. The product of the reduction matrix with the extension matrix should
be the identity matrix (ensuring in particular that the two matrices are of maximal rank). This
optional transformation can be used to reduce the finite element space to a certain region (tipically
a boundary) or to prescribe some matching conditions between non naturally compatible fems (for
instance fems with different degrees).

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

## 7.1 First level: manipulating fems on each elements

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

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

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

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

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

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

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

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

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

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

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

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

- "FEM_PK(n,k) ": Classical $P_K$ methods on simplexes of dimension $n$ with degree $k$ polyno-
mials.
- "FEM_QK(n,k) ": Classical $Q_K$ methods on parallelepiped of dimension $n$. Tensorial product
of degree $k$ $P_K$ method on the segment.
- "FEM_PK_PRISM(n,k) ": Classical methods on prism of dimension $n$. Tensorial product of
two degree $k$ $P_K$ method.
• "FEM_PRODUCT(a,b)": Tensorial product of the two polynomial finite element method a and b.

• "FEM_PK_DISCONTINUOUS(n,k)": discontinuous $P_K$ methods on simplexes of dimension $n$ with degree $k$ polynomials.

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

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

The `mesh_fem` can call directly these functions via:

```cpp
void mesh_fem::set_classical_finite_element(const dal::bit_vector &cvs,
                                           dim_type fem_degree);
void mesh_fem::set_classical_discontinuous_finite_element(const dal::bit_vector &cvs,
                                                           dim_type fem_degree);
void mesh_fem::set_classical_finite_element(dim_type fem_degree);
void mesh_fem::set_classical_discontinuous_finite_element(dim_type fem_degree);
```

Some other methods:

- `mf.convex_index()`
  Set of indexes (a `dal::bit_vector`) on which a finite element method is defined.

- `mf.linked_mesh()`
  gives a reference to the linked mesh.

- `mf.fem_of_element(i)`
  gives a descriptor on the finite element method defined on element of index $i$ (does not take into account the qdim nor the optional reduction).

- `mf.clear()`
  Clears the structure, no finite element method is still defined.

## 7.2 Examples

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

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

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

```cpp
mf.set_finite_element(mymesh.convex_index(),
                      getfem::fem_descriptor("FEM_PK(2, 1)"));
```
selects the method on all the elements of the mesh.

7.3 Second level: the optional “vectorization/tensorization”

If the finite element represents an unknown which is a vector field, the method `mf.set_qdim(Q)` allows set the target dimension for the definition of the target dimension $Q$.

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

- if the fem of the $ith$ element is intrinsically a vector FEM, then:

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

- if the fem has a `target_dim` equal to 1, then:

```
    mf.nb_dof_of_element(i) == mf.get_qdim()*mf.fem_of_element(i).nb_dof()
```

Additionally, if the field to be represented is a tensor field instead of a vector field (for instance the stress or strain tensor field in elasticity), it is possible to specify the tensor dimensions with the methods:

```
    mf.set_qdim(dim_type M, dim_type N)
    mf.set_qdim(dim_type M, dim_type N, dim_type O, dim_type P)
    mf.set_qdim(const bgeot::multi_index &mii)
```

respectively for a tensor field of order two, four and arbitrary (but limited to 6). For most of the operations, this is equivalent to declare a vector field of the size the product of the dimensions. However, the declared tensor dimensions are taken into account into the high level generic assembly. Remember that the components inside a tensor are stored in Fortran order.

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

- `mf.nb_basic_dof_of_element(i)` gives the number of basic degrees of freedom on the element of index $i$.
- `mf.ind_basic_dof_of_element(i)` gives a container (an array) with all the global indexes of the basic degrees of freedom of element of index $i$.
- `mf.point_of_basic_dof(i,j)` gives a `bgeot::base_node` which represents the point associated with the basic dof of local index $j$ on element of index $i$.
- `mf.point_of_basic_dof(j)` gives a `bgeot::base_node` which represents the point associated with the basic dof of global index $j$.
- `mf.reference_point_of_basic_dof(i,j)` gives a `bgeot::base_node` which represents the point associated with the basic dof of local index $j$ on element of index $i$ in the coordinates of the reference element.
mf. first_convex_of_basic_dof(j)
gives the index of the first element on which the basic degree of freedom of global index j is defined.

mf. nb_basic_dof()
gives the total number of different basic degrees of freedom.

mf. get_qdim()
gives the target dimension Q.

mf. basic_dof_on_region(i)
Return a dal::bit_vector which represents the indices of basic dof which are in the set of convexes or the set of faces of index i (see the getfem::mesh object).

mf. dof_on_region(i)
Return a dal::bit_vector which represents the indices of dof which are in the set of convexes or the set of faces of index i (see the getfem::mesh object). For a reduced mesh_fem, a dof is lying on a region if its potential corresponding shape function is nonzero on this region. The extension matrix is used to make the correspondence between basic and reduced dofs.

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

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

A natural condition is that \( RE = I \) where \( I \) is the identity matrix.

mf. nb_dof()
gives the total number of different degrees of freedom. If the optional reduction is used, this will be the number of columns of the reduction matrix. Otherwise it will return the number of basic degrees of freedom.

mf. is_reduced()
return a boolean. True if the reduction is used.

mf. reduction_matrix()
return a const reference to the reduction matrix \( R \).

mf. extension_matrix()
return a const reference to the extension matrix \( E \).

mf. set_reduction_matrices(R, E)
Set the reduction and extension matrices to \( R \) and \( E \) and validate their use.

mf. set_reduction(b)
Where \( b \) is a boolean. Cancel the reduction if \( b \) is false and validate it if \( b \) is true. If \( b \) is true, the extension and reduction matrices have to be set previously.

mf. reduce_to_basic_dof(idof)
Set the reduction and extension matrices corresponding to keep only the basic dofs present in idof. The parameter idof is either a dal::bit_vector or a std::set<size_type>. This is equivalent to the use of a getfem::partial_mesh_fem object.
7.5 Obtaining generic mesh_fem's

It is possible to use the function:

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

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

7.6 The partial_mesh_fem object

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

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

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

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

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

where kept_dof and rejected_elt are some dal::bit_vector. kept_dof is the list of dof indices of the original mesh_fem mf to be kept. rejected_elt is an optional parameter that contains a list of element indices on which the getfem::partial_mesh_fem states that there is no finite element method. This is to avoid unnecessary computations during assembly procedures.
The description of an integration method on a whole mesh is done thanks to the structure `getfem::mesh_im`, defined in the file `getfem/getfem_mesh_im.h`. Basically, this structure describes the integration method on each element of the mesh. One can instantiate a `getfem::mesh_im` object as follows:

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

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

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

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

```cpp
mim.set_integration_method(i, ppi);
```

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

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

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

The list of all available descriptors of integration methods is in the file `getfem/getfem_integration.h`. Descriptors for integration methods are available thanks to the following function:

```cpp
getfem::pintegration_method ppi = getfem::int_method_descriptor("name of method");
```
where "name of method" is to be chosen among the existing methods. A name of a method can be retrieved with:

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

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

Examples of exact integration methods:

- "IM_NONE ()": Dummy integration method (new in getfem++-1.7).
- "IM_EXACT_SIMPLEX(n)": Description of the exact integration of polynomials on the simplex of reference of dimension n.
- "IM_PRODUCT(a, b)": Description of the exact integration on the convex which is the direct product of the convex in a and in b.
- "IM_EXACT_PARALLELEPIPED(n)": Description of the exact integration of polynomials on the parallelepiped of reference of dimension n
- "IM_EXACT_PRISM(n)": Description of the exact integration of polynomials on the prism of reference of dimension n

Examples of approximated integration methods:

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

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

An alternative way to obtain integration methods:

```cpp
getfem::pintegration_method ppi =
    getfem::classical_exact_im(bgeot::pgemetric_trans pgt);
```
These functions return an exact (i.e. analytical) integration method, or select an approximate integration method which is able to integrate exactly polynomials of degree \( \leq d \) (at least) for convexes defined with the specified geometric transformation.

### 8.1 Methods of the `mesh_im` object

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

- **mim.convex_index()**
  - Set of indexes (`dal::bit_vector`) on which an integration method is defined.

- **mim.linked_mesh()**
  - Gives a reference to the linked mesh.

- **mim.int_method_of_element(i)**
  - Gives a descriptor on the integration method defined on element of index \( i \).

- **mim.clear()**
  - Clear the structure. There are no further integration method defined on the mesh.
Mesh refinement with the Bank et all method (see [bank1983]) is available in dimension 1, 2 or 3 for simplex meshes (segments, triangles and tetrahedrons). For a given object `mymesh` of type `getfem::mesh`, the method:

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

refines the elements whose indices are stored in `bv` (a `dal::bit_vector` object). The conformity of the mesh is kept thanks to additional refinement (the so called green triangles). Information about green triangles (in Figure Example of Bank refinement in 2D) is stored on the mesh object to gather them for further refinements (see [bank1983]).

![Example of Bank refinement in 2D](image)

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

```cpp
error_estimate(mim, mf, U, err, rg);
```

where `mim` is the integration method (a `getfem::mesh_im` object), `mf` is the finite element method on which the unknown has been computed (a `getfem::mesh_fem` object), `U` is the vector of degrees of freedom of the unknown, `err` is a sufficiently large vector in which the error estimate is computed for each element of the mesh, and `rg` is a mesh region built from elements on which the error estimate should be computed (a `getfem::mesh_region` object).

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

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

where $\| \partial_n u \|$ is the jump of the normal derivative. Then, the error estimate for a given element is the sum of the computed quantities on each internal face multiplied by the element diameter. This basic error estimate can be taken as a model for more elaborated ones. It uses the high-level generic assembly and the `neighbour_elt` interpolate transformation (see Evaluating discontinuities across inter-element edges/faces).
CHAPTER 10

Compute arbitrary terms - high-level generic assembly procedures - Weak-form language

This section presents what is now the main generic assembly of GetFEM++. It is a high-level generic assembly in the sense that it is based on a weak form language to describe the weak formulation of boundary value problems of partial differential equations. It mainly has been developed to circumvent the difficulties with the previous low-level generic assembly (see Compute arbitrary terms - low-level generic assembly procedures) for which nonlinear terms were quite difficult to describe. Conversely, a symbolic differentiation algorithm is used with this version. It simplifies a lot the approximation of nonlinear coupled problems since only the weak form is necessary to be described, the tangent system being automatically computed. Moreover, the weak form language is compiled into optimized instructions before the evaluation on each integration point in order to obtain an optimal computational cost.

The header file to be included to use the high-level generic assembly procedures in C++ is getfem/generic_assembly.h.

10.1 Differences in execution time between high and low level generic assembly

For basic linear assembly terms, the high level generic assembly is most of the time faster than the low-level one. This is due to the fact that the high-level generic assembly incorporates a compilation in basic optimized instructions and operates simplifications. On complexe terms it can be really faster due to the simplifications on repeated terms. On the other hand, the fact that the low-level generic assembly incorporates a mechanism to pre-compute on the reference element the linear term for elements with a linear transformation makes that it can be faster on a few simple linear terms. Of course, a possibility would be to incorporate the ability to pre-compute on the reference element the linear term for linear transformations in the high level generic assembly. However, it would be rather complicated due to the high genericity of the language. A consequence also is that exact integration is not allowed in the high level generic assembly.
10.2 Overview of the weak form language syntax

A specific weak form language has been developed to describe the weak formulation of boundary value problems. It is intended to be close to the structure of a standard weak formulation and it incorporates the following components:

- **Variable names:** A list of variables should be given. The variables are described on a finite element method or can be a simple vector of unknowns. For instance $u, v, p, \text{pressure}, \text{electric\_field}$ are valid variable names.

- **Constant names:** A list of constants could be given. The rules are the same as for the variables but no test functions can be associated to constants.

- **Test functions:** Can be used with respect to any of the variables. They are identified by the prefix `Test_` followed by the corresponding variable name. For instance `Test_u, Test_v, Test_p, Test_pressure, Test_electric\_field`. For the tangent system, second order test functions are denoted `Test2_` followed by the variable name.

- **Gradients:** Spatial gradients of variables or test functions are identified by the prefix `Grad_` followed by the variable name or by `Test_` or `Test2_` followed itself by the variable name. This is available for FEM variables only. For instance `Grad_u, Grad_pressure, Grad_electric\_field` and `Grad_Test_u, Grad_Test2_v`. For vector fields, `Div_u` and `Div_Test_u` are some shortcuts for `Trace(Grad_u)` and `Trace(Grad_Test_u)`, respectively.

- **Hessians:** The Hessian of a variable or test function is identified by the prefix `Hess_` followed by the variable name or by `Test_` or `Test2_` followed itself by the variable name. This is available for FEM variables only. For instance `Hess_u, Hess_v, Hess_p, Hess_Test2_v, Hess_Test_p, Hess_Test_pressure`.

- **A certain number of predefined scalar functions** ($\sin(t), \cos(t), \text{pow}(t,u), \sqrt{t}, \text{sqr}(t), \text{Heaviside}(t), \ldots$). A scalar function can be applied to scalar or vector/matrix/tensor expressions. It applies componentwise. For functions having two arguments ($\text{pow}(t,u), \text{min}(t,u), \ldots$) if two non-scalar arguments are passed, the dimension have to be the same. For instance “max([1;2],[0;3])” will return “[1;3]”.

- **A certain number of operators:** $+, -, *, /, :, \ldots, *, /, @, \,'$.

- **Some constants:** $\pi, \text{meshdim}$ (the dimension of the current mesh), $\text{qdim}(u)$ and $\text{qdims}(u)$ the dimensions of the variable $u$ (the size for fixed size variables and the dimension of the vector field for FEM variables), $\text{Id}(n)$ the identity $n \times n$ matrix.

- Parentheses can be used to change the operations order in a standard way. For instance $(1+2)\times4$ or $(u+v)\times\text{Test\_u}$ are valid expressions.

- The access to a component of a vector/matrix/tensor can be done by following a term by a left parenthesis, the list of components and a right parenthesis. For instance $[1,1,2]$ $(3)$ is correct and will return 2. Note that indices are assumed to begin by 1 (even in C++ and with the python interface). A colon can replace the value of an index in a Matlab like syntax.

- **Explicit vectors:** For instance $[1;2;3;4]$ is an explicit vector of size four. Each component can be an expression.

- **Explicit matrices:** For instance $[1,3;2,4]$ and $[[1,2],[3,4]]$ denote the same 2x2 matrix. Each component can be an expression.
• Explicit fourth order tensors: example of explicit 3x2x2x2 fourth order tensor in the nested format:
  
  \[
  \begin{array}{cccc}
  \{1,2,3\}, & \{1,2,3\}, & \{1,2,3\}, & \{1,2,3\} \\
  \{1,2,3\}, & \{1,2,3\}, & \{1,2,3\}, & \{1,2,3\} \\
  \end{array}
  \]

• \(X\) is the current coordinate on the real element, \(X(i)\) is its \(i\)-th component.

• Normal is the outward unit normal vector to a boundary, when integrating on a domain boundary, or the unit normal vector to a level-set when integrating on a level-set with a mesh_im_level_set method. In the latter case, the normal vector is in the direction of the level-set function gradient.

• Reshape(\(t, i, j, \ldots\)): Reshape a vector/matrix/tensor. Note that all tensors in GetFEM++ are stored in the Fortran order.

• A certain number of linear and nonlinear operators (Trace, Norm, Det, Deviator, Contract, \ldots). The nonlinear operators cannot be applied to test functions.

• Diff(expression, variable): The possibility to explicit differentiate an expression with respect to a variable (symbolic differentiation).

• Diff(expression, variable, direction): computes the derivative of expression with respect to variable in the direction direction.

• Grad(expression): When possible, symbolically derive the gradient of the given expression.

• Possibility of macro definition (in the model, the ga_workspace object or directly in the assembly string). The macros should be some valid expressions that are expanded inline at the lexical analysis phase (if they are used several times, the computation is automatically factorized at the compilation stage).

• Interpolate(variable, transformation): Powerful operation which allows to interpolate the variables, or test functions either on the same mesh on other elements or on another mesh. transformation is an object stored by the workspace or model object which describes the map from the current point to the point where to perform the interpolation. This functionality can be used for instance to prescribe periodic conditions or to compute mortar matrices for two finite element spaces defined on different meshes or more generally for fictitious domain methods such as fluid-structure interaction.

• Elementary_transformation(variable, transformation, dest): Allow a linear transformation defined at the element level (i.e. not possible to define at the gauss point level). This feature has been added mostly for defining a reduction for plate elements (projection onto low-level vector element such as rotated RT0). transformation is an object stored by the workspace or model object which describes the trasformation for a particular element. dest is an optional argument refering to a model variable or data whose fem will be the target fem of the transformation. If omitted, the target fem of the transformation is the one of the first variable.

• Possibility of integration on the direct product of two-domains for double integral computation or coupling of two variables with a Kernel / convolution / exchange integral. This allows terms like

  \[
  \int_{\Omega_1} \int_{\Omega_2} k(x,y)u(x)v(y)dydx
  \]

  with \(\Omega_1\) and \(\Omega_2\) two domains, different or not, having their own meshes, integration methods and with \(u\) a variable defined on \(\Omega_1\) and \(v\) a variable defined on \(\Omega_2\). The keyword Secondary_domain(variable) allows to access to the variables on the second domain of integration.
### 10.3 Some basic examples

The weak formulation for the Poisson problem on a domain $\Omega$

$$-\text{div} \ \nabla u = f, \text{ in } \Omega,$$

with Dirichlet boundary conditions $u = 0$ on $\partial \Omega$ is classically

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx,$$

for all test functions $v$ vanishing on $\partial \Omega$. The corresponding expression on the assembly string is:

```
Grad_u.Grad_Test_u - my_f*Test_u
```

where $my_f$ is the expression of the source term. If now the equation is

$$-\text{div} \ a \nabla u = f, \text{ in } \Omega,$$

for $a$ a scalar coefficient, the corresponding assembly string is:

```
a*Grad_u.Grad_Test_u - my_f*Test_u
```

where $a$ has to be declared as a scalar constant or a scalar field. Not that is is also possible to describe it explicitly. For instance the problem

$$-\text{div} \ \sin(x_1 + x_2) \nabla u = f, \text{ in } \Omega,$$

where $x_1, x_2$ are the coordinates on the mesh, can be expressed:

```
\sin(X(1)+X(2))*Grad_u.Grad_Test_u - my_f*Test_u
```

Another classical equation is linear elasticity:

$$-\text{div} \ \sigma(u) = f, \text{ in } \Omega,$$

for $u$ a vector field and $\sigma(u) = \lambda \text{div} \ u + \mu (\nabla u + (\nabla u)^T)$ when isotropic linear elasticity is considered.

The corresponding assembly string to describe the weak formulation can be written:

```
"(lambda*Trace(Grad_u)*Id(qdim(u)) + mu*(Grad_u+Grad_u')):Grad_Test_u - my_f.Test_u"
```

or:

```
"lambda*Div_u*Div_Test_u + mu*(Grad_u + Grad_u'):Grad_Test_u - my_f.Test_u"
```

Here again, the coefficients $\lambda$ and $\mu$ can be given constants, or scalar field or explicit expression or even expression coming from some other variables in order to couples some problems. For instance, if the coefficients depends on a temperature field one can write:

```
"my_f1(theta)*Div_u*Div_Test_u + my_f2(theta)*(Grad_u + Grad_u'):Grad_Test_u - my_f.Grad_Test_u"
```

where $\theta$ is the temperature which can be the solution to a Poisson equation:
and \( my_f1 \) and \( my_f2 \) are some given functions. Note that in that case, the problem is nonlinear due to the coupling, even if the two functions \( my_f1 \) and \( my_f2 \) are linear.

## 10.4 Derivation order and symbolic differentiation

The derivation order of the assembly string is automatically detected. This means that if no test functions are found, the order will be considered to be 0 (potential energy), if first order test functions are found, the order will be considered to be 1 (weak formulation) and if both first and second order test functions are found, the order will be considered to be 2 (tangent system).

In order to perform an assembly (see next section), one should specify the order (0, 1 or 2). If an order 1 string is furnished and an order 2 assembly is required, a symbolic differentiation of the expression is performed. The same if an order 0 string is furnished and if an order 1 or 2 assembly is required. Of course, the converse is not true. If an order 1 expression is given and an order 0 assembly is expected, no integration is performed. This should not be generally not possible since an arbitrary weak formulation do not necessary derive from a potential energy.

The standard way to use the generic assembly is to furnish order 1 expressions (i.e. a weak formulation). If a potential energy exists, one may furnish it. However, it will be derived twice to obtain the tangent system which could result in complicated expressions. For nonlinear problems, it is not allowed to furnish order 2 expressions directly. The reason is that the weak formulation is necessary to obtain the residual. So nothing could be done with a tangent term without having the corresponding order 1 term.

IMPORTANT REMARK: Note that for coupled problems, a global potential frequently do not exists. So that the part of problems directly defined with a potential may be difficult to couple. To illustrate this, if you defined a potential with some parameters (elasticity coefficients for instance), and the coupling consists in a variation of these coefficients with respect to another variable, then the weak formulation do not consist of course in the derivative of the potential with respect to the coefficients which has generally no sense. This is the reason why the definition through a potential should be the exception.

## 10.5 C++ Call of the assembly

Note that the most natural way to use the generic assembly is by the use of the generic assembly bricks of the model object, see Section *Generic assembly bricks*. It is however also possible to use the high level generic assembly on its own.

The generic assembly is driven by the object `getfem::ga_workspace` defined in `getfem/getfem_generic_assembly.h`.

There is two ways to define a `getfem::ga_workspace` object. It can depend on a model (see *The model description and basic model bricks*) and should be declared as:

```cpp
getfem::ga_workspace workspace(model);
```

with `model` a previously define `getfem::model` object. In that case the variable and constant considered are the one of the model. The second way it to define an independent `getfem::ga_workspace` object by:
getfem::ga_workspace workspace;

In that case, the variable and constant have to be added to the workspace. This can be done thanks to the following methods:

```cpp
workspace.add_fem_variable(name, mf, I, V);
workspace.add_fixed_size_variable(name, I, V);
workspace.add_fem_constant(name, mf, V);
workspace.add_fixed_size_constant(name, V);
workspace.add_im_data(name, imd, V);
```

where `name` is the variable/constant name (see in the next sections the restriction on possible names), `mf` is the `getfem::mesh_fem` object describing the finite element method, `I` is an object of class `gmm::sub_interval` indicating the interval of the variable on the assembled vector/matrix and `V` is a `getfem::base_vector` being the value of the variable/constant. The last method add a constant defined on an `im_data` object `imd` which allows to store scalar/vector/tensor field informations on the integration points of an `mesh_im` object.

Once it is declared and once the variables and constant are declared, it is possible to add assembly string to the workspace with:

```cpp
workspace.add_expression("my expression", mim, rg = all_convexes());
```

where "my_expression" is the assembly string, `mim` is a `getfem::mesh_im` object and `rg` if an optional valid region of the mesh corresponding to `mim`.

As it is explained in the previous section, the order of the string will be automatically detected and a symbolic differentiation will be performed to obtain the corresponding tangent term.

Once assembly strings are added to the workspace, it is possible to call:

```cpp
workspace.assembly(order);
```

where `order` should be equal to 0 (potential energy), 1 (residual vector) or 2 (tangent term, or stiffness matrix for linear problems). The result of the assembly is available as follows:

```cpp
workspace.assembled_potential() // For order = 0
workspace.assembled_vector()   // For order = 1
workspace.assembled_matrix()   // For order = 2
```

By default, the assembled potential, vector and matrix is initialized to zero at the beginning of the assembly. It is however possible (and recommended) to set the assembly vector and matrix to external ones to perform an incremental assembly. The two methods:

```cpp
workspace.set_assembled_vector(getfem::base_vector &V);
workspace.set_assembled_matrix(getfem::model_real_sparse_matrix &K);
```

allows to do so. Be aware to give a vector and a matrix of the right dimension.
Note also that the method:

```cpp
workspace.clear_expressions();
```

allows to cancel all furnished expressions and allows to re-use the same workspace for another assembly.

It is also possible to call the generic assembly from the Python/Scilab/Matlab interface. See `gf_asm` command of the interface for more details.

## 10.6 C++ assembly examples

As a first example, if one needs to perform the assembly of a Poisson problem

\[- \text{div} \nabla u = f, \text{ in } \Omega,\]

the stiffness matrix is given

\[K_{i,j} = \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j \, dx,\]

and will be assembled by the following code:

```cpp
gemf::ga_workspace workspace;
gemf::size_type nbdo = mf.nb_dof();
gemf::base_vector U(nbdo);
workspace.add_fem_variable("u", mf, gmm::sub_interval(0, nbdo), U);
workspace.add_expression("Grad_u.Grad_Test_u", mim);
gemf::model_real_sparse_matrix K(nbdo, nbdo);
workspace.set_assembled_matrix(K);
workspace.assembly(2);
```

where of course, `mf` is supposed to be an already declared `gemf::mesh_fem` object and `mim` a already declared `gemf::mesh_im` object on the same mesh. Note that the value of the variable do not really intervene because of the linearity of the problem. This allows to pass `gemf::base_vector(nbdo)` as the value of the variable which will not be used. Note also that two other possible expressions for exactly the same result for the assembly string are "Grad_Test2_u.Grad_Test_u" (i.e. an order 2 expression) or "Norm_sqr(Grad_u)/2" (i.e. a potential). In fact other possible assembly string will give the same result such as "Grad_u.Grad_u/2" or "[Grad_u(1), Grad_u(2)].[Grad_Test_u(1), Grad_Test_u(2)]" for two-dimensional problems. However, the recommendation is preferably to give an order 1 expression (weak formulation) if there is no particular reason to prefer an order 0 or an order 2 expression.

As a second example, let us consider a coupled problem, for instance the mixed problem of incompressible elasticity given by the equations

\[- \text{div} (\mu \nabla u + (\nabla u)^T - pI_d) = f, \text{ in } \Omega,\]

\[\text{div } u = 0.\]

where \(u\) is the vector valued displacement and \(p\) the pressure. The assembly of the matrix for the whole coupled system can be performed as follows:

```cpp
gemf::ga_workspace workspace;
gemf::size_type nbdofu = mf_u.nb_dof();
gemf::size_type nbdofp = mf_p.nb_dof();
```

(continues on next page)
getfem::base_vector U(nbdofu);
getfem::base_vector P(nbdofp);
getfem::base_vector vmu(1); vmu[0] = mu;
workspace.add_fem_variable("u", mf_u, gmm::sub_interval(0, nbdofu), U);
workspace.add_fem_variable("p", mf_p, gmm::sub_interval(nbdofu, nbdofp), P);
workspace.add_fixed_size_constant("mu", vmu);
workspace.add_expression("2*mu*Sym(Grad_u):Grad_Test_u - p*Trace(Grad_Test_u) - Test_p*Trace(Grad_u)", mim);
getfem::model_real_sparse_matrix K(nbdofu+nbdofp, nbdofu+nbdofp);
workspace.set_assembled_matrix(K);
workspace.assembly(2);

where, here, mf_u and mf_p are supposed to be some already declared getfem::mesh_fem objects defined on the same mesh, mim a already declared getfem::mesh_im object and mu is the Lame coefficient. It is also possible to perform the assembly of the sub-matrix of this system separately.

Let us see now how to perform the assembly of a source term. The weak formulation of a volumic source term is

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

where $f$ is the source term and $v$ the test function. The corresponding assembly can be written:

getfem::ga_workspace workspace;
getfem::size_type nbdofu = mf_u.nb_dof();
getfem::base_vector U(nbdofu);
workspace.add_fem_variable("u", mf_u, gmm::sub_interval(0, nbdofu), U);
workspace.add_fem_constant("f", mf_data, F);
workspace.add_expression("f*Test_u", mim);
getfem::base_vector L(nbdofu);
workspace.set_assembled_vector(L);
workspace.assembly(1);

if the source term is describe on a finite element mf_data and the corresponding vector of degrees of freedom F. Explicit source terms are also possible. For instance:

getfem::ga_workspace workspace;
getfem::size_type nbdofu = mf_u.nb_dof();
getfem::base_vector U(nbdofu);
workspace.add_fem_variable("u", mf_u, gmm::sub_interval(0, nbdofu), U);
workspace.add_expression("sin(X(1)+X(2))*Test_u", mim);
getfem::base_vector L(nbdofu);
workspace.set_assembled_vector(L);
workspace.assembly(1);

is also valid. If the source term is a boundary term (in case of a Neumann condition) the only difference is that the mesh region corresponding to the boundary have to be given as follows:

workspace.add_expression("sin(X(1)+X(2))*Test_u", mim, region);

where region is the mesh region number.

As another example, let us describe a simple nonlinear elasticity problem. Assume that we consider a Saint-Venant Kirchhoff constitutive law which means that we consider the following elastic energy on a
body of reference configuration $\Omega$:

$$
\int_{\Omega} \frac{\lambda}{2} (\text{tr}(E))^2 + \mu \text{tr}(E^2) dx
$$

where $\lambda, \mu$ are the Lamé coefficients and $E$ is the strain tensor given by $E = (\nabla u + (\nabla u)^T + (\nabla u)^T \nabla u)/2$.

This is possible to perform the assembly of the corresponding tangent problem as follows:

```cpp
getfem::ga_workspace workspace;
getfem::size_type nbdofu = mf_u.nb_dof();
getfem::base_vector vlambda(1); vlambda[0] = lambda;
getfem::base_vector vmu(1); vmu[0] = mu;
workspace.add_fem_variable("u", mf_u, gmm::sub_interval(0, nbdofu), U);
workspace.add_fixed_size_constant("lambda", vlambda);
workspace.add_fixed_size_constant("mu", vmu);
workspace.add_expression("lambda*sqr(Trace(Grad_u+Grad_u'+Grad_u'*Grad_u)) + mu*Trace((Grad_u+Grad_u'+Grad_u'*Grad_u) *(Grad_u+Grad_u'+Grad_u'*Grad_u))", mim);
getfem::base_vector L(nbdofu);
workspace.set_assembled_vector(V);
workspace.assembly(1);
getfem::model_real_sparse_matrix K(nbdofu, nbdofu);
workspace.set_assembled_matrix(K);
workspace.assembly(2);
```

and to adapt a Newton-Raphson algorithm to solve that nonlinear problem. Of course the expression is rather repetitive and it would be preferable to define some intermediate nonlinear operators. However, note that repeated expressions are automatically detected and computed only once in the assembly.

The last example is the assembly of the stiffness matrix of an order four problem, the Kirchhoff-Love plate problem:

```cpp
getfem::ga_workspace workspace;
getfem::size_type nbdofu = mf_u.nb_dof();
getfem::base_vector vD(1); vD[0] = D;
getfem::base_vector vnu(1); vnu[0] = nu;
workspace.add_fem_variable("u", mf_u, gmm::sub_interval(0, nbdofu), U);
workspace.add_fixed_size_constant("D", vD);
workspace.add_fixed_size_constant("nu", vnu);
workspace.add_expression("D*(1-nu)*(Hess_u:Hess_Test_u) - D*nu*Trace(Hess_u)*Trace(Hess_Test_u)" , mim);
getfem::model_real_sparse_matrix K(nbdofu, nbdofu);
workspace.set_assembled_matrix(K);
workspace.assembly(2);
```

with $D$ the flexion modulus and $\nu$ the Poisson ratio.

## 10.7 Script languages call of the assembly

For the use with Python, Scilab or Matlab interfaces, see the respective documentation, in particular the `gf_asm` command and the `model` object.
10.8 The tensors

Basically, what is manipulated in the weak form language are tensors. This can be order 0 tensors in scalar expressions (for instance in \(3+\sin(\pi/2)\)), order 1 tensors in vector expressions (such as \(X.X\) or \(\text{Grad}_u\) if \(u\) is a scalar variable), order 2 tensors for matrix expressions and so on. For efficiency reasons, the language manipulates tensors up to order six. The language could be easily extended to support tensors of order greater than six but it may lead to inefficient computations. When an expression contains test functions (as in \(\text{Trace}(\text{Grad}_\text{Test}_u)\) for a vector field \(u\)), the computation is done for each test functions, which means that the tensor implicitly have a supplementary component. This means that, implicitly, the maximal order of manipulated tensors are in fact six (in \(\text{Grad}_\text{Test}_u:\text{Grad}_\text{Test2}_u\) there are two components implicitly added for first and second order test functions).

Order four tensors are necessary for instance to express elasticity tensors or in general to obtain the tangent term for vector valued unknowns.

10.9 The variables

A list of variables should be given to the \texttt{ga_worspace} object (directly or through a model object). The variables are described on a finite element method or can be a simple vector of unknowns. This means that it is possible also to couple algebraic equations to pde ones on a model. A variable name should begin by a letter (case sensitive) or an underscore followed by a letter, a number or an underscore. Some name are reserved, this is the case of operators names (\texttt{Det}, \texttt{Norm}, \texttt{Trace}, \texttt{Deviator},...) and thus cannot be used as variable names. The name should not begin by \texttt{Test\_}, \texttt{Test2\_}, \texttt{Grad\_}, \texttt{Div\_} or \texttt{Hess\_}. The variable name should not correspond to a predefined function (\texttt{sin}, \texttt{cos}, \texttt{acos} ...) and to constants (\texttt{pi}, \texttt{Normal}, \texttt{X}, \texttt{Id} ...).

10.10 The constants or data

A list of constants could also be given to the \texttt{ga_worspace} object. The rule are the same as for the variables but no test function can be associated to constants and there is no symbolic differentiation with respect to constants. Scalar constants are often defined to represent the coefficients which intervene in constitutive laws. Additionally, constants can be some scalar/vector/tensor fields defined on integration points via \texttt{im_data} object (for instance for some implementation of the approximation of constitutive laws such as plasticity).

10.11 Test functions

Each variable is associated with first order and second order test functions. The first order test function are used in the weak formulation (which derive form the potential equation if it exists) and the second order test functions are used in the tangent system. For a variable \(u\) the associated test functions are \texttt{Test\_u} and \texttt{Test2\_u}. The assembly string have to be linear with respect to test functions. As a result of the presence of the term \texttt{Test\_u} on a assembly string, the expression will be evaluated for each shape function of the finite element corresponding to the variable \(u\). On a given element, if the finite element have \(N\) shape functions ans if \(u\) is a scalar field, the value of \texttt{Test\_u} will be the value of each shape function on the current point. So \texttt{Test\_u} return if face a vector of \(N\) values. But of course, this is implicit in the language. So one do not have to care about this.
10.12 Gradient

The gradient of a variable or of test functions are identified by Grad__ followed by the variable name or by Test__ followed itself by the variable name. This is available for FEM variables (or constants) only. For instance Grad_u, Grad_v, Grad_p, Grad_pressure, Grad_electric_field and Grad_Test_u, Grad_Test_v, Grad_Test_p, Grad_Test_pressure, Grad_Test_electric_field. The gradient is either a vector for scalar variables or a matrix for vector field variables. In the latter case, the first index corresponds to the vector field dimension and the second one to the index of the partial derivative. Div_u and Div_Test_u are some optimized shortcuts for Trace(Grad_u) and Trace(Grad_Test_u), respectively.

10.13 Hessian

Similarly, the Hessian of a variable or of test functions are identified by Hess__ followed by the variable name or by Test__ followed itself by the variable name. This is available for FEM variables only. For instance Hess_u, Hess_v, Hess_p, Hess_pressure, Hess_electric_field and Hess_Test_u, Hess_Test_v, Hess_Test_p, Hess_Test_pressure, Hess_Test_electric_field. The Hessian is either a matrix for scalar variables or a third order tensor for vector field variables. In the latter case, the first index corresponds to the vector field dimension and the two remaining to the indices of partial derivatives.

10.14 Predefined scalar functions

A certain number of predefined scalar functions can be used. The exhaustive list is the following and for most of them are equivalent to the corresponding C function:

- sqr(t) (the square of t, equivalent to t*t), pow(t, u) (t to the power u), sqrt(t) (square root of t), exp(t), log(t), log10(t)
- sin(t), cos(t), tan(t), asin(t), acos(t), atan(t), atan2(t, u)
- sinh(t), cosh(t), tanh(t), asinh(t), acosh(t), atanh(t)
- erf(t), erfc(t)
- sinc(t) (the cardinal sine function sin(t)/t)
- Heaviside(t) (0 for t < 0, 1 for t ≥ 0)
- sign(t)
- abs(t)
- pos_part(t) (tH(t))
- reg_pos_part(t, eps) (((t - eps/2 - t^2/(2*eps))H(t - eps) + t^2 H(t)/(2*eps))
- neg_part(t) (-tH(-t)), max(t, u), min(t, u)

A scalar function can be applied to a scalar expression, but also to a tensor one. If is is applied to a tensor expression, is is applied componentwise and the result is a tensor with the same dimensions. For functions having two arguments (pow(t,u), min(t,u) . . . ) if two non-scalar arguments are passed, the dimension have to be the same. For instance “max(1;2, [0;3])” will return “[0;3]”.

10.12. Gradient 45
10.15 User defined scalar functions

It is possible to add a scalar function to the already predefined ones. Note that the generic assembly consider only scalar function with one or two parameters. In order to add a scalar function to the generic assembly, one has to call:

```
ga_define_function(name, nb_args, expr, der1="", der2="");
ga_define_function(name, getfem::pscalar_func_onearg f1, der1="");
ga_define_function(name, getfem::pscalar_func_twoargs f2, der1="", der2="");
```

where `name` is the name of the function to be defined, `nb_args` is equal to 1 or 2. In the first call, `expr` is a string describing the function in the generic weak form language and using `t` as the first variable and `u` as the second one (if `nb_args` is equal to 2). For instance, `sin(2*t)+sqr(t)` is a valid expression. Note that it is not possible to refer to constant or data defined in a `ga_workspace` object. `der1` and `der2` are the expression of the derivatives with respect to `t` and `u`. They are optional. If they are not furnished, a symbolic differentiation is used if the derivative is needed. If `der1` and `der2` are defined to be only a function name, it will be understand that the derivative is the corresponding function.

In the second call, `f1` should be a C pointer on a scalar C function having one scalar parameter and in the third call, `f2` should be a C pointer on a scalar C function having two scalar parameters.

Additionally:

```
bool ga_function_exists(name)
```

return true is a function `name` is already defined and:

```
.ga_undefine_function(name)
```

cancel the definition of an already define function (it has no action if the function does not exist) which allow to redefine a function.

10.16 Derivatives of defined scalar functions

It is possible to refer directly to the derivative of defined functions by adding the prefix `Derivative_` to the function name. For instance, `Derivative_sin(t)` will be equivalent to `cos(t)`. For two arguments functions like `pow(t,u)` one can refer to the derivative with respect to the second argument with the prefix `Derivative_2_` before the function name.

10.17 Binary operations

A certain number of binary operations between tensors are available:

- + and – are the standard addition and subtraction of scalar, vector, matrix or tensors.
- * stands for the scalar, matrix-vector, matrix-matrix or (fourth order tensor)-matrix multiplication.
- / stands for the division by a scalar.
• stands for the scalar product of vectors, or more generally to the contraction of a tensor with respect to its last index with a vector or with the first index of another tensor. Note that * and . are equivalent for matrix-vector or matrix-matrix multiplication.

• : stands for the Frobenius product of matrices or more generally to the contraction of a tensor with respect to the two last indices with a matrix or the two first indices of a higher order tensor. Note that * and : are equivalent for (fourth order tensor)-matrix multiplication.

• .* stands for the multiplication of two vectors/matrix/tensor componentwise.

• ./ stands for the division of two vectors/matrix/tensor componentwise.

• @ stands for the tensor product.

• Contract(A, i, B, j) stands for the contraction of tensors A and B with respect to the ith index of A and jth index of B. The first index is numbered 1. For instance Contract(V,1,W,1) is equivalent to V.W for two vectors V and W.

• Contract(A, i, j, B, k, l) stands for the double contraction of tensors A and B with respect to indices i,j of A and indices k,l of B. The first index is numbered 1. For instance Contract(A,1,2,B,1,2) is equivalent to A:B for two matrices A and B.

10.18 Unary operators

• – the unary minus operator: change the sign of an expression.

• ' stands for the transpose of a matrix or line view of a vector. It a tensor A is of order greater than two, A' denotes the inversion of the two first indices.

• Contract(A, i, j) stands for the contraction of tensor A with respect to its ith and jth indices. The first index is numbered 1. For instance, Contract(A, 1, 2) is equivalent to Trace(A) for a matrix A.

• Swap_indices(A, i, j) exchange indices number i and j. The first index is numbered 1. For instance Swap_indices(A, 1, 2) is equivalent to A' for a matrix A.

• Index_move_last(A, i) move the index number i in order to be the last one. For instance, if A is a fourth order tensor A_{ij}^{k}l_{ij}^{k}j, then the result of Index_move_last(A, 2) will be the tensor B_{ijkl}^{ij} = A_{ij}^{k}l_{ij}^{k}. For a matrix, Index_move_last(A, 1) is equivalent to A'.

10.19 Parentheses

Parentheses can be used in a standard way to change the operation order. If no parentheses are indicated, the usually priority order are used. The operations + and – have the lower priority (with no distinction), then *, /, :, ..., *, ., @ with no distinction and the higher priority is reserved for the unary operators – and ‘.”

10.20 Explicit vectors

The weak form language allows to define explicit vectors (i.e. order 1 tensors) with the notation [a, b, c, d, e], i.e. an arbitrary number of components separated by a comma (note the separation with a semicolon [a;b;c;d;e] is also permitted), the whole vector beginning with a right bracket and
ended by a left bracket. The components can be some numeric constants, some valid expressions and may also contain test functions. In the latter case, the vector has to be homogeneous with respect to the test functions. This means that a construction of the type \([\text{Test}_u; \text{Test}_v]\) is not allowed. A valid example, with \(u\) as a scalar field variable is \([5\cdot\text{Grad}_\text{Test}_u(2), 2\cdot\text{Grad}_\text{Test}_u(1)]\). Note also that using the quite operator (transpose), an expression \([a, b, c, d, e]'\) stands for ‘row vector’, i.e. a 1x5 matrix.

### 10.21 Explicit matrices

Similarly to explicit vectors, it is possible to define explicit matrices (i.e. order 2 tensors) with the notation \([[a, b], [c, d]]\), i.e. an arbitrary number of columns vectors separated by a comma (the syntax \([a, c; b, d]\) of lines separated by a semicolon is also permitted). For instance \([[11, 21], [12, 22], [13, 23]]\) and \([11, 12, 13; 21, 22, 23]\) both represent the same 2x3 matrix. The components can be some numeric constants, some valid expressions and may also contain test functions.

### 10.22 Explicit tensors

Explicit tensors of any order are permitted with the nested format. A tensor of order \(n\) is written as a succession of tensor of order \(n-1\) of equal dimensions and separated by a comma. For instance \([[[[1, 2, 3], [1, 2, 3]], [[1, 2, 3], [1, 2, 3]]], [[[1, 2, 3], [1, 2, 3]], [[1, 2, 3], [1, 2, 3]]]]\) is a fourth order tensor. Another possibility is to use the syntax \(\text{Reshape}([1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3], 3, 2, 2, 2)\) where the components have to be given in Fortran order.

### 10.23 Access to tensor components

The access to a component of a vector/matrix/tensor can be done by following a term by a left parenthesis, the list of components and a right parenthesis. For instance \([1, 1, 2](3)\) is correct and is returning 2 as expected. Note that indices are assumed to begin by 1 (even in C++ and with the python interface). The expressions \([1, 1; 2, 3](2, 2)\) and \(\text{Grad}_u(2, 2)\) are also correct provided that \(u\) is a vector valued declared variable. Note that the components can be the result of a constant computation. For instance \([1, 1; 2, 3](1+1,a)\) is correct provided that \(a\) is a declared constant but not if it is declared as a variable. A colon can replace the value of an index in a Matlab like syntax for instance to access to a line or a column of a matrix. \([1, 1; 2, 3](1,:):\) denotes the first line of the matrix \([1, 1; 2, 3]\). It can also be used for a fourth order tensor.

### 10.24 Constant expressions

- Floating points with standards notations (for instance \(3, 1.456, 1E-6\))
- \(\pi\): the constant \(\Pi\).
- \(\text{meshdim}\): the dimension of the current mesh (i.e. size of geometrical nodes)
- \(\text{timestep}\): the main time step of the model on which this assembly string is evaluated (defined by \(\text{model.set_time_step}(dt)\)). Do not work on pure workspaces.
• **Id(n):** the identity matrix of size $n \times n$. $n$ should be an integer expression. For instance `Id(meshdim)` is allowed.

• **qdim(u):** the total dimension of the variable $u$ (i.e. the size for fixed size variables and the total dimension of the vector/tensor field for FEM variables)

• **qdims(u):** the dimensions of the variable $u$ (i.e. the size for fixed size variables and the vector of dimensions of the vector/tensor field for FEM variables)

### 10.25 Special expressions linked to the current position

• **$X$** is the current coordinate on the real element (i.e. the position on the mesh of the current Gauss point on which the expression is evaluated), $X(i)$ is its $i$-th component. For instance $\sin(X(1)+X(2))$ is a valid expression on a mesh of dimension greater or equal to two.

• **Normal** the outward unit normal vector to a boundary when integration on a boundary is performed.

• **element_size** gives an estimate of the current element diameter (using `getfem::convex_radius_estimate`).

• **element_K** gives the gradient of the geometric transformation (see `dp-transgeo`) from the reference (parent) element. Could be used only if the mesh do not contain elements of mixed dimensions.

• **element_B** gives the transpose of the pseudo-inverse of the gradient of the geometric transformation (see `dp-transgeo`) from the reference (parent) element. Could be used only if the mesh do not contain elements of mixed dimensions.

### 10.26 Print command

For debugging purpose, the command `Print(a)` is printing the tensor $a$ and pass it unchanged. For instance `Grad_u.Print(Grad_Test_u)` will have the same effect as `Grad_u.Grad_Test_u` but printing the tensor `Grad_Test_u` for each Gauss point of each element. Note that constant terms are printed only once at the beginning of the assembly. Note also that the expression could be derived so that the derivative of the term may be printed instead of the term itself.

### 10.27 Reshape a tensor

The command `Reshape(t, i, j, ...)` reshapes the tensor $t$ (which could be an expression). The only constraint is that the number of components should be compatible. For instance `Reshape(Grad_u, 1, meshdim)` is equivalent to `Grad_u'$ for $u$ a scalar variable. Note that the order of the components remain unchanged and are classically stored in Fortran order for compatibility with Blas/Lapack.

### 10.28 Trace, Deviator, Sym and Skew operators

Trace, Deviator, Sym and Skew operators are linear operators acting on square matrices:
• \textbf{Trace}(m) gives the trace (sum of diagonal components) of a square matrix \( m \).

• \textbf{Deviator}(m) gives the deviator of a square matrix \( m \). It is equivalent to \( m - \text{Trace}(m) \times \text{Id}(\text{m}\_\text{dim})/\text{m}\_\text{dim} \), where \( \text{m}\_\text{dim} \) is the dimension of \( m \).

• \textbf{Sym}(m) gives the symmetric part of a square matrix \( m \), i.e. \( (m + m')/2 \).

• \textbf{Skew}(m) gives the skew-symmetric part of a square matrix \( m \), i.e. \( (m - m')/2 \).

The four operators can be applied on test functions. Which means that for instance both \textbf{Trace} (Grad \( u \)) and \textbf{Trace} (Grad Test \( u \)) are valid when \textit{Grad} \( u \) is a square matrix (i.e. \( u \) a vector field of the same dimension as the mesh).

### 10.29 Nonlinear operators

The weak form language provide some predefined nonlinear operator. Each nonlinear operator is available together with its first and second derivatives. Nonlinear operator can be applied to an expression as long as this expression do not contain some test functions.

• \textbf{Norm}(v) for \( v \) a vector or a matrix gives the euclidean norm of a vector or a Frobenius norm of a matrix.

• \textbf{Norm}\_sqr(v) for \( v \) a vector or a matrix gives the square of the euclidean norm of a vector or of the Frobenius norm of a matrix. For a vector this is equivalent to \( v \cdot v \) and for a matrix to \( m:m \).

• \textbf{Normalized}(v) for \( v \) a vector or a matrix gives \( v \) divided by its euclidean (for vectors) or Frobenius (for matrices) norm. In order to avoid problems when \( v \) is close to 0, it is implemented as \textbf{Normalized}\_\text{reg}(v, 1E-25). Use with care. Think that the derivative at the origin of \textbf{Normalized}(v)\*\textbf{Norm}(v) is wrong (it vanishes) and very different from the derivative of \( v \).

• \textbf{Normalized}\_\text{reg}(v, \text{eps}) for \( v \) a vector or a matrix gives a regularized version of \textbf{Normalized}(v): \( v/\sqrt{|v|*|v|+\text{eps}^2} \).

• \textbf{Ball}\_\text{projection}(v, r) for \( v \) a vector or a matrix and \( r \) a scalar, gives the projection of \( v \) on the ball of radius \( r \) and center the origin.

• \textbf{Det}(m) gives the determinant of a square matrix \( m \).

• \textbf{Inv}(m) gives the inverse of a square matrix \( m \). The second derivative is not available since it is an order 6 tensor. This means that \textbf{Inv}(m) cannot be used in the description of a potential energy.

• \textbf{Expm}(m) gives the exponential of a square matrix \( m \).

• \textbf{Logm}(m) gives the logarithm of a square matrix \( m \).

• \textbf{Matrix}\_I2(m) gives the second invariants of a square matrix \( m \) which is defined by \( (\text{sqr(Trace}(m)) - \text{Trace}(m\*m))/2 \).

• \textbf{Matrix}\_J1(m) gives the modified first invariant of a square matrix defined by \( \text{Trace}(m)\text{pow(Det}(m),-1/3) \).

• \textbf{Matrix}\_J2(m) gives the modified first invariant of a square matrix defined by \( \text{Matrix}\_I2(m)\text{pow(Det}(m),-2/3) \).
10.30 Macro definition

The weak form language allows the use of macros that are either predefined in the model or `ga_workspace` object or directly defined at the beginning of an assembly string. The definition into a `ga_workspace` or model object is done as follows:

```python
workspace.add_macro(name, expr)
```

or:

```python
model.add_macro(name, expr)
```

The definition of a macro into an assembly string is inserted before any regular expression, separated by a semicolon with the following syntax:

```
"Def name:=expr; regular_expression"
```

where `name` is the macro name which then can be used in the weak form language and contains also the macro parameters, `expr` is a valid expression of the weak form language (which may itself contain some macro definitions). For instance, a valid macro with no parameter is:

```python
model.add_macro("my_transformation", "[cos(alpha)*X(1); sin(alpha)*X(2)]");
```

where `alpha` should be a valid declared variable or data. A valid macro with two parameters is for instance:

```python
model.add_macro("ps(a,b)", "a.b");
```

The following assembly string is then valid (if `u` is a valid variable):

```
"Def ps(a,b):=a.b; ps(Grad_u, Grad_Test_u)"
```

Parameter are allowed to be post-fixed to `Grad_`, `Hess_`, `Test_` and `Test2_` prefixes, so that the following assembly string is valid:

```
"Def psgrad(a,b):=Grad_a.Grad_b; psgrad(u, Test_u)"
```

or with an imbrication of two macros:

```
"Def ps(a,b):=a.b; Def psgrad(a,b):=ps(Grad_a,Grad_b); psgrad(u, Test_u)"
```

A macro can be deleted from a `ga_workspace` or model object as follows:

```python
workspace.del_macro(name)
model.del_macro(name)
```

Note that a macro defined at the beginning of an assembly string is only defined in the assembly string and cannot be used later without being added in a model or `ga_workspace` object.

The macros are expanded inline at the lexical analysis phase. Note that a the compilation phase, the repeated expressions are automatically factorized and computed only once.
10.31 Explicit Differentiation

The workspace object automatically differentiate terms that are of lower deriation order. However, it is also allowed to explicitly differentiate an expression with respect to a variable. One interest is that the automatic differentiation performs a derivative with respect to all the declared variables of model/workspace but this is not necessarily the expected behavior when using a potential energy, for instance. The syntax is:

\[
\text{Diff(expression, variable)}
\]

For instance, the following expression:

\[
\text{Diff(u.u, u)}
\]

will result in:

\[
2*(u.Test_u)
\]

So that:

\[
\text{Grad}_u: \text{Grad}_test_u + \text{Diff(u.u, u)}
\]

is a valid expression. A third argument can be added to the \text{Diff} command to specify the direction:

\[
\text{Diff(expression, variable, direction)}
\]

in that case, it replaces the \text{Test_variable} by the expression \text{direction} which has to be of the same dimension as \text{variable}. It computes the derivative of \text{expression} with respect to \text{variable} in the direction \text{direction}. For instance:

\[
\text{Diff(u.u, u, v)}
\]

will result in:

\[
2*(u.v)
\]

if \text{v} is any valid expression of the same dimension than \text{u}.

10.32 Explicit Gradient

It is possible to ask for symbolic computation of the gradient of an expression with:

\[
\text{Grad(expression)}
\]

It will be computed as far as it is possible. The limitations come from the fact that \text{GetFEM++} is limited to second order derivative of shape function and nonlinear operators are supposed to provide only first and second order derivatives.

Of course:

\[
\text{Grad(u)}
\]

is equivalent to:
for a variable $u$.

10.33 Interpolate transformations

The Interpolate operation allows to compute integrals between quantities which are either defined on different part of a mesh or even on different meshes. It is a powerful operation which allows to compute mortar matrices or take into account periodic conditions. However, one have to remember that it is based on interpolation which may have a non-negligible computational cost.

In order to use this functionality, the user have first to declare to the workspace or to the model object an interpolate transformation which described the map between the current integration point and the point lying on the same mesh or on another mesh.

Different kind of transformations can be described. Several kinds of transformations has been implemented. The first one, described hereafter is a transformation described by an expression. A second one corresponds to the raytracing contact detection (see Raytracing interpolate transformation). Some other transformations (neighbour element and element extrapolation) are describe in the next sections.

The transformation defined by an expression can be added to the workspace or the model thanks to the command:

```plaintext
add_interpolate_transformation_from_expression
  (workspace, transname, source_mesh, target_mesh, expr);
```

or:

```plaintext
add_interpolate_transformation_from_expression
  (model, transname, source_mesh, target_mesh, expr);
```

where `workspace` is a workspace object, `model` a model object, `transname` is the name given to the transformation, `source_mesh` the mesh on which the integration occurs, `target_mesh` the mesh on which the interpolation is performed and `expr` is a regular expression of the high-level generic weak form language which may contains reference to the variables of the workspace/model.

For instance, an expression:

```plaintext
add_interpolate_transformation_from_expression
  (model, "my_transformation", my_mesh, my_mesh, "X-[1;0]");
```

will allow to integrate some expressions at the current position with a shift of -1 with respect to the first coordinate. This simple kind of transformation can be used to prescribe a periodic condition.

Of course, one may used more complex expressions such as:

```plaintext
add_interpolate_transformation_from_expression
  (model, "my_transformation", my_mesh, my_second_mesh, "[X[1]cos(X[2]);
   -X[1]sin(X[2])]");
```

```plaintext
add_interpolate_transformation_from_expression
  (model, "my_transformation", my_mesh, my_mesh, "X+u");
```

where $u$ is a vector variable of the workspace/model.
Once a transformation is defined in the workspace/model, one can interpolate a variable or test functions, the position or the unit normal vector to a boundary thanks to one of these expressions:

\[
\begin{align*}
\text{Interpolate(Normal, transname)} \\
\text{Interpolate(X, transname)} \\
\text{Interpolate(u, transname)} \\
\text{Interpolate(Grad_u, transname)} \\
\text{Interpolate(Div_u, transname)} \\
\text{Interpolate(Hess_u, transname)} \\
\text{Interpolate(Test_u, transname)} \\
\text{Interpolate(Grad_Test_u, transname)} \\
\text{Interpolate(Div_Test_u, transname)} \\
\text{Interpolate(Hess_Test_u, transname)}
\end{align*}
\]

where \( u \) is the name of the variable to be interpolated.

For instance, the assembly expression to prescribe the equality of a variable \( u \) with its interpolation (for instance for prescribing a periodic boundary condition) thanks to a multiplier \( \lambda \) could be written:

\[
(\text{Interpolate(u,my_transformation)-u}) \cdot \lambda
\]

(see demo_periodic_laplacian.m in interface/tests/matlab directory).

In some situations, the interpolation of a point may fail if the transformed point is outside the target mesh. Both in order to treat this case and to allow the transformation to differentiate some other cases (see Raytracing interpolate transformation for the differentiation between rigid bodies and deformable ones in the Raytracing_interpolate_transformation) the transformation returns an integer identifier to the weak form language. A value 0 of this identifier means that no corresponding location on the target mesh has been found. A value of 1 means that a corresponding point has been found. This identifier can be used thanks to the following special command of the weak form language:

\[
\text{Interpolate_filter(transname, expr, i)}
\]

where \( \text{transname} \) is the name of the transformation, \( \text{expr} \) is the expression to be evaluated and \( i \) value of the returned integer identifier for which the expression have to be computed. Note that \( i \) can be ommited, in that case, the expression is evaluated for a nonzero identifier (i.e. when a corresponding point has been found). For instance, the previous assembly expression to prescribe the equality of a variable \( u \) with its interpolation could be writtne:

\[
\text{Interpolate_filter(transmane, Interpolate(u,my_transformation)-u)*lambda)} + \text{Interpolate_filter(transmane, lambda*lambda, 0)}
\]

In that case, the equality will only be prescribed in the part of the domain where the transformation succeed and in the other part, the mutliplier is enforced to vanish.

**CAUTION**: You have to think that when some variables are used in the transformation, the computation of the tangent system takes into account these dependence. However, the second derivative of a transformation with respect to a variable used has not been implemented. Thus, such a transformation is not allowed in the definition of a potential since it cannot be derived twice.

### 10.34 Element extrapolation transformation

A specific transformation (see previous section) is defined in order to allows the evaluation of certain quantities by extrapolation with respect to another element (in general a neighbour element). This is
not strictly speaking a transformation since the point location remain unchanged, but the evaluation is made on another element extrapolating the shape functions outside it. This transformation is used for stabilization term in fictitious domain applications (with cut elements) where it is more robust to extrapolate some quantities on a neighbour element having a sufficiently large intersection with the real domain than evaluating them on the current element if it has a small intersection with the real domain. The functions allowing to add such a transformation to a model or a workspace are:

```cpp
add_element_extrapolation_transformation(
    model, transname, my_mesh, std::map<size_type, size_type> &elt_corr);
add_element_extrapolation_transformation(
    workspace, transname, my_mesh, std::map<size_type, size_type> &elt_corr);
```

The map elt_corr should contain the correspondences between the elements where the transformation is to be applied and the respective elements where the extrapolation has to be made. On the element not listed in the map, no transformation is applied and the evaluation is performed normally on the current element.

The following functions allow to change the element correspondence of a previously added element extrapolation transformation:

```cpp
set_element_extrapolation_correspondence(
    model, transname, std::map<size_type, size_type> &elt_corr);
set_element_extrapolation_correspondence(
    workspace, transname, std::map<size_type, size_type> &elt_corr);
```

10.35 Evaluating discontinuities across inter-element edges/faces

A specific interpolate transformation (see previous sections), called neighbour_elt is defined by default in all models. This transformation can only be used when a computation is made on an internal edge/face of a mesh, i.e. an element face shared at least by two elements. It aims to compute discontinuity jumps of a variable across inter-element faces. It is particularly suitable to implement Discontinuous Galerkin and interior penalty methods, Ghost penalty terms or a posteriori estimators. The expressions:

```
Interpolate(Normal, neighbour_elt)
Interpolate(X, neighbour_elt)
Interpolate(u, neighbour_elt)
Interpolate(Grad_u, neighbour_elt)
Interpolate(Div_u, neighbour_elt)
Interpolate(Hess_u, neighbour_elt)
Interpolate(Test_u, neighbour_elt)
Interpolate(Grad_Test_u, neighbour_elt)
Interpolate(Div_Test_u, neighbour_elt)
Interpolate(Hess_Test_u, neighbour_elt)
```

are available (as with any other interpolate transformation) and compute a field on the current point but on the neighbour element. Of course, Interpolate(X, neighbour_elt) as no specific interest since it returns the same result as X. Similarly, in most cases, Interpolate(Normal, neighbour elt) will return the opposite of Normal except for instance for 2D shell element in a 3D mesh where it has an interest.

The jump on a variable u can be computed with:

10.35. Evaluating discontinuities across inter-element edges/faces 55
and a penalisation term of the jump can be written:

\[(u-\text{Interpolate}(u, \text{neighbour}_\text{elt}))(\text{Test}_\text{u-Interpolate}(\text{Test}_u, \text{neighbour}_\text{elt}))\]

Note that the region representing the set of all internal faces of a mesh can be obtained thanks to the function:

\[
\text{mr_internal_face} = \text{inner_faces_of_mesh}(\text{my}\_\text{mesh}, \text{mr})
\]

where \(\text{mr}\) is an optional mesh region. If \(\text{mr}\) is specified only the face internal with respect to this region are returned. An important aspect is that each face is represented only once and is arbitrarily chosen between the two neighbour elements.

See for instance interface/tests/python/demo_laplacian_DG.py or interface/tests/matlab/demo_laplacian_DG.m for an example of use.

Compared to other interpolate transformations, this transformation is more optimized and benefits from finite element and geometric transformation pre-computations.

### 10.36 Double domain integrals or terms (convolution - Kernel - Exchange integrals)

In some very special cases, it can be interesting to compute an integral on the direct product of two domains, i.e. a double integral such as for instance

\[
\int_{\Omega_1} \int_{\Omega_2} k(x, y)u(x)v(y)dydx,
\]

where \(k(x, y)\) is a given kernel, \(u\) a quantity defined on \(\Omega_1\) and \(v\) a quantity defined on \(\Omega_2\), eventually with \(\Omega_1\) and \(\Omega_2\) the same domain. This can be interesting either to compute such an integral or to define an interaction term between two variables defined on two different domains.

CAUTION: Of course, this kind of term have to be used with great care, since it naturally leads to fully populated stiffness or tangent matrices.

The weak form language of GetFEM++ furnishes a mechanism to compute such a term. First, the secondary domain has to be declared in the workspace/model with its integration methods. The addition of a standard secondary domain can be done with one of the two following functions:

```
add_standard_secondary_domain(model, domain_name, mim, region);
add_standard_secondary_domain(workspace, domain_name, mim, region);
```

where `model` or `workspace` is the model or workspace where the secondary domain has to be declared, `domain_name` is a string for the identification of this domain together with the mesh region and integration method, `mim` the integration method and `region` a mesh region. Note that with these standard secondary domains, the integration is done on the whole region for each element of the primary domain. It can be interesting to implement specific secondary domains restricting the integration to the necessary elements with respect to the element of the primary domain. A structure is dedicated to this in GetFEM++.
Once a secondary domain has been declared, it can be specified that a weak form language expression has to be assembled on the direct product of a current domain and a secondary domain, adding the name of the secondary domain to the `add_expression` method of the workspace object or using `add_linear_twodomain_term`, `add_nonlinear_twodomain_term` or `add_twodomain_source_term` functions:

```python
workspace.add_expression(expr, mim, region, derivative_order, secondary_domain)
add_twodomain_source_term(model, mim, expr, region, secondary_domain)
add_linear_twodomain_term(model, mim, expr, region, secondary_domain)
add_nonlinear_twodomain_term(model, mim, expr, region, secondary_domain)
```

For the utilisation with the Python/Scilab/Matlab interface, see the documentation on `gf_asm` command and the `model` object.

Inside an expression of the weak form language, one can refer to the unit normal vector to a boundary, to the current position or to the value of a variable thanks to the expressions:

```python
Secondary_domain(Normal)
Secondary_domain(X)
Secondary_domain(u)
Secondary_domain(Grad_u)
Secondary_domain(Div_u)
Secondary_domain(Hess_u)
Secondary_domain(Test_u)
Secondary_domain(Grad_Test_u)
Secondary_domain(Div_Test_u)
Secondary_domain(Hess_Test_u)
```

For instance, a term like

\[ \int_{\Omega_1} \int_{\Omega_1} e^{-\|x-y\|} u(x) u(y) dy dx, \]

would correspond to the following weak form language expression:

```python
exp(Norm(X-Secondary_domain(X)))*u*Secondary_domain(u)
```

### 10.37 Elementary transformations

An elementary transformation is a linear transformation of the shape functions given by a matrix which may depend on the element which is applied to the local degrees of freedom at the element level. An example of definition of elementary transformation can be found in the file `src/getfem_linearized_plates.cc`. It aims for instance to define a local projection of a finite element on a lower level element to perform a reduction such as the one used in MITC elements.

Once a transformation is defined, it can be added to the model/workspace with the method:

```python
model.add_elementary_transformation(transname, pelementary_transformation)
```

where `pelementary_transformation` is a pointer to an object deriving from `virtual_elementary_transformation`. Once it is added to the model/workspace, it is possible to use the following expressions in the weak form language:
Elementary_transformation(u, transname[, dest])
Elementary_transformation(Grad_u, transname[, dest])
Elementary_transformation(Div_u, transname[, dest])
Elementary_transformation(Hess_u, transname[, dest])
Elementary_transformation(Test_u, transname[, dest])
Elementary_transformation(Grad_Test_u, transname[, dest])
Elementary_transformation(Div_Test_u, transname[, dest])
Elementary_transformation(Hess_Test_u, transname[, dest])

where \( u \) is one of the FEM variables of the model/workspace, and \( \text{dest} \) is an optional parameter which should be a variable or data name of the model and will correspond to the target fem of the transformation. If omitted, by default, the transformation is from the fem of the first variable to itself.

A typical transformation is the one for the projection on rotated RT0 element for two-dimensional elements which is an ingredient of the MITC plate element. It can be added thanks to the function (defined in src/getfem/getfem_linearized_plates.h):

\[
\text{add}_2D\text{\_rotated\_RT0\_projection}(\text{model}, \text{transname})
\]

Some other transformations are available for the use into Hybrid High-Order methods (HHO methods, see Tools for HHO (Hybrid High-Order) methods for more information). These transformations correspond to the reconstruction of the gradient of a variable or the variable itself, the HHO methods having separated discretizations on the interior of the element and on its faces. The different transformations can be added with the functions (defined in src/getfem/getfem_HHO.h):

\[
\text{add}_\text{HHO\_reconstructed\_gradient}(\text{model}, \text{transname});
\text{add}_\text{HHO\_reconstructed\_symmetrized\_gradient}(\text{model}, \text{transname});
\text{add}_\text{HHO\_reconstructed\_value}(\text{model}, \text{transname});
\text{add}_\text{HHO\_reconstructed\_symmetrized\_value}(\text{model}, \text{transname});
\text{add}_\text{HHO\_stabilization}(\text{model}, \text{transname});
\text{add}_\text{HHO\_symmetrized\_stabilization}(\text{model}, \text{transname});
\]

### 10.38 Xfem discontinuity evaluation (with mesh\_fem\_level\_set)

When using a fem cut by a level-set (using fem\_level\_set or mesh\_fem\_level\_set objects), it is often interesting to integrate the discontinuity jump of a variable, or the jump in gradient or the average value. For this purpose, the weak form language furnishes the following expressions for \( u \) a FEM variable:

\[
\text{Xfem\_plus}(u)
\text{Xfem\_plus}(\text{Grad}\_u)
\text{Xfem\_plus}(\text{Div}\_u)
\text{Xfem\_plus}(\text{Hess}\_u)
\text{Xfem\_plus}(\text{Test}\_u)
\text{Xfem\_plus}(\text{Test\_Grad}\_u)
\text{Xfem\_plus}(\text{Test\_Div}\_u)
\text{Xfem\_plus}(\text{Test\_Hess}\_u)
\text{Xfem\_minus}(u)
\text{Xfem\_minus}(\text{Grad}\_u)
\text{Xfem\_minus}(\text{Div}\_u)
\text{Xfem\_minus}(\text{Hess}\_u)
\]

(continues on next page)
Xfem_minus(Test_u)
Xfem_minus(Test_Grad_u)
Xfem_minus(Test_Div_u)
Xfem_minus(Test_Hess_u)

which are only available when the evaluation (integration) is made on the curve/surface separating two zones of continuity, i.e. on the zero level-set of a considered level-set function (using a mesh_im_level_set object). For instance, a jump in the variable $u$ will be given by:

Xfem_plus(u) - Xfem_minus(u)

and the average by:

$\frac{Xfem_plus(u) + Xfem_minus(u)}{2}$

The value $Xfem_plus(u)$ is the value of $u$ on the side where the corresponding level-set function is positive and $Xfem_minus(u)$ the value of $u$ on the side where the level-set function is negative.

Additionally, note that, when integrating on a level-set with a mesh_im_level_set object, Normal stands for the normal unit vector to the level-set in the direction of the gradient of the level-set function.

10.39 Storage of sub-expressions in a getfem::im_data object during assembly

It is possible to store in a vector depending on a getfem::im_data object a part of an assembly computation, for instance in order to use this computation in another assembly. This is an alternative to the interpolation functions which allows not to compute twice the same expression.

The method to add such an assignment in the assembly is the following for a model or a ga_workspace:

```cpp
model.add_assembly_assignments(dataname, expr, region = size_type(-1),
 order = 1, before = false);
workspace.add_assignment_expression(dataname, expr,
 region = mesh_region::all_convexes(), order = 1, before = false)
```

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

Additionally, in a model, the method:

```cpp
model.clear_assembly_assignments()
```

allows to cancel all the assembly assignments previously added.
CHAPTER 11

Compute arbitrary terms - low-level generic assembly procedures

This section presents the first version of generic assembly procedure which has been implemented in GetFEM++. It allows to easily make the assembly of arbitrary matrices in the linear case. In the nonlinear case, some special “non_linear_term” object have to be implemented, which could be a bit tricky and obliges to use very low-level internal tools of GetFEM++. The high-level generic assembly has been developed to circumvent these difficulties (see Compute arbitrary terms - high-level generic assembly procedures - Weak-form language).

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

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

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

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

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

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

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

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

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

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

where \(\varphi^j\) is a \(j\)th component of the \(i\)th base function of \(mf\) and \(\psi^p\) is a (scalar) base function of the second mesh fem. Since we want to assemble

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

the reduction is:

\[
\sum_{j,k,p} \left( \int \partial_k \varphi^j \partial_m \psi^p \right) a^p
\]

In the \(\text{comp}\) function, \(\text{vGrad}\) was used instead of \(\text{Grad}\) since we said that we were assembling a vector Laplacian: that is why each \(\text{vGrad}\) part has three dimensions (dof number, component number, and derivative number). For a scalar Laplacian, we could have used \(\text{comp(Grad(#1).Grad(#1).Base(#2))}(:,k,:,k,p).a(p)\). But the vector form has the advantage to work in both vector and scalar case.

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

The third example shows how to compute the \(L^2\) norm of a scalar or vector field on a mesh boundary:
assem.push_im(mim);
assem.push_mf(mf);
assem.push_data(U);
std::vector<scalar_type> v(1);
assem.push_vec(v);
assem.set("u=data(#1);
    V()+=u(i).u(j).comp(vBase(#1).vBase(#1))(i,k,j,k)";
assem.assembly(boundary_number);

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

\[ \sum_{i,j,k} \left( \int_{\text{boundary}} u_i \varphi_i^k u_j \varphi_j^k \right) \]

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

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

The original equations are:

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

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

### 11.1 available operations inside the \texttt{comp} command

- \texttt{Base(#i)}: evaluate the value of the base functions of the \textit{ith} mesh\_fem
- \texttt{Grad(#i)}: evaluate the value of the gradient of the base functions of the \textit{ith} mesh\_fem
- \texttt{Hess(#i)}: evaluate the value of the Hessian of the base functions of the \textit{ith} mesh\_fem
- \texttt{Normal()}: evaluate the unit normal (should not be used for volumic integrations !)
- \texttt{NonLin$x(#mf1, \ldots, #mfn)}: evaluate the \textit{xth} non-linear term (inserted with \texttt{push_nonlinear_term(pnonlinear_elem_term)}) using the listed mesh\_fem objects.
- \texttt{GradGT()}, \texttt{GradGTInv}(): evaluate the gradient (and its inverse) of the geometric transformation of the current convex.
Note: you may reference any data object inside the `comp` command, and perform reductions inside the `comp()`. This feature is mostly interesting for speeding up assembly of nonlinear terms (see the file `getfem/getfem_nonlinear_elasticity.h` for an example of use).

11.2 others operations

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

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

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

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

If there is more than one data array, output array or output sparse matrix, one can use `data$2`, `data$3`, `V$2`, `M$2`,...
Some Standard assembly procedures (low-level generic assembly)

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

CAUTION: The assembly procedures do not clean the matrix/vector at the beginning of the assembly in order to keep the possibility to perform several assembly operations on the same matrix/vector. Consequently, one has to clean the matrix/vector before the first assembly operation.

12.1 Laplacian (Poisson) problem

An assembling procedure is defined to solve the problem:

\[
-\text{div}(a(x) \cdot \text{grad}(u(x))) = f(x) \quad \text{in } \Omega,
\]
\[
u(x) = U(x) \quad \text{on } \Gamma_D,
\]
\[
\frac{\partial u}{\partial \eta}(x) = F(x) \quad \text{on } \Gamma_N,
\]

where \( \Omega \) is an open domain of arbitrary dimension, \( \Gamma_D \) and \( \Gamma_N \) are parts of the boundary of \( \Omega \), \( u(x) \) is the unknown, \( a(x) \) is a given coefficient, \( f(x) \) is a given source term, \( U(x) \) the prescribed value of \( u(x) \) on \( \Gamma_D \) and \( F(x) \) is the prescribed normal derivative of \( u(x) \) on \( \Gamma_N \). The function to be called to assemble the stiffness matrix is:

\[
\texttt{getfem::asm\_stiffness\_matrix\_for\_laplacian(SM, mim, mfu, mfd, A)};
\]

where

- \texttt{SM} is a matrix of any type having the right dimension (i.e. \texttt{mfu.nbdof()}),
- \texttt{mim} is a variable of type \texttt{getfem::mesh\_im} defining the integration method used,
- \texttt{mfu} is a variable of type \texttt{getfem::mesh\_fem} and should define the finite element method for the solution,
• mfd is a variable of type `getfem::mesh_fem` (possibly equal to mfu) describing the finite element method on which the coefficient $a(x)$ is defined,

• A is the (real or complex) vector of the values of this coefficient on each degree of freedom of mfd.

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

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

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

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

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

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

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

is used again, with nbound is the index of the boundary $\Gamma_N$ in the linked mesh of mim, mfu and mfd.

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

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

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

$$ [SM]U = B, $$

where $U$ is the discrete unknown.

For the second manner, one should use the more general:

```cpp
getfem::asm_dirichlet_constraints(H, R, mim, mf_u, mf_mult, mf_r, r, nbound).
```

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

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

Then, one should call:

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

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

\[
(N^t[SM]N)U_{int} = N^t(B - [SM]U_d),
\]

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

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

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

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

### 12.2 Linear Elasticity problem

The following function assembles the stiffness matrix for linear elasticity:

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

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

**Caution:** Linear elasticity problem is a vectorial problem, so the target dimension of \(mfu\) (see \(mf.set\_qdim(Q)\)) should be the same as the dimension of the mesh.

In order to assemble source term, Neumann and Dirichlet conditions, same functions as in previous section can be used.
12.3 Stokes Problem with mixed finite element method

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

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

12.4 Assembling a mass matrix

Assembly of a mass matrix between two finite elements:

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

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

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

where \( nbound \) is the region index in \( mim.linked_mesh() \), or a `mesh_region` object.
Once a solution has been computed, it is quite easy to extract any quantity of interest on it with the interpolation functions for instance for post-treatment.

## 13.1 Basic interpolation

The file `getfem/getfem_interpolation.h` defines the function

```cpp
getfem::interpolation(...) to interpolate a solution from a given mesh/finite element method on another mesh and/or another Lagrange finite element method:
```

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

where `mf1` is a variable of type `getfem::mesh_fem` and describes the finite element method on which the source field `U` is defined, `mf2` is the finite element method on which `U` will be interpolated. `extrapolation` is an optional parameter. The values are 0 not to allow the extrapolation, 1 for an extrapolation of the exterior points near the boundary and 2 for the extrapolation of all exterior points (could be expensive).

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

... important:

```
```

If you need to perform more than one interpolation between the same finite element methods, it might be more efficient to use the function:
getfem::interpolation(mf1, mf2, M, extrapolation = 0);

where \(M\) is a row matrix which will be filled with the linear map representing the interpolation (i.e. such that \(V = MU\)). The matrix should have the correct dimensions (i.e. \(mf2.nb\_dof() \times mf1.nb\_dof()\)). Once this matrix is built, the interpolation is done with a simple matrix multiplication:

gmm::mult(M, U, V);

### 13.2 Interpolation based on the high-level weak form language

It is possible to extract some arbitrary expressions on possibly several fields thanks to the weak form language and the interpolation functions.

This is specially dedicated to the model object (but it can also be used with a ga_workspace object). For instance if \(md\) is a valid object containing some defined variables \(u\) (vectorial) and \(p\) (scalar), one can interpolate on a Lagrange finite element method an expression such as \(p \cdot \text{Trace}(\text{Grad}_u)\). The resulting expression can be scalar, vectorial or tensorial. The size of the resulting vector is automatically adapted.

The high-level generic interpolation functions are defined in the file `getfem/getfem_generic_assembly.h`.

There is different interpolation functions corresponding to the interpolation on a Lagrange fem on the same mesh, the interpolation on a cloud on points or on a `getfem::im_data` object.

**Interpolation on a Lagrange fem:**

```cpp
void getfem::ga_interpolation_Lagrange_fem(workspace, mf, result);
```

where `workspace` is a `getfem::ga_workspace` object which aims to store the different variables and data (see [Compute arbitrary terms - high-level generic assembly procedures - Weak-form language](#)), `mf` is the `getfem::mesh_fem` object reresenting the Lagrange fem on which the interpolation is to be done and `result` is a `beot::base_vector` which store the interpolatin. Note that the workspace should contain the epression to be interpolated.

```cpp
void getfem::ga_interpolation_Lagrange_fem(md, expr, mf, result, rg=mesh_region::all_convexes());
```

where `md` is a `getfem::model` object (containing the variables and data), `expr` (std::string object) is the expression to be interpolated, `mf` is the `getfem::mesh_fem` object reresenting the Lagrange fem on which the interpolation is to be done, `result` is the vector in which the interpolation is stored and `rg` is the optional mesh region.

**Interpolation on a cloud of points:**

```cpp
void getfem::ga_interpolation_mti(md, expr, mti, result, extrapolation = 0, rg=mesh_region::all_convexes(), nbpoints = size_type(-1));
```

where `md` is a `getfem::model` object (containing the variables and data), `expr` (std::string object) is the expression to be interpolated, `mti` is a `getfem::mesh_trans_inv` object which stores the cloud of points (see `getfem/getfem_interpolation.h`), `result` is the vector in which the interpolation is stored, `extrapolation` is an option for extrapolating the field outside the mesh for outside points, `rg` is the optional mesh region and `nbpoints` is the optional maximal number of points.
Interpolation on an im_data object (on the Gauss points of an integration method):

```cpp
void getfem::ga_interpolation_im_data(md, expr, im_data &imd,
    base_vector &result, const mesh_region &rg=mesh_region::all_convexes());
```

where `md` is a `getfem::model` object (containing the variables and data), `expr` (std::string object) is the expression to be interpolated, `imd` is a `getfem::im_data` object which refers to a integration method (see getfem/getfem_im_data.h), `result` is the vector in which the interpolation is stored and `rg` is the optional mesh region.

**13.2. Interpolation based on the high-level weak form language**
Incorporate new finite element methods in *GetFEM++*

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

Please read dp for more details and see the files `getfem/getfem_fem.h`, `getfem_fem.cc` for practical implementation.
Chapter 14. Incorporate new finite element methods in GetFEM++
Incorporate new approximated integration methods in *GetFEM++*

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

```plaintext
NAME = IM_TETRAHEDRON(5)
N = 3
GEOTRANS = GT_PK(3,1)
NBPT = 4
0, 0.25, 0.25, 0.25, 0.008818342151675485
1, 0.31979362782962991, 0.31979362782962991, 0.31979362782962991, 0.
   0.0115113678104538
1, 0.09197107805273033, 0.09197107805273033, 0.09197107805273033, 0.
   0.0119895139631697
1, 0.056350832689629156, 0.056350832689629156, 0.44364916731037084, 0.
   0.008818342151675485
NBF = 4 IM_TRIANGLE(5)
IM_TRIANGLE(5)
IM_TRIANGLE(5)
```

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

Composition of the integration node definition:

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

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

The file format is inspired from [EncylOpCubature].
Since v2.0, GetFEM++ offers a certain number of facilities to support Xfem and fictitious domain methods with a cut-fem strategy. Most of these tools have been initially mainly developed by Julien Pommier for the study published in [LA-PO-RE-SA2005].

The implementation is a fairly large generality, based on the use of level-sets, as suggested in [SU-CH-MO-BE2001] and allows simultaneous use of a large number of level-sets which can cross.

The Xfem implementation for the discretization of the jump follows the strategy of [HA-HA2004] although we had no knowledge of this work during implementation. This means that there is no degree of freedom representing the jump across the level-set. Instead, the degrees of freedom represent the displacement of each side of the level-set. This is essential in any way in the presence of level-set that intersect each other because it may exist more than two different zones of continuity inside a single element.

The cut fem strategy for fictitious domain method has been used for the first time with GetFEM++ for the study published in [HA-RE2009] where a quite simple stabilization strategy is proposed. Here also, before knowing the existence of the Work of E. Burman and P. Hanbo [bu-ha2010] on that topic.

The tools for Xfem have been then enriched by the PhD works of J. Larsy (see for instance [LA-RE-SA2010]) the one of E. Chahine (see for instance [CH-LA-RE2011], [NI-RE-CH2011]), of S. Amdouni (see for instance [AM-MO-RE2014], [AM-MO-RE2014b]) and of M. Fabre (see for instance [Fa-Po-Re2015]).

**Important:** All the tools listed below needs the package qhull installed on your system. This package is widely available. It computes convex hull and Delaunay triangulations in arbitrary dimension.

The programs tests/crack.cc, interface/tests/matlab/crack.m and interface/tests/python/crack.py are some good examples of use of these tools.
16.1 Representation of level-sets

Some structure are defined to manipulate level-set functions defined by piecewise polynomial function on a mesh. In the file `getfem/getfem_levelset.h` a level-set is represented by a function defined on a Lagrange fem of a certain degree on a mesh. The constructor to define a new `getfem::level_set` is the following:

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

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

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

Note that, in applications, the level-set function often evolves thanks to an Hamilton-Jacobi equation (for its re-initialization for instance). See the A pure convection method which can be used in the approximation of a Hamilton-Jacobi equation.

16.2 Mesh cut by level-sets

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

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

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

In particular a subdivision of each element cut by the level-set is made with simplices. Note that the whole cut-mesh is generally not conformal.

The cut-mesh can be obtained for instance for post-treatment thanks to `mls.global_cut_mesh(m)` which fill `m` with the cut-mesh.
16.3 Adapted integration methods

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

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

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

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

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

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

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

When more than one level-set is declared on the `getfem::mesh_level_set` object, it is possible
to set more precisely the integration domain using the method:

```cpp
mim.set_level_set_boolean_operations("desc");
```

where “desc” is a string containing the description of the boolean operation which defines the integration
domain. The syntax is simple, for example if there are 3 different level-set,

- “a*b*c” is the intersection of the domains defined by each level-set (this is the default
  behavior if this function is not called).
- “a+b+c” is the union of their domains.
- “c-(a+b)” is the domain of the third level-set minus the union of the domains of the two
  others.
- “!a” is the complementary of the domain of a (i.e. it is the domain where \( a(x)>0 \))

The first level-set is always referred to with “a”, the second with “b”, and so on.
16.4 Cut-fem

The implementation of a cut finite element method such as described in [bu-ha2010], i.e. a finite element on a fictitious domain restricted to a smaller real domain, is possible just using the previous tools and mainly the adapted integration method. Several examples are available on *GetFEM++* test programs. See for instance `interface/tests/python/demo_fictitious_domain.py` or `interface/tests/matlab/demo_fictitious_domain.m`.

In this context, one often needs to restrict the unknown finite element field to the degrees of freedom whose corresponding shape function supports have an intersection with the real domain. This can be done using the `partial_mesh_fem` object. See for instance `interface/tests/matlab/demo_structural_optimization.m`.

Note that often, a stabilization technique have to be considered in order to treat eventual locking phenomena due to element with very small intersection with the real domain for example when applying a Dirichlet condition. See for instance [bu-ha2010], [HA-RE2009] and [Fa-Po-Re2015].

16.5 Discontinuous field across some level-sets

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

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

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

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

To represent discontinuous fields, the finite element method is enriched with discontinuous functions which are the product of some Heaviside functions by the shape functions of the finite element method represented by `mf` (see [HA-HA2004] and [Xfem] for more details).

16.6 Xfem

The Xfem (see [Xfem]) consists not only in the enrichment with some Heaviside functions (which is done by the object `getfem::mesh_fem_level_set`) but also the enrichment with asymptotic displacement at the crack tip. There is several manner to enrich with an asymptotic displacement: enrichment only on the element containing the crack tip as in [Xfem], enrichment in a fixed size zone as in [LA-PO-RE-SA2005] or [Be-Mi-Mo-Bu2005], enrichment with a cut-off function as in [CH-LA-RE2008] or [NI-RE-CH2011] or with an integral matching condition between the enriched and non-enriched zones as in [CH-LA-RE2011]. The choice in Getfem fell on maximum flexibility to easily implement all possibilities. As it is mainly a transformation of the finite element method itself, two tools have been defined to produce some enriched finite elements:

```
getfem::mesh_fem_product mf_asympt(mf_part_unity, mf_sing)
getfem::mesh_fem_sum mf_sum(mf1, mf2)
```
where \( \text{mf\_sing} \) should be a global ‘finite element method’, in fact just a collection of global functions (with or without a cut-off function) defined thanks to the object \texttt{getfem::mesh\_fem\_global\_function} (see the file \texttt{src/getfem/getfem\_mesh\_fem\_global\_function.h}) and \( \text{mf\_part\_unity} \) a basic scalar finite element method. The resulting ‘getfem::mesh\_fem\_product’ is the linear combination of all the product of the shape function of the two given finite element methods, possibly restricted to a sub-set of degrees of freedom of the first finite element method given by the method \texttt{mf\_asympt\_set\_enrichment(\text{enriched\_dofs})}.

Once the asymptotic enrichment is defined, the object \texttt{getfem::mesh\_fem\_sum} allows to produce the direct sum of two finite element methods. For instance of the one enriched by the Heaviside functions (\texttt{getfem::mesh\_fem\_level\_set} object) and the asymptotic enrichment.

See \texttt{interface/tests/matlab/demo\_crack.m}, \texttt{interface/tests/python/demo\_crack.py} or \texttt{tests/crack.cc} for some examples of use of these tools.

Additionally, the weak form language defines the two commands \texttt{Xfem\_plus} and \texttt{Xfem\_minus} allowing to take into account the jump of any field or derivative of any field across a level-set (see \texttt{Xfem\_discontinuity\_evaluation\_with\_mesh\_fem\_level\_set}). This a priori allows to write any interface law easily.

Note also that some procedures are available in the file \texttt{src/getfem/getfem\_crack\_sif.h} to compute the stress intensity factors in 2D (restricted to homogeneous isotropic linearized elasticity).

### 16.7 Post treatment

Several tools are available to represent the solution only on a side of a levels-set or on both taking into account the discontinuity (for Xfem approximation).

When a cut-mesh \texttt{mls} is used (i.e. a \texttt{getfem::mesh\_level\_set} object), is is possible to obtain the set of all sub-elements with the command:

\[
\texttt{mls.global\_cut\_mesh(mcut);} 
\]

where \texttt{mcut} has to be an empty mesh which will be fill by the sub-elements. Note that the resulting mesh is a non-regular one in the sense that the sub-mesh of all elements are not conformal at the element edges/faces. It is however possible to interolate on a Lagrange fem on this mesh and make a post-treatment with it to correctly represent a discontinuous field.

Another mean to represent only the interesting part of the solution when a fictitious domain method is used is to use the mesh slices defined by an isovalue level-set (see \texttt{Producing\_mesh\_slices}).

see for instance files \texttt{interface/tests/matlab/demo\_crack.m}, \texttt{interface/tests/python/demo\_fictitious\_domain.py} and \texttt{interface/tests/matlab/demo\_structural\_optimization.m}. 

16.7. Post treatment 81
Tools for HHO (Hybrid High-Order) methods

HHO method are hybrid methods in the sense that they have both degrees of freedom located on the element of a mesh and on the faces of the elements which represent separated approximations. HHO method are primal methods in the sense that both the degree of freedom in the element and on the faces represent the main unknown of the problem (no lagrange multipliers is introduced). The interest of these methods, first developped in [Di-Er2015], [Di-Er2017] is their accuracy and their great robustness, in particular with respect to the element shapes and their locking-free properties. Moreover, they can be extended without difficulty to the approximation of nonlinear problems (see [AB-ER-PI2018] for hyper-elasticity, [AB-ER-PI2019] for plasticity and [ca-ch-er2019] for contact problems).

HHO methods can be applied to arbitrary shape elements. However, the implementation in GetFEM++ is for the moment limited to standard elements: simplices, quadrilaterals, hexahedrons, ... Moreover this implementation is still experimental and not pretending to optimality. For the moment, there is no tool to make an automatic condensation of internal dofs.

17.1 HHO elements

HHO elements are composite ones having a polynomial approximation space for the interior of the element and a polynomial approximation for each face of the element. Moreover, this is a discontinuous approximation, in the sense that no continuity is prescribed between the approximation inside the element and the approximation on the faces, neither than between the approximations on two different faces of the element. However, when two neighbour elements share a face, the approximation on this face is shared by the two elements. GetFEM++ provide a specific method simply called FEM_HHO(fem_int, fem_face1, fem_face2, ...) which allows to build an hybrid method from standard finite element spaces. For instance, on a triangle, a possible HHO method can be obtained with:

```cpp
getfem::pfem pf = getfem::fem_descriptor("HHO(FEM_SIMPLEX_IPK(2,2), FEM_SIMPLEX_CIPK(1,2))");
```

The first argument to FEM_HHO(...) is the fem for the interior of the element. It has to be a discontinuous FEM. The method FEM_SIMPLEX_IPK(2,2) is a discontinuous method having its degrees of freedom in the strict interior of the element, which ensure that no dof identification will be done. The
second argument is the fem for the faces (if only one method is given, it will be applied to all faces, but it is also possible to give a different method for each face). Their is no verification on the fact that the given method are of discontinuous type (In fact, a method like FEM_HHO(FEM_PK(2,2), FEM_PK(1, 2)) will have no difference with FEM_PK(2,2) since the degree of freedom on the faces will be identified with the interior ones).

For the moment, the furnished element for interior and faces are - FEM_SIMPLEX_IPK(n, k) : interior PK element of degree k for the simplices in dimension n (equivalent to FEM_PK_DISCONTINUOUS(n, k, 0.1)). - FEM_QUAD_IPK(n, k) : interior PK element of degree k for the quadrilaterals in dimension n. - FEM_PRISM_IPK(n, k) : interior PK element of degree k for the prisms in dimension n. - FEM_SIMPLEX_CIPK(n, k) : interior PK element on simplices which is additionally connectable. Designed to be used on HHO element face. - FEM_QUAD_CIPK(k) : interior PK element on a quadrilateral which is additionally connectable. Designed to be used on HHO element face.

17.2 Reconstruction operators

For a variable $u$, we will note $u_T$ its value in the interior of the element $T$ and $u_{\partial T}$ its value on the boundary of $T$ (corresponding to the two different approximations). The reconstruction operators are implemented in GetFEM++ as elementary transformations, as described in the section Elementary transformations.

17.2.1 Reconstructed gradient

The first reconstruction operator is the reconstructed gradient. Given a certain polynomial space $V_G$, the reconstructed gradient $G(u)$ will be the solution to the local problem

$$\int_T G(u) : \tau dx = \int_T \nabla u_T : \tau dx + \int_{\partial T} (u_{\partial T} - u_T). (\tau n_T) d\Gamma, \forall \tau \in V_G$$

where $n_T$ is the outward unit normal to $T$ on $\partial T$. Note that the space $V$ is a vector-valued one if $u$ is a scalar field variable (in that case, $G(u) : \tau$ reduces to $G(u) \tau$) and a matrix-valued one if $u$ is a vector field variable.

In order to be used, the elementary transformation corresponding to this operator has first to be added to the model by the command:

```cpp
add_HHO_reconstructed_gradient(model, transname);
```

where `transname` is an arbitrary name which will designate the transformation in the weak form language. Then, it will be possible to refer to the reconstructed gradient of a variable $u$ into the weak form language as `Elementary_transformation(u, HHO_grad, Gu)`, if `transname="HHO_grad"`. The third parameter of the transformation $Gu$ should be a fem variable or a data of the model. This variable will not be used on itself but will determine the finite element space of the reconstruction (the space $V_G$).

This is an example of use with the Python interface for a two-dimensional triangule mesh $m$

```python
mfu = gf.MeshFem(m, 1)
mfgu = gf.MeshFem(m, N)
mfu.set_fem(gf.Fem('FEM_HHO(FEM_SIMPLEX_IPK(2,2),FEM_SIMPLEX_CIPK(1,2))'))
mfgu.set_fem(gf.Fem('FEM_PK(2,2)'))
```

(continues on next page)
The macro definitions allowing to use the gradient of the variable inside weak formulations as usual. For instance, the addition of a weak term for the Laplace equation can then be simply written:

```
md.add_linear_term(mim, 'HHO_Grad_u.HHO_Grad_Test_u')
```

Two complete examples of use are given in the test programs `interface/tests/demo_laplacian_HHO.py` and `interface/tests/demo_elasticity_HHO.py`.

### 17.2.2 Reconstructed symmetrized gradient

The symmetrized gradient is only for vector field variables and additionally when the vector field dimension is the same as the domain dimension. This is usually the case for instance for elasticity problems. With the same notation as in the previous section, the reconstructed gradient $G^s(u)$ will be the solution to the local problem

$$
\int_T G^s(u) : \tau dx = \int_T \nabla^s u_T : \tau dx + \int_{\partial T} (u_{\partial T} - u_T) \cdot (\tau^s n_T) d\Gamma, \quad \forall \tau \in V_G
$$

where $\nabla^s u_T = (\nabla u_T + (\nabla u_T)T)/2$ and $\tau^s = (\tau + \tau^T)/2$.

The elementary transformation corresponding to this operator can be added to the model by the command:

```
add_HHO_reconstructed_symmetrized_gradient(model, transname);
```

and then be used into the weak form language as `Elementary_transformation(u, HHO_sym_grad, Gu), if transname="HHO_sym_grad", with Gu still determining the reconstruction space`.

### 17.2.3 Reconstructed variable

A reconstruction of higher order can be done using both the approximation on the interior and the approximation on the faces. The reconstructed variable $D(u)$ will be the solution to the local Neumann problem on a chosen space $V_D$

$$
\int_T \nabla D(u) \cdot \nabla v dx = \int_T \nabla u_T \cdot \nabla v dx + \int_{\partial T} (u_{\partial T} - u_T) \cdot (\nabla v n_T) d\Gamma, \quad \forall v \in V_D
$$

with the additional constraint $\int_T D(u) dx = \int_T u_T dx$

The corresponding elementary transformation can be added to the model by the command:
add_HHO_reconstructed_value(model, transname);

and used into the weak form language as Elementary_transformation(u, HHO_val, ud), if transname="HHO_val", with ud determining the reconstruction space.

### 17.2.4 Reconstructed variable with symmetrized gradient

A variant of the reconstruction of a variable is the one using a symmetrized gradient. It can be used only for vector field variables and additionally when the vector field dimension is the same as the domain dimension. The reconstructed variable \(D(u)\) will be the solution to the local Neumann problem on a chosen space \(V_D\)

\[
\int_T \nabla^s D(u). \nabla^s v dx = \int_T \nabla^s u_T. \nabla^s v dx + \int_{\partial T} (u_{\partial T} - u_T). (\nabla^s v n_T) d\Gamma, \ \forall v \in V_D
\]

with the additional constraints

\[
\int_T D(u) dx = \int_T u_T dx \\
\int_T \text{Skew}(\nabla D(u)) dx = \int_{\partial T} (n_T \otimes u_{\partial T} - u_{\partial T} \otimes n_T)/2d\Gamma
\]

where \(\text{Skew}(\nabla D(u)) = (\nabla D(u) - (\nabla D(u))^T)/2\).

The corresponding elementary transformation can be added to the model by the command:

add_HHO_reconstructed_value(model, transname);

and used into the weak form language as Elementary_transformation(u, HHO_val, ud), if transname="HHO_val", with ud determining the reconstruction space.

### 17.3 Stabilization operators

The stabilization operators is an operator that measure in a sense the discontinuity of the approximation. A stabilization is obtained by a penalization term using this operator. The stabilization operator \(S(u)\) is defined on the boundary space \(V_{\partial T}\) of the element, with the formula

\[
S(u) = \Pi_{\partial T}(u_{\partial T} - D(u)) - \Pi_T(u_T - D(u))
\]

where \(D(u)\) is the reconstruction operator on a polynomial space one degree higher that the finite element space used for the variable, \(\Pi_{\partial T}\) is the \(L^2\) projection onto the space of the face approximations and \(\Pi_T\) the \(L^2\) projection onto the space of the interior of the element.

For vector field variables having the same dimension as the domain, there exists also a stabilization operator using the symmetrized gradient, which is defined by

\[
S^s(u) = \Pi_{\partial T}(u_{\partial T} - D^s(u)) - \Pi_T(u_T - D^s(u))
\]

The corresponding elementary transformations can be added to the model by the two commands:

add_HHO_stabilization(model, transname);
add_HHO_symmetrized_stabilization(model, transname);
and used into the weak form language as `Elementary_transformation(u, HHO_stab), if transname="HHO_stab"`. A third argument is optional to specify the target (HHO) space (the default is one of the variable itself). An example of use is also given in the test programs `interface/tests/demo_laplacian_HHO.py` and `interface/tests/demo_elasticity_HHO.py`.
A special finite element method is defined in `getfem/getfem_interpolated_fem.h` which is not a real finite element method, but a pseudo-fem which interpolates a finite element method defined on another mesh. If you need to assemble a matrix with finite element methods defined on different meshes, you may use the “interpolated fem” or “projected fem” for that purpose:

```cpp
// interpolation within a volume
getfem::new_interpolated_fem(getfem::mesh_fem mf, getfem::mesh_im mim);
// projection on a surface
getfem::new_projected_fem(getfem::mesh_fem mf, getfem::mesh_im mim);
```

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

The interpolation is made on each Gauss point of the integration methods of `mim`, so only this integration method can be used in assembly procedures.

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

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

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

Instead of using the previous tools (interpolated and projected fems), it is possible to use a finite element variable defined on another mesh than the one on which an assembly is computed using the “interpolate transformation” tool of the weak form language (see Interpolate transformations), the finite element variables will be interpolated on each Gauss point. There is no restriction on the dimensions of the mesh used, which means in particular that a two-dimensional fem variable can be interpolated on a one-dimensional mesh (allowing the coupling of shell and beam elements, for instance). It is also possible to use some transformations like polar coordinates to euclidean ones.

18.2 mortar methods

Mortar methods are supported by GetFEM++. The coupling term between non matching meshes can in particular be computed using the interpolate transformations of the weak form language (see Interpolate transformations).
CHAPTER 19

Compute $L^2$ and $H^1$ norms

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

```cpp
getfem::asm_L2_norm(mim, mf, U, region = mesh_region::all_convexes());
getfem::asm_H1_semi_norm(mim, mf, U, region = mesh_region::all_convexes());
getfem::asm_H1_norm(mim, mf, U, region = mesh_region::all_convexes());
```

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

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

```cpp
getfem::asm_L2_dist(mim, mf1, U1, mf2, U2, region = mesh_region::all_convexes());
getfem::asm_H1_dist(mim, mf1, U1, mf2, U2, region = mesh_region::all_convexes());
```

These functions return the $L^2$ and $H^1$ norms of $u_1 - u_2$. 
Chapter 19. Compute $L^2$ and $H^1$ norms
CHAPTER 20

Compute derivatives

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

```cpp
gtetfem::compute_gradient(mf1, mf2, U, V);
```

where `mf1` is a variable of type `mesh_fem` and describes the finite element method on which the solution is defined, `mf2` describes the finite element method to compute the gradient, `U` is a vector representing the solution and should be of size `mf1.nb_dof()`, `V` is the vector on which the gradient will be computed and should be of size `N * mf2.nb_dof()`, with `N` the dimension of the domain.
Export and view a solution

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

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

The objects that can be exported are, mesh, mesh_fem objects, and stored_mesh_slice.

21.1 Saving mesh and mesh_fem objects for the Matlab interface

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

```cpp
std::fstream f("solution.U",std::ios::out);
for (unsigned i=0; i < gmm::vect_size(U); ++i)
    f << U[i] << \verb+\n+;
// when the 2nd arg is true, the mesh is saved with the |mf|
mf.write_to_file("solution.mf", true);
```

and then, under Matlab:

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

See the getfem-matlab interface documentation for more details.
Two other file formats are supported for export: the VTK file format, the OpenDX file format and the Gmsh post-processing file format. Both can export either a `getfem::mesh` or `getfem::mesh_fem`, but also the more versatile `getfem::stored_mesh_slice`. Examples of use can be found in the examples of the tests directory.

## 21.2 Producing mesh slices

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

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

```cpp
getfem::slicer_none()
  empty slicer.
getfem::slicer_boundary(const mesh &m, ldots)
  extract the boundary of a mesh.
getfem::slicer_apply_deformation(mesh_slice_cv_dof_data_base &)
  apply a deformation to the mesh , the deformation field is defined on a mesh_fem.
getfem::slicer_half_space(base_node x0, base_node n, int orient)
  cut the mesh with a half space (if orient = -1 or +1), or a plane (if orient = 0), x0 being a node of the plane, and n being a normal of the plane.
getfem::slicer_sphere(base_node x0, scalar_type R, int orient)
  cut with the interior (orient``=-1), boundary (``orient``=0) or exterior (``orient``=+1) or a sphere of center ``x0 and radius R.
getfem::slicer_cylinder(base_node x0, base_node x1, scalar_type R, int orient)
  slice with the interior/boundary/exterior of a cylinder of axis (x0,x1) and radius R.
getfem::slicer_isovalue(const mesh_slice_cv_dof_data_base& mfU, scalar_type val, int orient)
  cut with the isosurface defined by the scalar field mfU and val. Keep only simplices where :math:`u(x) < val` (``orient``=-1), :math:`u(x)=val` (``orient``=0 or `u(x) > val.`
getfem::slicer_mesh_with_mesh(const mesh &m2)
  cut the convexes with the convexes of the mesh m2.
getfem::slicer_union(const slicer_action &sA, const slicer_action &sB)
  merges the output of two slicer operations.
getfem::slicer_intersect(slicer_action &sA, slicer_action &sB)
  intersect the output of two slicer operations.
```
\texttt{getfem::slicer\_complementary(slicer\_action \&s)}

return the complementary of a slicer operation.

\texttt{getfem::slicer\_build\_edges\_mesh(mesh\& \textit{edges\_m})}

slicer whose side-effect is to build the mesh \textit{edges\_m} with the edges of the sliced mesh.

\texttt{getfem::slicer\_build\_mesh(mesh \&m)}

in some (rare) occasions, it might be useful to build a mesh from a slice. Note however that there is absolutely no guaranty that the mesh will be conformal (although it is often the case).

\texttt{getfem::slicer\_build\_stored\_mesh\_slice(stored\_mesh\_slice\& \textit{sl})}

record the output of the slicing operation into a \textit{stored\_mesh\_slice} object. Note that it is often more convenient to use the \textit{stored\_mesh\_slice::build(\ldots)} method to achieve the same result.

\texttt{getfem::slicer\_explode(c)}

shrink or expand each convex with respect to its gravity center.

In order to apply these slicers, a \texttt{getfem::mesh\_slicer(mesh\&)} object should be created, and the \texttt{getfem::slicer\_action} are then stacked with \texttt{mesh\_slicer::push\_back\_action(slicer\_action\&)} and \texttt{mesh\_slicer::push\_front\_action(slicer\_action\&)}. The slicing operation is finally executed with \texttt{mesh\_slicer::exec(int \textit{nrefine})} (or \texttt{mesh\_slicer::exec(int \textit{nrefine}, const mesh\_region \&c\textit{cvlst})} to apply the operation to a subset of the mesh, or its boundary etc.).

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

Example of use (cut the boundary of a mesh \textit{m} with a half-space, and save the result into a \textit{stored\_mesh\_slice}): \[
\text{getfem::slicer\_boundary a0(m);}
\text{getfem::slicer\_half\_space a1(base\_node(0,0), base\_node(1, 0), -1);}
\text{getfem::stored\_mesh\_slice sl;}
\text{getfem::slicer\_build\_stored\_mesh\_slice a2(sl);}
\text{getfem::mesh\_slicer slicer(m);}
\text{slicer.push\_back\_action(a1);}
\text{slicer.push\_back\_action(a2);}
\text{int nrefine = 3;}
\text{slicer.exec(nrefine);}
\]

In order to build a \texttt{getfem::stored\_mesh\_slice} object during the slicing operation, the \texttt{stored\_mesh\_slice::build()} method is often more convenient than using explicitly the \texttt{slicer\_build\_stored\_mesh\_slice} slicer:

\[
\text{getfem::stored\_mesh\_slice sl;}
\text{sl.build(m, getfem::slicer\_boundary(m),}
\text{ getfem::slicer\_half\_space(base\_node(0,0), base\_node(1, 0), -1),}
\text{ nrefine);}
\]
The simplest way to use these slices is to export them to VTK, OpenDX, or Gmsh. The file getfem/getfem_export.h contains three classes: getfem::vtk_export, getfem::dx_export and getfem::pos_export.

21.3 Exporting mesh, mesh_fem or slices to VTK

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

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

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

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

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

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

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

21.4 Exporting mesh, mesh_fem or slices to OpenDX

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

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

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

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

```cpp
g fem::dx_export exp("output.dx");
g fem::stored_mesh_slice sl;
if (N <= 2) sl.build(mesh, getfem::slicer_none(), 4);
else sl.build(mesh, getfem::slicer_boundary(mesh), 4);
exp.exporting(sl, true);
// for each mesh object, a corresponding `mesh` object will be created in the data file
// for the edges of the original mesh
exp.exporting_mesh_edges();
while (t <= T) {
  ...
  exp.write_point_data(mf_u, U);
  exp.serie_add_object("deformation");
  exp.write_point_data(mf_vm, VM);
  exp.serie_add_object("von_mises_stress");
}
```

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

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

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

A method to compute a pure convection is defined in the file `getfem/getfem_convect.h`. The call of the function is:

```cpp
getfem::convect(mf, U, mf_v, V, dt, nt, option = CONVECT_EXTRAPOLATION);
```

where `mf` is a variable of type `getfem::mesh_fem`, `U` is a vector which represent the field to be convected, `mf_v` is a `getfem::mesh_fem` for the velocity field, `V` is the dof vector for the velocity field, `dt` is the pseudo time of convection and `nt` the number of iterations for the computation of characteristics. `option` is an option for the boundary condition where there is a re-entrant convection. The possibilities are `getfem::CONVECT_EXTRAPOLATION` (extrapolation of the field on the nearest element) or `getfem::CONVECT_UNCHANGED` (no change of the value on the boundary).

The method integrate the partial differential equation

\[
\frac{\partial U}{\partial t} + V \cdot \nabla U = 0,
\]

on the time interval \([0, dt]\).

The method used is of Galerkin-Characteristic kind. It is a very simple version which is inconditionnally stable but rather dissipative. See [ZT1989] and also the Freefem++ documentation on convect command.

The defined method works only if `mf` is a pure Lagrange finite element method for the moment. The principle is to convect backward the finite element nodes by solving the ordinary differential equation:

\[
\frac{dX}{dt} = -V(X),
\]

with an initial condition corresponding to each node. This convection is made with `nt` steps. Then the solution is interpolated on the convected nodes.

In order to make the extrapolation not too expensive, the product `dt \times V` should not be too large.

Note that this method can be used to solve convection dominant problems coupling it with a splitting scheme.
CHAPTER 23

The model description and basic model bricks

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

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

The kernel of the model description is contained in the file getfem/getfem_models.h. The two main objects are the model and the brick.

23.1 The model object

The aim of the model object, defined in file getfem/getfem_models.h, is to globally describe a PDE model. It mainly contains two lists: a list of variables (related or not to the mesh_fem objects) and data (also related or not to the mesh_fem objects) and a list of bricks. The role of the model object is to coordinate the module and make them produce a linear system of equations. If the model is linear, this will simply be the linear system of equation on the corresponding dofs. If the model is nonlinear, this will be the tangent linear system. There are two versions of the model object: a real one and complex one.

The declaration of a model object is done by:

```
getfem::model md(complex_version = false);
```
The parameter of the constructor is a boolean which determines whether the model deals with complex number or real numbers. The default is false for a model dealing with real numbers.

There are different kinds of variables/data in the model. The variables are the unknown of the model. They will be (generally) computed by solving the (tangent) linear system built by the model. Generally, the model will have several variables. Each variable has a certain size (number of degrees of freedom) and the different variables are sorted in alphanumeric order to form the global unknown ($U$ in Fig. 1: The (tangent) linear system). Each variable will be associated to an interval $I = [n_1, n_2]$ which will represent the degrees of freedom indices corresponding to this variable in the global system. The model stores also some data (in the same format as the variables). The difference between data and variables is that data is not an unknown of the model. The value of the data should be provided. In some cases (nonlinear models) some variables can be considered as some data for certain terms. Variables and data are of two kinds. They can have a fixed size, or they can depend on a finite element method (be the d.o.f. of a finite element method).

For instance, in the situation described in Fig. 1: The (tangent) linear system, there are four variables in the model, namely $X, Y, V$ and $W$. The role of the model object will be to assemble the linear system, i.e. to fill the sub matrices corresponding to each variable ($R_{X,X}, R_{Y,Y}, R_{V,V},$ and $R_{W,W}$) and the coupling terms between two variables ($R_{X,Y}, R_{X,V}, R_{W,V}, \cdots$). This different contributions will be given by the different bricks added to the model.

The main useful methods on a model object are

- `m.is_complex()`
  A boolean which says if the model deals with real or complex unknowns and data.
add_fixed_size_variable (name, size, niter=1)
Add a variable of fixed size. name is a string which designate the variable. niter is the number of copy of the variable.

add_fixed_size_variable (name, sizes, niter=1)
Add a variable of fixed size. name is a string which designate the variable. sizes is a vector of dimension for matrix or tensor fixed size variables. niter is the number of copy of the variable.

add_fixed_size_data (name, size, niter=1)
Add a data of fixed size. name is a string which designate the data. niter is the number of copy of the data.

add_fixed_size_data (name, sizes, niter=1)
Add a data of fixed size. name is a string which designate the data. sizes is a vector of dimension for matrix or tensor fixed size variables. niter is the number of copy of the data.

add_initialized_fixed_size_data (name, V)
Add a data of fixed size initialized with the given vector V. name is a string which designate the data.

add_initialized_scalar_data (name, e)
Add a data of size 1 initialized with the given scalar value e. name is a string which designate the data.

add_fem_variable (name, mf, niter=1)
Add a variable being the dofs of a finite element method mf. name is a string which designate the variable. niter is the number of copy of the variable.

add_filtered_fem_variable (name, mf, region)
Add a variable being the dofs of a finite element method mf only specific to a given region. (The standard way to define mf in GetFEM++ is to define in the whole domain.) name is a string which designate the variable. region is the region number. This function will select the degree of freedom whose shape function is non-zero on the given region. Internally, a partial_mesh_fem object will be used. The method mesh_fem_of_variable('name') allows to access to the partial_mesh_fem built.

add_fem_data (name, mf, niter=1)
Add a data being the dofs of a finite element method mf. name is a string which designate the data. niter is the number of copy of the data.

add_initialized_fem_data (name, mf, V, niter=1)
Add a data being the dofs of a finite element method mf initialized with the given vector V. name is a string which designate the data. niter is the number of copy of the data.

add_multiplier (name, mf, primal_name, niter=1)
Add a special variable linked to the finite element method mf and being a multiplier for certain constraints (Dirichlet condition for instance) on a primal variable primal_name. The most important is that the degrees of freedom will be filtered thanks to a partial_mesh_fem object in order to retain only a set of linearly independent constraints. To ensure this, a call to the bricks having a term linking the multiplier and the primal variable is done and a special algorithm is called to extract independent constraints. This algorithm is optimized for boundary multipliers (see gmm::range_basis). Use it with care for volumic multipliers. niter is the number of copy of the variable. Note that for complex terms, only the real part is considered to filter the multiplier.

real_variable (name, niter=1)
Gives the access to the vector value of a variable or data. Real version.

23.1. The model object
complex_variable (name, niter=1)
  Gives the access to the vector value of a variable or data. Complex version.

mesh_fem_of_variable (name)
  Gives a reference on the mesh_fem on which the variable is defined. Throw an exception if this is not a fem variable.

real_tangent_matrix ()
  Gives the access to tangent matrix. Real version. A computation of the tangent system have to be done first.

complex_tangent_matrix ()
  Gives the access to tangent matrix. Complex version. A computation of the tangent system have to be done first.

real_rhs ()
  Gives the access to right hand side vector of the linear system. real version. A computation of the tangent system have to be done first.

complex_rhs ()
  Gives the access to right hand side vector of the linear system. Complex version. A computation of the tangent system have to be done first.

23.2 The brick object

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

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

23.3 How to build a new brick

Note first that the design of a new brick is only necessary for special terms not covered by existing bricks and not covered by the wide range of accessible terms (including complex coupling terms) of the generic assembly brick (see Generic assembly bricks).

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

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

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

    void asm_real_tangent_terms(const getfem::model &md, size_type ib, const getfem::model::varnamelist &varl, const getfem::model::varnamelist &datal, const getfem::model::mimlist &mims, getfem::model::real_matlist &matl, getfem::model::real_veclist &vecl, getfem::model::real_veclist &vecl_sym, size_type region, build_version nl) const {

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

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

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

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

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

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

The method `asm_real_tangent_terms` will be called by the model object for the assembly of
the tangent system. The model object gives the whole framework to the brick to build its terms. The parameter `md` of the `asm_real_tangent_terms` method is the model that called the brick, `ib` being the brick number in the model. The parameter `varl` is an array of variable/data names defined in this model and needed in the brick. `mims` is an array of `mesh_im` pointers. It corresponds to the integration methods needed to assemble the terms. `matl` is an array of matrices to be computed. `vecl` is an array of vectors to be computed (rhs or residual vectors). `vecl_sym` is an array of vectors to be computed only for symmetric terms and corresponding to the rhs of the second variable. A brick can have an arbitrary number of terms. For each term, at least the corresponding matrix or the corresponding vector has to be filled (or both the two, but only in the nonlinear case, see the description of the terms below, next section). `region` is a mesh region number indicated that the terms have to be assembled on a certain region. `nl` is for nonlinear bricks only. It says if the tangent matrix or the residual or both the two are to be computed (for linear bricks, all is to be computed at each call).

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

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

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

The lines:

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

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

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

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

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

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

\[ u = u_D \text{ on } \Gamma, \]

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

\[ \int_{\Gamma} u \mu \, d\Gamma = \int_{\Gamma} u_D \mu \, d\Gamma, \forall \mu \in M, \]
where $M$ is an appropriate multiplier space. The contributions to the global linear system can be viewed in Fig. *Contributions of the simple Dirichlet brick*. The matrix $B$ is the “mass matrix” between the finite element space of the variable $u$ and the finite element space of the multiplier $\mu$. $L_u$ is the right hand side corresponding to the data $u_D$.

![Fig. 2: Contributions of the simple Dirichlet brick](image)

The brick can be defined as follows:

```cpp
struct my_Dirichlet_brick: public getfem::virtual_brick {

  void asm_real_tangent_terms(const getfem::model &md, size_type ib,
                              const getfem::model::varnamelist &varl,
                              const getfem::model::varnamelist &datal,
                              const getfem::model::mimlist &mims,
                              getfem::model::real_matlist &matl,
                              getfem::model::real_veclist &vecl,
                              getfem::model::real_veclist &vecl_sym,
                              size_type region, build_version nl) const {
    GMM_ASSERT1(matl.size() == 1,
                "My Dirichlet brick has one and only one term");
    GMM_ASSERT1(mims.size() == 1,
                "My Dirichlet brick need one and only one mesh_im");
    GMM_ASSERT1(varl.size() == 2 && datal.size() == 1,
                "Wrong number of variables for my Laplacian brick");

    const getfem::mesh_fem &mf_u = md.mesh_fem_of_variable(varl[0]);
    const getfem::mesh_fem &mf_mult = md.mesh_fem_of_variable(varl[1]);
    const getfem::mesh_im &mim = *mims[0];
    const getfem::model_real_plain_vector &A = md.real_variable(datal[0]);
    const getfem::mesh_fem &mf_data = md.pmesh_fem_of_variable(datal[0]);

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

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

  my_Dirichlet_brick(void)
};
```

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

The lines:

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

allow to have the access to the value of the data corresponding to the right hand side of the Dirichlet condition and to the `mesh_fem` on which this data is defined. If the data is constant (not described on a fem) then `mf_data` is a null pointer.

The lines:

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

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

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

### 23.4 How to add the brick to a model

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

- A pointer to the brick itself.
- The set of variable names concerned with the terms of the brick.
- The set of data names concerned with the terms of the brick.
- A list of terms description.
- A list of integration methods.
- Eventually the concerned mesh region.

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

```cpp
md.add_brick(pbr, const getfem::model::varnamelist &varnames,
              const getfem::model::varnamelist &datanames,
              const getfem::model::termvector &terms,
              const getfem::model::intmethodvector &intmethods,
              const getfem::model::meshvector &regions);
```
The method returns the index of the brick in the model. The call of this method is rather complex because it can be adapted to many situations. The construction of a new brick should be accompanied to the definition of a function that adds the new brick to the model calling this method and more simple to use.

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

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

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

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

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

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

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

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

The function allowing to add the simple Dirichlet brick described above can be defined as follows:
```cpp
size_t add_my_Dirichlet_condition_brick(model &md, const mesh_im &mim,
    const std::string &varname,
    const std::string &multname,
    size_t region,
    const std::string &dataname) {
    pbrick pbr = std::make_shared<my_Dirichlet_brick>();
    model::termlist tl;
    tl.push_back(model::term_description(multname, varname, true));
    model::varnamelist vl(1, varname);
    vl.push_back(multname);
    model::varnamelist dl;
    if (dataname.size()) dl.push_back(dataname);
    return md.add_brick(pbr, vl, dl, tl, model::mimlist(1, &mim), region);
}
```

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

### 23.5 Generic assembly bricks

A mean to add a term either on one variable or on several ones is to directly use the weak form language described in Section Compute arbitrary terms - high-level generic assembly procedures - Weak-form language. The more general way is to use:

```cpp
size_type getfem::add_nonlinear_term(md, mim, expr,
    region = -1, is_sym = false, is_coercive = false);
```

This adds a brick to the model md, using the integration method mim, the assembly string expr on the mesh region region. If the result is symmetric, you can specify it on the 5th argument and if it is coercive on the 6th argument. The latter indications of symmetry and coercivity are used to determine the right linear solver. If you are not so sure, it is preferable not to indicate anything.

However, this brick consider that the expression is nonlinear. This brick is especially indicated to obtain nonlinear coupled terms between several variables. This means in particular that the assembly of the term is performed at each call of the assembly of the model and that a Newton algorithm will be used to solve the problem. If the term is indeed linear, you should use instead:

```cpp
size_type getfem::add_linear_term(md, mim, expr,
    region = -1, is_sym = false, is_coercive = false);
```

with the same arguments. Conversely, this brick alway assume that the term corresponding to expr is linear and the assembly will be performed only once if the data used do not change. Thus, you have to care that your expression is indeed linear (affine in fact) with respect to each variable. Otherwise, the result is of course not guaranted. Source terms in the expression are taken into account. Still for linear problem, it is possible to perform the assembly of a sole source term thanks to:

```cpp
size_type getfem::add_source_term(md, mim, expr, region = -1);
```

with again the same arguments except the symmetry and coercivness. This brick performs the assembly of the corresponding order 1 term (residual vector) and add it as a right hand side to the problem. The assembly will be performed only once, so the term should not depend on the variables of the model (but could depend of course on the constants).
For instance, if one wants to solve a Poisson problem on a predefined variable $u$ of the model, one may use the corresponding pre-defined bricks (see below) or simply use:

```cpp
getfem::add_nonlinear_term(md, mim, "Grad_u.Grad_Test_u - F*Test_u", -1, \
                          true, true);
```

where $F$ is a pre-defined constant of the model representing the right hand side. Of course, doing so, Newton’s algorithms will be called. So, the more appropriate manner is to use the linear bricks as follows:

```cpp
getfem::add_linear_term(md, mim, "Grad_u.Grad_Test_u", -1, true, true); \
getfem::add_source_term(md, mim, "F*Test_u");
```

Note that for the moment, the use of the weak form language is not possible for complex valued problems.

### 23.6 Generic elliptic brick

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

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

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

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

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

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

The second function is:

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

It adds a term with an arbitrary coefficient given by the expression $dataexpr$ which has to be a regular expression of the weak form language (like “1”, “sin(X[0])” or “Norm(u)” for instance) even depending on model variables (except for the complex version where it has to be a declared data of the model).

Note that very general equations can be obtained with this brick. For instance, linear anisotropic elasticity can be obtained with a tensor data. When an order four tensor is used, the corresponding weak term
is the following
\[ \int_\Omega \sum_{i,j,k,l} a_{i,j,k,l} \partial_i u_j \partial_k v_l dx \]

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

The brick has a working complex version.

23.7 Dirichlet condition brick

The aim of the Dirichlet condition brick is to prescribe a Dirichlet condition on a part of the boundary of the domain for a variable of the model. This means that the value of this variable is prescribed on the boundary. There is three versions of this brick (see also the section  *Nitsche’s method for dirichlet and contact boundary conditions*). The first version prescribe the Dirichlet thank to a multiplier. The associated weak form of the term is the following:

\[ \int_\Gamma u \mu d\Gamma = \int_\Gamma u_D \mu d\Gamma, \forall \mu \in M. \]

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

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

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

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

The only difference is that \( \text{multname} \) is replaced by \( \text{mf_mult} \) which means that only the finite element on which the multiplier will be built is given. The function adds itself the multiplier variable to the model. The third function is very similar:

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

The parameter \( \text{mf_mult} \) is replaced by an integer \( \text{degree} \) indicating that the multiplier will be built on a classical finite element method of that degree.
Note, that in all the cases, when a variable is added by the method `add_multiplier` of the model object, the `mesh_fem` will be filtered (thank to a `partial_mesh_fem_object` in order to retain only the degrees of freedom having a non vanishing contribution on the considered boundary.

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

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

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

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

```cpp
add_Dirichlet_condition_with_penalization(md, mim, varname,
penalization_coeff, region,
dataname = std::string(),
+mf_mult = 0);
```

The penalization consists in computing the mass matrix of the variable and add it multiplied by the penalization coefficient to the stiffness matrix. The parameter `mf_mult` (a pointer to a `getfem::mesh_fem` object) is optional. It allows to weaken the Dirichlet condition for locking situations. In that case, the penalization matrix is of the form $B^T B$ where $B$ is the “mass matrix” on the boundary between the shape functions of the variable `varname` and the shape function of the multiplier space. The penalization coefficient is added as a data of the model and can be changed thanks to the function:

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

The third version of the Dirichlet condition brick use a simplification of the linear system (tangent linear system for nonlinear problems). Basically, it enforces a 1 on the diagonal components of the lines corresponding to prescribed degrees of freedom, it completes the lines with some zeros (for symmetric problems, it also complete the columns with some zeros) and it adapts the right-hand side accordingly. This is a rather simple and economic way to prescribe a Dirichlet condition. However, it can only be applied when one can identify the degrees of freedom prescribed by the the Dirichlet condition. So, it has to be use with care with reduced finite element methods, Hermite element methods and cannot be applied for a normal (or generalized) Dirichlet condition on vectorial problems. The function allowing to add this brick is:

```cpp
add_Dirichlet_condition_with_simplification(md, varname, region,
dataname = std::string());
```

If `dataname` is ommited, an homogeneous Dirichlet condition is applied. If `dataname` is given, the constraint is that it has to be constant or described on the same finite element method as the variable `varname` on which the Dirichlet condition is applied. Additionally, If `dataname` is constant, it can only be applied to Lagrange finite element methods.

### 23.8 Generalized Dirichlet condition brick

The generalized Dirichlet condition is a boundary condition of a vector field $u$ of the type

$$Hu = r$$

where $H$ is a matrix field. The functions adding the corresponding bricks are similar to the ones of the standard Dirichlet condition except that they need the supplementary parameter `Hname` which gives the
name of the data corresponding to $H$. This data can be a matrix field described on a scalar fem or a constant matrix.

```c
add_generalized_Dirichlet_condition_with_multipliers(md, mim, varname,
    mutname, region,
    dataname, Hname);
add_generalized_Dirichlet_condition_with_multipliers(md, mim, varname,
    mf_mult, region,
    dataname, Hname);
add_generalized_Dirichlet_condition_with_multipliers(md, mim, varname,
    degree, region,
    dataname, Hname);
add_generalized_Dirichlet_condition_with_penalization(md, mim, varname,
    penalization_coeff, region,
    dataname, Hname);
```

## 23.9 Pointwise constraints brick

The pointwise constraints brick is a Dirichlet condition like brick which allows to prescribe the value of an unknown on given points of the domain. These points are not necessarily some vertex of the mesh or some points corresponding to degrees of freedom of the finite element method on which the unknown is described.

For scalar field variables, given a set of $N_p$ points $x_i, i = 1 \cdots N_p$, the brick allows to prescribe the value of the variable on these points, i.e. to enforce the condition

$$u(x_i) = l_i, \quad i = 1 \cdots N_p,$$

where $u$ is the scalar field and $l_i$ the value to be prescribed on the point $x_i$.

For vector field variables, given a set of $N_p$ points $x_i, i = 1 \cdots N_p$, the brick allows to prescribe the value of one component of the variable on these points, i.e. to enforce the condition

$$u(x_i) \cdot n_i = l_i, \quad i = 1 \cdots N_p,$$

where $n_i$ is the vector such that $u(x_i) \cdot n_i$ represent the component to be prescribed.

The brick has two versions: a penalized version and a version with multipliers. The call is the following:

```c
add_pointwise_constraints_with_penalization(md, varname, penalisation_coef,
    dataname_pt, dataname_unitv = std::string(),
    dataname_val = std::string());
add_pointwise_constraints_with_given_multipliers(md, varname, mutname,
    dataname_pt, dataname_unitv = std::string(),
    dataname_val = std::string());
add_pointwise_constraints_with_multipliers(md, varname, dataname_pt,
    dataname_unitv = std::string(),
    dataname_val = std::string());
```
respectively for the penalized version, the one with a given multiplier fixed size variable and the one which automatically adds a multiplier variable of the right size to the model. The data `dataname_pt`, `dataname_unitv` and `dataname_val` should be added first to the model. `dataname_pt` should be a vector containing the coordinates of the points where to prescribed the value of the variable `varname`. It is thus of size $N N_p$, where $N$ is the dimension of the mesh. `dataname_unitv` is ignored for a scalar field variable. For a vector field variable, it should contain the vector $n_i$. In that case, its size should be $Q N_p$, where $Q$ is the dimension of the vector field. `dataname_val` is optional and represents the right hand side, it should contain the components $l_i$. The default value for $l_i$ is 0.

This brick is mainly designed to prescribe the rigid displacements for pure Neumann problems.

### 23.10 Source term bricks (and Neumann condition)

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

$$\int_\Omega f v \, dx$$

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

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

The function to add a source term to a model is:

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

where `md` is the model object, `mim` is the integration method, `varname` is the variable of the model for which the source term is added, `dataexpr` has to be a regular expression of the weak form language (except for the complex version where it has to be a declared data of the model). It has to be scalar or vector valued depending on the fact that the variable is scalar or vector valued itself. `region` is a mesh region on which the term is added. If the region corresponds to a boundary, the source term will represent a Neumann condition. `directdataname` is an optional additional data which will directly be added to the right hand side without assembly.

The brick has a working complex version.

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

```cpp
add_normal_source_term_brick(md, mim, 
varname, dataexpr, region);
```

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

### 23.11 Predefined solvers

Although it will be more convenient to build a specific solver for some problems, a generic solver is available to test your models quickly. It can also be taken as an example to build your own solver. It is
defined in `src/getfem/getfem_model_solvers.h` and `src/getfem_model_solvers.cc` and the call is:

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

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

Note that `SuperLU` is used as a default linear solver on “small” problems. You can also link `MUMPS` with `GetFEM++` (see section `Linear algebra procedures`) and use the parallel version. For nonlinear problems, A Newton method (also called Newton-Raphson method) is used.

Note also that it is possible to disable some variables (with the method `md.disable_variable(varname)` of the model object) in order to solve the problem only with respect to a subset of variables (the disabled variables are the considered as data) for instance to replace the global Newton strategy with a fixed point one.

Let us recall that a standard initialization for the `iter` object is the following (see `Gmm++` documentation on `gmm-iter`):

```cpp
gmm::iteration iter(1E-7, 1, 200);
```

where `1E-7` is the relative tolerance for the stopping criterion, `1` is the noisy option and `200` is the maximum number of iterations. The stopping criterion of Newton’s method is build as follows. For a relative tolerance `ε`, the algorithm stops when:

\[
\min\left(\frac{\|F(u)\|_1}{\max(L, 10^{-25})}, \frac{\|h\|_1}{\max(\|u\|_1, 10^{-25})}\right) < \varepsilon
\]

where `F(u)` is the residual vector, `\|\cdot\|_1` is the classical 1-norm in `\mathbb{R}^n`, `h` is the search direction given by Newton’s algorithm, `L` is the norm of an estimated external loads (coming from source term and Dirichlet bricks) and `u` is the current state of the searched variable. The maximum taken with `10^{-25}` is to avoid pathological cases when `L` and/or `u` are vanishing.

### 23.12 Example of a complete Poisson problem

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

```cpp
using bgeot::base_small_vector;
scalar_type sol_u(const base_node &x) { return sin(x[0]+x[1]); }
// Right hand side. Allows an interpolation for the source term.
scalar_type sol_f(const base_node &x) { return 2*sin(x[0]+x[1]); }
// Gradient of the solution. Allows an interpolation for the Neumann term.
base_small_vector sol_grad(const base_node &x)
    { return base_small_vector(cos(x[0]+x[1]), cos(x[0]+x[1])); }
int main(void) {
```
// ... definition of a mesh
// ... definition of a finite element method mf_u
// ... definition of a finite element method mf_rhs
// ... definition of an integration method mim
// ... definition of boundaries NEUMANN_BOUNDARY_NUM
// and DIRICHLET_BOUNDARY_NUM

// Model object
getfem::model laplacian_model;

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

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

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

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

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

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

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

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

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

23.12. Example of a complete Poisson problem 119
23.13 Nitsche’s method for dirichlet and contact boundary conditions

GetFEM++ provides a generic implementation of Nitsche’s method which allows to account for Dirichlet type or contact with friction boundary conditions in a weak sense without the use of Lagrange multipliers. The method is very attractive because it transforms a Dirichlet boundary condition into a weak term similar to a Neumann boundary condition. However, this advantage is at the cost that the implementation of Nitsche’s method is model dependent, since it requires an approximation of the corresponding Neumann term. In order to add a boundary condition with Nitsche’s method on a variable of a model, the corresponding brick needs to have access to an approximation of the Neumann term of all partial differential terms applied to this variable. In the following, considering a variable \( u \), we will denote by \( G \) the sum of all Neumann terms on this variable. Note that the Neumann term \( G \) will often depend on the variable \( u \) but it may also depend on other variables of the model. This is the case for instance for mixed formulations of incompressible elasticity. The Neumann terms depend also frequently on some parameters of the model (elasticity coefficients . . . ) but this is assumed to be contained in its expression.

For instance, if there is a Laplace term (\( \Delta u \)), applied on the variable \( u \), the Neumann term will be \( G = \frac{\partial u}{\partial n} \) where \( n \) is the outward unit normal on the considered boundary. If \( u \) represents the displacements of a deformable body, the Neumann term will be \( G = \sigma(u)n \), where \( \sigma(u) \) is the stress tensor depending on the constitutive law. Of course, in that case \( G \) also depends on some material parameters. If additionally a mixed incompressibility brick is added with a variable \( p \) denoting the pressure, the Neumann term on \( u \) will depend on \( p \) in the following way: \( G = \sigma(u)n - pn \).

In order to allow a generic implementation in which the brick imposing Nitsche’s method will work for every partial differential term applied to the concerned variables, each brick adding a partial differential term to a model is required to give its expression via an assembly string (weak form language).

These expressions are utilized in a special method of the model object:

```python
expr = md.Neumann_term(variable, region)
```

which allows to automatically derive an expression for the sum of all Neumann terms, by scanning the expressions provided by all partial differential term bricks and performing appropriate manipulations. Of course it is required that all volumic bricks were added to the model prior to the call of this method. The derivation of the Neumann term works only for second order partial differential equations. A generic implementation for higher order pde would be more complicated.

23.13.1 Generic Nitsche’s method for a Dirichlet condition

Assume that the variable \( u \) is considered and that one wants to prescribe the condition

\[ Hu = g \]

on a part \( \Gamma_D \) of the boundary of the considered domain. Here \( H \) is considered equal to one in the scalar case or can be either the identity matrix in the vectorial case either a singular matrix having only 1 or 0 as eigenvalues. This allow here to prescribe only the normal or tangent component of \( u \). For instance if one wants to prescribe only the normal component, \( H \) will be chosen to be equal to \( nn^T \) where \( n \) is the outward unit normal on \( \Gamma_D \).
Nitsche’s method for prescribing this Dirichlet condition consists in adding the following term to the weak formulation of the problem

\[
\int_{\Gamma_D} \gamma \left( Hu - g - \gamma HG \right) (Hv) - \theta (Hu - g) \left( HD_u G[v] \right) d\Gamma,
\]

where \(\gamma\) and \(\theta\) are two parameters of Nitsche’s method and \(v\) is the test function corresponding to \(u\). The parameter \(\theta\) can be chosen positive or negative. \(\theta = 1\) corresponds to the more standard method which leads to a symmetric tangent term in standard situations, \(\theta = 0\) corresponds to a non-symmetric method which has the advantage of a reduced number of terms and not requiring the second derivatives of \(G\) in the nonlinear case, and \(\theta = -1\) is a kind of skew-symmetric method which ensures an unconditional coercivity (which means independent of \(\gamma\)) at least in standard situations. The parameter \(\gamma\) is a kind of penalization parameter (although the method is consistent) which is taken to be \(\gamma = \gamma_0 h_T\) where \(\gamma_0\) is taken uniform on the mesh and \(h_T\) is the diameter of the element \(T\). Note that, in standard situations, except for \(\theta = -1\) the parameter \(\gamma_0\) has to be taken sufficiently small in order to ensure the convergence of Nitsche’s method.

The bricks adding a Dirichlet condition with Nitsche’s method to a model are the following:

```cpp
getfem::add_Dirichlet_condition_with_Nitsche_method
(model &md, const mesh_im &mim, const std::string &varname,
 const std::string &Neumannterm,
 const std::string &gamma0name, size_type region,
 scalar_type theta = scalar_type(1),
 const std::string &dataname = std::string());
```

This function adds a Dirichlet condition on the variable \(\text{varname}\) and the mesh region \(\text{region}\). This region should be a boundary. \(\text{Neumannterm}\) is the expression of the Neumann term (obtained by the Green formula) described as an expression of the weak form language. This term can be obtained with \(\text{md.Neumann_term(varname, region)}\) once all volumic bricks have been added to the model. The Dirichlet condition is prescribed with Nitsche’s method. \(\text{dataname}\) is the optional right hand side of the Dirichlet condition. It could be constant or described on a fem; scalar or vector valued, depending on the variable on which the Dirichlet condition is prescribed. \(\text{gamma0name}\) is the Nitsche’s method parameter. \(\text{theta}\) is a scalar value which can be positive or negative. \(\text{theta} = 1\) corresponds to the standard symmetric method which is conditionally coercive for \(\text{gamma0}\) small. \(\text{theta} = -1\) corresponds to the skew-symmetric method which is inconditionally coercive. \(\text{theta} = 0\) is the simplest method for which the second derivative of the Neumann term is not necessary even for nonlinear problems. Returns the brick index in the model.

```cpp
getfem::add_normal_Dirichlet_condition_with_Nitsche_method
(model &md, const mesh_im &mim, const std::string &varname,
 const std::string &Neumannterm,
 const std::string &gamma0name, size_type region,
 scalar_type theta = scalar_type(1),
 const std::string &dataname = std::string());
```

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

```cpp
getfem::add_generalized_Dirichlet_condition_with_Nitsche_method(model &md, const mesh_im &mim, const std::string &varname,
        const std::string &Neumannterm,
        const std::string &gamma0name, size_type region, scalar_type theta,
        const std::string &dataname, const std::string &Hname);
```

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

### 23.13.2 Generic Nitsche’s method for contact with friction condition

We describe here the use of Nitsche’s method to prescribe a contact with Coulomb friction condition in the small deformations framework. This corresponds to a weak integral contact condition which as some similarity with the ones which use Lagrange multipliers describe in the corresponding section, see Weak integral contact condition

In order to simplify notations, let use denote by $P_{n,\mathcal{F}}$ the following map which corresponds to a couple of projections:

$$P_{n,\mathcal{F}}(x) = -(x.n)_-n + P_{B(0,\mathcal{F}(x.n)_-)}(x - (x.n)n)$$

This application make the projection of the normal part of $x$ on $\mathbb{R}_-$ and the tangential part on the ball of center 0 and radius $\mathcal{F}(x.n)_-$, where $\mathcal{F}$ is the friction coefficient.

Using this, and considering that the sliding velocity is approximated by $\alpha(u_T - w_T)$ where the expression of $\alpha$ and $w_T$ depend on the time integration scheme used (see Weak integral contact condition), Nitsche’s term for contact with friction reads as:

$$- \int_{\Gamma_C} \theta \gamma G \cdot D_n G[v] d\Gamma + \int_{\Gamma_C} \gamma P_{n,\mathcal{F}}(G - \frac{Au}{\gamma} + \frac{gap}{\gamma} n + \frac{\alpha w}{\gamma} (\theta D_n G[v] - \frac{v}{\gamma}) d\Gamma,$$

where $\Gamma_C$ is the contact boundary, $G$ is the Neumann term which represents here $\sigma n$ the stress at the contact boundary and $A$ is the $d \times d$ matrix

$$A = \alpha I_d + (1 - \alpha)nn^T$$

Note that for the variant with $\theta = 0$ a majority of terms vanish.
The following function adds a contact condition with or without Coulomb friction on the variable `varname_u` and the mesh boundary `region`. `Neumannterm` is the expression of the Neumann term (obtained by the Green formula) described as an expression of the weak form language. This term can be obtained with `md.Neumann_term(varname, region)` once all volumic bricks have been added to the model. The contact condition is prescribed with Nitsche’s method. The rigid obstacle should be described with the data `dataname_obstacle` being a signed distance to the obstacle (interpolated on a finite element method). `gamma0name` is the Nitsche’s method parameter. `theta` is a scalar value which can be positive or negative. `theta = 1` corresponds to the standard symmetric method which is conditionally coercive for `gamma0` small. `theta = -1` corresponds to the skew-symmetric method which is inconditionally coercive. `theta = 0` is the simplest method for which the second derivative of the Neumann term is not necessary. The optional parameter `dataexpr_friction_coeff` is the friction coefficient which could be any expression of the weak form language. Returns the brick index in the model:

```
getfem::add_Nitsche_contact_with_rigid_obstacle_brick
(model &md, const mesh_im &mim, const std::string &varname_u,
 const std::string &Neumannterm,
 const std::string &expr_obs, const std::string &dataname_gamma0,
 scalar_type theta_,
 std::string dataexpr_friction_coeff,
 const std::string &dataname_alpha,
 const std::string &dataname_wt,
 size_type region);
```

### 23.14 Constraint brick

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

$$BU = L,$$

can be added with the two following functions:

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

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

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

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

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

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

where `indbrick` is the index of the brick in the model.
23.15 Other “explicit” bricks

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

The function:

\[
\text{indbrick} = \text{getfem}::\text{add} \_\text{explicit} \_\text{matrix}(\text{md}, \text{varname}1, \text{varname}2, B, \\
\text{issymmetric} = \text{false}, \\
\text{iscoercive} = \text{false});
\]

adds a brick which just adds the matrix \( B \) to the tangent system relatively to the variables \( \text{varname}1 \) and \( \text{varname}2 \). The given matrix should have as many rows as the dimension of \( \text{varname}1 \) and as many columns as the dimension of \( \text{varname}2 \). If the two variables are different and if \text{issymmetric} is set to true then the transpose of the matrix is also added to the tangent system (default is false). Set \text{iscoercive} to true if the term does not affect the coercivity of the tangent system (default is false). The matrix can be changed by the command:

\[
\text{getfem}::\text{set} \_\text{private} \_\text{data} \_\text{matrix}(\text{md}, \text{indbrick}, B);
\]

The function:

\[
\text{getfem}::\text{add} \_\text{explicit} \_\text{rhs}(\text{md}, \text{varname}, L);
\]

adds a brick which just add the vector \( L \) to the right hand side of the tangent system relatively to the variable \( \text{varname} \). The given vector should have the same size as the variable \( \text{varname} \). The value of the vector can by changed by the command:

\[
\text{getfem}::\text{set} \_\text{private} \_\text{data} \_\text{rhs}(\text{md}, \text{indbrick}, L);
\]

23.16 Helmholtz brick

This brick represents the complex or real Helmholtz problem:

\[
\Delta u + k^2 u = \ldots
\]

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

The function adding a Helmholtz brick to a model is:

\[
\text{getfem}::\text{add} \_\text{Helmholtz} \_\text{brick}(\text{md}, \text{mim}, \text{varname}, \text{dataexpr}, \text{region});
\]

where \( \text{varname} \) is the variable on which the Helmholtz term is added and \( \text{dataexpr} \) is the wave number.

23.17 Fourier-Robin brick

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

\[
\frac{\partial u}{\partial \nu} = Qu
\]
for scalar problems, or
\[ \sigma \cdot \nu = Qu \]

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

The function adding this brick to a model is:

```cpp
add_Fourier_Robin_brick(md, mim, varname, dataexpr, region);
```

where `dataexpr` is the data of the model which represents the coefficient \( Q \). It can be an arbitrary valid expression of the weak form language (except for the complex version for which it should be a data of the model)

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

### 23.18 Isotropic linearized elasticity brick

This brick represents a term

\[ -\text{div}(\sigma) = \ldots \]

with

\[ \sigma = \lambda \text{tr}(\varepsilon(u))I + 2\mu \varepsilon(u) \]

\[ \varepsilon(u) = (\nabla u + \nabla u^T)/2 \]

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

Let us recall that the relation between the Lamé coefficients and Young modulus \( E \) and Poisson ratio \( \nu \) is

\[ \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)}, \]

except for the plane stress approximation (2D model) where

\[ \lambda^* = \frac{E
\nu}{(1-\nu^2)}, \quad \mu^* = \frac{E}{2(1+\nu^2)}, \]

The function which adds this brick to a model and parametrized with the Lamé coefficients is:

```cpp
ind_brick = getfem::add_isotropic_linearized_elasticity_brick
(md, mim, varname, data_lambda, data_mu,
region = size_type(-1));
```

where `dataname_lambda` and `dataname_mu` are the data of the model representing the Lamé coefficients.

The function which adds this brick to a model and parametrized with Young modulus and Poisson ratio is:
This brick represent a plane strain approximation when it is applied to a 2D mesh (and a standard model on a 3D mesh). In order to obtain a plane stress approximation for 2D meshes, one can use:

```cpp
ind_brick = getfem::add_isotropic_linearized_elasticity_brick_pstress
           (md, mim, varname, data_E, data_nu, region = size_type(-1));
```

For 3D meshes, the two previous bricks give the same result.

The function:

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

compute the Von Mises criterion (or Tresca if `tresca_flag` is set to true) on the displacement field stored in `varname`. The stress is evaluated on the `mesh_fem` `mf_vm` and stored in the vector `VM`. It is not valid for 2D plane stress approximation and is parametrized with Lamé coefficients. The functions:

```cpp
getfem::compute_isotropic_linearized_Von_Mises
       (md, varname, data_E, data_nu, mf_vm, VM);
getfem::compute_isotropic_linearized_Von_Mises
       (md, varname, data_E, data_nu, mf_vm, VM);
```

compute the Von Mises stress, parametrized with Young modulus and Poisson ratio, the second one being valid for 2D plane stress approximation when it is applied on a 2D mesh (the two functions give the same result for 3D problems).

The program `tests/elastostatic.cc` can be taken as a model of use of a linearized isotropic elasticity brick.

### 23.19 Linear incompressibility (or nearly incompressibility) brick

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

\[ \text{div}(\mathbf{u}) = 0, \quad (\text{or} \quad \text{div}(\mathbf{u}) = \varepsilon p) \]

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

The function adding this incompressibility condition is:

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

where `varname` is the variable on which the incompressibility condition is prescribed, `multname_pressure` is a variable which should be described on a scalar fem representing the multiplier (the pressure) and `dataexpr_penal_coeff` is an optional penalization coefficient for the nearly incompressible condition.
In nearly incompressible homogeneous linearized elasticity, one has $\varepsilon = 1/\lambda$ where $\lambda$ is one of the Lamé coefficient and $\varepsilon$ the penalization coefficient.

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

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

laplacian_model.add_fem_variable("u", mf_u);

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

getfem::add_isotropic_linearized_elasticity_brick(Stokes_model, mim, 
  "u", "lambda", "mu");

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

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

getfem::add_source_term_brick(Stokes_model, mim, "u", "VolumicData");

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

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

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

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

### 23.20 Mass brick

This brick represents a weak term of the form

$$ \int_{\Omega} \rho u \cdot v \, dx + \ldots $$

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

The function which adds this brick to a model is:

```cpp
ind_brick = getfem::add_mass_brick
    (md, mim, varname, dataexpr_rho="", region = size_type(-1));
```
where `dataexpr_rho` is an optional expression representing the density $\rho$. If it is omitted, the density is assumed to be equal to one.

## 23.21 Bilaplacian and Kirchhoff-Love plate bricks

The following function

```cpp
ind = add_bilaplacian_brick(md, mim, varname, dataname,
                            region = size_type(-1));
```

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

For the Kirchhoff-Love plate model, the weak form is a bit different (and more stable than the previous one). the function to add that term is

```cpp
ind = add_bilaplacian_brick_KL(md, mim, varname, dataname1, dataname2,
                                region = size_type(-1));
```

It adds a bilaplacian brick on the variable `varname` and on the mesh region `region`. This represent a term $\Delta(D\Delta u)$ where $D(x)$ is the flexion modulus determined by `dataname1`. The term is integrated by part following a Kirchhoff-Love plate model with `dataname2` the poisson ratio.

There is specific bricks to add appropriate boundary conditions for fourth order partial differential equations. The first one is

```cpp
ind = add_normal_derivative_source_term_brick(md, mim, varname, dataname, region);
```

which adds a normal derivative source term brick $F = \int b.\partial_n v$ on the variable `varname` and on the mesh region `region`. It updates the right hand side of the linear system. `dataname` represents $b$ and `varname` represents $v$.

A Neumann term can be added thanks to the following bricks

```cpp
ind = add_Kirchhoff_Love_Neumann_term_brick(md, mim, varname, dataname1, dataname2, region);
```

which adds a Neumann term brick for Kirchhoff-Love model on the variable `varname` and the mesh region `region`. `dataname1` represents the bending moment tensor and `dataname2` its divergence.

And a Dirichlet condition on the normal derivative can be prescribed thanks to the following bricks

```cpp
ind = add_normal_derivative_Dirichlet_condition_with_multipliers
     (md, mim, varname, mutname, region, dataname = std::string(),
      R_must_be_derivated = false);
ind = add_normal_derivative_Dirichlet_condition_with_multipliers
     (md, mim, varname, mf_mult, region, dataname = std::string(),
      R_must_be_derivated = false);
ind = add_normal_derivative_Dirichlet_condition_with_multipliers
     (md, mim, varname, degree, region, dataname = std::string(),
      R_must_be_derivated = false);
```
These bricks add a Dirichlet condition on the normal derivative of the variable varname and on the mesh region region (which should be a boundary). The general form is $\int \partial_n u(x)v(x) = \int r(x)v(x)\forall v$ where $r(x)$ is the right hand side for the Dirichlet condition (0 for homogeneous conditions) and $v$ is in a space of multipliers defined by the variable multname (first version) or defined on the finite element method mf_mult (second version) or simply on a Lagrange finite element method of degree degree (third version) on the part of boundary determined by region. dataname is an optional parameter which represents the right hand side of the Dirichlet condition. If $R_{must\_be\_derivated}$ is set to true then the normal derivative of dataname is considered.

The test program bilaplacian.cc is a good example of the use of the previous bricks.

### 23.22 Mindlin-Reissner plate model

This brick implements the classical Mindlin-Reissner bending model for isotropic plates.

#### 23.22.1 The Mindlin-Reissner plate model

Let $\Omega \subset \mathbb{R}^2$ be the reference configuration of the mid-plane of a plate of thickness $\epsilon$.

The weak formulation of the Mindlin-Reissner model for isotropic material can be written as follows for $u_3$ the transverse displacement and $\theta$ the rotation of fibers normal to the mid-plane:

$$
\int_{\Omega} D e^A \left( (1 - \nu) \gamma(\theta) \cdot \gamma(\psi) + \nu \text{div}(\theta) \text{div}(\psi) \right) dx \\
+ \int_{\Omega} Ge(\nabla u_3 - \theta) \cdot (\nabla v_3 - \psi) dx = \int_{\Omega} F_3 v_3 + M.\psi dx,
$$

for all admissible test functions $v_3 : \Omega \rightarrow \mathbb{R}$, $\psi : \Omega \rightarrow \mathbb{R}^2$ and where:

- $D = \frac{E}{12(1 - \nu^2)}$, $G = \frac{E \kappa}{2(1 + \nu)}$,
- $\gamma(\theta) = (\nabla \theta + \nabla \theta^T)/2$,
- $F_3 = \int_{-\epsilon/2}^{\epsilon/2} f_3 dx_3 + g^+_3 + g^-_3$,
- $M_\alpha = \epsilon (g^+_\alpha - g^-_\alpha)/2 + \int_{-\epsilon/2}^{\epsilon/2} x_3 f_\alpha dx_3, \alpha \in \{1, 2\}$,

$f$ being a volumic force applied inside the three dimensional plate, $g^+$ and $g^-$ a force applied on the top and bottom surface of the plate, $E$ Young’s modulus, $\nu$ Poisson’s ratio and $\kappa$ the shear correction factor (usually set to 5/6).

The classical boundary conditions are the following:

- Simple support : a dirichlet condition on $u_3$.
- Clamped support : a dirichlet condition on both $u_3$ and $\theta$.
- Prescribed transverse force : boundary source term on $u_3$.
- Prescribed moment : boundary source term on $\theta$.

An important issue of this model is that it is subjected to the so called shear locking so that a direct Galerkin procedure do not give a satisfactory approximation. There is several ways to circumvent the shear locking problem : reduced integration, projection of the transverse shear term, mixed methods. The two first method are proposed.
Reduced integration of the transverse shear term

This strategy consists simply to use a lower order integration method to numerically compute the term

\[ \int \Omega Ge(\nabla u_3 - \theta) \cdot (\nabla v_3 - \psi) dx \]

This strategy is working properly at least when both the rotation and the transverse displacement is approximated with Q1 quadrilateral element with a degree one reduced integration method (the so-called QUAD4 element).

Projection of the transverse shear term

Another strategy comes from the MITC elements (Mixed Interpolation of Tensorial Components) which correspond to a re-interpretation in terms of projection of some mixed methods. The most popular element of this type is the MITC4 which correspond to the quadrilateral element Q1 with a projection of the transverse shear term on a rotated Raviart-Thomas element of lowest degree (RT0) (see [ba-dv1985], [br-ba-fo1989]). This means that the transverse shear term becomes

\[ \int \Omega Ge P^h(\nabla u_3 - \theta) \cdot P^h(\nabla v_3 - \psi) dx \]

where \( P^h(T) \) is the elementwise \( L^2 \)-projection onto the rotated RT0 space. For the moment, the only projection implemented is the previous described one (projection on rotated RT0 space for quadrilateral element). Higher degree elements and triangular elements can be found in the litterature (see [Mi-Zh2002], [br-ba-fo1989], [Duan2014]) and will be under consideration for a future implementation. Note also that since \( P^h(\nabla u_3) = \nabla u_3 \), the term reduces to

\[ \int \Omega Ge(\nabla u_3 - P^h(\theta)) \cdot (\nabla v_3 - P^h(\psi)) dx \]

The principle of the definition of an elementary projection is explained if the description of the weak form language (see Elementary transformations) and an example can be found in the file src/getfem_linearized_plates.cc.

23.22.2 Add a Mindlin-Reissner plate model brick to a model

The following function defined in src/getfem/getfem_linearized_plates.h allows to add a Mindlin-Reissner plate model term to a transverse displacement \( u_3 \) and a rotation \( \theta \):

```cpp
size_type add_Mindlin_Reissner_plate_brick(model, mim, mim_reduced, name_u3, name_theta, param_E, param_nu, param_epsilon, param_kappa, variant = 2, region)
```

where `name_u3` is name of the variable which represents the transverse displacement, `name_theta` the variable which represents the rotation, `param_E` the Young Modulus, `param_nu` the poisson ratio, `param_epsilon` the plate thickness, `param_kappa` the shear correction factor. Note that since this brick uses the weak form language, the parameter can be regular expression of this language. There are three variants. `variant = 0` corresponds to the an unreduced formulation and in that case only the integration method `mim` is used. Practically this variant is not usable since it is subject to a strong locking phenomenon. `variant = 1` corresponds to a reduced integration where `mim` is used for the rotation term and `mim_reduced` for the transverse shear term. `variant = 2` (default) corresponds to the projection onto a rotated RT0 element of the transverse shear term. For the moment, this is adapted to quadrilateral only.
(because it is not sufficient to remove the locking phenomenon on triangle elements). Note also that if you use high order elements, the projection on RT0 will reduce the order of the approximation. Returns the brick index in the model.

The projection on rotated RTO element can be added to a model thanks to the following function:

```c
void add_2D_rotated_RT0_projection(model, transname);
```

### 23.23 The model tools for the integration of transient problems

Although time integration scheme can be written directly using the model object by describing the problem to be solved at each iteration, the model object furnishes some basic tools to facilitate the writing of such schemes. These tools are based on the following basic principles:

- The original variables of the model represent the state of the system to be solved at the current time step (say step $n$). This is the case even for a middle point scheme, mainly because if one needs to apply different schemes to different variables of the system, all variable should describe the system at a unique time step.

- Some data are added to the model to represent the state of the system at previous time steps. For classical one-step schemes (for the moment, only one-step schemes are provided), only the previous time step is stored. For instance if $u$ is a variable (thus represented at step $n$), $\text{Previous}_u$, $\text{Previous2}_u$, $\text{Previous3}_u$ will be the data representing the state of the variable at the previous time step (step $n-1$, $n-2$ and $n-3$).

- Some intermediate variables are added to the model to represent the time derivative (and the second order time derivative for second order problem). For instance, if $u$ is a variable, $\dot{u}$ will represent the first order time derivative of $u$ and $\dot{\dot{u}}$ the second order one. One can refer to these variables in the model to add a brick on it or to use it in the weak form language. However, these are not considered to be independent variables, they will be linked to their corresponding original variable (in an affine way) by the time integration scheme. Most of the schemes need also the time derivative at the previous time step and add the data $\text{Previous}_\dot{u}$ and possibly $\text{Previous}_\dot{\dot{u}}$ to the model.

- A different time integration scheme can be applied on each variable of the model. Note that most of the time, multiplier variable and more generally variables for which no time derivative is used do not need a time integration scheme.

- The data $t$ represent the time parameter and can be used (either in the weak form language or as parameter of some bricks). Before the assembly of the system, the data $t$ is automatically updated to the time step $n$.

- The problem to be solved at each iteration correspond to the formulation of the transient problem in its natural (weak) formulation in which the velocity and the acceleration are expressed by the intermediate variables introduced. For instance, the translation into the weak form language of the problem

\[
\dot{u}(t,x) - \Delta u(t,x) = \sin(t)
\]

can simply be:

```c
\text{Dot}_u*\text{Test}_u + \text{Grad}_u.Grad_{\text{Test}_u} - \sin(t)*\text{Test}_u
```

(even though, of course, in this situation, the use of linear bricks is preferable for efficiency reasons)
• For all implemented one-step schemes, the time step can be changed from an iteration to another for both order one and order two in time problems (or even quasi-static problems).

• A scheme for second order in time problem (resp. first order in time) can be applied to a second or first order in time or even to a quasi-static problem (resp. to a first order or quasi-static problem) without any problem except that the initial data corresponding to the velocity/displacement have to be initialized with respect to the order of the scheme. Conversely, of course, a scheme for first order problem cannot be applied to a second order in time problem.

23.23.1 The implicit theta-method for first-order problems

For a problem which reads

$$ M\dot{U} = F(U) $$

where $F(U)$ might be nonlinear (and may depend on some other variables for coupled problems), for $dt$ a time step, $V = \dot{U}$ and $U^n, V^n$ the approximation of $U, V$ at time $ndt$, theta-method reads

$$ U^n = U^{n-1} + dt(\theta V^n + (1-\theta)V^{n-1}), $$

$$ MV^n = F(U^n), $$

for $\theta \in (0, 1]$ the parameter of the theta-method (for $\theta = 0$, the method corresponds to the forward Euler method and is not an implicit scheme) and for $U^{n-1}, V^{n-1}$ given.

Before the first time step, $U^0$ should be initialized, however, $V^0$ is also needed (except for $\theta = 1$). In this example, it should correspond to $M^{-1}F(U^0)$. For a general coupled problem where $M$ might be singular, a generic precomputation of $V^0$ is difficult to obtain. Thus $V^0$ have to be furnished also. Alternatively (see below) the model object (and the standard solve) furnishes a mean to evaluate them thanks to the application of a Backward Euler scheme on a (very) small time step.

The following formula can be deduced for the time derivative:

$$ V^n = \frac{U^n - U^{n-1}}{\theta dt} - \frac{1 - \theta}{\theta} V^{n-1} $$

When applying this scheme to a variable “u” of the model, the following affine dependent variable is added to the model:

"Dot_u"

which represent the time derivative of the variable and can be used in some brick definition.

The following data are also added:

"Previous_u", "Previous_Dot_u"

which correspond to the values of “u” and “Dot_u” at the previous time step.

Before the solve, the data “Previous_u” (corresponding to $U^0$ in the example) has to be initialized (except for $\theta = 1$). Again, “Previous_Dot_u” has to be either initialized or pre-computed as described in the next section. The affine dependence of “Dot_u” is thus given by:

$$ \text{Dot}_u = (u - \text{Previous}_u)/(\theta \times dt) - \text{Previous}_\text{Dot}_u \times (1 - \theta) / \theta $$

Which means that “Dot_u” will be replaced at assembly time by its expression in term of “u” (multiplied by $1/(\theta \times dt)$) and in term of a constant remaining part depending on the previous time step. The addition of this scheme to a variable is to be done thanks to:
23.23.2 Precomputation of velocity/acceleration

Most of the time integration schemes (except, for instance, the backward Euler scheme) needs the pre-computation of the first or second order time derivative before the initial time step (for instance $V^0$ for the theta-method for first order problems, $A^0$ for second order problems . . .).

The choice is let to the user to either initialize these derivative or to ask to the model to automatically approximate them.

The method used (for the moment) to approximate the supplementary derivatives may be explained in the example of the solve of

$$M \dot{U} = F(U)$$

with a theta-method (see the previous section). In order to approximate $V_0$, the theta-method is applied for $\theta = 1$ (i.e. a backward Euler scheme) on a very small time step. This is possible since the backward Euler do not need an initial time derivative. Then the time derivative computed thanks to the backward Euler at the end of the very small time step is simply used as an approximation of the initial time derivative.

For a model $md$, the following instructions:

```cpp
model.perform_init_time_derivative(ddt);
standard_solve(model, iter);
```

allows to perform automatically the approximation of the initial time derivative. The parameter $ddt$ corresponds to the small time step used to perform the aproximation. Typically, $ddt = dt/20$ could be used where $dt$ is the time step used to approximate the transient problem (see the example below).

23.23.3 The implicit theta-method for second-order problems

For a problem which reads

$$M \ddot{U} = F(U)$$

where $F(U)$ might be nonlinear (and may depend on some other variables for coupled problems), for $dt$ a time step, $\dot{V} = \dot{U}$, $\ddot{A} = \ddot{U}$ and $U^n, V^n, A^n$ the approximation of $\dot{U}, \ddot{V}, \ddot{A}$ at time $n dt$, the first order theta-method reads

$$\begin{cases}
U^n &= U^{n-1} + dt(\theta V^n + (1 - \theta)V^{n-1}), \\
V^n &= V^{n-1} + dt(\theta A^n + (1 - \theta)A^{n-1}), \\
MA^n &= F(U^n),
\end{cases}$$

for $\theta \in (0, 1]$ the parameter of the theta-method (for $\theta = 0$, the method correspond to the forward Euler method and is not an implicit scheme) and for $U^{n-1}, V^{n-1}, A^{n-1}$ given.

At the first time step, $U^0$, $V^0$ should be given and $A^0$ is to be given or pre-computed (except for $\theta = 1$).
The following formula can be deduced for the time derivative:

\[ V^n = \frac{U^n - U^{n-1}}{\theta dt} - \frac{1 - \theta}{\theta} V^{n-1} \]
\[ A^n = \frac{U^n - U^{n-1}}{\theta^2 dt^2} - \frac{1 - \theta}{\theta^2} V^{n-1} - \frac{1 - \theta}{\theta} A^{n-1} \]

When applying this scheme to a variable “u” of the model, the following affine dependent variables are added to the model:

"Dot_u", "Dot2_u"

which represent the first and second order time derivative of the variable and can be used in some brick definition.

The following data are also added:

"Previous_u", "Previous_Dot_u", "Previous_Dot2_u"

which correspond to the values of “u”, “Dot_u” and “Dot2_u” at the previous time step.

Before the solve, the data “Previous_u” and “Previous_Dot_u” (corresponding to \( U^0 \) in the example) have to be initialized and “Previous_Dot2_u” should be either initialized or precomputed (see the previous section, and except for \( \theta = 1 \)). The affine dependences are thus given by:

\[
\begin{align*}
\text{Dot}_u & = (u - \text{Previous}_u)/(\theta + \text{dt}) - \text{Previous}_\text{Dot}_u(1-\theta)/\theta \\
\text{Dot2}_u & = (u - \text{Previous}_u)/(\theta + \text{theta} + \text{dt} + \text{dt}) - \text{Previous}_\text{Dot}_u/(\text{theta} + \text{theta} + \text{dt} + \text{dt}) - \text{Previous}_\text{Dot2}_u(1-\theta)/\theta
\end{align*}
\]

The addition of this scheme to a variable is to be done thanks to:

\textbf{add_theta_method_for_second_order(model &md, const std::string &varname, scalar_type theta)}

### 23.23.4 The implicit Newmark scheme for second order problems

For a problem which reads

\[ M\ddot{U} = F(U) \]

where \( F(U) \) might be nonlinear (and may depend on some other variables for coupled problems), for \( dt \) a time step, \( V = \dot{U} \), \( A = \ddot{U} \) and \( U^n, V^n, A^n \) the approximation of \( U, V, A \) at time \( n dt \), the first order theta-method reads

\[
\begin{align*}
U^n & = U^{n-1} + dt V^n + \frac{dt^2}{2} (2\beta V^n + (1 - 2\beta)V^{n-1}), \\
V^n & = V^{n-1} + dt (\gamma A^n + (1 - \gamma)A^{n-1}), \\
MA^n & = F(U^n),
\end{align*}
\]

for \( \beta \in (0, 1] \) and \( \gamma \in [1/2, 1] \) are the parameters of the Newmark scheme and for \( U^{n-1}, V^{n-1}, A^{n-1} \) given.

At the first time step, \( U^0, V^0 \) should be given and \( A^0 \) is to be given or pre-computed (except for \( \beta = 1/2, \gamma = 1 \)).
The following formula can be deduced for the time derivative:

\[ V^n = \frac{\gamma}{\beta \Delta t}(U^n - U^{n-1}) + \frac{\beta - \gamma}{\beta} V^{n-1} + \Delta t(1 - \frac{\gamma}{2\beta}) A^{n-1} \]

\[ A^n = \frac{U^n - U^{n-1}}{\beta \Delta t^2} - \frac{1}{\beta \Delta t} V^{n-1} - (1/2 - \beta) A^{n-1} \]

When applying this scheme to a variable "u" of the model, the following affine dependent variables are added to the model:

"Dot_u", "Dot2_u"

which represent the first and second order time derivative of the variable and can be used in some brick definition.

The following data are also added:

"Previous_u", "Previous_Dot_u", "Previous_Dot2_u"

which correspond to the values of “u”, “Dot_u” and “Dot2_u” at the previous time step.

Before the first solve, the data “Previous_u” and “Previous_Dot_u” (corresponding to \( U^0 \) in the example) have to be initialized. The data “Previous_Dot2_u” is to be given or precomputed (see Precomputation of velocity/acceleration and except for \( \beta = 1/2, \gamma = 1 \)).

The addition of this scheme to a variable is to be done thanks to:

```cpp
add_Newmark_scheme(model &md, const std::string &varname, 
                   scalar_type beta, scalar_type gamma);
```

### 23.23.5 The implicit Houbolt scheme

For a problem which reads

\[(K + \frac{11}{6 \Delta t} C + \frac{2}{\Delta t^2} M)u_n = F_n + (\frac{5}{\Delta t^2} M + \frac{3}{\Delta t} C)u_{n-1} - (\frac{4}{\Delta t^2} M + \frac{3}{2 \Delta t} C)u_{n-2} + (\frac{1}{\Delta t^2} M + \frac{1}{3 \Delta t} C)u_{n-3}\]

where \( \Delta t \) means a time step, \( M \) the matrix in term of “Dot2_u”, \( C \) the matrix in term of “Dot_u” and \( K \) the matrix in term of “u”. The affine dependences are thus given by:

\[
\text{Dot}_u = 1/(6*\Delta t)*(11*u-18*\text{Previous}_u+9*\text{Previous2}_u-2*\text{Previous3}_u) \\
\text{Dot2}_u = 1/(\Delta t*2)*(2*u-5*\text{Previous}_u+4*\text{Previous2}_u-\text{Previous3}_u)
\]

When applying this scheme to a variable “u” of the model, the following affine dependent variables are added to the model:

"Dot_u", "Dot2_u"

which represent the first and second order time derivative of the variable and can be used in some brick definition.

The following data are also added:

"Previous_u", "Previous2_u", "Previous3_u"
which correspond to the values of “u” at the time step n-1, n-2, n-3.

Before the solve, the data “Previous_u”, “Previous2_u” and “Previous3_u” (corresponding to \( U^0 \) in the example) have to be initialized.

The addition of this scheme to a variable is to be done thanks to:

\[
\text{add_Houbolt_scheme(model \&md, const std::string \&varname);}
\]

### 23.23.6 Transient terms

As it has been explained in previous sections, some intermediate variables are added to the model in order to represent the time derivative of the variables on which the scheme is applied. Once again, if “u” is such a variable, “Dot_u” will represent the time derivative of “u” approximated by the used scheme.

This also means that “Dot_u” (and “Dot2_u” in order two in time problems) can be used to express the transient terms. In the weak form language, the term:

\[
\int_{\Omega} \dot{u} v dx
\]

can be simply expressed by:

\[
\text{Dot_u*Test_u}
\]

Similarly, every existing model brick of \textit{GetFEM++} can be applied to “Dot_u”. This is the case for instance with:

\[
\text{getfem::add_mass_brick(model, mim, "Dot_u");}
\]

which adds the same transient term.

**VERY IMPORTANT:** When adding an existing model brick applied to an affine dependent variable such as “Dot_u”, it is always assumed that the corresponding test function is the one of the corresponding original variable (i.e., “Test_u” here). In other words, “Test_Dot_u”, the test variable corresponding to the velocity, is not used. This corresponds to the choice made to solve the problem in terms of the original variable, so that the test function corresponds to the original variable.

Another example of model brick which can be used to account for a Kelvin-Voigt linearized viscosity term is the linearized elasticity brick:

\[
\text{getfem::add_isotropic_linearized_elasticity_brick(model, mim, "Dot_u", \\
"lambda_viscosity", "mu_viscosity");}
\]

when applied to an order two transient elasticity problem.

### 23.23.7 Computation on the sequence of time steps

Typically, the solve on the different time steps will take the following form:

```
for (scalar_type t = 0.; t < T; t += dt) {
  // time loop
  // Eventually compute here some time dependent terms
}
```

(continues on next page)
iter.init();
getfem::standard_solve(model, iter);

// + Do something with the solution (plot or store it)
model.shift_variables_for_time_integration();

Note that the call of the method:

model.shift_variables_for_time_integration();

is needed between two time step since it will copy the current value of the variables ($u$ and $Dot_u$ for instance) to the previous ones ($Previous_u$ and $Previous_Dot_u$).

### 23.23.8 Boundary conditions

Standard boundary conditions can of course be applied normally to the different variables of the unknown. By default, applying Dirichlet, Neumann or contact boundary conditions to the unknown simply means that the conditions are prescribed on the variable at the current time step $n$.

### 23.23.9 Small example: heat equation

The complete compilable program corresponds to the test program tests/heat_equation.cc of GetFEM++ distribution. See also /interface/tests/matlab/demo_wave_equation.m for an example of order two in time problem with the Matlab interface.

Assuming that $mf_u$ and $mim$ are valid finite element and integration methods defined on a valid mesh, the following code will perform the approximation of the evolution of the temperature on the mesh assuming a unitary diffusion coefficient:

```cpp
getfem::model model;
model.add_fem_variable("u", mf_u, 2); // Main unknown of the problem
getfem::add_generic_elliptic_brick(model, mim, "u"); // Laplace term

// Volumic source term.
getfem::add_source_term_generic_assembly_brick(model, mim, "sin(t)*Test_u \rightarrow");

// Dirichlet condition.
getfem::add_Dirichlet_condition_with_multipliers(model, mim, "u", mf_u, DIRICHLET_BOUNDARY_NUM);

// transient part.
getfem::add_theta_method_for_first_order(model, "u", theta);
getfem::add_mass_brick(model, mim, "Dot_u");

gmm::iteration iter(residual, 0, 40000);
model.set_time(0.); // Init time is 0 (not mandatory)
```

(continues on next page)
model.set_time_step(dt);  // Init of the time step.

// Null initial value for the temperature.
gmm::clear(model.set_real_variable("Previous_u");

// Automatic computation of Previous_Dot_u
model.perform_init_time_derivative(dt/20.);
iter.init();
standard_solve(model, iter);

// Iterations in time
for (scalar_type t = 0.; t < T; t += dt) {
    iter.init();
    getfem::standard_solve(model, iter);

    // + Do something with the solution (plot or store it)
    // Copy the current variables "u" and "Dot_u" into "Previous_u"
    // and "Previous_Dot_u"
    model.shift_variables_for_time_integration();
}

23.23.10 Implicit/explicit some terms

...

23.23.11 Explicit schemes

...

23.23.12 Time step adaptation

...

23.23.13 Quasi-static problems

...

23.24 Small sliding contact with friction bricks

The aim of these bricks is to take into account a contact condition with or without friction of an elastic structure on a rigid foundation or between two elastic structures. These bricks are restricted to small deformation approximation of contact (this may include large deformations on a flat obstacle).
23.24.1 Approximation of contact

For small deformation problems submitted a simple (compared to large deformation) expression of the contact with friction condition is usually used where the tangential displacement do not influence the normal one. This is an approximation in the sense that if an obstacle is not perfectly flat, the tangential displacement of course influence the point where the contact holds. This will not be the case in small deformation where the contact condition can be considered to be described on the reference configuration.

There are mainly two largely used discretizations of the contact with friction condition in this framework: a direct nodal contact condition (usually prescribed on the displacement finite element nodes) or a weak nodal contact condition (usually prescribed on the multiplier finite element nodes). The two discretization leads to similar system. However, the interpretation of quantities is not the same. A third approach is developed on Getfem contact bricks: a weak integral contact condition. It needs the computation of a non-linear integral on the contact boundary at each iteration but the numerical resolution is potentially more scalable because it derives directly from continuous principles.

More details can be found for instance in [KI-OD1988], [KH-PO-RE2006] and [LA-RE2006].

23.24.2 Direct nodal contact condition

A nodal contact condition consists in a certain number of contact nodes \( a_i, i = 1..N_c \) on which a contact with (or without) friction condition is applied. The contact condition reads

\[
\begin{align*}
    u_N(a_i) - \text{gap}_i \leq 0, & \quad \lambda^i_N \leq 0, & \quad (u_N(a_i) - \text{gap}_i)\lambda^i_N = 0,
\end{align*}
\]

where \( \lambda^i_N \) is the equivalent nodal contact force on \( a_i \) and \( u_N(a_i) \) is the normal relative displacement between the elastic solid and an obstacle or between two elastic solids. The term \( \text{gap} \) represents the normal gap between the two solids in the reference configuration. The friction condition reads

\[
\begin{align*}
    \|\lambda^i_T\| \leq -\mathcal{F}\lambda^i_N, & \quad \lambda^i_T = \mathcal{F}\lambda^i_N \frac{\dot{u}_T}{\|\dot{u}_T\|} \quad \text{when } \dot{u}_T \neq 0,
\end{align*}
\]

where \( \dot{u}_T \) is the relative slip velocity, \( \mathcal{F} \) is the friction coefficient and \( \lambda^i_T \) the equivalent nodal friction force on \( a_i \). The friction condition can be summarized by the inclusion

\[
\lambda^i_T \in \mathcal{F}\lambda^i_N \text{Dir}(\dot{u}_T),
\]

where \( \text{Dir}(\dot{u}_T) \) is the multivalued map being the sub-differential of \( x \mapsto \|x\| \) (i.e. \( \text{Dir}(x) = \frac{x}{\|x\|} \) when \( x \neq 0 \) and \( \text{Dir}(0) \) the closed unit ball). For two dimensional cases, \( \text{Dir}(\dot{u}_T) \) reduces to \( \text{Sign}(\dot{u}_T) \) where \( \text{Sign} \) is the multivalued sign map.

A complete linearized elasticity problem with contact with friction reads as

Given an augmentation parameter \( r \), the contact and friction conditions can be equivalently expressed in term of projection as

\[
\begin{align*}
    \frac{1}{r} (\lambda^i_N - P_{[-\infty,0]}(\lambda^i_N - r(u_N(a_i) - \text{gap}_i))) = 0, & \quad \frac{1}{r} (\lambda^i_T - P_{[-\mathcal{F}(-\mathcal{F}\lambda^i_N),0]}(\lambda^i_T - r(u_N(a_i) - \text{gap}_i))) = 0,
\end{align*}
\]

where \( P_K \) is the projection on the convex \( K \) and \( \mathcal{B}(-\mathcal{F}\lambda^i_N) \) is the ball of center 0 and radius \( -\mathcal{F}\lambda^i_N \). These expressions will be used to perform a semi-smooth Newton method.
Suppose now that you approximate a linearized elasticity problem submitted to contact with friction. Then, if $U$ is the vector of the unknown for the displacement you will be able to express the matrices $B_N$ and $B_T$ such that

$$u_N(a_i) = (B_N U)_i,$$

$$(\dot{u}_T(a_i))_k = (B_T \dot{U})_{(d-1)(i-1)+k},$$

where $d$ is the dimension of the domain and $k = 1..d-1$. The expression of the elasticity problem with contact with friction can be written as

$$K U = L + B_N^T \lambda_N + B_T^T \lambda_T,$$

$$-\frac{1}{\tau \alpha_i} (\lambda_N^i - P_{\infty,0} (\lambda_N^i - \alpha_i r((B_N U)_i - \text{gap}_{ij}))) = 0, \quad i = 1..N_c,$$

$$-\frac{1}{\tau \alpha_i} (\lambda_T^i - P e^{-\beta_0} P_{\infty,0} (\lambda_T^i - \alpha_i r((B_T U - B_T U^0)_i))) = 0, \quad i = 1..N_c,$$

where $\alpha_i$ is a parameter which can be added for the homogenization of the augmentation parameter, $(B_T U)_i$ denotes here the sub-vector of indices from $(d-1)(i-1) + 1$ to $(d-1)i$ for the sake of simplicity and the sliding velocity $B_T \dot{U}$ have been discretized into $\frac{(B_T U - B_T U^0)}{\Delta t}$ with $U^0$ the displacement at the previous time step. Note that of course another discretization of the sliding velocity is possible and that the time step $\Delta t$ do not appear in the expression of the friction condition since it does not influence the direction of the sliding velocity.

In that case, the homogenization coefficient $\alpha_i$ can be taken proportional to $h^{d-2}$ ($h$ being the diameter of the element). In this way, the augmentation parameter $\lambda$ can be expressed in $N/m^2$ and chosen closed to the Young modulus of the elastic body. Note that the solution is not sensitive to the value of the augmentation parameter.

### 23.24.3 Weak nodal contact condition

The direct nodal condition may have some drawback: locking phenomena, over-constraint. It is in fact often more stable and for the same accuracy to use multiplier of reduced order compared to the displacement (the direct nodal contact condition corresponds more or less to a multiplier described on the same finite element method than the displacement).

Let $\varphi_i$ be the shapes functions of the finite element describing the displacement, and $\psi_i$ the shape functions of a finite element describing a multiplier on the contact boundary $\Gamma_c$. It is assumed that the set of admissible multiplier describing the normal stress will be

$$\Lambda^h_N = \{ \mu_N^h = \sum \mu_N^j \psi_j : \mu_N^j(a_i) \leq 0, \quad i = 1..N_c \}$$

where $a_i, \quad i = 1..N_c$ are the finite element nodes corresponding to the multiplier. The discrete contact condition is now expressed in a weak form by

$$\int_{\Gamma_c} (\mu_N^h - \lambda_N^h)(u_N - \text{gap}) d\Gamma \geq 0 \quad \forall \mu_N^h \in \Lambda^h_N.$$  

In that case, the component $\lambda_N^i$ is a contact stress ($N/m^2$) and the matrix $B_N$ can be written

$$(B_N)_{ij} = \int_{\Gamma_c} \psi_i \varphi_j d\Gamma.$$  

The matrix $B_T$ can also be written in a similar way. The friction condition can be written in a weak form

$$\int_{\Gamma_c} (\mu_T^h - \lambda_T^h) \dot{u}_T d\Gamma \geq 0 \quad \forall \mu_T^h \in \Lambda^h_T(\mathcal{F} \lambda_T^h),$$  

where $\mathcal{F}$ is a functional containing the join and break laws.
where $\mathcal{A}_h^i(\mathcal{F} \lambda_N^h)$ is the discrete set of admissible friction stress.

Finally, the expression of the direct nodal contact condition are recovered

$$KU = L + B_N^T \lambda_N + B_T^T \lambda_T,$$

$$-\frac{1}{r \alpha_i}(\lambda_N^i - P_{(\mathcal{F} P_{gap})}((\lambda_N^i - \alpha_i r((B_N U)_i - gap_i)))) = 0, \ i = 1...N_c,$$

$$-\frac{1}{r \alpha_i}(\lambda_T^i - P_{\mathcal{F} P_{gap}}((\lambda_N^i - \alpha_i r((B_N U)_i - gap_i))))(\lambda_T^i - \alpha_i r(B_T U - B_T U_0)) = 0, \ i = 1...N_c,$$

except that now $\lambda_N^i$ and $\lambda_T^i$ are force densities, and $\alpha_i$ has to be now chosen proportional to $1/h^d$ such that the augmentation parameter $r$ can still be chosen close to the Young modulus of the elastic body.

Note that without additional stabilization technique (see [HI-RE2010]) an inf-sup condition have to be satisfied between the finite element of the displacement and the one for the multipliers. This means in particular that the finite element for the multiplier have to be “less rich” than the one for the displacement.

### 23.24.4 Weak integral contact condition

The weak integral contact formulation allows not to explicitly describe the discrete set of admissible stress. See also *Generic Nitsche’s method for contact with friction condition*. The contact stress (including the friction one) is described on a finite element space $W^h$ on the contact boundary $\Gamma_c$:

$$\lambda^h \in W^h = \left\{ \sum \lambda_i \psi_i, \lambda_i \in IR^d \right\}$$

where $d$ is the dimension of the problem and $\psi_i$ still the shapes functions on which the contact stress is developed. Now, given a outward unit vector $n$ on the contact boundary $\Gamma_c$ (usually the normal to the obstacle), we make the standard decompositions:

$$\lambda_N^h = \lambda^h \cdot n, \quad \lambda_T^h = \lambda^h - \lambda_N^h n, \quad u_N^h = u^h \cdot n, \quad u_T^h = u^h - u_N^h n,$$

where $u^h$ is the displacement field approximated on a finite element space $V^h$. This allows to express the contact condition in the following way

$$\int_{\Gamma_c} (\lambda_N^h + (\lambda_N^h - r(u_N^h - gap)) - \mu^h d\Gamma = 0 \ \forall \mu^h \in W^h,$$

where $gap$ is a given initial gap in reference configuration, $r$ is an augmentation parameter and $(\cdot)_-$ : $IR \rightarrow IR_-$ is the negative part. The friction condition can similarly be written:

$$\int_{\Gamma_c} (\lambda_T^h - P_{B(\mathcal{F}(\lambda_N^h - r(u_N^h - gap)))}(\lambda_T^h - r\alpha(u_T^h - w_T^h))) \cdot \mu^h d\Gamma = 0 \ \forall \mu^h \in W^h,$$

where $B(\rho)$ is the closed ball of center 0 and radius $\rho$ and $P_{B(\rho)}$ is the orthogonal projection on it (By convention, the ball reduces to the origin for $\rho \leq 0$). The term $\alpha(u_T^h - w_T^h)$ represent here an approximation of the sliding velocity. The parameter $\alpha$ and the field $w_T^h$ have to be adapted with respect to the chosen approximation. For instance, if the standard finite difference

$$(u_T^h)^{n+1} \approx \frac{(u_T^h)^{n+1} - (u_T^h)^n}{dt}$$

is chosen, then one has to take $\alpha = 1/dt$ and $w_T^h = (u_T^h)^n$. Note that due to the symmetry of the ball, the parameter $\alpha$ do not play an important role in the formulation. It can simply be viewed as a scaling between the augmentation parameter for the contact condition and the one for the friction condition. Note also that contrarily to the previous formulations of contact, here there is not a strict
indépendance of the conditions with respect to the augmentation parameter (the independance only occurs at the continuous level).

GetFEM++ bricks implement four versions of the contact condition derived from the Alart-Curnier augmented Lagrangian formulation [AL-CU1991]. The first one corresponds to the non-symmetric version. It consists in solving:

$$a(u^h, v^h) + \int_{\Gamma_c} \lambda^h \cdot v^h d\Gamma = \ell(v^h) \quad \forall v^h \in V^h,$$

$$- \frac{1}{r} \int_{\Gamma_c} (\lambda^h_N + (\lambda^h_N - r(u^h_N - gap)) \mu^h_N) d\Gamma$$

$$- \frac{1}{r} \int_{\Gamma_c} (\lambda^h_T - P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T))) \cdot \mu^h_T d\Gamma = 0 \quad \forall \mu^h \in W^h,$$

where $a(\cdot, \cdot)$ and $\ell(v)$ represent the remaining parts of the problem in $u$, for instance linear elasticity and $\rho = \mathcal{F}(\lambda^h_N - r(u^h_N - gap))$. Note that in this case, the mathematical analysis leads to choose a value for the augmentation parameter of the kind $r = r_0/r$ with $r_0$ having the dimension of a elasticity modulus (a classical choice is the value of Young’s modulus). In order to write a Newton iteration, one has to derive the tangent system. It can be written, reporting only the contact and friction terms and not the right hand side:

$$\cdots - \int_{\Gamma_c} \delta_{\lambda} \cdot v d\Gamma = \cdots \quad \forall v^h \in V^h,$$

$$- \frac{1}{r} \int_{\Gamma_c} (1 - H(r(u^h_N - gap) - \lambda^h_N)) \delta_{\lambda} \lambda^h_N d\Gamma - \int_{\Gamma_c} H(r(u^h_N - gap) - \lambda^h_N) \delta u_N \lambda^h_N d\Gamma$$

$$- \frac{1}{r} \int_{\Gamma_c} (\delta_{\lambda} - D_x P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T))) \delta_{\lambda} \lambda^h_T d\Gamma$$

$$\int_{\Gamma_c} \alpha D_x P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T)) \delta u_T \cdot \mu^h_T d\Gamma$$

$$+ \int_{\Gamma_c} (\mathcal{F} D_{\rho} P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T)) \delta u_N) \cdot \mu^h_T d\Gamma$$

$$- \int_{\Gamma_c} \left( \mathcal{F} D_{\rho} P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T)) \delta_{\lambda} \lambda^h_T \right) \cdot \mu^h_T d\Gamma = \cdots \quad \forall \mu^h \in W^h,$$

where $H(\cdot)$ is the Heaviside function (0 for a negative argument and 1 for a non-negative argument), $D_x P_B(\rho)(x)$ and $D_{\rho} P_B(\rho)(x)$ are the derivatives of the projection on $B(\rho)$ (assumed to vanish for $\rho \leq 0$) and $\delta_{\lambda}$ and $\delta u$ are the unknown corresponding to the tangent problem.

The second version corresponds to the “symmetric” version. It is in fact symmetric in the frictionless case only (because in this case it directly derives from the augmented Lagrangian formulation). It reads:

$$a(u^h, v^h) + \int_{\Gamma_c} (\lambda^h_N - r(u^h_N - gap)) \cdot v^h d\Gamma$$

$$- \int_{\Gamma_c} P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T)) \cdot v^h d\Gamma = \ell(v^h) \quad \forall v^h \in V^h,$$

$$- \frac{1}{r} \int_{\Gamma_c} (\lambda^h_N + (\lambda^h_N - r(u^h_N - gap)) \mu^h_N) d\Gamma$$

$$- \frac{1}{r} \int_{\Gamma_c} (\lambda^h_T - P_B(\rho) (\lambda^h_T - r \alpha(u^h_T - w^h_T))) \cdot \mu^h_T d\Gamma = 0 \quad \forall \mu^h \in W^h,$$
and the tangent system:

\[
\begin{aligned}
&\cdots + \int_{\Gamma_e} rH(u_N^h - gap)\delta_{u_N}v_N d\Gamma \\
&- \int_{\Gamma_e} r\alpha D_T P_B(\lambda_T^h - r\alpha(u_T^h - w_T^h))\delta_{u_T}v_T^h d\Gamma \\
&+ \int_{\Gamma_e} (r\mathcal{F} P_B(\lambda_T^h - r\alpha(u_T^h - w_T^h))\delta_{u_N}v_N^h) - d\Gamma = \cdots \quad \forall v^h \in V^h,
\end{aligned}
\]

still with \( \rho = \mathcal{F}(\lambda_N^h - r(u_N^h - gap)) \).

The third version corresponds to a penalized contact and friction condition. It does not require the use of a multiplier. In this version, the parameter \( r \) is a penalization parameter and as to be large enough to perform a good approximation of the non-penetration and the Coulomb friction conditions. The formulation reads:

\[
\begin{aligned}
&\left\{ a(u^h, v^h) + \int_{\Gamma_e} r(u_N^h - gap)_+ v_N^h d\Gamma \\
&\quad + \int_{\Gamma_e} P_B(\mathcal{F}(u_N^h - gap)_+) (r\alpha(u_T^h - w_T^h)) \cdot v_T^h d\Gamma = \ell(v^h) \quad \forall v^h \in V^h,
\end{aligned}
\]

and the tangent system:

\[
\begin{aligned}
&\cdots + \int_{\Gamma_e} rH(u_N^h - gap)\delta_{u_N}v_N d\Gamma \\
&- \int_{\Gamma_e} r\alpha D_T P_B(\mathcal{F}(u_N^h - gap)_+) (r\alpha(u_T^h - w_T^h))\delta_{u_T}v_T^h d\Gamma \\
&+ \int_{\Gamma_e} (r\mathcal{F} H(u_N^h - gap)_+ P_B(\mathcal{F}(u_N^h - gap)_+) (r\alpha(u_T^h - w_T^h))\delta_{u_N}v_N^h) - d\Gamma = \cdots \quad \forall v^h \in V^h;
\end{aligned}
\]

### 23.24.5 Numerical continuation

In addition, GetFEM++ develops a method of numerical continuation for finding numerical solutions of discretized evolutionary contact problems based on the weak integral contact condition (see Numerical continuation and bifurcation for a general introduction). For this purpose, a parameter-dependent sliding velocity may be added to the friction condition so that it becomes:

\[
\int_{\Gamma_e} \left( \lambda_N^h - P_B(\mathcal{F}\lambda_N^h) (\lambda_T^h - r(\alpha(u_T^h - w_T^h) + (1 - \gamma)z_T^h)) \right) \cdot \mu_T^h d\Gamma = 0 \quad \forall \mu^h \in W^h.
\]
Here, $\gamma$ is a parameter and $z^h_T$ is an initial sliding velocity. It is worth mentioning that if one chooses

$$\alpha = \frac{1}{d^t}, \quad w^h_T = (u^h_T)^n, \quad z^h_T = \frac{(u^h_T)^n - (u^h_{T-1})^n}{d^t},$$

then he recovers the standard friction condition at time $t_n$ and $t_{n+1}$ for $\gamma$ equal to 0 and 1, respectively.

### 23.24.6 Friction law

Apart from pure Coulomb friction $\rho = F |\sigma_n|$, the weak integral contact framework in GetFEM++ also supports a more generic friction law description:

$$\rho = \begin{cases} 
\tau_{ad\text{h}} + F |\sigma_n| & \text{if } \tau_{ad\text{h}} + F |\sigma_n| < \tau_{tresca} \\
\tau_{tresca} & \text{otherwise}
\end{cases}$$

In this equation $\rho$ is the admissible friction stress for a given normal stress $\sigma_n$, $F$ is the coefficient of friction, $\tau_{ad\text{h}}$ is an adhesional (load-independent) shear stress and $\tau_{tresca}$ is a maximum shear stress limit.

![Friction law diagram]

### 23.24.7 Add a contact with or without friction to a model

### 23.24.8 Frictionless basic contact brick

In order to add a frictionless contact brick you call the model object method:

```cpp
getfem::add_basic_contact_brick
    (md, varname_u, multname_n, dataname_r, BN, dataname_gap, dataname_alpha, aug_version);
```

This function adds a frictionless contact brick on `varname_u` thanks to a multiplier variable `multname_n`. If $U$ is the vector of degrees of freedom on which the unilateral constraint is applied, the matrix $B_N$ have to be such that this condition is defined by $B_N U \leq 0$. The constraint is prescribed thanks to a multiplier `multname_n` whose dimension should be equal to the number of lines of $B_N$. The variable `dataname_r` is the name of the augmentation parameter $r$ should be chosen in a range of acceptable values. `dataname_gap` is an optional parameter representing the initial gap. It can be a single value or a vector of value. `dataname_alpha` is an optional homogenization parameter for the augmentation parameter.

The parameter `aug_version` indicates the augmentation strategy : 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method based on augmented multipliers.

Note that is possible to change the basic contact matrix $B_N$ by using: 144
23.24.9 Basic contact brick with friction

```cpp
getfem::contact_brick_set_BN(md, indbrick);
```

23.24.9.1 Basic contact brick with friction

```cpp
getfem::add_basic_contact_brick(md, varname_u, multname_n, multname_t,
    dataname_r, BN, dataname_friction_coeff, dataname_gap, dataname_alpha,
    aug_version);
```

This function adds a contact brick with friction on `varname_u` thanks to two multiplier variables `multname_n` and `multname_t`. If $U$ is the vector of degrees of freedom on which the condition is applied, the matrix $B_N$ has to be such that the contact condition is defined by $B_N U \leq \text{gap}$ and $B_T$ have to be such that the relative tangential displacement is $B_T U$. The matrix $B_T$ should have as many rows as $B_N$ multiplied by $d - 1$ where $d$ is the domain dimension. The contact condition is prescribed thanks to a multiplier `multname_n` whose dimension should be equal to the number of rows of $B_N$ and the friction condition by a multiplier `multname_t` whose size should be the number of rows of $B_T$. The parameter `dataname_friction_coeff` describes the friction coefficient. It could be a scalar or a vector describing the coefficient on each contact condition. The augmentation parameter $r$ should be chosen in a range of acceptable values (see Getfem user documentation). `dataname_gap` is an optional parameter representing the initial gap. It can be a single value or a vector of value. `dataname_alpha` is an optional homogenization parameter for the augmentation parameter.

The parameter `aug_version` indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method based on augmented multipliers and 4 for the unsymmetric method based on augmented multipliers with De Saxce projection.

Note that is possible to change the basic contact matrices $B_N$ and $B_T$ by using:

```cpp
getfem::contact_brick_set_BN(md, indbrick);
getfem::contact_brick_set_BT(md, indbrick);
```

23.24.10 Frictionless nodal contact with a rigid obstacle brick

```cpp
getfem::add_nodal_contact_with_rigid_obstacle_brick(md, mim, varname_u, multname_n, dataname_r, region, obstacle, aug_version);
```

This function adds a direct nodal frictionless contact condition with a rigid obstacle to the model. The condition is applied on the variable `varname_u` on the boundary corresponding to `region`. The rigid obstacle should be described with the string `obstacle` being a signed distance to the obstacle. This string should be an expression where the coordinates are ‘x’, ‘y’ in 2D and ‘x’, ‘y’, ‘z’ in 3D. For instance, if the rigid obstacle correspond to $z \leq 0$, the corresponding signed distance will be simply ‘z’. `multname_n` should be a fixed size variable whose size is the number of degrees of freedom on boundary `region`. It represents the contact equivalent nodal forces. The augmentation parameter $r$ should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method based on augmented multipliers.

23.24.11 Nodal contact with a rigid obstacle brick with friction

```cpp
getfem::add_nodal_contact_with_rigid_obstacle_brick(md, mim, varname_u, multname_n, multname_t, dataname_r, dataname_friction_coeff, region, obstacle,
```

23.24. Small sliding contact with friction bricks
This function adds a direct nodal contact with friction condition with a rigid obstacle to the model. The condition is applied on the variable \texttt{varname\_u} on the boundary corresponding to \texttt{region}. The rigid obstacle should be described with the string \texttt{obstacle} being a signed distance to the obstacle. This string should be an expression where the coordinates are \texttt{\_x}, \texttt{\_y} in 2D and \texttt{\_x}, \texttt{\_y}, \texttt{\_z} in 3D. For instance, if the rigid obstacle correspond to \(z \leq 0\), the corresponding signed distance will be simply \texttt{\_z}. \texttt{multname\_n} should be a fixed size variable whose size is the number of degrees of freedom on boundary \texttt{region}. It represents the contact equivalent nodal forces. \texttt{multname\_t} should be a fixed size variable whose size is the number of degrees of freedom on boundary \texttt{region} multiplied by \(d - 1\) where \(d\) is the domain dimension. It represents the friction equivalent nodal forces. The augmentation parameter \(r\) should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). \texttt{dataname\_friction\_coeff} is the friction coefficient. It could be a scalar or a vector of values representing the friction coefficient on each contact node.

The parameter \texttt{aug\_version} indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method based on augmented multipliers and 4 for the unsymmetric method based on augmented multipliers with De Saxce projection.

### 23.24.12 Frictionless nodal contact between non-matching meshes brick

```cpp
getfem::add_nodal_contact_between_nonmatching_meshes_brick (md, mim1, mim2, varname\_u1, varname\_u2, multname\_n, dataname\_r, rg1, rg2, slave1=true, slave2=false, aug\_version=1);
```

This function adds a frictionless contact condition between two faces of one or two elastic bodies. The condition is applied on the variable \texttt{varname\_u} or the variables \texttt{varname\_u1} and \texttt{varname\_u2} depending if a single or two distinct displacement fields are given. Vectors \texttt{rg1} and \texttt{rg2} contain pairs of regions expected to come in contact with each other. In case of a single region per side, \texttt{rg1} and \texttt{rg2} can be given as normal integers. In the single displacement variable case the regions defined in both \texttt{rg1} and \texttt{rg2} refer to the variable \texttt{varname\_u}. In the case of two displacement variables, \texttt{rg1} refers to \texttt{varname\_u1} and \texttt{rg2} refers to \texttt{varname\_u2}. \texttt{multname\_n} should be a fixed size variable whose size is the number of degrees of freedom on those regions among the ones defined in \texttt{rg1} and \texttt{rg2} which are characterized as “slaves”. It represents the contact equivalent nodal forces. The augmentation parameter \(r\) should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). The optional parameters \texttt{slave1} and \texttt{slave2} declare if the regions defined in \texttt{rg1} and \texttt{rg2} are correspondingly considered as “slaves”. By default \texttt{slave1} is true and \texttt{slave2} is false, i.e. \texttt{rg1} contains the slave surfaces, while \texttt{rg2} the master surfaces. Preferably only one of \texttt{slave1} and \texttt{slave2} is set to true.

The parameter \texttt{aug\_version} indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method with augmented multiplier. Basically, this brick computes the matrix \(B_N\) and the vectors gap and alpha and calls the basic contact brick.

### 23.24.13 Nodal contact between non-matching meshes brick with friction

```cpp
getfem::add_nodal_contact_between_nonmatching_meshes_brick
```

This function adds a direct nodal contact with friction condition with a rigid obstacle to the model. The condition is applied on the variable \texttt{varname\_u} on the boundary corresponding to \texttt{region}. The rigid obstacle should be described with the string \texttt{obstacle} being a signed distance to the obstacle. This string should be an expression where the coordinates are \texttt{\_x}, \texttt{\_y} in 2D and \texttt{\_x}, \texttt{\_y}, \texttt{\_z} in 3D. For instance, if the rigid obstacle correspond to \(z \leq 0\), the corresponding signed distance will be simply \texttt{\_z}. \texttt{multname\_n} should be a fixed size variable whose size is the number of degrees of freedom on boundary \texttt{region}. It represents the contact equivalent nodal forces. \texttt{multname\_t} should be a fixed size variable whose size is the number of degrees of freedom on boundary \texttt{region} multiplied by \(d - 1\) where \(d\) is the domain dimension. It represents the friction equivalent nodal forces. The augmentation parameter \(r\) should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). \texttt{dataname\_friction\_coeff} is the friction coefficient. It could be a scalar or a vector of values representing the friction coefficient on each contact node.

The parameter \texttt{aug\_version} indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method with augmented multipliers and 4 for the unsymmetric method based on augmented multipliers with De Saxce projection.
aug_version=1);

This function adds a contact with friction condition between two faces of one or two elastic bodies. The condition is applied on the variable varname_u or the variables varname_u1 and varname_u2 depending if a single or two distinct displacement fields are given. Vectors rg1 and rg2 contain pairs of regions expected to come in contact with each other. In case of a single region per side, rg1 and rg2 can be given as normal integers. In the single displacement variable case the regions defined in both rg1 and rg2 refer to the variable varname_u. In the case of two displacement variables, rg1 refers to varname_u1 and rg2 refers to varname_u2. multname_n should be a fixed size variable whose size is the number of degrees of freedom on those regions among the ones defined in rg1 and rg2 which are characterized as “slaves”. It represents the contact equivalent nodal normal forces. multname_t should be a fixed size variable whose size corresponds to the size of multname_n multiplied by qdim - 1. It represents the contact equivalent nodal tangent (frictional) forces. The augmentation parameter r should be chosen in a range of acceptable values (close to the Young modulus of the elastic body, see Getfem user documentation). The friction coefficient stored in the parameter friction_coeff is either a single value or a vector of the same size as multname_n. The optional parameters slave1 and slave2 declare if the regions defined in rg1 and rg2 are correspondingly considered as “slaves”. By default slave1 is true and slave2 is false, i.e. rg1 contains the slave surfaces, while rg2 the master surfaces. Preferably only one of slave1 and slave2 is set to true.

The parameter aug_version indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method with augmented multiplier and 4 for the unsymmetric method with augmented multiplier and De Saxce projection.

Basically, this brick computes the matrices \( B_N \) and \( B_T \) as well the vectors gap and alpha and calls the basic contact brick.

### 23.24.14 Hughes stabilized frictionless contact condition

In order to add a Hughes stabilized frictionless contact brick you call the model object method:

```cpp
getfem::add_Hughes_stab_basic_contact_brick
  (md, varname_u, multname_n, dataname_r, BN, DN, dataname_gap, dataname_
  \( \rightarrow \) alpha, aug_version);
```

This function adds a Hughes stabilized frictionless contact brick on varname_u thanks to a multiplier variable multname_n. If we take \( U \) is the vector of degrees of freedom on which the unilateral constraint is applied, and \( \lambda \) the multiplier Vector of contact force. Then Hughes stabilized frictionless contact condition is defined by the matrix \( B_N \) and \( D_N \) have to be such that this condition is defined by \( B_N U - D_N \lambda \leq 0 \). Where \( D_N \) is the mass matrix relative to stabilized term. The variable dataname_r is the name of the augmentation parameter \( r \) should be chosen in a range of acceptable values. dataname_gap is an optional parameter representing the initial gap. It can be a single value or a vector of value. dataname_alpha is an optional homogenization parameter for the augmentation parameter.

The parameter aug_version indicates the augmentation strategy: 1 for the non-symmetric Alart-Curnier augmented Lagrangian, 2 for the symmetric one, 3 for the unsymmetric method based on augmented multipliers.

Note that the matrix \( D_N \) is a sum of the basic contact term and the Hughes stabilised term. You can change it with:

```cpp
getfem::contact_brick_set_DN(md, indbrick);
```

### 23.24. Small sliding contact with friction bricks
23.24.15 Frictionless integral contact with a rigid obstacle brick

```cpp
getfem::add_integral_contact_with_rigid_obstacle_brick
    (md, mim, varname_u, multname_n, dataname_obs, dataname_r, region,
     → option = 1);
```

This function adds a frictionless contact condition with a rigid obstacle to the model, which is defined in an integral way. It is the direct approximation of an augmented Lagrangian formulation defined at the continuous level. The advantage should be a better scalability: the number of Newton iterations should be more or less independent of the mesh size. The condition is applied on the variable `varname_u` on the boundary corresponding to `region`. The rigid obstacle should be described with the data `dataname_obs` being a signed distance to the obstacle (interpolated on a finite element method). `multname_n` should be a fem variable representing the contact stress. An inf-sup condition between `multname_n` and `varname_u` is required. The augmentation parameter `dataname_r` should be chosen in a range of acceptable values.

Possible values for `option` is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new unsymmetric method. The default value is 1.

`mim` represents of course the integration method. Note that it should be accurate enough to integrate efficiently the nonlinear terms involved.

23.24.16 Integral contact with a rigid obstacle brick with friction

```cpp
getfem::add_integral_contact_with_rigid_obstacle_brick
    (md, mim, varname_u, multname_n, dataname_obs, dataname_r,
     dataname_friction_coeffs, region, option = 1, dataname_alpha = "",
     dataname_wt = "", dataname_gamma = "", dataname_vt = "");
```

This function adds a contact with friction condition with a rigid obstacle to the model, which is defined in an integral way. It is the direct approximation of an augmented Lagrangian formulation defined at the continuous level. The advantage should be a better scalability: the number of Newton iterations should be more or less independent of the mesh size. The condition is applied on the variable `varname_u` on the boundary corresponding to `region`. The rigid obstacle should be described with the data `dataname_obs` being a signed distance to the obstacle (interpolated on a finite element method). `multname_n` should be a fem variable representing the contact stress. An inf-sup condition between `multname_n` and `varname_u` is required. The augmentation parameter `dataname_r` should be chosen in a range of acceptable values.

The parameter `dataname_friction_coeffs` contains the Coulomb friction coefficient and optionally an adhesional shear stress threshold and the tresca limit shear stress. For constant coefficients its size is from 1 to 3. For coefficients described on a finite element method, this vector contains a number of single values, value pairs or triplets equal to the number of the corresponding mesh_fem’s basic dofs.

Possible values for `option` is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new unsymmetric method. The default value is 1. Option 4, assumes pure Coulomb friction and ignores any adhesional stress and tresca limit coefficients.

`dataname_alpha` and `dataname_wt` are optional parameters to solve evolutionary friction problems. `dataname_gamma` and `dataname_vt` denote optional data for adding a parameter-dependent
sliding velocity to the friction condition. \texttt{mim} represents of course the integration method. Note that it should be accurate enough to integrate efficiently the nonlinear terms involved.

### 23.24.17 Frictionless integral contact between non-matching meshes brick

```cpp
getfem::add_integral_contact_between_nonmatching_meshes_brick
(md, mim, varname_u1, varname_u2, multname_n, dataname_r,
 region1, region2, option = 1);
```

This function adds a frictionless contact condition between nonmatching meshes to the model, which is defined in an integral way. It is the direct approximation of an augmented Lagrangian formulation defined at the continuous level. The advantage should be a better scalability: the number of Newton iterations should be more or less independent of the mesh size. The condition is applied on the variables \texttt{varname_u1} and \texttt{varname_u2} on the boundaries corresponding to \texttt{region1} and \texttt{region2}. \texttt{multname_n} should be a fem variable representing the contact stress. An inf-sup condition between \texttt{multname_n} and \texttt{varname_u1} and \texttt{varname_u2} is required. The augmentation parameter \texttt{dataname_r} should be chosen in a range of acceptable values.

Possible values for \texttt{option} is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new nonsymmetric method. The default value is 1.

\texttt{mim} represents of course the integration method. Note that it should be accurate enough to integrate efficiently the nonlinear terms involved.

### 23.24.18 Integral contact between non-matching meshes brick with friction

```cpp
getfem::add_integral_contact_between_nonmatching_meshes_brick
(md, mim, varname_u1, varname_u2, multname, dataname_r,
 dataname_friction_coeffs, region1, region2, option = 1,
 dataname_alpha = "", dataname_wt1 = "", dataname_wt2 = "");
```

This function adds a contact with friction condition between nonmatching meshes to the model. This brick adds a contact which is defined in an integral way. It is the direct approximation of an augmented Lagrangian formulation defined at the continuous level. The advantage should be a better scalability: the number of Newton iterations should be more or less independent of the mesh size. The condition is applied on the variables \texttt{varname_u1} and \texttt{varname_u2} on the boundaries corresponding to \texttt{region1} and \texttt{region2}. \texttt{multname} should be a fem variable representing the contact and friction stress. An inf-sup condition between \texttt{multname} and \texttt{varname_u1} and \texttt{varname_u2} is required. The augmentation parameter \texttt{dataname_r} should be chosen in a range of acceptable values.

The parameter \texttt{dataname_friction_coeffs} contains the Coulomb friction coefficient and optionally an adhesional shear stress threshold and the tresca limit shear stress. For constant coefficients its size is from 1 to 3. For coefficients described on a finite element method on the same mesh as \texttt{varname_u1}, this vector contains a number of single values, value pairs or triplets equal to the number of the corresponding mesh\_fem’s basic dofs.

Possible values for \texttt{option} is 1 for the non-symmetric Alart-Curnier augmented Lagrangian method, 2 for the symmetric one, 3 for the non-symmetric Alart-Curnier method with an additional augmentation and 4 for a new unsymmetric method. The default value is 1. \texttt{dataname_alpha}, \texttt{dataname_wt1} and \texttt{dataname_wt2} are optional parameters to solve evolutionary friction problems. \texttt{mim} represents
the integration method on the same mesh as `varname_u1`. Note that it should be accurate enough to integrate efficiently the nonlinear terms involved.

### 23.24.19 Frictionless penalized contact with a rigid obstacle brick

```cpp
getfem::add_penalized_contact_with_rigid_obstacle_brick
    (md, mim, varname_u, dataname_obs, dataname_r, region,
     option = 1, dataname_lambda_n = "");
```

This function adds a frictionless penalized contact condition with a rigid obstacle to the model. The condition is applied on the variable `varname_u` on the boundary corresponding to `region`. The rigid obstacle should be described with the data `dataname_obstacle` being a signed distance to the obstacle (interpolated on a finite element method). The penalization parameter `dataname_r` should be chosen large enough to prescribe an approximate non-penetration condition but not too large not to deteriorate too much the conditioning of the tangent system. `dataname_n` is an optional parameter used if option is 2. In that case, the penalization term is shifted by `lambda_n` (this allows the use of an Uzawa algorithm on the corresponding augmented dLagrangian formulation).

### 23.24.20 Penalized contact with a rigid obstacle brick with friction

```cpp
getfem::add_penalized_contact_with_rigid_obstacle_brick
    (md, mim, varname_u, dataname_obs, dataname_r, dataname_friction_coeffs,
     region, option = 1, dataname_lambda = "", dataname_alpha = "",
     dataname_wt = "");
```

This function adds a penalized contact condition with Coulomb friction with a rigid obstacle to the model. The condition is applied on the variable `varname_u` on the boundary corresponding to `region`. The rigid obstacle should be described with the data `dataname_obstacle` being a signed distance to the obstacle (interpolated on a finite element method). The parameter `dataname_friction_coeffs` contains the Coulomb friction coefficient and optionally an adhesional shear stress threshold and the tresca limit shear stress. For constant coefficients its size is from 1 to 3. For coefficients described on a finite element method, this vector contains a number of single values, value pairs or triplets equal to the number of the corresponding mesh_fem’s basic dofs.

The penalization parameter `dataname_r` should be chosen large enough to prescribe approximate non-penetration and friction conditions but not too large not to deteriorate too much the conditioning of the tangent system. `dataname_lambda` is an optional parameter used if option is 2. In that case, the penalization term is shifted by `lambda` (this allows the use of an Uzawa algorithm on the corresponding augmented Lagrangian formulation). `dataname_alpha` and `dataname_wt` are optional parameters to solve evolutionary friction problems.

### 23.24.21 Frictionless penalized contact between non-matching meshes brick

```cpp
getfem::add_penalized_contact_between_nonmatching_meshes_brick
    (md, mim, varname_u1, varname_u2, dataname_r,
     region1, region2, option = 1, dataname_lambda_n = "");
```
This function adds a penalized contact frictionless condition between nonmatching meshes to the model. The condition is applied on the variables `varname_u1` and `varname_u2` on the boundaries corresponding to `region1` and `region2`. The penalization parameter "dataname_r" should be chosen large enough to prescribe an approximate non-penetration condition but not too large not to deteriorate too much the conditioning of the tangent system. `dataname_n` is an optional parameter used if option is 2. In that case, the penalization term is shifted by `lambda_n` (this allows the use of an Uzawa algorithm on the corresponding augmented Lagrangian formulation).

### 23.24.22 Penalized contact between non-matching meshes brick with friction

```cpp
getfem::add_penalized_contact_between_nonmatching_meshes_brick
    (md, mim, varname_u1, varname_u2, dataname_r, dataname_friction_coeffs,
    region1, region2, option = 1, dataname_lambda = "",
    dataname_alpha = "", dataname_wt1 = "", dataname_wt2 = "");
```

This function adds a penalized contact condition with Coulomb friction between nonmatching meshes to the model. The condition is applied on the variables `varname_u1` and `varname_u2` on the boundaries corresponding to `region1` and `region2`. The penalization parameter "dataname_r" should be chosen large enough to prescribe an approximate non-penetration condition but not too large not to deteriorate too much the conditioning of the tangent system.

The parameter `dataname_friction_coeffs` contains the Coulomb friction coefficient and optionally an adhesional shear stress threshold and the tresca limit shear stress. For constant coefficients its size is from 1 to 3. For coefficients described on a finite element method on the same mesh as `varname_u1`, this vector contains a number of single values, value pairs or triplets equal to the number of the corresponding mesh_fem’s basic dofs.

`dataname_lambda` is an optional parameter used if `option` is 2. In that case, the penalization term is shifted by `lambda` (this allows the use of an Uzawa algorithm on the corresponding augmented Lagrangian formulation) `dataname_alpha`, `dataname_wt1` and `dataname_wt2` are optional parameters to solve evolutionary friction problems. `mim` represents the integration method on the same mesh as `varname_u1`. Note that it should be accurate enough to integrate efficiently the nonlinear terms involved.

### 23.25 Large sliding/large deformation contact with friction bricks

The basic tools to deal with large sliding/large deformation contact of deformable structures are accessible in the weak form language. Some interpolate transformations (see Interpolate transformations) are defined to perform the contact detection and allow to integrate from a contact boundary to the opposite contact boundary. Some other useful tools such as the unit normal vector in the real configuration and projections to take into account contact with Coulomb friction are also defined as operators in the weak form language.

Of course, the computational cost of large sliding/large deformation contact algorithms is greatly higher than small sliding-small deformation ones.
In order to incorporate the contact detection in the high-level generic assembly, a specific interpolate transformation has been defined (see *Interpolate transformations* for more explanations on interpolate transformations). It is based on a raytracing contact detection has described in [KO-RE2014] and uses the criteria described below. The interpolate transformation stores the different potential contact surfaces. On most of methods, potential contact surface are classified into two categories: master and slave surface (see *figure*).

The slave surface is the “contactor” and the master one the “target”. Rigid obstacle are also considered. They are always master surfaces. The basic rule is that the contact is considered between a slave surface and a master one. However, the multi-contact frame object and the *GetFEM++* bricks allow multi-contact situations, including contact between two master surfaces, self-contact of a master surface and an arbitrary number of slave and master surfaces.

Basically, in order to detect the contact pairs, Gauss points or f.e.m. nodes of slave surfaces are projected on master surfaces (see *figure*). If self-contact is considered, Gauss points or f.e.m. nodes of master surface are also projected on master surfaces.

The addition of a raytracing transformation to a model:

```c++
void add_raytracing_transformation(model &md, const std::string &transname, scalar_type d)
```

where *transname* is a name given to the transformation which allows to refer to it in the weak form language and *d* is the release distance (see above).

The raytracing transformation is added without any slave or master contact boundary. The following functions allows to add some boundaries to the transformation:
where \texttt{dispname} is the variable name which represent the displacement on that contact boundary. The difference between master and slave contact boundary is that the contact detection is to be performed starting from a slave or master boundary toward a master boundary. The contact detection is not performed toward a slave boundary. Consequently, only the influence boxes of the elements of the master surfaces are computed and stored.

It is also possible to add a rigid obstacle (considered as a master surface) thanks to the function:

\begin{verbatim}
add_rigid_obstacle_to_raytracing_transformation(model &md,  
    const std::string &transname,  
    const std::string &expr,  
    size_type N)
\end{verbatim}

where \texttt{expr} is the expression of a signed distance to the obstacle using the syntax of the weak form language (\(X\) being the current position, \(X(0), X(1)\ldots\) the corresponding components). For instance an expression \(X(0) + 5\) will correspond to a flat obstacle lying on the right of the position \(-5\) of the first coordinate. Be aware that the expression have to be close to a signed distance, which in particular means that the gradient norm have to be close to 1.

In order to distinguish between non-contact situations and the occurrence of a contact with another deformable body or with a rigid obstacle, the transformation returns an integer identifier which can be used by the \texttt{Interpolate\_filter} command of the weak form language (see \textit{Interpolate transformations}). The different values:

- 0 : no contact found on this Gauss point
- 1 : contact occurs on this Gauss point with a deformable body
- 2 : contact occurs on this Gauss point with a rigid obstacle.

such that it is possible to differentiate the treatment of these three cases using:

\begin{verbatim}
Interpolate\_filter(transname, expr1, 0)
Interpolate\_filter(transname, expr2, 1)
Interpolate\_filter(transname, expr3, 2)
\end{verbatim}

in the weak form language, where \texttt{expr1, expr2} and \texttt{expr3} correspond to the different terms to be computed. The matlab interface demo program \texttt{/interface/tests/matlab/demo\_large\_sliding\_contact.m} presents an example of use.

Note that the transformation could also be directly used with a \texttt{ga\_workspace} object if model object are not used. See \texttt{getfem/getfem\_contact\_and\_friction\_common.h} for more details. Note also that in the framework of the model object, a interfaced use of this transformation is allowed by the model bricks described below.

\section*{23.25.2 The contact pair detection algorithm}

A contact pair is formed by a point of a slave (or master in case of self-contact) surface and a projected point on the nearest master surface (or rigid obstacle). The Algorithm used is summerized in \textit{figure}
23. The model description and basic model bricks

Generate deformed points corresponding to F.E.M. nodes or Gauss points on master and slave surfaces

Run a Delaunay triangulation on the set of points

Examine the set of edges and with a compatibility test of unit normal or normal cones, produce potential pairs

Generate an influence box for each face of element on a master surface

Generate deformed points corresponding to F.E.M. nodes or Gauss points on slave surfaces (and master ones in case of self-contact)

For each computed deformed point select the influence boxes containing the point, test the compatibility of unit normal or normal cone and produce potential pairs

For each slave point (or master point in the case of self-contact), detect the potential contact with a rigid obstacle. Then apply a set of criteria to each potential pair detected. Select the nearest potential pair or rigid obstacle and store it.
It is impossible to distinguish without fail between valid and invalid contact situations without a global topological criterion (such as in [Pantz2008]), a fortiori for self-contact detection. However, this kind of criterion can be very costly to implement. Thus, one generally implements some simple heuristic criteria which cannot cover all the possible cases. We present such a set of criteria here. They are of course perfectible and subject to change. First, in *figure* one can see a certain number of situations of valid or invalid contact that criteria have to distinguish.

Some details on the algorithm:

- **Computation of influence boxes.** The influence box of an element is just an offset to its bounding box at a distance equal to the release distance. If this strategy is used, the release distance should not be too large compared to the element size. Otherwise, a point would correspond to a large number of influence box which can considerably slow down the search of contact pairs. The influence boxes are stored in a region tree object in order to find the boxes containing a point with an algorithm having a mean complexity in $O(\log(N))$.

- **What is a potential contact pair.** A potential contact pair is a pair slave point - master element face which will be investigated. The projection of the slave point on the master surface will be done and criteria will be applied.
Chapter 23. The model description and basic model bricks
**Projection algorithm.** The projection of the slave point onto a master element face is done by a parametrization of the surface on the reference element via the geometric transformation and the displacement field. During the projection, no constraint is applied to remain inside the element face, which means that the element face is prolonged analytically. The projection is performed by minimizing the distance between the slave point and the projected one using the parametrization and Newton’s and/or BFGS algorithms. If *raytrace* is set to true, then no projection is computed. Instead a ray tracing from the point x in the direction of the unit normal vector at x to find y. This means the reverse of the usual situation (x will be the projection of y).

The list of criteria:

- **Criterion 1:** *the unit normal cone/vector should be compatible, and the two points do not share the same element.* Two unit normal vector are compatible if their scalar product are non-positive. In case of f.e.m. node contact, since a fem node is shared generally by several elements, a normal cone constituted of the unit normal vectors of each element is considered. Two normal cones are compatible if at least one pair of unit normal vector have their scalar product non-positive. In order to simplify the computation, a normal cone is reduced to a mean normal vector if the solid angle of the normal cone is less than *cut_angle* a parameter of the multi-contact frame object. This criterion allows to treat cases (B) and (K1).

- **Criterion 2:** *the contact pair is eliminated when the search of the projection/raytrace point do not converge.* When Newton’s algorithms (and BFGS one for projection) used to compute the projection/raytrace of the slave point on the master element surface fails to converge, the pair is not considered. A warning is generated.

- **Criterion 3:** *the projected point should be inside the element.* The slave point is projected on the surface of the master element without the constraint to remain inside the face (which means that the face is prolonged). If the orthogonal projection is outside the face, the pair is not considered. This is the present state, however, to treat case (J3) an additional treatment will have to be considered (projection on the face with the constraint to remain inside it and test of the normal cone at this point) This criterion allows to treat cases (F2), (K₂), (M₁) and (M₂).

- **Criterion 4:** *the release distance is applied.* If the distance between the slave point and its projection on the master surface is greater than the release distance, the contact pair is not considered. This can treat cases (C), (E), (F₁), (G), (H) if the release distance is adapted and the deformation not too important.

- **Criterion 5:** *comparison with rigid obstacles.* If the signed distance between the slave point and its projection on the master surface is greater than the one with a rigid obstacle (considering that the release distance is also first applied to rigid obstacle) then the contact pair is not considered.

- **Criterion 6:** *for self-contact only : apply a test on unit normals in reference configuration.* In case of self contact, a contact pair is eliminated when the slave point and the master element belong to the same mesh and if the slave point is behind the master surface (with respect to its unit outward normal vector) and not four times farther than the release distance. This can treat cases (A), (C), (D), (H).

- **Criterion 7:** *smallest signed distance on contact pairs.* Between the retained contact pairs (or rigid obstacle) the one corresponding to the smallest signed distance is retained.

### Nodal contact brick with projection

Notations: \( \Omega \subset \mathbb{R}^d \) denotes the reference configuration of a deformable body, possibly constituted by several unconnected parts (see *figure*). \( \Omega_δ \) is the deformed configuration and \( \varphi^h : \Omega \rightarrow \Omega_δ \) is the approximated deformation on a finite element space \( V^h \). The displacement \( u^h : \Omega \rightarrow \mathbb{R}^d \) is defined...
by $\varphi^h(X) = X + u^h(X)$. A generic point of the reference configuration $\Omega$ is denoted by $X$ while the corresponding point of the deformed configuration is denoted by $x = \varphi^h(X)$. $\Gamma^S$ denotes a slave boundary of $\Omega$ and $\Gamma^M$ a master one. The corresponding boundaries on the deformed configuration are $\Gamma_{def}^S$ and $\Gamma_{def}^M$, respectively. The outward unit normal vector to the boundary (in the deformed configuration) at a point $x = \varphi^h(X)$ of that boundary is denoted by $n_x$. Finally, the notation $\delta A[B]$ denotes the directional derivative of the quantity $A$ with respect to the deformation and in the direction $B$. Similarly, the notation $\delta^2 A[B, C]$ is the second derivative in the directions $B$ and $C$.

Let $J(\varphi^h)$ be the potential energy of the system, without taking into account contact and friction contributions. Typically, it includes elastic and external load potential energy. Let $X_i$ for $i \in I_{\text{nodes}}$ be the set of finite element nodes on the slave boundary in the reference configuration. Let $X_i$ for $i \in I_{\text{def}}$ be the contact nodes in potential contact with the master surface of a deformable body. Let $X_i$ for $i \in I_{\text{rig}}$ be the contact nodes in potential contact with a rigid obstacle.

We denote by $x_i = \varphi^h(X_i)$ the corresponding node on the deformed configuration and $y_i$ the projection on the master surface (or rigid obstacle) on the deformed configuration. Let $Y_i$ the point on the master surface verifying $y_i = \varphi^h(Y_i)$. This allows to define the normal gap as

$$g_i = n_y.(\varphi^h(X_i) - \varphi^h(Y_i)) = \|\varphi^h(X_i) - \varphi^h(Y_i)\| \text{Sign}(n_y.(\varphi^h(X_i) - \varphi^h(Y_i))),$$

where $n_y$ is the outward unit normal vector of the master surface at $y$.

Considering only stationary rigid obstacles and applying the principle of Alart-Curnier augmented Lagrangian [AL-CU1991], the problem with nodal contact with friction condition can be expressed as follows in an unsymmetric version (see [renard2013] for the linear elasticity case).

\[
\left\{ \begin{array}{l}
\text{Find } \varphi^h \in V^h \text{ such that } \\
\delta J(\varphi^h)[\delta u^h] - \sum_{i \in I_{\text{def}}} \lambda_i \cdot (\delta u^h(X_i) - \delta u^h(Y_i)) - \sum_{i \in I_{\text{rig}}} \lambda_i \delta u^h(X_i) = 0 \quad \forall \delta u^h \in V^h, \\
\frac{1}{r} \left[ \lambda_i + P_{n_y, \varphi} (\lambda_i + r \left( g_i n_y - \alpha (\varphi^h(X_i) - \varphi^h(Y_i)) - W_T(X_i) + W_T(Y_i) \right) \right] = 0 \quad \forall i \in I_{\text{def}}, \\
\frac{1}{r} \left[ \lambda_i + P_{n_y, \varphi} (\lambda_i + r \left( g_i n_y - \alpha (\varphi^h(X_i) - W_T(X_i)) \right) \right] = 0 \quad \forall i \in I_{\text{rig}},
\end{array} \right.
\]

where $W_T$, $\alpha$, $P_{n_y, \varphi}$ ... + tangent system

Sorry, for the moment the brick is not working.

### 23.25.3 Tools of the high-level generic assembly for contact with friction

The following nonlinear operators are defined in the weak form language (see Compute arbitrary terms - high-level generic assembly procedures - Weak-form language):

- **Transformed_unit_vector(Grad_u, n)** where Grad_u is the gradient of a displacement field and $n$ a unit vector in the reference configuration. This nonlinear operator corresponds to

$$n_{\text{trans}} = \frac{(I + \nabla u)^{-T} n}{\|(I + \nabla u)^{-T} n\|}$$

with the following partial derivatives

$$\partial_u n_{\text{trans}}[\delta u] = -(I - n_{\text{trans}} \otimes n_{\text{trans}})(I + \nabla u)^{-T}(\nabla \delta u)^T n_{\text{trans}}$$

$$\partial_n n_{\text{trans}}[\delta n] = \frac{(I + \nabla u)^{-T} \delta n - n_{\text{trans}} (n_{\text{trans}} \cdot \delta n)}{\|(I + \nabla u)^{-T} n\|}$$
• `Coulomb_friction_coupled_projection(lambda, n, Vs, g, f, r)` where 
  `lambda` is the contact force, `n` is a unit normal vector, `Vs` is the sliding velocity, `g` is the gap, `f` the friction coefficient and `r` a positive augmentation parameter. The expression of the operator is

\[
P(\lambda, n, V_s, g, f, r) = -(\lambda \cdot n + r g) n + P_B(n, r)(\lambda - r V_s)
\]

with \(\tau = \min(f_3 + f_1(\lambda \cdot n + r g), f_2)\)

where \((\cdot)_-\) is the negative part \((x)_- = (-x)_+\) and \(f_1, f_2, f_3\) are the three components of the friction coefficient. Note that the components \(f_2, f_3\) are optional. If a scalar fiction coefficient is given (only \(f_1\)) then this corresponds to the classical Coulomb friction law. If a vector of two components is given (only \(f_1, f_2\)) then this corresponds to a Coulomb friction with a given threshold. Finally, if a vector of three components is given, the friction law corresponds to the expression of \(\tau\) given above.

The expression \(P_B(n, r)(q)\) refers to the orthogonal projection (this is link to the return mapping algorithm) on the tangential ball (with respect to \(n\) of radius \(\tau\)).

The derivatives can be expressed as follows with \(T_n = (I - n \otimes n)\) and \(q_r = T_n q\):

\[
\partial_q P_B(n, \tau)(q) = \begin{cases} 
  0 & \text{for } \tau \leq 0 \\
  T_n & \text{for } \|q_r\| \leq \tau \\
  \|q_r\|^{-1} \left( T_n - \frac{q_r}{\|q_r\|} \otimes \frac{q_r}{\|q_r\|} \right) & \text{otherwise}
\end{cases}
\]

\[
\partial_r P_B(n, \tau)(q) = \begin{cases} 
  0 & \text{for } \tau \leq 0 \\
  \frac{q_r}{\|q_r\|} & \text{for } \|q_r\| \leq \tau \\
  \|q_r\|^{-1} \left( T_n - \frac{q_r}{\|q_r\|} \otimes \frac{q_r}{\|q_r\|} \right) + n \otimes q_r & \text{otherwise}
\end{cases}
\]

\[
\partial_\lambda P(\lambda, n, V_s, g, f, r) = \partial_q P_B(n, \tau) + \partial_r P_B(n, \tau) \otimes \partial_n \tau + H(\lambda \cdot n - r g) n \otimes n,
\]

\[
\partial_n P(\lambda, n, V_s, g, f, r) = \begin{bmatrix} \partial_n P_B(n, \tau) + \partial_r P_B(n, \tau) \otimes \partial_n \tau \\
  + H(\lambda \cdot n - r g) (n \otimes n - (2 \lambda \cdot n + r g) n \otimes n + (\lambda \cdot n + r g) I) \\
  \partial_g P(\lambda, n, V_s, g, f, r) = \partial_r P_B(n, \tau) \partial_g \tau + H(\lambda \cdot n - r g) r n \\
  \partial_f P(\lambda, n, V_s, g, f, r) = \partial_r P_B(n, \tau) \partial_f \tau \\
\end{bmatrix}
\]

\[
\partial_r P(\lambda, n, V_s, g, f, r) = H(\lambda \cdot n - r g) gn + \partial_q P_B(n, \tau)V_s + \partial_r P_B(n, \tau) \partial_r \tau
\]

### 23.25.4 Integral contact brick with raytrace

Add of the brick:

```cpp
indbrick = add_integral_large_sliding_contact_brick_raytracing
  (model &md, const std::string &dataname_r,
   scalar_type release_distance,
   const std::string &dataname_friction_coeff = "0",
   const std::string &dataname_alpha = "1");
```

This brick allows to deal with a multi-contact situation. It adds to the model a raytracing interpolate transformation as described in a previous section whose name can be obtained by the command:
`const std::string &transformation_name_of_large_sliding_contact_brick(model &md, size_type indbrick);`

Once the brick is added to the model, the master and slave contact boundaries have to be added with the following function:

```cpp
add_contact_boundary_to_large_sliding_contact_brick(model &md, size_type indbrick, const mesh_im &mim, size_type region, bool is_master, bool is_slave, const std::string &u, const std::string &lambda = "", const std::string &w = "", bool frame_indifferent = false)
```

where `region` should be a valid mesh region number representing a boundary, `is_master` should be set to `true` if the contact detection is to be done on that contact boundary, `is_slave` should be set to `true` if the integration of contact terms is to be done on that boundary. Note that a contact boundary is allowed to be both master and slave, in particular to allow self-contact detection. `u` is the displacement variable. If `is_slave` is set to `true`, `lambda` should describe a multiplier variable with degrees of freedom on the contact boundary (typically added to the model with the `md.add_filtered_fem_variable(...)` method). Pure master contact boundary do not need the definition of a multiplier. Additionally, `"w` is for the evolutionary case and represents the displacement at the previous time step.

A rigid obstacle can be added to the brick with:

```cpp
add_rigid_obstacle_to_large_sliding_contact_brick(model &md, size_type indbrick, std::string expr, size_type N)
```

where `expr` is an expression using the weak form language (with `X` is the current position) which should be a signed distance to the obstacle. `N` is the mesh dimension.
Numerical continuation and bifurcation

Let an algebraic problem coming from discretisation of a FEM-model can be written in the form

\[ F(U) = 0. \]

In what follows, we shall suppose that the model depends on an additional scalar parameter \( \lambda \) so that \( F(U) = F(U, \lambda) \).

### 24.1 Numerical continuation

Methods of numerical continuation serve for tracing solutions of the system

\[ F(U, \lambda) = 0, \quad F : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N. \]

In GetFEM++, a continuation technique for piecewise \( C^1 \) (\( PC^1 \)) solution curves is implemented (see [Li-Re2014] for more details). Since it does not make an explicit difference between the state variable \( U \) and the parameter \( \lambda \), we shall denote \( Y := (U, \lambda) \) for brevity. Nevertheless, to avoid bad scaling when calculating tangents, for example, we shall use the following weighted scalar product and norm:

\[
\langle Y, \tilde{Y} \rangle_w := \kappa \langle U, \tilde{U} \rangle + \lambda \tilde{\lambda}, \quad \| Y \|_w := \sqrt{\kappa \| U \|^2 + \lambda^2}, \quad Y = (U, \lambda), \quad \tilde{Y} = (\tilde{U}, \tilde{\lambda}).
\]

Here, \( \kappa \) should be chosen so that \( \kappa \langle U, \tilde{U} \rangle \) is proportional to the scalar product of the corresponding space variables, usually in \( L^2 \). One can take, for example, \( \kappa = h^d \), where \( h \) is the mesh size and \( d \) stands for the dimension of the underlying problem. Alternatively, \( \kappa \) can be chosen as \( 1/N \) for simplicity.

The idea of the continuation strategy is to continue smooth pieces of solution curves by a classical predictor-corrector method and to join the smooth pieces continuously.

The particular predictor-corrector method employed is a slight modification of the \textit{inexact Moore-Penrose} continuation implemented in MATCONT [Dh-Go-Ku2003]. It computes a sequence of consecutive points \( Y_j \) lying approximately on a solution curve and a sequence of the corresponding unit tangent vectors \( T_j \):

\[
\| F(Y_j) \| \leq \varepsilon, \quad F'(Y_j; T_j) = 0, \quad \| T_j \|_w = 1, \quad j = 0, 1, \ldots.
\]
To describe it, let us suppose that we have a couple \((Y_j, T_j)\) satisfying the relations above at our disposal. In the prediction, an initial approximation of \((Y_{j+1}, T_{j+1})\) is taken as

\[ Y^0_{j+1} := Y_j + h_j T_j, \quad T^0_{j+1} := T_j, \]

where \(h_j\) is a step size. Its choice will be discussed later on.

In the correction, one computes a sequence \(\{(Y^l_j, T^l_j)\}\), where \(T^l_{j+1} := \tilde{T}^l_{j+1}/\|\tilde{T}^l_{j+1}\|_w\) and the couple \((Y^l_{j+1}, T^l_{j+1})\) is given by one iteration of the Newton method applied to the equation \(F^l(Y, T) = 0\) with

\[ F^l(Y, T) := \begin{pmatrix} \frac{F(Y)}{(T^l_{j+1})^\top (Y - Y^l_{j+1})} \\ \frac{\nabla F(Y^l_{j+1})^\top T}{\langle T^l_{j+1}, T\rangle_w - \langle T^l_{j+1} - T^{l-1}_{j+1}, T^{l-1}_{j+1}\rangle_w} \end{pmatrix} \]

and the initial approximation \((Y^{l-1}_{j+1}, T^{l-1}_{j+1})\). Due to the potential non-differentiability of \(F\), a piecewise-smooth variant of the Newton method is used (Algorithm 7.2.14 in [Fa-Pa2003]).

A couple \((Y^l_{j+1}, T^l_{j+1})\) is accepted for \((Y_{j+1}, T_{j+1})\) if \(\|F(Y^l_{j+1})\| \leq \varepsilon\), \(\|Y^l_{j+1} - Y^{l-1}_{j+1}\|_w \leq \varepsilon\)', and the cosine of the angle between \(T^l_{j+1}\) and \(T_j\) is greater or equal to \(c_{\text{min}}\). Let us note that the partial gradient of \(F\) (or of one of its selection functions in the case of the non-differentiability) with respect to \(U\) is assembled analytically whereas the partial gradient with respect to \(\lambda\) is evaluated by forward finite differences with an increment equal to \(1e-8\).

The step size \(h_{j+1}\) in the next prediction depends on how the Newton correction has been successful. Denoting the number of iterations needed by \(l_{it}\), it is selected as

\[ h_{j+1} := \begin{cases} \max\{h_{\text{dec}} h_j, h_{\text{min}}\} & \text{if no new couple has been accepted}, \\ \min\{h_{\text{inc}} h_j, h_{\max}\} & \text{if a new couple has been accepted and } l_{it} < l_{\text{thr}}, \\ h_j & \text{otherwise}, \end{cases} \]

where \(0 < h_{\text{dec}} < 1 < h_{\text{inc}}, 0 < l_{\text{thr}}\) and \(0 < h_{\text{min}} < h_{\text{max}}\) are given constants. At the beginning, one sets \(h_1 := h_{\text{init}}\) for some \(h_{\text{min}} \leq h_{\text{init}} \leq h_{\text{max}}\).

Now, let us suppose that we have approximated a piece of a solution curve corresponding to one sub-domain of smooth behaviour of \(F\) and we want to recover a piece corresponding to another sub-domain of smooth behaviour. Let \((Y_j, T_j)\) be the last computed couple.

To approximate the tangent to the other smooth piece, we first take a point \(Y_j + h T_j\) with \(h\) a bit greater than \(h_{\text{min}}\) so that this point belongs to the interior of the other sub-domain of smooth behaviour. Then
we find $\tilde{T}$ such that 
\[
\nabla F(Y_j + hT_j)\tilde{T} = 0, \quad \|\tilde{T}\|_w = 1,
\]
and it remains to determine an appropriate direction of this vector. This can be done on the basis of the following observations: First, there exists $r \in \{\pm 1\}$ such that $Y_j - r\tilde{h}\tilde{T}$ remains in the same sub-domain as $Y_j$ for any $\tilde{h}$ positive. This is characterised by the fact that $\frac{|T\top\tilde{T}|}{\|T\|\|\tilde{T}\|}$ is significantly smaller than 1 for $T_-$ with $\nabla F(Y_j - r\tilde{h}\tilde{T})T_- = 0$. Second, $Y_j + r\tilde{h}\tilde{T}$ appears in the other sub-domain for $\tilde{h}$ larger than some positive threshold, and, for such values, $\frac{|T\top\tilde{T}|}{\|T\|\|\tilde{T}\|}$ is close to 1 for $T_+$ with $\nabla F(Y_j + r\tilde{h}\tilde{T})T_+ = 0$.

This suggests the following procedure for selecting the desired direction of $\tilde{T}$: Increase the values of $\tilde{h}$ successively from $\tilde{h}_{\text{min}}$, and when you arrive at $\tilde{h}$ and $r \in \{\pm 1\}$ such that 
\[
\frac{|T\top\tilde{T}|}{\|T\|\|\tilde{T}\|} \approx 1 \quad \text{if} \quad \nabla F(Y_j + r\tilde{h}\tilde{T})T = 0,
\]
take $r\tilde{T}$ as the approximation of the tangent to the other smooth piece.

Having this approximation at our disposal, we restart the predictor-corrector with $(Y_j, r\tilde{T})$.

In GetFEM++, the continuation is implemented for two ways of parameterization of the model:

1. The parameter $\lambda$ is directly a scalar datum, which the model depends on.

2. The model is parametrised by the scalar parameter $\lambda$ via a vector datum $P$, which the model depends on. In this case, one takes the linear path 
\[
\lambda \mapsto P(\lambda) := (1 - \lambda)P^0 + \lambda P^1,
\]
where $P^0$ and $P^1$ are given values of $P$, and one traces the solution set of the problem 
\[
F(U, P(\lambda)) = 0.
\]

### 24.2 Detection of limit points

When tracing solutions of the system $F(U, \lambda) = 0$, one may be interested in limit points (also called fold or turning points), where the number of solutions with the same value of $\lambda$ changes. These points
can be detected by a sign change of a test function $\tau_{LP}$:

$$\tau_{LP}(T_j)\tau_{LP}(T_{j+1}) < 0,$$

where $\tau_{LP}$ is defined by

$$\tau_{LP}(T) := T_\lambda, \quad T = (T_U, T_\lambda) \in \mathbb{R}^N \times \mathbb{R}.$$

![Fig. 3: Limit point.](image)

### 24.3 Numerical bifurcation

A point $\tilde{Y}$ is called a bifurcation point of the system $F(Y) = 0$ if $F(\tilde{Y}) = 0$ and two or more distinct solution curves pass through it. The following result gives a test for smooth bifurcation points (see, e.g., [Georg2001]):

Let $s \mapsto Y(s)$ be a parameterization of a solution curve and $\tilde{Y} := Y(\tilde{s})$ be a bifurcation point. Moreover, let $T^\top \dot{Y}(\tilde{s}) > 0$, $B \notin \text{Im}(J(\tilde{Y}))$, $C \notin \text{Im}(J(\tilde{Y})^\top)$, $d \in \mathbb{R}$ and

$$J(Y) := \begin{pmatrix} \nabla F(Y) \\ T^\top \end{pmatrix}.$$

Define $\tau_{BP}(Y)$ via

$$\begin{pmatrix} J(Y) \\ C^\top \end{pmatrix} \begin{pmatrix} B \\ d \end{pmatrix} \begin{pmatrix} V(Y) \\ \tau_{BP}(Y) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Then $\tau_{BP}(Y(s))$ changes its sign at $s = \tilde{s}$.

Obviously, if one takes $B$, $C$ and $d$ randomly, it is highly possible that they satisfy the requirements above. Consequently, the numerical continuation method is able to detect bifurcation points by taking the vectors $Y$ and $T$ supplied by the correction at each continuation step and monitoring the signs of $\tau_{BP}$. 

164 Chapter 24. Numerical continuation and bifurcation
Once a bifurcation point $\bar{Y}$ is detected by a sign change $\tau_{BP}(Y_j)\tau_{BP}(Y_{j+1}) < 0$, it can be approximated more precisely by the predictor-corrector steps described above with a special step-length adaptation (see Section 8.1 in [Al-Ge1997]). Namely, one can take the subsequent step lengths as

$$h_{j+1} := -\frac{\tau_{BP}(Y_{j+1})}{\tau_{BP}(Y_{j+1}) - \tau_{BP}(Y_j)} h_j$$

until $|h_{j+1}| < h_{\text{min}}$, which corresponds to the secant method for finding a zero of the function $s \mapsto \tau_{BP}(Y(s))$.

Finally, it would be desirable to switch solution branches. To this end, we shall consider the case of the so-called \textit{simple bifurcation point}, where only two distinct solution curves intersect.

Let $\tilde{Y}$ be an approximation of $\bar{Y}$ that we are given and $V(\tilde{Y})$ be the first part of the solution of the augmented system for computing the test function $\tau_{BP}(\tilde{Y})$. As proposed in [Georg2001], one can take $V(\tilde{Y})$ as a predictor direction and do one continuation step starting with $(\tilde{Y}, V(\tilde{Y}))$ to obtain a point on a new branch. After this continuation step has been performed successfully and a point on the new branch has been recovered, one can proceed with usual predictor-corrector steps to trace this branch.

Recently, tools for numerical $PC^1$-bifurcation have been developed in GetFEM++. Let $J$ be a matrix function of a real parameter now defined by

$$J(\alpha) := (1 - \alpha) \left( \nabla F(Y_j) \right) + \alpha \left( \nabla F(Y_{j+1}) \right).$$

As proposed in [Li-Re2014hal], the following test can be used for detection of a $PC^1$ bifurcation point between $Y_j$ and $Y_{j+1}$:

$$\det J(0) \det J(1) < 0.$$ 

To perform this test numerically, introduce

$$M(\alpha) := \begin{pmatrix} J(\alpha) & B \\ C^\top & d \end{pmatrix}$$

and $\tau_{BP}(\alpha)$ analogously as above via

$$M(\alpha) \begin{pmatrix} V(\alpha) \\ \tau_{BP}(\alpha) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$ 

It follows from Cramer’s rule that

$$\tau_{BP}(\alpha) = \frac{\det J(\alpha)}{\det M(\alpha)}$$

24.3. Numerical bifurcation
provided that \( \det M(\alpha) \) is non-zero. Hence if \( B, C \) and \( d \) are chosen so that \( \det M(\alpha) \) is non-zero whenever \( \det J(\alpha) \) is zero, then the sign changes of \( \det J(\alpha) \) are characterised by passings of \( \tau_{BP}(\alpha) \) through 0 whereas the sign changes of \( \det M(\alpha) \) by sign changes of \( \tau_{BP}(\alpha) \) caused by singularities. To conclude, the sign of \( \det J(0) \det J(1) \) is determined by following the behaviour of \( \tau_{BP}(\alpha) \) and monitoring the sign changes of \( \det J(\alpha) \) when \( \alpha \) passes through \( [0, 1] \).

As justified in [Li-Re2014hal], \( B, C \) and \( d \) can be chosen randomly again. The increments \( \delta \) of the current values of \( \alpha \) are changed adaptively so that singularities of \( \tau_{BP} \) are treated effectively. After each calculation of \( \tau_{BP}(\alpha) \), \( \delta \) is set as follows:

\[
\delta := \begin{cases} 
\min\{2\delta, \delta_{\text{max}}\} & \text{if } |\tau_{BP}(\alpha) - \tau_{BP}(\alpha - \delta)| < 0.5\tau_{\text{fac}}\tau_{\text{ref}}, \\
\max\{0.1\delta, \delta_{\text{min}}\} & \text{if } |\tau_{BP}(\alpha) - \tau_{BP}(\alpha - \delta)| > \tau_{\text{fac}}\tau_{\text{ref}}, \\
\delta & \text{otherwise,}
\end{cases}
\]

where \( \delta_{\text{max}} > \delta_{\text{min}} > 0 \) and \( \tau_{\text{fac}} > 0 \) are given constants and \( \tau_{\text{ref}} := \max\{|\tau_{BP}(1) - \tau_{BP}(0)|, 10^{-8}\} \).

When a \( PC^1 \) bifurcation point is detected between \( Y_j \) and \( Y_{j+1} \), it is approximated more precisely by a bisection-like procedure. The obtained approximation lies on the same smooth branch as \( Y_j \), and the corresponding unit tangent that points out from the corresponding region of smoothness is calculated too.

Contrary to the smooth case, it is not clear how many branches can emanate from the \( PC^1 \) bifurcation point and in which directions they could be sought. For this reason, continuation steps for a whole sequence of predictor directions are tried out for finding points on new branches.

Denoting \( \tilde{Y} \), \( \tilde{T} \) the approximation of the bifurcation point and the corresponding tangent, respectively, the predictor directions are taken as follows: For a couple of reference vectors \( \tilde{V}_1 \) and \( \tilde{V}_2 \), one takes \( \pm V \) with \( V \) satisfying

\[
\nabla F(\tilde{Y} + h_{\text{min}}\tilde{V})V = 0, \quad \|V\|_w = 1,
\]

where \( \tilde{V} \) passes through a set of linear combinations of \( \tilde{V}_1 \) and \( \tilde{V}_2 \). The total number of the linear combinations is given by \( n_{\text{dir}} \), and the reference vectors are chosen successively according to the following strategy:

1. One takes \( \tilde{V}_1 := -\tilde{T} \) and \( \tilde{V}_2 \) such that

\[
\nabla F(\tilde{Y} + h_{\text{min}}\tilde{T})\tilde{V}_2 = 0, \quad \|\tilde{V}_2\|_w = 1.
\]

2. Let \( \{\tilde{T}_1, \ldots, \tilde{T}_{n_{\text{dir}}}\} \) denote the set of unit tangents that correspond to the points from the branches found so far and that are oriented in the directions of branching from the bifurcation point. Then \( \tilde{V}_1 \) and \( \tilde{V}_2 \) are taken successively as different combinations from \( \{\tilde{T}_1, \ldots, \tilde{T}_{n_{\text{dir}}}\} \).

3. If all combinations that are available so far have already been used, let \( \tilde{V}_1 \) be unchanged and take \( \tilde{V}_2 := \tilde{V}_2^+ \) with \( \tilde{V}_2^+ \) satisfying

\[
\nabla F\left(\tilde{Y} + h_{\text{min}}\left(\tilde{V}_2^- + 0.1\frac{\tilde{V}_3}{\|\tilde{V}_3\|_w}\right)\right)\tilde{V}_2^+ = 0, \quad \|\tilde{V}_2^+\|_w = 1.
\]

Here, \( \tilde{V}_2^- \) equals the vector \( \tilde{V}_2 \) employed previously and \( \tilde{V}_3 \) is chosen randomly.

The total number of selections of \( \tilde{V}_1 \) and \( \tilde{V}_2 \) is given by \( n_{\text{span}} \).

More details on \( PC^1 \) numerical branching can be found in [Li-Re2015hal].
24.4 Approximation of solution curves of a model

The numerical continuation is defined in getfem/getfem_continuation.h. In order to use it, one has to set it up via the corresponding object first:

```cpp
getfem::cont_struct_getfem_model S(model, parameter_name, sfac, ls, h_init, 
  ↔ h_max, h_min, h_inc, h_dec, 
  ↔ maxres_solve, noisy, singularities, 
  ↔ thrvar, ndir, nspan);
```

where `parameter_name` is the name of the model datum representing $\lambda$, `sfac` represents the scale factor $\kappa$, and `ls` is the name of the solver to be used for the linear systems incorporated in the process (e.g., `getfem::default_linear_solver<getfem::model_real_sparse_matrix, getfem::model_real_plain_vector>(model)`). The real numbers $h_{\text{init}}$, $h_{\text{max}}$, $h_{\text{min}}$, $h_{\text{dec}}$ denote $h_{\text{init}}$, $h_{\text{max}}$, $h_{\text{min}}$, $h_{\text{inc}}$, and $h_{\text{dec}}$, the integer `maxit` is the maximum number of iterations allowed in the correction and $\theta_{\text{thr}}$, $\varepsilon$, $\varepsilon'$, $c_{\text{min}}$, and the target residual value for the linear systems to be solved, respectively. The non-negative integer `noisy` determines how detailed information has to be displayed in the course of the continuation process (the larger value the more details), the integer `singularities` determines whether the tools for detection and treatment of singular points have to be used (0 for ignoring them completely, 1 for detecting limit points, and 2 for detecting and treating bifurcation points, as well), and the boolean value of `non-smooth` determines whether only tools for smooth continuation and bifurcation have to be used or even tools for non-smooth ones do. The real numbers `delta_max`, `delta_min` and `thrvar` represent $\delta_{\text{max}}$, $\delta_{\text{min}}$ and $\theta_{\text{fac}}$, and the integers `ndir` and `nspan` stand for $n_{\text{dir}}$ and $n_{\text{span}}$, respectively.

Optionally, parameterization by a vector datum is then declared by:

```cpp
S.set_parametrised_data_names(initdata_name, finaldata_name, currentdata_name);
```

Here, the data names `initdata_name` and `finaldata_name` should represent $P^0$ and $P^1$, respectively. Under `currentdata_name`, the values of $P(\lambda)$ have to be stored, that is, actual values of the datum the model depends on.

Next, the continuation is initialised by:

```cpp
S.init_Moore_Penrose_continuation(U, lambda, T_U, T_lambda, h);
```

where `U` should be a solution for the value of the parameter $\lambda$ equal to `lambda` so that $Y_0 = (U,\lambda)$. During this initialisation, an initial unit tangent $T_0$ corresponding to $Y_0$ is computed in accordance with the sign of the initial value $T_{\text{lambda}}$, and it is returned in $T_U, T_{\text{lambda}}$. Moreover, $h$ is set to the initial step size $h_{\text{init}}$.

Subsequently, one step of the continuation is called by

```cpp
S.Moore_Penrose_continuation(U, lambda, T_U, T_lambda, h, h0);
```

After each call, a new point on a solution curve and the corresponding tangent are returned in the variables `U`, `lambda` and `T_U, T_{\text{lambda}}`. The step size for the next prediction is returned in `h`. The size of the current step is returned in the optional argument `h0`. According to the chosen value of `singularities`, the test functions for limit and bifurcation points are evaluated at the end of each continuation step. Furthermore, if a smooth bifurcation point is detected, the procedure for numerical
bifurcation is performed and an approximation of the branching point as well as tangents to both bifurcating curves are saved in the continuation object S. From there, they can easily be recovered with member functions of S so that one can initialise the continuation to trace either of the curves next time.

Complete examples of use on a smooth problem are shown in the test programs tests/test_continuation.cc, interface/tests/matlab/demo_continuation.m and interface/src/scilab/demos/demo_continuation.sce, whereas interface/src/scilab/demos/demo_continuation_vee.sce and interface/src/scilab/demos/demo_continuation_block.sce employ also non-smooth tools.
25.1 Some recalls on finite strain elasticity

Let $\Omega$ be the reference configuration and $\Omega_t$ the deformed configuration of an elastic media. Then for $X \in \Omega$ we will denote by $\Phi(x) = u(X) + X$ the deformation. the vector field $u$ is the displacement with respect to the initial position.

The Cauchy-Green tensor is defined by

$$C = \nabla \Phi^T \nabla \Phi$$

The deformation tensor (Green-Lagrange)

$$E = \frac{1}{2} (\nabla \Phi^T \nabla \Phi - I) = \frac{1}{2} (\nabla u^T \nabla u + \nabla u^T + \nabla u)$$

(In the case of linear elasticity, $\nabla u^T \nabla u$ is neglected).

One has

$$C = \nabla \Phi^T \nabla \Phi = 2E + I.$$

Both tensors $E$ and $C$ are used to describe finite strain elasticity constitutive laws.

25.1.1 Main invariants and derivatives

The description of finite strain elasticity constitutive laws often requires the principal invariants of the deformation tensors:
\( i_1, i_2, i_3 \) are the invariants of orders 1, 2 and 3:

\[
\begin{align*}
    i_1(E) &= \text{tr } E & i_1(C) &= 2 \text{tr } E + 3 \\
    i_2(E) &= \frac{(\text{tr } E)^2 - \text{tr } E^2}{2} & i_2(C) &= 4i_2(E) + 4i_1(E) + 3 \\
    i_3(E) &= \det E & i_3(C) &= 8i_3(E) + 4i_2(E) + 2i_1(E) + 1
\end{align*}
\]

The derivatives of the invariants with respect to the tensor \( E \) in the direction \( H \) are:

\[
\begin{align*}
    \frac{\partial i_1}{\partial E}(E; H) &= I : H = \text{tr } H \\
    \frac{\partial i_2}{\partial E}(E; H) &= (i_1(E)I - E^T) : H = (\text{tr } E)(\text{tr } H) - E^T : H \\
    \frac{\partial i_3}{\partial E}(E; H) &= i_3(E)(E^{-T}) : H = (i_2(E)I - i_1(E)E + E^2) : H \text{ in 3D.}
\end{align*}
\]

We will write

\[
\begin{align*}
    \frac{\partial i_1}{\partial E}(E) &= I \\
    \frac{\partial i_2}{\partial E}(E) &= i_1(E)I - E^T \\
    \frac{\partial i_3}{\partial E}(E) &= i_3(E)E^{-T}.
\end{align*}
\]

Let us also recall that

\[
\frac{\partial (M^{-1})}{\partial M}(M; H) = -M^{-1}HM^{-1}
\]

The second derivatives of the invariants are fourth order tensors defined by

\[
\begin{align*}
    \frac{\partial^2 i_1}{\partial E^2}(E) &= 0 \\
    \frac{\partial^2 i_2}{\partial E^2}(E)_{ijkl} &= \delta_{ij}\delta_{kl} - \delta_{il}\delta_{jk} \\
    \frac{\partial^2 i_3}{\partial E^2}(E)_{ijkl} &= i_3(E)(E^{-1})_{ij}E^{-1}_{kl} - E^{-1}_{jk}E^{-1}_{li}).
\end{align*}
\]

The notation \( A : B \) denotes the Frobenius product \( A : B = \sum_{ij} A_{ij}B_{ij} \). This product has the following properties:

\[
\begin{align*}
    A : B &= \text{tr } (A^T B) = \text{tr } (AB^T) = \text{tr } (BA^T) = \text{tr } (B^TA), \\
    A : BC &= B^TA : C, \\
    A : BC &= AC^T : B, \\
    \text{tr } (ABC) &= \text{tr } (B^TA^TC^T)
\end{align*}
\]

Note also that

\[
\frac{\partial i_j}{\partial E}(C; H) = 2\frac{\partial i_j}{\partial C}(C; H).
\]

This property enables us to write the constitutive laws as a function of the Cauchy-Green tensor invariants, especially for the case of the generalized Blatz-Ko strain energy.
25.1.2 Potential elastic energy and its derivative

The stress in the reference configuration can be described by the second Piola-Kirchhoff stress tensor
\[ \hat{\sigma} = \nabla \Phi^{-1} \sigma \nabla \Phi^{-T} \det \nabla \Phi \] where \( \sigma \) is the Cauchy stress tensor in the deformed configuration \( \Omega_t \). An hyper-elastic constitutive law is given by
\[ \hat{\sigma} = \frac{\partial}{\partial E} W(E) = 2 \frac{\partial}{\partial C} W(C) \]
where \( W \) is the density of strain energy of the material. The total strain energy is given by
\[ \mathcal{I}(u) = \int_{\Omega} W(E(u))dX \]
and the derivative of the energy in a direction \( v \) can be written
\[ D\mathcal{I}(u; v) = \int_{\Omega} \frac{\partial W}{\partial E}(E(u)) : (I + \nabla u^T) \nabla v dX \]
because in particular
\[ DE(u; v) = \frac{1}{2}(\nabla u^T \nabla v + \nabla v^T \nabla u + \nabla v^T + \nabla v) \]
and \( A : B = A : (B + B^T)/2 \) when \( A \) is symmetric which is the case for \( \hat{\sigma} \).

Another way is to consider the static equilibrium which can be written as follows in the reference configuration:
\[ -\text{div} \left( (I + \nabla u)\hat{\sigma} \right) = f. \]
Integrating by parts, one obtains:
\[ \int_{\Omega} (I + \nabla u)\hat{\sigma} : \nabla v dX = l(v). \]

25.1.3 Tangent matrix

The displacement \( u \) is fixed. In order to obtain the tangent matrix, one substitutes \( u \) with \( u + h \)
\[ \int_{\Omega} (I + \nabla u + \nabla h)\hat{\sigma}(E(u) + E(h) + \frac{1}{2}(\nabla h^T \nabla u + \nabla u^T \nabla h)) : \nabla v dX = l(v) \]
and considers the linear part w.r.t. \( h \), which is
\[ \int_{\Omega} \nabla h \hat{\sigma}(E(u)) : \nabla v dX + \int_{\Omega} \frac{\partial^2 W}{\partial E^2} \left( \nabla h + \nabla h^T + \nabla h^T \nabla u + \nabla u^T \nabla h \right) : (I + \nabla u^T) \nabla v dX \]
which is symmetric w.r.t. \( v \) and \( h \). It can be rewritten as
\[ \int_{\Omega} \nabla h \hat{\sigma}(E(u)) : \nabla v + \mathcal{A}((I + \nabla u^T) \nabla h) : (I + \nabla u^T) \nabla v dX \]
where \( \mathcal{A} \) is the symmetric \( 3 \times 3 \times 3 \times 3 \) tensor given by \( \mathcal{A}_{ijkl} = \left( \left( \frac{\partial^2 W}{\partial E^2} \right)_{ijkl} + \left( \frac{\partial^2 W}{\partial E^2} \right)_{jikl} \right)/2. \)
25.1.4 Some classical constitutive laws

**Linearized: Saint-Venant Kirchhoff law (small deformations)**

\[ W = \frac{\lambda}{2} i_1(E)^2 + \mu i_1(E^2) \]

\[ \hat{\sigma} = \lambda i_1(E) I + 2\mu E \]

\[ A = \lambda i_1(H) I + \mu (H + H^T) \]

**Three parameters Mooney-Rivlin law**

Compressible material.

\[ W = c_1(j_1(C) - 3) + c_2(j_2(C) - 3) + d_1(i_3(C)^{1/2} - 1)^2 \]

where \(c_1, c_2\) and \(d_1\) are given coefficients and

\[ j_1(C) = i_1(C)i_3(C)^{-1/3} \]
\[ j_2(C) = i_2(C)i_3(C)^{-2/3} \]
\[ \frac{\partial j_1}{\partial C}(C) = i_3(C)^{-1/3} \left( \frac{\partial i_1}{\partial C}(C) - \frac{i_1(C)}{3i_3(C)} \frac{\partial i_3}{\partial C}(C) \right) \]
\[ \frac{\partial j_2}{\partial C}(C) = i_3(C)^{-2/3} \left( \frac{\partial i_2}{\partial C}(C) - \frac{2i_2(C)}{3i_3(C)} \frac{\partial i_3}{\partial C}(C) \right) \]
\[ \frac{\partial^2 j_1}{\partial C^2}(C) = i_3(C)^{-1/3} \left( \frac{4i_1(C)}{9i_3(C)^2} \frac{\partial i_3}{\partial C}(C) \otimes \frac{\partial i_3}{\partial C}(C) - \frac{1}{3i_3(C)} \left( \frac{\partial i_3}{\partial C}(C) \otimes \frac{\partial i_1}{\partial C}(C) \right) + \frac{i_1(C)}{3i_3(C)} \frac{\partial^2 i_3}{\partial C^2}(C) \right) \]

and then

\[ \hat{\sigma} = 2c_1 \frac{\partial j_1}{\partial C}(C) + 2c_2 \frac{\partial j_2}{\partial C}(C) + 2d_1 \left( 1 - i_3(C)^{-1/2} \right) \frac{\partial i_3}{\partial C}(C) \]
\[ B = 4c_1 \frac{\partial^2 j_1}{\partial C^2}(C) + 4c_2 \frac{\partial^2 j_2}{\partial C^2}(C) + 4d_1 \left( 1 - i_3(C)^{-1/2} \right) \frac{\partial^2 i_3}{\partial C^2}(C) + \frac{1}{2} i_3(C)^{-3/2} \frac{\partial i_3}{\partial C}(C) \otimes \frac{\partial i_3}{\partial C}(C) \]

\[ A_{ijkl} = (B_{ijkl} + B_{jikl})/2 \]

Incompressible material.

\[ d_1 = 0 \]

with the additional constraint:

\[ i_3(C) = 1 \]

The incompressibility constraint \(i_3(C) = 1\) is handled with a Lagrange multiplier \(p\) (the pressure)


\( \sigma = -pI \Rightarrow \dot{\sigma} = -p\nabla \Phi \nabla \Phi^T \) \( \det \nabla \Phi \)

\( 1 - i_3(\nabla \Phi) = 0 \)

\( -\int_{\Omega_0} (\det \nabla \Phi - 1) q dX = 0 \quad \forall q \)

\( B = -\int_{\Omega_0} p(\nabla \Phi)^{-T} \det \nabla \Phi : \nabla v dX \)

\( K = \int_{\Omega_0} (p(\nabla \Phi)^{-T}(\nabla h)^T(\nabla \Phi)^{-T} \det \nabla \Phi : \nabla v dX - p(\nabla \Phi)^{-T}(\det \nabla \Phi(\nabla \Phi)^{-T} : \nabla h) : \nabla v) dX \)

\( = \int_{\Omega_0} p(\nabla h^T \nabla \Phi^{-T}) : (\nabla \Phi^{-1} \nabla v) \det \nabla \Phi dX - \int_{\Omega_0} p(\nabla \Phi^{-T} : \nabla h)(\nabla \Phi^{-T} : \nabla v) \det \nabla \Phi dX \)

**Ciarlet-Geymonat law**

\( W = a i_1(C) + (\mu - a)i_2(C) + \left( \frac{\lambda}{4} - \frac{\mu}{2} + a \right)i_3(C) - \left( \frac{\mu}{2} + \frac{\lambda}{4} \right) \log \det(C) \)

with \( \lambda, \mu \) the Lamé coefficients and \( \max(0, \mu - \frac{\lambda}{2}) < a < \frac{\mu}{2} \) (see [ciarlet1988]).

**Generalized Blatz-Ko law**

\( W = (ai_1(C) + bi_3(C))^{1/2} + \frac{B_2(C)}{B_3(C)} + d^n \)

Since \( \frac{\partial W}{\partial C}(C) = \sum_j \frac{\partial W}{\partial i_j(C)} \frac{\partial i_j(C)}{\partial C} \), and \( \frac{\partial^2 W}{\partial C^2}(C) = \sum_j \sum_k \frac{\partial^2 W}{\partial i_j(C) \partial i_k(C)} \frac{\partial i_j(C)}{\partial C} \frac{\partial i_k(C)}{\partial C} + \sum_j \frac{\partial W}{\partial i_j(C)} \frac{\partial^2 i_j(C)}{\partial C^2} \), we must compute the derivatives of the strain energy function with respect to the Cauchy-Green tensor invariants (we don’t need to compute the invariants derivatives with respect to \( E \) since \( \frac{\partial i_j(C)}{\partial C}(C; H) = 2\frac{\partial i_j(C)}{\partial C}(C; H) \)):

\[
\begin{align*}
\frac{\partial W}{\partial i_1(C)} & = naZ^{n-1} & \text{with } Z = (ai_1(C) + bi_3(C))^{1/2} + \frac{B_2(C)}{B_3(C)} + d^n \\
\frac{\partial W}{\partial i_2(C)} & = n\left( \frac{c}{2i_3(C)^{\gamma_2}} - \frac{ci_2(C)}{i_3(C)^{\gamma_2}} \right) Z^{n-1} \\
\frac{\partial W}{\partial i_3(C)} & = n(n - 1)A^2 Z^{n-2} \\
\frac{\partial W}{\partial i_1(C)^2} & = n(n - 1)A^2 \frac{c}{i_3(C)^{\gamma_2}} Z^{n-2} \\
\frac{\partial W}{\partial i_2(C)^2} & = n(n - 1)\left( \frac{b}{2i_3(C)^{\gamma_2}} - \frac{ci_2(C)}{i_3(C)^{\gamma_2}} \right) Z^{n-2} \\
\frac{\partial W}{\partial i_3(C)^2} & = n(n - 1)\left( \frac{b}{2i_3(C)^{\gamma_2}} - \frac{ci_2(C)}{i_3(C)^{\gamma_2}} \right) Z^{n-2} \\
\frac{\partial W}{\partial i_1(C)i_2(C)} & = n(n - 1)\left( \frac{b}{2i_3(C)^{\gamma_2}} - \frac{ci_2(C)}{i_3(C)^{\gamma_2}} \right)^2 Z^{n-2} + n\left( -\frac{b}{4i_3(C)^{\gamma_2}} + \frac{ci_2(C)}{2i_3(C)^{\gamma_2}} \right) Z^{n-1}
\end{align*}
\]

**Plane strain hyper-elasticity**

All previous models are valid in volumic domains. Corresponding plane strain 2D models can be obtained by restricting the stress tensor and the fourth order tensor \( A \) to their plane components.
25.2 Add an nonlinear elasticity brick to a model

This brick represents a large strain elasticity problem. It is defined in the files `getfem/getfem_nonlinear_elasticity.h` and `getfem/getfem_nonlinear_elasticity.cc`. The function adding this brick to a model is

```cpp
ind = getfem::add_nonlinear_elasticity_brick(md, mim, varname, AHL, dataname, region = -1);
```

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

```cpp
getfem::SaintVenant_Kirchhoff_hyperelastic_law AHL;
getfem::Ciarlet_Geymonat_hyperelastic_law AHL;
getfem::Mooney_Rivlin_hyperelastic_law AHL(compressible, neohookean);
getfem::plane_strain_hyperelastic_law AHL(pAHL);
getfem::generalized_Blatz_Ko_hyperelastic_law AHL;
```

The Saint-Venant Kirchhoff law is a linearized law defined with the two Lame coefficients, Ciarlet Geymonat law is defined with the two Lame coefficients and an additional coefficient \((\lambda, \mu, a)\).

The Mooney-Rivlin law accepts two optional flags, the first one determines if the material will be compressible \((d_1 \neq 0)\) and the second one determines if the material is neo Hookean \((c_2 = 0)\). Depending on these flags one to three coefficients may be necessary. By default it is defined as incompressible and non neo Hookean, thus it needs two material coefficients \((c_1, c_2)\). In this case, it is to be used with the large strain incompressibility condition.

The plane strain hyperelastic law takes a pointer on a hyperelastic law as a parameter and performs a 2D plane strain approximation.

`md` is the model variable, `mim` the integration method, `varname` the string being the name of the variable on which the term is added, `dataname` the string being the name of the data in the model representing the coefficients of the law (can be constant or describe on a finite element method) and `region` is the region on which the term is considered (by default, all the mesh).

The program `nonlinear_elastostatic.cc` in `tests` directory and `demo_nonlinear_elasticity.m` in `interface/tests/matlab` directory are some examples of use of this brick with or without an incompressibility condition.

Note that the addition of a new hyperelastic constitutive law consists in furnishing the expression of the strain energy, the stress tensor and the derivative of the stress tensor. See the file `getfem/getfem_nonlinear_elasticity.cc` for more details. In particular, expression of the invariants and their derivatives are available.

A function which computes the Von Mises or Tresca stresses is also available:

```cpp
VM = compute_Von_Mises_or_Tresca(md, varname, AHL, dataname, mf_vm, VM, tresca)
```

It returns a vector of the degrees of freedom of the Von Mises or Tresca stress on the finite element method `mf_vm`. `tresca` is a boolean whose value should be `true` for Tresca stress and `false` for Von Mises stress.
25.3 Add a large strain incompressibility brick to a model

This brick adds an incompressibility condition in a large strain problem of type

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

A Lagrange multiplier representing the pressure is introduced in a mixed formulation. The function adding this brick to a model is

```python
ind = add_nonlinear_incompressibility_brick(md, mim, varname, multname, region = -1)
```

where `md` is the model, `mim` the integration method, `varname` the variable of the model on which the incompressibility condition is added, `multanme` the multiplier variable corresponding to the pressure (be aware that at least a linear Ladyzhenskaja-Babuska-Brezzi inf-sup condition is satisfied between the f.e.m. of the variable and the one of the multiplier). `region` is an optional parameter corresponding to the mesh region on which the term is considered (by default, all the mesh).

25.4 High-level generic assembly versions

The weak form language gives access to the hyperelastic potential and constitutive laws implemented in `GetFEM++`. This allows to directly use them in the language, for instance using a generic assembly brick in a model or for interpolation of certain quantities (the stress for instance).

Here is the list of nonlinear operators in the language which can be useful for nonlinear elasticity:

```plaintext
Det(M) % determinant of the matrix M
Trace(M) % trace of the matrix M
Matrix_i2(M) % second invariant of M (in 3D):
  → (sqr(Trace(m)) - Trace(m*m))/2
Matrix_j1(M) % modified first invariant of M:
  → Trace(m)*pow(Det(m), -1/3).
Matrix_j2(M) % modified second invariant of M:
  → Matrix_I2(m)*pow(Det(m), -2/3).
Right_Cauchy_Green(F) % F' * F
Left_Cauchy_Green(F) % F * F'
Green_Lagrangian(F) % (F'F - Id(meshdim))/2
Cauchy_stress_from_PK2(sigma, Grad_u) % (Id+Grad_u)*sigma*(I+Grad_u')/
  → det(I+Grad_u)
```

The potentials:

```plaintext
Saint_Venant_Kirchhoff_potential(Grad_u, [lambda; mu])
Plane_Strain_Saint_Venant_Kirchhoff_potential(Grad_u, [lambda; mu])
Generalized_Blatz_Ko_potential(Grad_u, [a;b;c;d;n])
Plane_Strain_Generalized_Blatz_Ko_potential(Grad_u, [a;b;c;d;n])
Ciarlet_Geymonat_potential(Grad_u, [lambda;mu;a])
Plane_Strain_Ciarlet_Geymonat_potential(Grad_u, [lambda;mu;a])
Incompressible_Mooney_Rivlin_potential(Grad_u, [c1;c2])
Plane_Strain_Incompressible_Mooney_Rivlin_potential(Grad_u, [c1;c2])
Compressible_Mooney_Rivlin_potential(Grad_u, [c1;c2;d1])
Plane_Strain_Compressible_Mooney_Rivlin_potential(Grad_u, [c1;c2;d1])
Incompressible_Neo_Hookean_potential(Grad_u, [c1])
```

(continues on next page)
Plane_Strain_Incompressible_Neo_Hookean_potential(Grad_u, [c1])
Compressible_Neo_Hookean_potential(Grad_u, [c1;d1])
Plane_Strain_Compressible_Neo_Hookean_potential(Grad_u, [c1;d1])
Compressible_Neo_Hookean_Bonet_potential(Grad_u, [lambda;mu])
Plane_Strain_Compressible_Neo_Hookean_Bonet_potential(Grad_u, [lambda;mu])
Compressible_Neo_Hookean_Ciarlet_potential(Grad_u, [lambda;mu])
Plane_Strain_Compressible_Neo_Hookean_Ciarlet_potential(Grad_u, [lambda;mu])

The second Piola-Kirchhoff stress tensors:

Saint_Venant_Kirchhoff_PK2(Grad_u, [lambda; mu])
Plane_Strain_Saint_Venant_Kirchhoff_PK2(Grad_u, [lambda; mu])
Generalized_Blatz_Ko_PK2(Grad_u, [a;b;c;d;n])
Plane_Strain_Generalized_Blatz_Ko_PK2(Grad_u, [a;b;c;d;n])
Ciarlet_Geymonat_PK2(Grad_u, [lambda;mu;a])
Plane_Strain_Ciarlet_Geymonat_PK2(Grad_u, [lambda;mu;a])
Incompressible_Mooney_Rivlin_PK2(Grad_u, [c1;c2])
Plane_Strain_Incompressible_Mooney_Rivlin_PK2(Grad_u, [c1;c2])
Compressible_Saint_Venant_Kirchhoff_PK2(Grad_u, [c1;c2;d1])
Plane_Strain_Compressible_Saint_Venant_Kirchhoff_PK2(Grad_u, [c1;c2;d1])
Incompressible_Neo_Hookean_PK2(Grad_u, [c1])
Plane_Strain_Incompressible_Neo_Hookean_PK2(Grad_u, [c1])
Compressible_Neo_Hookean_PK2(Grad_u, [c1;d1])
Plane_Strain_Compressible_Neo_Hookean_PK2(Grad_u, [c1;d1])
Compressible_Neo_Hookean_Bonet_PK2(Grad_u, [lambda;mu])
Plane_Strain_Compressible_Neo_Hookean_Bonet_PK2(Grad_u, [lambda;mu])
Compressible_Neo_Hookean_Ciarlet_PK2(Grad_u, [lambda;mu])
Plane_Strain_Compressible_Neo_Hookean_Ciarlet_PK2(Grad_u, [lambda;mu])

Note that the derivatives with respect to the material parameters have not been implemented apart for the Saint Venant Kirchhoff hyperelastic law. Therefore, it is not possible to make the parameter depend on other variables of a model (derivatives are not necessary complicated to implement but for the moment, only a wrapper with old implementations has been written).

Note that the coupling of models is to be done at the weak formulation level. In a general way, it is recommended not to use the potential to define a problem. Main couplings cannot be obtained at the potential level. Thus the use of potential should be restricted to the actual computation of the potential.

An example of use to add a Saint Venant-Kirchhoff hyperelastic term to a variable \( u \) in a model or a ga_workspace is given by the addition of the following assembly string:

```
"((Id(meshdim)+Grad_u)*(Saint_Venant_Kirchhoff_PK2(Grad_u,[lambda;mu]))):Grad_Test_u"
```

Note that in that case, \( \lambda \) and \( \mu \) have to be declared data of the model/ga_workspace. It is of course possible to replace them by explicit constants or expressions depending on several data.

Concerning the incompressible Mooney-Rivlin law, it has to be completed by an incompressibility term. For instance by adding the following incompressibility brick:

```python
ind = add_finite_strain_incompressibility_brick(md, mim, varname, multname, region = -1);
```

This brick just adds the term \( p*(1-Det(Id(meshdim)+Grad_u)) \) if \( p \) is the multiplier and \( u \) the variable which represents the displacement.
The addition of an hyperelastic term to a model can also be done thanks to the following function:

```c
ind = add_finite_strain_elasticity_brick(md, mim, lawname, varname, params,
  region = size_type(-1));
```

where `md` is the model, `mim` the integration method, `varname` the variable of the model representing the large strain displacement, `lawname` is the constitutive law name which could be `Saint_Venant_Kirchhoff`, `Generalized_Blatz_Ko`, `Ciarlet_Geymonat`, `Incompressible_Mooney_Rivlin`, `Compressible_Mooney_Rivlin`, `Incompressible_Neo_Hookean`, `Compressible_Neo_Hookean_Bonet` or `Compressible_Neo_Hookean_Ciarlet`. `params` is a string representing the parameters of the law defined as a small vector or a vector field.

The Von Mises stress can be interpolated with the following function:

```c
void compute_finite_strain_elasticity_Von_Mises(md, varname, lawname, params, mf_vm, VM,
  rg=mesh_region::all_convexes());
```

where `md` is the model, `varname` the variable of the model representing the large strain displacement, `lawname` is the constitutive law name (see previous brick), `params` is a string representing the parameters of the law, `mf_vm` a (preferably discontinuous) Lagrange finite element method on which the interpolation will be done and `VM` a vector of type `model_real_plain_vector` in which the interpolation will be stored.
Small strain plasticity

A framework for the approximation of plasticity models in GetFEM++. See in src/getfem_plasticity.cc and interface/src/gf_model_set.cc for the brick implementation and to extend the implementation to new plasticity models.

26.1 Theoretical background

We present a short introduction to small strain plasticity. We refer mainly to [SI-HU1998] and [SO-PE-OW2008] for a more detailed presentation.

26.1.1 Additive decomposition of the small strain tensor

Let $\Omega \subset \mathbb{R}^3$ be the reference configuration of a deformable body and $u : \Omega \to \mathbb{R}^3$ be the displacement field. Small strain plasticity is based on the additive decomposition of the small strain tensor $\varepsilon(u) = \nabla u + \nabla u^T$ in

$$\varepsilon(u) = \varepsilon^e + \varepsilon^p$$

where $\varepsilon^e$ is the elastic part of the strain tensor and $\varepsilon^p$ the plastic one.

26.1.2 Internal variables, free energy potential and elastic law

We consider

$$\alpha : \Omega \to \mathbb{R}^{d_\alpha},$$

a vector field of $d_\alpha$ strain type internal variables ($d_\alpha = 0$ if no internal variables are considered). We consider also a free energy potential

$$\psi(\varepsilon^e, \alpha),$$
such that corresponding stress type variables are determined by

\[ \sigma = \frac{\partial \psi}{\partial \varepsilon} (\varepsilon^e, \alpha), \quad A = \frac{\partial \psi}{\partial \alpha} (\varepsilon^e, \alpha), \]

where \( \sigma \) is the Cauchy stress tensor and \( A \) the stress type internal variables. The plastic dissipation is given by

\[ \sigma : \dot{\varepsilon}^p - A \dot{\alpha} \geq 0. \]

In the standard cases, \( \psi(\varepsilon^e, \alpha) \) is decomposed into

\[ \psi(\varepsilon^e, \alpha) = \psi^f(\varepsilon^e) + \psi^p(\alpha). \]

In the case of linearized elasticity, one has \( \psi^f(\varepsilon^e) = \frac{1}{2} (A \varepsilon^e) : \varepsilon^e \) where \( A \) is the fourth order elasticity tensor. For isotropic linearized elasticity this expression reduces to \( \psi^f(\varepsilon^e) = \mu \text{dev}(\varepsilon^e) : \text{dev}(\varepsilon^e) + \frac{1}{2} K (\text{tr}(\varepsilon^e))^2 \) where \( \mu \) is the shear modulus and \( K = \lambda + 2\mu/3 \) is the bulk modulus.

### 26.1.3 Plastic potential, yield function and plastic flow rule

Plastic yielding is supposed to occur when the stress attains a critical value. This limit is determined by a yield function \( f(\sigma, A) \) and the condition

\[ f(\sigma, A) \leq 0. \]

The surface \( f(\sigma, A) = 0 \) is the yield surface where the plastic deformation may occur.

Let us also consider the plastic potential \( \Psi(\sigma, A) \), (convex with respect to its both variables) which determines the plastic flow direction in the sense that the flow rule is defined as

\[ \dot{\varepsilon}^p = \gamma \frac{\partial \Psi}{\partial \sigma}(\sigma, A), \quad \dot{\alpha} = -\gamma \frac{\partial \Psi}{\partial A}(\sigma, A), \]

with the additional complementarity condition

\[ f(\sigma, A) \leq 0, \quad \gamma \geq 0, \quad f(\sigma, A) \gamma = 0. \]

The variable \( \gamma \) is called the plastic multiplier. Note that when \( \psi(\varepsilon^e, \alpha), f(\sigma, A) \) or \( \Psi(\sigma, A) \) are not differentiable, subdifferentials have to be used. Associated plasticity corresponds to the choice \( \Psi(\sigma, A) = f(\sigma, A) \).

### 26.1.4 Initial boundary value problem

The weak formulation of a dynamic elastoplastic problem can be written, for an arbitrary kinematically admissible test function \( v \), as follows:

\[ \int_\Omega \rho \ddot{u} \cdot v + \sigma : \nabla \dot{v} dx = \int_\Omega f \dot{v} dx + \int_{\Gamma_N} g \dot{v} dx, \]

\[ u(0, x) = u_0(x), \quad \dot{u}(0) = v_0(x), \]

\[ \varepsilon^p(0, x) = \varepsilon^p_0, \quad \alpha(0, x) = \alpha_0, \]

for \( u_0, v_0, \varepsilon^p_0, \alpha_0 \) being initial values and \( f \) and \( g \) being prescribed forces in the interior of domain \( \Omega \) and on the part of the boundary \( \Gamma_N \).
Note that plasticity models are often applied on quasi-static problems which correspond to the term $\rho \dot{u}$ being neglected.

Given a time step $\Delta t = t_{n+1} - t_n$, from time $t_n$ to $t_{n+1}$, we will denote in the sequel $u_n$, $\varepsilon_n^p$ and $\alpha_n$ the approximations at time $t_n$ of $u(t_n)$, $\varepsilon_n^p(t_n)$ and $\alpha(t_n)$ respectively. These approximations correspond to the chosen time integration scheme (for instance one of the proposed schemes in *The model tools for the integration of transient problems*) which can be different than the time integration scheme used for the integration of the flow rule (see below).

### 26.2 Flow rule integration

The plastic flow rule has to be integrated with its own time integration scheme. Among standards schemes, the backward Euler scheme, the $\theta$-scheme (or generalized trapezoidal rule) and the generalized mid-point scheme are the most commonly used in that context. We make here the choice of the $\theta$-scheme ($\theta = 1$ corresponds to the backward Euler scheme as a special case).

Let $u_{n+1}$ be the displacement at the considered time step and $u_n$ at the previous one.

The $\theta$-scheme for the integration of the plastic flow rules reads as

\[
\varepsilon_{n+1} = (1 - \theta) \Delta t \gamma_n \frac{\partial \Psi}{\partial \sigma}(\sigma_n, A_n) + \theta \Delta t \gamma_{n+1} \frac{\partial \Psi}{\partial \sigma}(\sigma_{n+1}, A_{n+1}),
\]

(26.1)

\[
\alpha_{n+1} = (1 - \theta) \Delta t \gamma_n \frac{\partial \Psi}{\partial A}(\sigma_n, A_n) + \theta \Delta t \gamma_{n+1} \frac{\partial \Psi}{\partial A}(\sigma_{n+1}, A_{n+1}),
\]

(26.2)

with the complementary condition

\[f(\sigma_{n+1}, A_{n+1}) \leq 0, \quad \gamma_{n+1} \geq 0, \quad f(\sigma_{n+1}, A_{n+1}) \gamma_{n+1} = 0.\]

where $0 < \theta \leq 1$ is the parameter of the $\theta$-scheme. We exclude $\theta = 0$ because we will not consider explicit integration of plasticity. Let us recall that $\theta = 1$ corresponds to the backward Euler scheme and $\theta = 1/2$ to the Crank-Nicolson scheme (or trapezoidal rule) which is a second order consistent scheme.

Note that the complementarity condition for the quantities at time step $n$ is prescribed at the previous time step ($\sigma_n$, $\alpha_n$, and $\gamma_n$ are supposed to be already determined).

A solution would be to solve the whole problem with all the unknowns, that is $u_{n+1}$, $\gamma_{n+1}$, $\varepsilon_{n+1}^p$ and $A_{n+1}$. This is of course possible but would be a rather expensive strategy because of the resulting high number of degrees of freedom. A classical strategy (the return mapping one for instance, see [SO-PE-OW2008] or the closest point projection one) consist in integrating locally the plastic flow on each Gauss point of the considered integration method separately, or more precisely to consider on each Gauss point the maps

\[E^p : (u_{n+1}, \zeta_n, \eta_n) \mapsto \varepsilon_{n+1}^p\]

\[\mathcal{A} : (u_{n+1}, \zeta_n, \eta_n) \mapsto \alpha_{n+1}\]

with $\eta_n$, $\zeta_n$ the right hand side of equations (26.1), (26.2), i.e.

\[\zeta_n = \varepsilon_n^p + (1 - \theta) \Delta t \gamma_n \frac{\partial \Psi}{\partial \sigma}(\sigma_n, A_n),\]

\[\eta_n = \alpha_n - (1 - \theta) \Delta t \gamma_{n} \frac{\partial \Psi}{\partial A}(\sigma_n, A_n)\]

This means in particular that $(\varepsilon_{n+1}^p, \alpha_{n+1}) = (E^p(u_{n+1}, \zeta_n, \eta_n), \mathcal{A}(u_{n+1}, \zeta_n, \eta_n))$ is the solution to equations (26.1) and (26.2). Both these maps and their tangent moduli (usually called consistent tangent
moduli) are then used in the global solve of the problem with a Newton method and for \( u_{n+1} \) the unique remaining variable. The advantage of the return mapping strategy is that the unique variable of the global solve is the displacement \( u_{n+1} \). A nonlinear solve on each Gauss point is often necessary which is usually performed with a local Newton method.

In GetFEM++ we propose both the return mapping strategy and also an alternative strategy developed below which is mainly inspired from [PO-NI2016], [SE-PO-WO2015] and [HA-WO2009] and allow more simple tangent moduli. It consists in keeping (a multiple of) \( \gamma_{n+1} \) as an additional unknown with respect to \( u_{n+1} \). As we will see, this will allow a more generic treatment of the yield functions, the price for the simplicity being this additional unknown scalar field.

First, we consider an additional (and optional) given function \( \alpha(\sigma_{n+1}, A_{n+1}) > 0 \) whose interest will appear later on (it will allow simple local inverses) and the new unknown scalar field

\[
\xi_{n+1} = \frac{\gamma_{n+1}}{\alpha(\sigma_{n+1}, A_{n+1})},
\]

so that our two main unknowns are now \( u_{n+1} \) and \( \xi_{n+1} \). The discretized plastic flow rule integration now reads:

\[
\varepsilon_{n+1} - \varepsilon_n = \left( 1 - \theta \right) \alpha(\sigma_n, A_n) \Delta t \xi_n \frac{\partial \Psi}{\partial \sigma}(\sigma_n, A_n) + \theta \alpha(\sigma_{n+1}, A_{n+1}) \Delta t \xi_{n+1} \frac{\partial \Psi}{\partial \sigma}(\sigma_{n+1}, A_{n+1}),
\]

\[
\alpha_{n+1} - \alpha_n = \left( 1 - \theta \right) \alpha(\sigma_n, A_n) \Delta t \xi_n \frac{\partial \Psi}{\partial A}(\sigma_n, A_n) + \theta \alpha(\sigma_{n+1}, A_{n+1}) \Delta t \xi_{n+1} \frac{\partial \Psi}{\partial A}(\sigma_{n+1}, A_{n+1}),
\]

\[
f(\sigma_{n+1}, A_{n+1}) \leq 0, \quad \xi_{n+1} \geq 0, \quad f(\sigma_{n+1}, A_{n+1}) \xi_{n+1} = 0.
\]

For \( u_{n+1} \) and \( \xi_{n+1} \) be given, we define the two maps

\[
\delta^p : (u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n, \eta_n) \mapsto \varepsilon_{n+1}^p
\]

\[
\omega : (u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n, \eta_n) \mapsto \alpha_{n+1}
\]

where the pair \( (\varepsilon_{n+1}^p, \alpha_{n+1}) = (\delta^p(u_{n+1}, \theta \xi_{n+1}, \zeta_n, \eta_n), \omega(u_{n+1}, \theta \xi_{n+1}, \zeta_n, \eta_n)) \) is the solution to equations (26.3), (26.4) (without the consideration of (26.5)). We will see later, that, at least for simple isotropic plastic flow rules, these maps have a simple expression, even sometimes a linear one with respect to \( u_{n+1} \).

Still \( u_{n+1} \) and \( \xi_{n+1} \) be given the stress \( \sigma_{n+1} \) reads

\[
\sigma_{n+1} = \frac{\partial \delta^\varepsilon}{\partial \varepsilon}(\varepsilon(u_{n+1}) - \varepsilon_{n+1}^p),
\]

\[
A_{n+1} = \frac{\partial \omega^p}{\partial \alpha}(\alpha_{n+1}).
\]

The complementarity equation (26.5) is then prescribed with the use of a well chosen complementarity function, as in [HA-WO2009] for \( r > 0 \) such as:

\[
\int_\Omega (\xi_{n+1} - (\xi_{n+1} + r f(\sigma_{n+1}, A_{n+1}))_+) \lambda dx = 0, \forall \lambda
\]

or

\[
\int_\Omega (f(\sigma_{n+1} + (-f(\sigma_{n+1}, A_{n+1}) - \xi_{n+1}/r)_+) + A_{n+1}) \lambda dx = 0, \forall \lambda
\]

NOTE : The notation \( \Delta \xi_{n+1} = \Delta t \xi_{n+1} \) is often used in the literature. The choice here is to preserve the distinction between the two quantities, mainly because of the possible use of adaptative time step : when the time step is changing, the value \( \xi_n \) has to be multiplied by the new time step, so that it is preferable to store \( \xi_n \) instead of \( \Delta \xi_n \) when using the \( \theta \)-scheme.
26.2.1 Plane strain approximation

A plane strain approximation is a 2D problem which corresponds to the deformation of a long cylindrical object where the strain in the length direction (assumed to be along the \( z \) axis) is considered small compared to the ones in the other directions and is neglected. It result in a plane strain tensor of the form

\[
\varepsilon(u) = \begin{pmatrix}
\varepsilon_{1,1} & \varepsilon_{1,2} & 0 \\
\varepsilon_{1,2} & \varepsilon_{2,2} & 0 \\
0 & 0 & 0
\end{pmatrix}.
\]

We denote

\[
\bar{\varepsilon}(u) = \begin{pmatrix}
\varepsilon_{1,1} \\
\varepsilon_{1,2} \\
\varepsilon_{2,2}
\end{pmatrix}
\]

the non neglected components of the strain tensor. In the decomposition of plastic and elastic part of the strain tensor, we assume

\[
\varepsilon_{1,3}^p = \varepsilon_{2,3}^p = \varepsilon_{1,3}^e = \varepsilon_{2,3}^e = 0
\]

and

\[
\varepsilon_{3,3}^e + \varepsilon_{3,3}^p = \varepsilon_{3,3} = 0.
\]

The adaptation to the plane strain approximation to plastic model is most of the time an easy task. An isotropic linearized elastic response reads

\[
\sigma = \lambda \text{tr}(\varepsilon(u)) I + 2\mu (\varepsilon(u) - \varepsilon^p),
\]

and thus

\[
\bar{\sigma} = \lambda \text{tr}(\bar{\varepsilon}(u)) \bar{I} + 2\mu (\bar{\varepsilon}(u) - \bar{\varepsilon}^p),
\]

The nonzero \( \sigma_{3,3} \) component of the stress tensor is given by

\[
\sigma_{3,3} = \lambda \text{tr}(\bar{\varepsilon}(u)) - 2\mu \varepsilon_{3,3}^p
\]

Note that in the common case where isochoric plastic strain is assumed, one has

\[
\text{tr}(\varepsilon^p) = 0 \Rightarrow \varepsilon_{3,3}^p = - (\varepsilon_{1,1}^p + \varepsilon_{2,2}^p).
\]

26.2.2 Plane stress approximation

The plane stress approximation describe generally the 2D membrane deformation of a thin plate. It consist in prescribing the stress tensor to have only in-plane nonzero components, i.e.

\[
\sigma = \begin{pmatrix}
\sigma_{1,1} & \sigma_{1,2} & 0 \\
\sigma_{1,2} & \sigma_{2,2} & 0 \\
0 & 0 & 0
\end{pmatrix}.
\]

We will still denote

\[
\bar{\sigma} = \begin{pmatrix}
\sigma_{1,1} \\
\sigma_{1,2} \\
\sigma_{1,2}
\end{pmatrix}
\]
the in-plane components of the stress tensor. For elastoplasticity, it consists generally to apply the 2D plastic flow rule, prescribing the out-plane components of the stress tensor to be zero with the additional variables $\varepsilon^{e}_{1,3}, \varepsilon^{e}_{2,3}, \varepsilon^{e}_{3,3}$ being unknown (see for instance [SO-PE-OW2008]).

For an isotropic linearized elastic response, one has $\sigma = \lambda \text{tr}(\varepsilon^e) + 2\mu \varepsilon^e$ such that

$$\varepsilon^e = \begin{pmatrix} \varepsilon^{e}_{1,1} & \varepsilon^{e}_{1,2} & 0 \\ \varepsilon^{e}_{1,2} & \varepsilon^{e}_{2,2} & 0 \\ 0 & 0 & \varepsilon^{e}_{3,3} \end{pmatrix}.$$ 

with

$$\varepsilon^{e}_{3,3} = -\frac{\lambda}{\lambda + 2\mu}(\varepsilon^{e}_{1,1} + \varepsilon^{e}_{2,2})$$

so that

$$\bar{\sigma} = \lambda^* \text{tr}(\varepsilon^e) + 2\mu \varepsilon^e \quad \text{with} \quad \lambda^* = \frac{2\mu \lambda}{\lambda + 2\mu} \quad (26.6)$$

Moreover

$$\|\text{Dev}(\sigma)\| = \left(\|\sigma\|^2 - \frac{1}{3}(\text{tr}(\sigma))^2\right)^{1/2}. \quad (26.7)$$

Note that in the case where isochoric plastic strain is assumed, one still has

$$\text{tr}(\varepsilon^p) = 0 \quad \Rightarrow \quad \varepsilon^{p}_{3,3} = -(\varepsilon^{p}_{1,1} + \varepsilon^{p}_{2,2}).$$

### 26.3 Some classical laws

**Tresca** : $\rho(\sigma) \leq \sigma_y$ where $\rho(\sigma)$ spectral radius of the Cauchy stress tensor and $\sigma_y$ the uniaxial yield stress (which may depend on some hardening internal variables).

**Von Mises** : $\|\text{Dev}(\sigma)\| \leq \sqrt{\frac{2}{3}} \sigma_y$ where $\text{Dev}(\sigma) = \sigma - \frac{1}{3} \text{tr}(\sigma) I$ the deviatoric part of $\sigma$ and $\|\sigma\| = \sqrt{\sigma : \sigma}$.

#### 26.3.1 Perfect isotropic associated elastoplasticity with Von-Mises criterion (Prandl-Reuss model)

There is no internal variables and we consider an isotropic elastic response. The flow rule reads

$$\dot{\varepsilon}^p = \gamma \frac{\text{Dev}(\sigma)}{\|\text{Dev}(\sigma)\|}$$

This corresponds to $\Psi(\sigma) = f(\sigma) = \|\text{Dev}(\sigma)\| - \sqrt{\frac{2}{3}} \sigma_y$.

The $\theta$-scheme for the integration of the plastic flow rule reads:

$$\varepsilon^{p}_{n+1} - \varepsilon^{p}_{n} = (1 - \theta) \alpha(\sigma_n) \Delta t \xi_n \frac{\text{Dev}(\sigma_n)}{\|\text{Dev}(\sigma_n)\|} + \theta \alpha(\sigma_{n+1}) \Delta t \xi_{n+1} \frac{\text{Dev}(\sigma_{n+1})}{\|\text{Dev}(\sigma_{n+1})\|}.$$ 

Choosing the factor $\alpha(\sigma_n) = \|\text{Dev}(\sigma_n)\|$ and still with $\xi_n = \frac{\gamma_n}{\alpha(\sigma_n)}$ this gives the equation

$$\varepsilon^{p}_{n+1} - \varepsilon^{p}_{n} = (1 - \theta) \Delta t \xi_n \text{Dev}(\sigma_n) + \theta \Delta t \xi_{n+1} \text{Dev}(\sigma_{n+1}).$$
Since \( \text{Dev}(\sigma_{n+1}) = 2\mu\text{Dev}(\varepsilon(u_{n+1})) - 2\mu\varepsilon_{p,n+1} \) this directly gives:

\[
\delta^p_{n+1}(u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n) = \zeta_n + \left( 1 - \frac{1}{1 + 2\mu \theta \Delta t \xi_{n+1}} \right) (\text{Dev}(\varepsilon(u_{n+1})) - \zeta_n),
\]

which is a linear expression with respect to \( u_{n+1} \) (but not with respect to \( \xi_{n+1} \)).

Moreover, \( \zeta_n \) is defined by

\[
\zeta_n = \varepsilon_n^p + (1 - \theta) \Delta t \xi_n (\text{Dev}(\sigma_n)) = \varepsilon_n^p + (1 - \theta) \Delta t \xi_n 2\mu (\text{Dev}(\varepsilon(u_n)) - \varepsilon_n^p).
\]

**Elimination of the multiplier (for the return mapping approach)**

One has

\[
\|\text{Dev}(\sigma_{n+1})\| = 2\mu\|\text{Dev}(\varepsilon(u_{n+1})) - \varepsilon_{n+1}\| = \frac{2\mu}{1 + 2\mu \theta \Delta t \xi_{n+1}} \|\text{Dev}(\varepsilon(u_{n+1})) - \zeta_n\|,
\]

Thus, denoting \( B = \text{Dev}(\varepsilon(u_{n+1})) - \zeta_n \), either

\[
2\mu \|B\| \leq \sqrt{\frac{2}{3}} \sigma_y,
\]

and \( \xi_{n+1} = 0 \), i.e. we are in the elastic case, or \( \|\text{Dev}(\sigma_{n+1})\| = \sqrt{\frac{2}{3}} \) and one obtains

\[
1 + 2\mu \theta \Delta t \xi_{n+1} = \frac{2\mu \|B\|}{\sqrt{\frac{2}{3}} \sigma_y},
\]

and thus

\[
\varepsilon_{n+1}^p = \zeta_n + \left( 1 - \sqrt{\frac{2}{3}} \frac{\sigma_y}{2\mu \|B\|} \right) B.
\]

The two options can be summarized by

\[
\varepsilon_{n+1}^p = \delta^p_{n+1}(u_{n+1}, \zeta_n) = \zeta_n + \left( 1 - \sqrt{\frac{2}{3}} \frac{\sigma_y}{2\mu \|B\|} \right) B.
\]

The multiplier \( \xi_{n+1} \) (needed for the \( \theta \)-scheme for \( \theta \neq 1 \)) is given by

\[
\xi_{n+1} = \frac{1}{\theta \Delta t} \left( \sqrt{\frac{3}{2}} \frac{\|B\|}{\sigma_y} - \frac{1}{2\mu} \right). \]

**Plane strain approximation**

The plane strain approximation has the same expression replacing the 3D strain tensors by the in-plane ones \( \bar{\varepsilon}^p \) and \( \bar{\varepsilon}(u_{n+1}) \).

\[
\tilde{\delta}^p_{n+1}(u_{n+1}, \theta \Delta t \xi_{n+1}, \bar{\zeta}_n) = \bar{\zeta}_n + \left( 1 - \frac{1}{1 + 2\mu \theta \Delta t \xi_{n+1}} \right) \left( \text{Dev}(\bar{\varepsilon}(u_{n+1})) - \bar{\zeta}_n \right),
\]

where \( \text{Dev}(\bar{\varepsilon}) = \bar{\varepsilon} - \frac{1}{3} \text{tr}(\bar{\varepsilon}) \bar{I} \) is the 2D restriction of the 3D deviator.

Moreover, for the yield condition,

\[
\|\text{Dev}(\sigma)\|^2 = 4\mu^2 \left( \|\text{Dev}(\bar{\varepsilon}) - \bar{\varepsilon}^p\|^2 + \left( \frac{\text{tr}(\bar{\varepsilon}(u))}{3} - \text{tr}(\bar{\varepsilon}^p) \right)^2 \right).
\]
And for the elimination of the multiplier,

\[
\bar{E}_p(\bar{u}_{n+1}, \bar{\varepsilon}_n^p) = \zeta_n + \left(1 - \sqrt{\frac{2}{3}} \frac{\sigma_y}{2\mu\|B\|}\right) \bar{B}
\]

with \(\bar{B} = \text{Dev}(\bar{\varepsilon}(u_{n+1})) - \zeta_n\) and \(||B||^2 = ||\text{Dev}(\bar{\varepsilon}(u_{n+1})) - \zeta_n||^2 + \left(\frac{\text{tr}(\bar{\varepsilon}(u_{n+1}))}{3} - \text{tr}(\zeta_n)\right)^2\).

**Plane stress approximation**

For plane stress approximation, using (26.6) we deduce from the expression of the 3D case

\[
\bar{\varepsilon}_{n+1}^p = \frac{1}{1 + 2\mu\theta\Delta\xi} \left(\zeta_n + 2\mu\theta\Delta\xi \left(\bar{\varepsilon}(u_{n+1}) - \frac{2\mu}{3(\lambda + 2\mu)}(\text{tr}(\bar{\varepsilon}(u_{n+1})) - \text{tr}(\bar{\varepsilon}_{n+1}^p))\right)\right),
\]

since \(\text{Dev}(\varepsilon(u)) = \varepsilon(u) - \frac{2\mu}{3(\lambda + 2\mu)}(\text{tr}(\varepsilon(u)) - \text{tr}(\varepsilon^p))\). Of course, this relation still has to be inverted. Denoting \(\alpha = 1 + 2\mu\theta\Delta\xi\), \(\beta = \frac{4\mu^2\theta\Delta\xi}{3\lambda + 6\mu}\) and \(C = \zeta_n + 2\mu\theta\Delta\xi \left(\bar{\varepsilon}(u_{n+1}) - \frac{2\mu}{3(\lambda + 2\mu)}(\text{tr}(\bar{\varepsilon}(u_{n+1}))\right)\) one obtains

\[
\bar{\varepsilon}_{n+1}^p = \frac{\beta \text{tr}(C)}{\alpha(\alpha - 2\beta)} \bar{I} + \frac{1}{\alpha} C.
\]

Moreover, for the yield condition, expression (26.7) can be used.

### 26.3.2 Isotropic elastoplasticity with linear isotropic and kinematic hardening and Von-Mises criterion

We consider an isotropic elastic response and the internal variable \(\alpha : \Omega \rightarrow \mathbb{R}\) being the accumulated plastic strain which satisfies

\[
\dot{\alpha} = \sqrt{\frac{2}{3}} \gamma
\]

For \(H_i\) the isotropic hardening modulus, the linear hardening consists in

\[
\psi^p(\alpha) = \frac{1}{2} H_i \alpha^2
\]

i.e. \(A = H_i\alpha\) and a uniaxial yield stress defined by

\[
\sigma_y(\alpha) = \sigma_{y0} + A = \sigma_{y0} + H_i\alpha,
\]

for \(\sigma_{y0}\) the initial uniaxial yield stress. The yield function (and plastic potential since this is an associated plastic model) can be defined by

\[
\Psi(\sigma, A) = f(\sigma, A) = ||\text{Dev}(\sigma - \frac{2}{3} H_k \bar{\varepsilon}^p)|| - \sqrt{\frac{2}{3}(\sigma_{y0} + A)},
\]

where \(H_k\) is the kinematic hardening modulus. The same computation as in the previous section leads to

\[
\bar{E}_p(u_{n+1}, \theta\Delta\xi_{n+1}, \zeta_n) = \zeta_n + \frac{1}{2(\mu + H_k/3)} \left(1 - \frac{1}{1 + 2(\mu + H_k/3)\theta\Delta\xi_{n+1}}\right) \left(2\mu\text{Dev}(\bar{\varepsilon}(u_{n+1})) - 2(\mu + H_k/3)\zeta_n\right)
\]
The two cases can be summarized by

\[
\begin{align*}
\mathcal{F}(u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n, \eta_n) &= \eta_n + \sqrt{\frac{2}{3}} \theta \Delta t \xi_{n+1} \| \text{Dev}(\sigma_{n+1} - \frac{2}{3} H_k \varepsilon_p^{n+1}) \| \\
&= \eta_n + \sqrt{\frac{2}{3}} \theta \Delta t \xi_{n+1} \| 2 \mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k/3) \varepsilon_p^{n+1} \| \\
&= \eta_n + \sqrt{\frac{2}{3}} \frac{\theta \Delta t \xi_{n+1}}{3} \| 2 \mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k/3) \zeta_n \| \\
&= \eta_n + \sqrt{\frac{2}{3}} \frac{\theta \Delta t \xi_{n+1}}{2(\mu + H_k/3)} \left(1 - \frac{1}{1 + 2(\mu + H_k/3) \Delta t \xi_{n+1}}\right) \| 2 \mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k/3) \zeta_n \|
\end{align*}
\]

where \( \zeta_n \) and \( \eta_n \) are defined by

\[
\zeta_n = \varepsilon_p^n + (1 - \theta) \Delta t \xi_n \| \text{Dev}(\sigma_n) - \frac{2}{3} H_k \varepsilon_p^{n+1} \| = \varepsilon_p^n + (1 - \theta) \Delta t \xi_n \| 2 \mu \text{Dev}(\varepsilon(u_n)) - 2(\mu + H_k/3) \varepsilon_p^n \|,
\]

\[
\eta_n = \alpha_n + (1 - \theta) \sqrt{\frac{2}{3}} \Delta t \xi_n \| \text{Dev}(\sigma_n) - \frac{2}{3} H_k \varepsilon_p^n \| = \alpha_n + (1 - \theta) \sqrt{\frac{2}{3}} \Delta t \xi_n \| 2 \mu \text{Dev}(\varepsilon(u_n)) - 2(\mu + H_k/3) \varepsilon_p^n \|.
\]

Note that the isotropic hardening modulus do not intervene in \( \mathcal{F}(u_{n+1}, \theta \Delta \xi, \varepsilon_p^n) \) but only in \( f(\sigma, A) \).

**Elimination of the multiplier (for the return mapping approach)**

Denoting \( \delta = \frac{1}{1 + 2(\mu + H_k/3) \Delta t \xi_{n+1}}, \beta = \frac{1 - \delta}{2(\mu + H_k/3)} \) and \( B = 2 \mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k/3) \zeta_n \) the expression for \( \varepsilon_p^{n+1} \) and \( \alpha_{n+1} \) becomes

\[
\varepsilon_p^{n+1} = \zeta_n + \beta B, \quad \alpha_{n+1} = \eta_n + \sqrt{\frac{2}{3}} \beta \| B \|, \quad (26.8)
\]

and the plastic constraint

\[
\delta \| B \| \leq \sqrt{\frac{2}{3}} (\sigma_{y0} + H_i \alpha_{n+1}).
\]

Thus, either we are in the elastic case, i.e. \( \xi_{n+1} = 0, \delta = 1 \) and

\[
\| B \| \leq \sqrt{\frac{2}{3}} (\sigma_{y0} + H_i \eta_n),
\]

or we are in the plastic case and \( \xi_{n+1} > 0, \delta < 1, \| B \| = \sqrt{\frac{2}{3}} (\sigma_{y0} + H_i \alpha_{n+1}) \) and \( (1 - \delta) \) solves the equation

\[
\| B \| - (1 - \delta) \| B \| = \sqrt{\frac{2}{3}} \left(\sigma_{y0} + H_i \eta_n + \sqrt{\frac{2}{3}} 2(\mu + H_k/3) (1 - \delta) \| B \|\right),
\]

which leads to

\[
1 - \delta = \frac{2(\mu + H_k/3)}{\| B \| (2\mu + \frac{2}{3}(H_k + H_i))} \left(\| B \| - \sqrt{\frac{2}{3}} (\sigma_{y0} + H_i \eta_n)\right).
\]

The two cases can be summarized by

\[
\beta = \frac{1}{\| B \| (2\mu + \frac{2}{3}(H_k + H_i))} \left(\| B \| - \sqrt{\frac{2}{3}} (\sigma_{y0} + H_i \eta_n)\right).
\]

26.3. Some classical laws

187
which directly gives \( \varepsilon^p(u_{n+1}, \zeta, \eta_n) \) and \( \mathcal{A}(u_{n+1}, \zeta, \eta_n) \) thanks to (26.8). The multiplier \( \xi_{n+1} \) being given by
\[
\xi_{n+1} = \frac{1}{(2(\mu + H_k / 3))\theta \Delta t} \left( \frac{1}{\delta} - 1 \right) = \frac{1}{\theta \Delta t} \frac{\beta}{1 - 2(\mu + H_k / 3)\beta}.
\]

**Plane strain approximation**

Still denoting \( \delta = \frac{1}{1 + 2(\mu + H_k / 3)\theta \Delta t \xi_{n+1}} \), \( \beta = \frac{1 - \delta}{2(\mu + H_k / 3)} \), \( B = 2\mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k / 3)\zeta_n \) and \( \overline{B} = 2\mu \text{Dev}(\bar{\varepsilon}(u_{n+1})) - 2(\mu + H_k / 3)\zeta_n \) its in-plane part, one has
\[
\bar{\varepsilon}^p(u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n) = \bar{\zeta}_n + \beta \overline{B},
\]
\[
\mathcal{A}(u_{n+1}, \theta \Delta t \xi_{n+1}, \zeta_n, \eta_n) = \eta_n + \sqrt{2 \beta \|B\|},
\]
with
\[
\|B\|^2 = 2\mu \text{Dev}(\varepsilon(u_{n+1})) - 2(\mu + H_k / 3)\zeta_n)^2 + \left( 2\mu \text{tr}(\bar{\varepsilon}(u_{n+1})) - 2(\mu + H_k / 3)\text{tr}(\zeta_n) \right)^2.
\]
The yield condition still reads
\[
\delta \|B\| \leq \sqrt{2 \overline{\sigma}_y(\sigma_{y0} + H_i \alpha_{n+1}).}
\]
and for the elimination of the multiplier, \( \beta \) has the same expression as in the previous section adapting the value of \( \|B\| \). The expressions of \( \bar{\zeta}_n \) and \( \eta_n \) have to be adapted accordingly.

### 26.3.3 Souza-Auricchio elastoplasticity law (for shape memory alloys)

See for instance [GR-ST2015] for the justification of the construction of this flow rule. A Von-Mises stress criterion together with an isotropic elastic response, no internal variables and a special type of kinematic hardening is considered with a constraint \( \|\varepsilon^p\| \leq c_3 \). The plastic potential and yield function have the form
\[
\Psi(\sigma) = f(\sigma) = \left\| \text{Dev} \left( \sigma - c_1 \frac{\varepsilon^p}{\|\varepsilon^p\|} - c_2 \varepsilon^p - \delta \frac{\varepsilon^p}{\|\varepsilon^p\|} \right) \right\| - \sqrt{2 \overline{\sigma}_y},
\]
with the complementarity condition
\[
\delta \geq 0, \|\varepsilon^p\| \leq c_3, \delta(\|\varepsilon^p\| - c_3) = 0,
\]
where \( c_1, c_2 \) and \( c_3 \) are some physical parameters. Note that \( \frac{\varepsilon^p}{\|\varepsilon^p\|} \) has to be understood to be the whole unit ball for \( \varepsilon^p = 0 \).

to be done . . .

### 26.4 Elasto-plasticity bricks

See the test programs tests/plasticity.cc, interface/tests/matlab/demo_plasticity.m, interface/tests/matlab/demo_plasticity.py and in contrib/test_plasticity.
26.4.1 Generic brick

There are two versions of the generic brick. A first one when the plastic multiplier is kept as a variable of the problem where the added term is of the form:

\[ \int_{\Omega} \sigma_{n+1} : \nabla \delta u \, dx + \int_{\Omega} (\xi_{n+1} - (\xi_{n+1} + rf(\sigma_{n+1}, A_{n+1})) \cdot \delta \xi \, dx = 0, \]

with \( r > 0 \) having a specific value chosen by the brick (in terms of the elasticity coefficients), and when the return mapping strategy is selected (plastic multiplier is just a data), just the added term:

\[ \int_{\Omega} \sigma_{n+1} : \nabla v \, dx. \]

The function which adds the brick to a model \( md \) is

```cpp
getfem::add_small_strain_elastoplasticity_brick
(md, mim, lawname, unknowns_type,
 const std::vector<std::string> &varnames,
 const std::vector<std::string> &params, region = size_type(-1));
```

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

The available plasticity laws are:

- “Prandtl Reuss” (or “isotropic perfect plasticity”). Isotropic elasto-plasticity with no hardening. The variables are the displacement, the plastic multiplier and the plastic strain. The displacement should be a variable and have a corresponding data having the same name preceded by “Previous_” corresponding to the displacement at the previous time step (typically “u” and “Previous_u”). The plastic multiplier should also have two versions (typically “xi” and “Previous_xi”) the first one being defined as data if \( unknowns_type = DISPLACEMENT_ONLY \) or as a variable if \( unknowns_type = DISPLACEMENT_AND_PLASTIC_MULTIPLIER \). The plastic strain should represent a n x n data tensor field stored on mesh_fem or (preferably) on an im_data (corresponding to \( mim \)). The data are the first Lame coefficient, the second one (shear modulus) and the uniaxial yield stress. IMPORTANT: Note that this law implements the 3D expressions. If it is used in 2D, the expressions are just transposed to the 2D. For the plane strain approximation, see below.

- “plane strain Prandtl Reuss” (or “plane strain isotropic perfect plasticity”) The same law as the previous one but adapted to the plane strain approximation. Can only be used in 2D.
• “Prandtl Reuss linear hardening” (or “isotropic plasticity linear hardening”). Isotropic elastoplasticity with linear isotropic and kinematic hardening. An additional variable compared to “Prandtl Reuss” law: the accumulated plastic strain. Similarly to the plastic strain, it is only stored at the end of the time step, so a simple data is required (preferably on an im_data). Two additional parameters: the kinematic hardening modulus and the isotropic one. 3D expressions only.

• “plane strain Prandtl Reuss linear hardening” (or “plane strain isotropic plasticity linear hardening”). The same law as the previous one but adapted to the plane strain approximation. Can only be used in 2D.

IMPORTANT: remember that small_strain_elastoplasticity_next_iter has to be called at the end of each time step, before the next one (and before any post-treatment: this sets the value of the plastic strain and plastic multiplier).

Additionally, the following function allow to pass from a time step to another for the small strain plastic brick:

```cpp
getfem::small_strain_elastoplasticity_next_iter
    (md, mim, lawname, unknowns_type,
     const std::vector<std::string> &varnames,
     const std::vector<std::string> &params, region = size_type(-1));
```

The parameters have to be exactly the same as the ones of the add_small_strain_elastoplasticity_brick, so see the documentation of this function for any explanations. Basically, this brick computes the plastic strain and the plastic multiplier and stores them for the next step. Additionally, it copies the computed displacement to the data that stores the displacement of the previous time step (typically “u” to “Previous_u”). It has to be called before any use of compute_small_strain_elastoplasticity_Von_Mises.

The function

```cpp
getfem::compute_small_strain_elastoplasticity_Von_Mises
    (md, mim, lawname, unknowns_type,
     const std::vector<std::string> &varnames,
     const std::vector<std::string> &params,
     const mesh_fem &mf_vm, model_real_plain_vector &VM,
     region = size_type(-1));
```

computes the Von Mises stress field with respect to a small strain elastoplasticity term, approximated on mf_vm, and stores the result into VM. All other parameters have to be exactly the same as for add_small_strain_elastoplasticity_brick. Remember that small_strain_elastoplasticity_next_iter has to be called before any call of this function.

26.4.2 A specific brick based on the low-level generic assembly for perfect plasticity

This is an previous version of a elastoplasticity brick which is restricted to isotropic perfect plasticity and is based on the low-level generic assembly. Its specificity which could be interesting for testing is that the flow rule is integrated on finite element nodes (not on Gauss points).

The function adding this brick to a model is:

```cpp
getfem::add_elastoplasticity_brick
    (md, mim, ACP, varname, previous_varname, datalambda, datamu, „
     datathreshold, datasigma, region);
```
where:

- `varname` represents the main displacement unknown on which the brick is added (u).
- `previous_varname` is the displacement at the previous time step.
- `datalambda` and `datamu` are the data corresponding to the Lame coefficients.
- `datathreshold` represents the plastic threshold of the studied material.
- `datasigma` represents the stress constraint values supported by the material. It should be composed of 2 iterates for the time scheme needed for the Newton algorithm used. Note that the finite element method on which `datasigma` is defined should be able to represent the derivative of `varname`.
- `ACP` corresponds to the type of projection to be used. It has an `abstract_constraints_projection` type and for the moment, only exists the `VM_projection` corresponding to the Von Mises one.

Be careful: `datalambda`, `datamu` and `datathreshold` could be constants or described on the same finite element method.

This function assembles the tangent matrix and the right hand side vector which will be solved using a Newton algorithm.

Additionally, The function:

```cpp
getfem::elastoplasticity_next_iter
    (md, mim, varname, previous_varname, ACP, datalambda, datamu, ¬
    ¬datathreshold, datasigma);
```

computes the new stress constraint values supported by the material after a load or an unload (once a solve has been done earlier) and upload the variables `varname` and `datasigma` as follows:

\[ u^{n+1} \Rightarrow u^n \quad \text{and} \quad \sigma^{n+1} \Rightarrow \sigma^n \]

Then, \( u^n \) and \( \sigma^n \) contains the new values computed and one can restart the process.

The function:

```cpp
getfem::compute_elastoplasticity_Von_Mises_or_Tresca
    (md, datasigma, mf_vm, VM, tresca=false);  
```

computes the Von Mises (or Tresca if `tresca = true`) criterion on the stress tensor stored in `datasigma`. The stress is evaluated on the `mesh_fem` `mf_vm` and stored into the vector `VM`. Of course, this function can be used if and only if the previous function `elastoplasticity_next_iter` has been called earlier.

The function:

```cpp
getfem::compute_plastic_part
    (md, mim, mf_pl, varname, previous_varname, ACP, datalambda, datamu, ¬
    ¬datathreshold, datasigma, Plast);
```

computes on `mf_pl` the plastic part of the material, that could appear after a load and an unload, into the vector `Plast`.

Note that `datasigma` should be the vector containing the new stress constraint values, i.e. after a load or an unload of the material.
ALE Support for object having a large rigid body motion

27.1 ALE terms for rotating objects

This section presents a set of bricks facilitating the use of an ALE formulation for rotating bodies having a rotational symmetry (typically a train wheel).

27.1.1 Theoretical background

This strategy consists in adopting an intermediary description between an Eulerian and a Lagrangian ones for a rotating body having a rotational symmetry. This intermediary description consists in a rotating axes with respect to the reference configuration. See for instance [Dr-La-Ek2014] and [Nackenhorst2004].

It is supposed that the considered body is submitted approximately to a rigid body motion

$$\tau(X) = R(t)X + Z(t)$$

and may have additional deformation (expected smaller) with respect to this rigid motion, where $R(t)$ is a rotation matrix

$$R(t) = \begin{pmatrix} \cos(\theta(t)) & \sin(\theta(t)) & 0 \\ -\sin(\theta(t)) & \cos(\theta(t)) & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

and $Z(t)$ is a translation. Note that, in order to be consistent with a positive translation for a positive angle for a rolling contact, the rotation is \textit{clockwise}. This illustrated in the following figure:

Note that the description is given for a three-dimensional body. For two-dimensional bodies, the third axes is neglected so that $R(t)$ is a $2 \times 2$ rotation matrix. Let us denote $r(t)$ the rotation:

$$r(t, X) = R(t)X, \quad \text{and} \quad A = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. $$
We have then

\[ \dot{r}(t, X) = \dot{\theta}AR(t)X \]

If \( \varphi(t, X) \) is the deformation of the body which maps the reference configuration \( \Omega^0 \) to the deformed configuration \( \Omega_t \) at time \( t \), the ALE description consists in the decomposition of the deformation of the cylinder in

\[ \varphi(t, X) = (\tau(t) \circ \bar{\varphi}(t) \circ r(t))(X) = \bar{\varphi}(t, r(t, X)) + Z(t) \]

With \( \bar{X} = R(t)X \) the new considered deformation is

\[ \bar{\varphi}(t, \bar{X}) = \varphi(X) - Z(t) \]

Thanks to the rotation symmetry of the reference configuration \( \Omega^0 \) we note that \( \bar{\Omega}^0 = r(t, \Omega^0) \) is independent of \( t \) and will serve as the new reference configuration. This is illustrated in the following figure:

The denomination ALE of the method is justified by the fact that \( \bar{\Omega}^0 \) is an intermediate configuration which is of Euler type for the rigid motion and a Lagrangian one for the additional deformation of the solid. If we denote

\[ \bar{u}(t, X) = \bar{\varphi}(t, X) - \bar{X} \]

the displacement with respect to this intermediate configuration, the advantage is that if this additional displacement with respect to the rigid body motion is small, it is possible to use a small deformation model (for instance linearized elasticity).

Due to the objectivity properties of standard constitutive laws, the expression of these laws in the intermediate configuration is most of the time identical to the expression in a standard reference configuration.
except for the expression of the time derivative which are modified because the change of coordinate is nonconstant in time:

\[
\frac{\partial \varphi}{\partial t} = \frac{\partial \varphi}{\partial t} + \dot{\theta} \nabla \varphi A \dot{X} + \dot{Z}(t),
\]

\[
\frac{\partial^2 \varphi}{\partial t^2} = \frac{\partial^2 \varphi}{\partial t^2} + 2\dot{\theta} \nabla \frac{\partial \varphi}{\partial t} A \dot{X} + \ddot{\theta} \nabla^2 (\nabla \varphi A \dot{X}) \otimes (A \dot{X}) + \dot{\theta} \nabla \varphi A \dot{X} + \dot{Z}(t).
\]

Note that the term \( \dot{\theta} A \dot{X} \) is the rigid motion velocity vector. Now, if \( \Theta(t, X) \) is a quantity attached to the material points (for instance the temperature), then, with \( \Theta(t, \tilde{X}) = \Theta(t, X) \), one simply has

\[
\frac{\partial \Theta}{\partial t} = \frac{\partial \tilde{\Theta}}{\partial t} + \dot{\theta} \nabla \varphi A \dot{X}
\]

This should not be forgotten that a correction has to be provided for each evolving variable for which the time derivative intervene in the considered model (think for instance to plastic flow for plasticity). So that certain model bricks cannot be used directly (plastic bricks for instance).

\( \text{GetFEM++} \) bricks for structural mechanics are mainly considering the displacement as the unknown. The expression for the displacement is the following:

\[
\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 \bar{u}}{\partial t^2} + 2\dot{\theta} \nabla \frac{\partial \bar{u}}{\partial t} A \dot{X} + \ddot{\theta} \nabla^2 ((I_d + \nabla \bar{u}) A \dot{X}) \otimes (A \dot{X}) + \dot{\theta} (I_d + \nabla \bar{u}) A \dot{X} + \dot{Z}(t).
\]

### Weak formulation of the transient terms

Assuming \( \rho^0 \) the density in the reference configuration having a rotation symmetry, the term corresponding to acceleration in the weak formulation reads (with \( \nu(X) = \nu(X) \) a test function):

\[
\int_{\Omega^0} \rho^0 \frac{\partial^2 u}{\partial t^2} \cdot \nu dX = \int_{\Omega^0} \rho^0 [\frac{\partial^2 \bar{u}}{\partial t^2} + 2\dot{\theta} \nabla \frac{\partial \bar{u}}{\partial t} A \dot{X} + \ddot{\theta} \nabla^2 ((I_d + \nabla \bar{u}) A \dot{X}) \otimes (A \dot{X}) + \dot{\theta} (I_d + \nabla \bar{u}) A \dot{X} + \dot{Z}(t)] \cdot \nu dX.
\]

The third term in the right hand side can be integrated by part as follows:

\[
\int_{\Omega^0} \rho^0 \ddot{\theta} \nabla^2 ((I_d + \nabla \bar{u}) A \dot{X}) \otimes (A \dot{X}) \cdot \nu dX = - \int_{\Omega^0} \rho^0 [\dot{\theta} (I_d + \nabla \bar{u}) A \dot{X}) \cdot (\nabla (\rho^0 \nu) A \dot{X}) dX + \int_{\partial \Omega^0} \rho^0 \ddot{\theta} ((I_d + \nabla \bar{u}) A \dot{X}) \otimes (A \dot{X}) N \cdot \nu d\Gamma.
\]

Since \( \tilde{N} \) the outward unit normal vector on \( \partial \Omega^0 \) is orthogonal to \( A \dot{X} \) the boundary term is zero and \( \nabla (\rho^0 \nu) = \tilde{v} \nabla \rho^0 + \rho^0 \nabla \tilde{v} \) and since \( \nabla \rho^0 \cdot (A \dot{X}) = 0 \) because of the assumption on \( \rho^0 \) to have a rotation symmetry, we have

\[
\int_{\Omega^0} \rho^0 \ddot{\theta} \nabla^2 ((I_d + \nabla \bar{u}) A \dot{X}) \otimes (A \dot{X}) \cdot \nu dX = - \int_{\Omega^0} \rho^0 \ddot{\theta} (\nabla \bar{u} A \dot{X}) \cdot (\nabla \tilde{v} A \dot{X}) dX - \int_{\Omega^0} \rho^0 \ddot{\theta} (A^2 \dot{X}) \cdot \nu dX.
\]

Thus globally

\[
\int_{\Omega^0} \rho^0 \frac{\partial^2 u}{\partial t^2} \cdot \nu dX = \int_{\Omega^0} \rho^0 [\frac{\partial^2 \bar{u}}{\partial t^2} + 2\dot{\theta} \nabla \frac{\partial \bar{u}}{\partial t} A \dot{X} + \dot{\theta} \nabla \bar{u} A \dot{X}] \cdot \nu dX
- \int_{\Omega^0} \rho^0 \ddot{\theta} (\nabla \bar{u} A \dot{X}) \cdot (\nabla \tilde{v} A \dot{X}) dX - \int_{\Omega^0} \rho^0 (\ddot{\theta} A^2 \dot{X} + \dot{\theta} A \dot{X} + \dot{Z}(t)) \cdot \nu dX.
\]
Note that two terms can deteriorate the coercivity of the problem and thus its well posedness and the stability of time integration schemes: the second one (convection term) and the fifth one. This may oblige to use additional stabilization techniques for large values of the angular velocity $\dot{\theta}$.

### 27.1.2 The available bricks . . .

To be adapted

```cpp
ind = getfem::brick_name(parameters);
```

where parameters are the parameters . . .

### 27.2 ALE terms for a uniformly translated part of an object

This section presents a set of bricks facilitating the use of an ALE formulation for an object being potentially infinite in one direction and which whose part of interests (on which the computation is considered) is translated uniformly in that direction (typically a bar).

#### 27.2.1 Theoretical background

Let us consider an object whose reference configuration $\Omega^0 \in \mathbb{R}^d$ is infinite in the direction $E_1$, i.e. $\Omega^0 = \mathbb{R} \times \omega^0$ where $\omega^0 \in \mathbb{R}^{d-1}$. At a time $t$, only a “windows” of this object is considered

$$
\Omega^0_t = (\alpha + z(t), \beta + z(t)) \times \omega^0
$$

where $z(t)$ represents the translation.

If $\varphi(t, X)$ is the deformation of the body which maps the reference configuration $\Omega^0$ to the deformed configuration $\Omega_t$ at time $t$, the ALE description consists in considering the intermediary reference configuration

$$
\bar{\Omega}^0 = (\alpha, \beta) \times \omega^0
$$

and $\bar{\varphi}(t, X) : \mathbb{R}^d \times \bar{\Omega}^0 \to \mathbb{R}^d$ defined by

$$
\bar{\varphi}(t, \bar{X}) = \varphi(t, X), \quad \text{with } \bar{X} = X - Z(t),
$$

where $Z(t) = z(t)E_1$. The interest of $\bar{\Omega}^0$ is of course to be time independant. Of course, some special boundary conditions have to be defined on $\{\alpha\} \times \omega^0$ and $\{\beta\} \times \omega^0$ (absorbing or periodic boundary conditions) in order to approximate the fact that the body is infinite.

If we denote

$$
\bar{u}(t, \bar{X}) = \bar{\varphi}(t, \bar{X}) - X = u(t, X),
$$

the displacement on the intermediary configuration, then it is easy to check that

$$
\frac{\partial \varphi}{\partial t} = \frac{\partial \bar{u}}{\partial t} - \nabla \bar{u} Z
$$

$$
\frac{\partial^2 \varphi}{\partial t^2} = \frac{\partial^2 \bar{u}}{\partial t^2} - \nabla \frac{\partial \bar{u}}{\partial t} Z + \frac{\partial^2 \bar{u}}{\partial Z^2} - \nabla \bar{u} \dot{Z}.
$$
Weak formulation of the transient terms

Assuming $\rho^0$ the density in the reference being invariant with the considered translation, the term corresponding to acceleration in the weak formulation reads (with $v(X) = \bar{v}(\bar{X})$ a test function and after integration by part):

$$
\int_{\Omega^0} \rho^0 \frac{\partial^2 u}{\partial t^2} \cdot v dX = \int_{\Omega^0} \rho^0 \left[ \frac{\partial^2 \bar{u}}{\partial t^2} - 2 \nabla \frac{\partial \bar{u}}{\partial t} \dot{Z} - \nabla \ddot{u} \dddot{Z} \right] \cdot \bar{v} - \rho^0 (\nabla \bar{u} \dot{Z} \cdot (\nabla \bar{v}) \dot{Z}) d\bar{X} + \int_{\partial \Omega^0} \rho^0 (\nabla \bar{u} \dot{Z} \cdot \bar{v} (\dot{Z} \cdot \bar{N})) d\Gamma,
$$

where $\bar{N}$ is the outward unit normal vector on $\partial \Omega^0$. Note that the last term vanishes on $(\alpha, \beta) \times \partial \omega^0$ but not necessarily on $\{\alpha\} \times \omega^0$ and $\{\beta\} \times \omega^0$. 

27.2. ALE terms for a uniformly translated part of an object 197
## Appendix A. Finite element method list

### Table 1: Symbols representing degree of freedom types

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Symbol" /></td>
<td>Value of the function at the node.</td>
</tr>
<tr>
<td><img src="image2.png" alt="Symbol" /></td>
<td>Value of the gradient along the first coordinate.</td>
</tr>
<tr>
<td><img src="image3.png" alt="Symbol" /></td>
<td>Value of the gradient along the second coordinate.</td>
</tr>
<tr>
<td><img src="image4.png" alt="Symbol" /></td>
<td>Value of the whole gradient at the node.</td>
</tr>
<tr>
<td><img src="image5.png" alt="Symbol" /></td>
<td>Value of the normal derivative to a face.</td>
</tr>
<tr>
<td><img src="image6.png" alt="Symbol" /></td>
<td>Value of the gradient along the third coordinate for 3D elements.</td>
</tr>
<tr>
<td><img src="image7.png" alt="Symbol" /></td>
<td>Value of the whole second derivative (hessian) at the node.</td>
</tr>
<tr>
<td><img src="image8.png" alt="Symbol" /></td>
<td>Scalar product with a certain vector (for instance an edge) for a vector elements.</td>
</tr>
<tr>
<td><img src="image9.png" alt="Symbol" /></td>
<td>Scalar product with the normal to a face for a vector elements.</td>
</tr>
<tr>
<td><img src="image10.png" alt="Symbol" /></td>
<td>Bubble function on an element or a face, to be specified.</td>
</tr>
<tr>
<td><img src="image11.png" alt="Symbol" /></td>
<td>Lagrange hierarchical d.o.f. value at the node in a space of details.</td>
</tr>
<tr>
<td><img src="image12.png" alt="Symbol" /></td>
<td>Value of the second derivative along the first coordinate (twice).</td>
</tr>
<tr>
<td><img src="image13.png" alt="Symbol" /></td>
<td>Value of the second derivative along the second coordinate (twice).</td>
</tr>
<tr>
<td><img src="image14.png" alt="Symbol" /></td>
<td>Value of the second cross derivative in 2D or second derivative along the third coordinate (twice) in 3D.</td>
</tr>
</tbody>
</table>
Let us recall that all finite element methods defined in GetFEM++ are declared in the file `getfem_fem.h` and that a descriptor on a finite element method is obtained thanks to the function:

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

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

### 28.1 Classical \( P_K \) Lagrange elements on simplices

It is possible to define a classical \( P_K \) Lagrange element of arbitrary dimension and arbitrary degree. Each degree of freedom of such an element corresponds to the value of the function on a corresponding node. The grid of node is the so-called Lagrange grid. Figures "Examples of classical \( P_K \) Lagrange elements on a segment."
Table 2: Examples of classical $P_K$ Lagrange elements on a triangle.

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>3 d.o.f., $C^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_2$</td>
<td>6 d.o.f., $C^0$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>10 d.o.f., $C^0$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>28 d.o.f., $C^0$</td>
</tr>
</tbody>
</table>

The number of degrees of freedom for a classical $P_K$ Lagrange element of dimension $P$ and degree $K$ is $\frac{(P + K)!}{P!K!}$. For instance, in dimension 2 ($P = 2$), this value is $\frac{(K + 1)(K + 2)}{2}$ and in dimension 3 ($P = 3$), it is $\frac{(K + 1)(K + 2)(K + 3)}{6}$. 


The particular way used in GetFEM++ to numerate the nodes are also shown in figures segment, triangle and tetrahedron. Using another numeration, let

\[ i_0, i_1, \ldots, i_P, \]

be some indices such that

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

Then, the coordinate of a node can be computed as

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

where \( S_0, S_1, \ldots, S_N \) are the vertices of the simplex (for \( K = 0 \) the particular choice \( a_{0,0,\ldots,0} = \sum_{n=0}^{P} \frac{1}{P+1} S_n \) has been chosen). Then each base function, corresponding of each node \( a_{i_0, i_1, \ldots, i_P} \) is

---

**Table 3: Examples of classical \( P_K \) Lagrange elements on a tetrahedron.**

<table>
<thead>
<tr>
<th>( P_1 ) element, 4 d.o.f., ( C^0 )</th>
<th>( P_2 ) element, 10 d.o.f., ( C^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram 1" /></td>
<td><img src="image2.png" alt="Diagram 2" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( P_3 ) element, 35 d.o.f., ( C^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Diagram 3" /></td>
</tr>
</tbody>
</table>
defined by
\[
\phi_{i_0,i_1,\ldots,i_P} = \prod_{n=0}^{P} \prod_{j=0}^{i_n-1} \left( \frac{K\lambda_n - j}{j+1} \right).
\]

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

\[
\lambda_n = x_n, \quad 0 \leq n < P, \\
\lambda_P = 1 - x_0 - x_1 - \ldots - x_{P-1}.
\]

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

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

**Table 4: Classical \(P_K\) Lagrange element "FEM_PK(P, K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>(\tau)-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K, 0 \leq K \leq 255)</td>
<td>(P, 1 \leq P \leq 255)</td>
<td>((K + P)!)</td>
<td>(C^0)</td>
<td>No ((Q = 1))</td>
<td>Yes ((M = Id))</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 5: Discontinuous \(P_K\) Lagrange element "FEM_PK_DISCONTINUOUS(P, K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>(\tau)-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K, 0 \leq K \leq 255)</td>
<td>(P, 1 \leq P \leq 255)</td>
<td>((K + P)!) (K!)</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes ((M = Id))</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 6: Discontinuous $P_K$ Lagrange element with internal dofs "FEM_PK_DISCONTINUOUS(P, K, alpha)". The method "FEM_PK_DISCONTINUOUS(P, K, 0)" is identical to "FEM_PK_DISCONTINUOUS(P, K)". For $alpha > 0$, "FEM_PK_DISCONTINUOUS(P, K, alpha)" corresponds to a Lagrange method with all finite element nodes in the interior of the domain located at the position $(alpha)g + (1 - alpha)a_i$ for $g$ the centroid of the element and $a_i$ the node of the standard $P_K$ method.

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K, 0 \leq K \leq 255$</td>
<td>$P, 1 \leq P \leq 255$</td>
<td>$(K + P)!/K!P!$</td>
<td>discontinuous</td>
<td>No $(Q = 1)$</td>
<td>Yes $(M = Id)$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

### 28.2 Classical Lagrange elements on other geometries

Classical Lagrange elements on parallelepipeds or prisms are obtained as tensor product of Lagrange elements on simplices. When two elements are defined, one on a dimension $P_1$ and the other in dimension $P_2$, one obtains the base functions of the tensorial product (on the reference element) as

$$\tilde{\varphi}_{ij}(x, y) = \tilde{\varphi}_i^1(x)\tilde{\varphi}_j^2(y), \quad x \in \mathbb{R}^{P_1}, y \in \mathbb{R}^{P_2},$$

where $\tilde{\varphi}_i^1$ and $\tilde{\varphi}_i^2$ are respectively the base functions of the first and second element.

Table 7: Examples of classical $Q_K$ Lagrange elements in dimension 2.

![Diagram](image)

$Q_1$ element, 4 d.o.f., $C^0$ $Q_3$ element, 16 d.o.f., $C^0$
The $Q_K$ element on a parallelepiped of dimension $P$ is obtained as the tensorial product of $P$ classical $P_K$ elements on the segment. Examples in dimension 2 are shown in figure dimension 2 and in dimension 3 in figure dimension 3.

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

Table 8: Examples of classical Lagrange elements in dimension 3.

<table>
<thead>
<tr>
<th>Element Type</th>
<th>Degrees of Freedom</th>
<th>Continuity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$ element</td>
<td>8 d.o.f.</td>
<td>$C^0$</td>
</tr>
<tr>
<td>$Q_3$ element</td>
<td>64 d.o.f.</td>
<td>$C^0$</td>
</tr>
<tr>
<td>$P_1 \otimes P_1$ element</td>
<td>6 d.o.f.</td>
<td>$C^0$</td>
</tr>
<tr>
<td>$P_3 \otimes P_3$ element</td>
<td>40 d.o.f.</td>
<td>$C^0$</td>
</tr>
</tbody>
</table>
Table 9: $Q_K$ Lagrange element on parallelepipeds
"FEM_QK(P, K)"

<table>
<thead>
<tr>
<th>degree $KP$, $0 \leq K \leq 255$</th>
<th>dimension $P$, $1 \leq P \leq 255$</th>
<th>d.o.f. number $(K+1)^P$</th>
<th>class $C^0$</th>
<th>vector No ($Q = 1$)</th>
<th>$\tau$-equivalent Yes ($M = Id$)</th>
<th>Polynomia Yes</th>
</tr>
</thead>
</table>

Table 10: $P_K \otimes P_K$ Lagrange element on prisms
"FEM_PK_PRISM(P, K)"

<table>
<thead>
<tr>
<th>degree $2K$, $0 \leq K \leq 255$</th>
<th>dimension $P$, $2 \leq P \leq 255$</th>
<th>d.o.f. number $(K+1) \times \frac{(K+P-1)!}{K!(P-1)!}$</th>
<th>class $C^0$</th>
<th>vector No ($Q = 1$)</th>
<th>$\tau$-equivalent Yes ($M = Id$)</th>
<th>Polynomia Yes</th>
</tr>
</thead>
</table>

Table 11: $P_{K_1} \otimes P_{K_2}$ Lagrange element on prisms
"FEM_PRODUCT(FEM_PK(P-1, K1), FEM_PK(1, K2))"

<table>
<thead>
<tr>
<th>degree $K_1 + K_2$, $0 \leq K_1, K_2 \leq 255$</th>
<th>dimension $P$, $2 \leq P \leq 255$</th>
<th>d.o.f. number $(K_2 + 1) \times \frac{(K_1 + P - 1)!}{K_1!(P - 1)!}$</th>
<th>class $C^0$</th>
<th>vector No ($Q = 1$)</th>
<th>$\tau$-equivalent Yes ($M = Id$)</th>
<th>Polynomia Yes</th>
</tr>
</thead>
</table>

28.2. Classical Lagrange elements on other geometries
Fig. 2: $P_2 \otimes P_1$ Lagrange element on a prism, 12 d.o.f., $C^0$
Fig. 3: Incomplete $Q_2$ elements in dimension two and three, 8 or 20 d.o.f., $C^0$

Table 12: Incomplete $Q_2$ Lagrange element on parallelepipeds (Quad 8 and Hexa 20 serendipity elements)
"FEM_Q2_INCOMPLETE(P)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension d.o.f. number</th>
<th>class $C^0$</th>
<th>vector $\tau$-equivalent</th>
<th>Polynomia</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$P$, $2 \leq P \leq 3$</td>
<td>8 for $P = 2$ 20 for $P = 3$</td>
<td>No ($Q = 1$) Yes ($M = Id$)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.3 Elements with hierarchical basis

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

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

28.3.1 Hierarchical elements with respect to the degree
Fig. 4: $P_K$ Hierarchical element on a segment, $C^0$

Table 13: $P_K$ Classical Lagrange element on simplices but with a hierarchical basis with respect to the degree "FEM_PK_HIERARCHICAL(P,K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$, $0 \leq K \leq 255$</td>
<td>$P$, $1 \leq P \leq 255$</td>
<td>$\frac{(K + P)!}{K!P!}$</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes ($M = Id$)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 14: $Q_K$ Classical Lagrange element on parallelepipeds but with a hierarchical basis with respect to the degree "FEM_QK_HIERARCHICAL(P,K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$, $0 \leq K \leq 255$</td>
<td>$P$, $1 \leq P \leq 255$</td>
<td>$(K + 1)^P$</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes ($M = Id$)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 15: $P^K$ Classical Lagrange element on prisms but with a hierarchical basis with respect to the degree $"FEM_PK_PRISM_HIERARCHICAL(P,K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K, 0 \leq K \leq 255$</td>
<td>$P, 2 \leq P \leq 255$</td>
<td>$(K + 1) \times \frac{(K + P - 1)!}{K!(P - 1)!}$</td>
<td>$C^K$</td>
<td>No ($Q = 1$)</td>
<td>Yes ($M = Id$)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

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

28.3.2 Composite elements

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

![Composite element diagram](image-url)
Table 16: Composition of a finite element method on an element with $S$ subdivisions
"FEM_STRUCTURED_COMPOSITE(FEM1, S)"

<table>
<thead>
<tr>
<th>degree of FEM1</th>
<th>dimension of FEM1</th>
<th>d.o.f. variable</th>
<th>class variable</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree of FEM1</td>
<td>dimension of FEM1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>piecewise</td>
</tr>
</tbody>
</table>

It is important to use a corresponding composite integration method.

### 28.3.3 Hierarchical composite elements

![Diagram of hierarchical composite element](image)

Fig. 6: hierarchical composite element "FEM_PK_HIERARCHICAL_COMPOSITE(2,1,3)"


Table 17: Hierarchical composition of a $P_K$ finite element method on a simplex with $S$ subdivisions
"FEM_PK_HIERARCHICAL_COMPOSITE(P,K,S)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>$P$</td>
<td>$(SK+P)$</td>
<td>variable</td>
<td>No $(Q = 1)$</td>
<td>Yes $(M = Id)$</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

Table 18: Hierarchical composition of a hierarchical $P_K$ finite element method on a simplex with $S$ subdivisions
"FEM_PK_FULL_HIERARCHICAL_COMPOSITE(P,K,S)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>$P$</td>
<td>$(SK+P)$</td>
<td>variable</td>
<td>No $(Q = 1)$</td>
<td>Yes $(M = Id)$</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

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

It is important to use a corresponding composite integration method.

28.4 Classical vector elements

28.4.1 Raviart-Thomas of lowest order elements

Table 19: Raviart-Thomas of lowest order element on simplices
"FEM_RT0(P)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$P$</td>
<td>$P+1$</td>
<td>H(div)</td>
<td>Yes $(Q = P)$</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Fig. 7: RT0 elements in dimension two and three. (P+1 dof, H(div))
Table 20: Raviart-Thomas of lowest order element on parallelepipeds (quadrilaterals, hexahedrals) "FEM_RT0Q(P)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P )</td>
<td>( 2P )</td>
<td>H(div)</td>
<td>Yes ((Q = \frac{P}{P}))</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.4.2 Nedelec (or Whitney) edge elements

Fig. 8: Nedelec edge elements in dimension two and three. \((P(P+1)/2\) dof, H(rot))

Table 21: Nedelec (or Whitney) edge element "FEM_NEDELEC(P)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( P )</td>
<td>( P(P+1)/2 )</td>
<td>H(rot)</td>
<td>Yes ((Q = \frac{P}{P}))</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.5 Specific elements in dimension 1

28.5.1 GaussLobatto element

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

The polynomials coefficients have been pre-computed with Maple (they require the inversion of an ill-conditioned system), hence they are only available for the following values of \( K \):
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 24, 32. Note that for \( K = 1 \) and \( K = 2 \), this is the classical \( P1 \) and \( P2 \) fem.

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynoma</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K )</td>
<td>1</td>
<td>( K + 1 )</td>
<td>( C^0 )</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 28.5.2 Hermite element

![Diagram of a segment with vertices labeled 1, 2, 3, 0]  

Fig. 9: \( P_3 \) Hermite element on a segment, 4 d.o.f., \( C^1 \)

Base functions on the reference element

\[
\begin{align*}
\hat{\varphi}_0 &= (2x + 1)(x - 1)^2, \\
\hat{\varphi}_1 &= x(x - 1)^2, \\
\hat{\varphi}_2 &= x^2(3 - 2x), \\
\hat{\varphi}_3 &= x^2(x - 1).
\end{align*}
\]

This element is close to be \( \tau \)-equivalent but it is not. On the real element the value of the gradient on vertices will be multiplied by the gradient of the geometric transformation. The matrix \( M \) is not equal to identity but is still diagonal.

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynoma</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>( C^1 )</td>
<td>No ((Q = 1))</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 28.5.3 Lagrange element with an additional bubble function

![Diagram of a segment with vertices labeled 0, 1, 2]  

Fig. 10: \( P_1 \) Lagrange element on a segment with additional internal bubble function, 3 d.o.f., \( C^0 \)
28.6 Specific elements in dimension 2

28.6.1 Elements with additional bubble functions

Table 24: Lagrange $P_1$ element with an additional internal bubble function "FEM_PK_WITH_CUBIC_BUBBLE(1, 1)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f.</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 25: Lagrange element on a triangle with additional internal bubble function

Table 26: Lagrange $P_1$ or $P_2$ element with an additional internal bubble function "FEM_PK_WITH_CUBIC_BUBBLE(2, K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f.</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>4 or 7</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Fig. 11: $P_3$ Lagrange element on a triangle with additional internal piecewise linear bubble function
Table 27: Lagrange $P_1$ with an additional internal piecewise linear 
bubble function "FEM_P1 PIECEWISE_LINEAR_BUBBLE"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4 or 7</td>
<td>$C^0$</td>
<td>No $(Q = 1)$</td>
<td>Yes</td>
<td>Piecewise</td>
</tr>
</tbody>
</table>

Fig. 12: $P_1$ Lagrange element on a triangle with additional bubble function on face 0, 4 d.o.f., $C^0$

Table 28: Lagrange $P_1$ element with an additional bubble function 
on face 0 "FEM_P1_BUBBLE_FACE(2)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>$C^0$</td>
<td>No $(Q = 1)$</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.6. Specific elements in dimension 2
28.6.2 Non-conforming \( P_1 \) element
Fig. 14: $P_1$ non-conforming element on a triangle, 3 d.o.f., discontinuous
Table 30: \( P_1 \) non-conforming element on a triangle
"FEM_P1_NONCONFORMING"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomia</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.6.3 Hermite element

Fig. 15: Hermite element on a triangle, \( P_3 \), 10 d.o.f., \( C^0 \)

Base functions on the reference element:

\[
\begin{align*}
\hat{\varphi}_0 &= (1 - x - y)(1 + x + y - 2x^2 - 2y^2 - 11xy), & (\hat{\varphi}_0(0,0) = 1), \\
\hat{\varphi}_1 &= x(1 - x - y)(1 - x - 2y), & (\partial_x \hat{\varphi}_1(0,0) = 1), \\
\hat{\varphi}_2 &= y(1 - x - y)(1 - 2x - y), & (\partial_y \hat{\varphi}_2(0,0) = 1), \\
\hat{\varphi}_3 &= -2x^3 + 7x^2y + 7xy^2 + 3x^2 - 7xy, & (\hat{\varphi}_3(1,0) = 1), \\
\hat{\varphi}_4 &= x^3 - 2x^2y - 2xy^2 - x^2 + 2xy, & (\partial_x \hat{\varphi}_4(1,0) = 1), \\
\hat{\varphi}_5 &= xy(y + 2x - 1), & (\partial_y \hat{\varphi}_5(1,0) = 1), \\
\hat{\varphi}_6 &= 7x^2y + 7xy^2 - 2y^3 + 3y^2 - 7xy, & (\hat{\varphi}_6(0,1) = 1), \\
\hat{\varphi}_7 &= xy(x + 2y - 1), & (\partial_x \hat{\varphi}_7(0,1) = 1), \\
\hat{\varphi}_8 &= y^4 - 2x^2y - 2xy^2 - y^2 + 2xy, & (\partial_y \hat{\varphi}_8(0,1) = 1), \\
\hat{\varphi}_9 &= 27xy(1 - x - y), & (\hat{\varphi}_9(1/3,1/3) = 1),
\end{align*}
\]
This element is not $\tau$-equivalent (The matrix $M$ is not equal to identity). On the real element linear combinations of $\hat{\phi}_4$ and $\hat{\phi}_7$ are used to match the gradient on the corresponding vertex. Idem for the two couples ($\hat{\phi}_5, \hat{\phi}_8$) and ($\hat{\phi}_6, \hat{\phi}_9$) for the two other vertices.

Table 31: Hermite element on a triangle "FEM_HERMITE(2)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>10</td>
<td>$C^0$</td>
<td>No $(Q = 1)$</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.6.4 Morley element

Fig. 16: triangle Morley element, $P_2$, 6 d.o.f., $C^0$

This element is not $\tau$-equivalent (The matrix $M$ is not equal to identity). In particular, it can be used for non-conforming discretization of fourth order problems, despite the fact that it is not $C^1$. 

28.6. Specific elements in dimension 2
Table 32: Morley element on a triangle "FEM_MORLEY"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>discontinuo</td>
<td>No ((Q = 1))</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

28.6.5 Argyris element

Fig. 17: Argyris element, \(P_5\), 21 d.o.f., \(C^1\)
The base functions on the reference element are:

\[ \hat{\varphi}_0(x, y) = 1 - 10x^3 - 10y^3 + 15x^4 - 30x^2y^2 + 15y^4 - 6x^5 + 30x^3y^2 + 30x^2y^3 - 6y^5, \quad (\hat{\varphi}_0(0, 0) = 1) \]
\[ \hat{\varphi}_1(x, y) = x - 6x^3 - 11xy^2 + 8x^2 + 10x^2y^2 + 18x^4y - 3x^5 + 3x^3y^2 - 10x^2y^3 - 8xy^4, \quad (\partial_x \hat{\varphi}_1(0, 0) = 1) \]
\[ \hat{\varphi}_2(x, y) = y - 11x^2y - 6y^3 + 18x^3y + 10x^2y^2 + 8y^3 - 8xy - 10x^3y^2 + 2x^2y^3 - 3y^5, \quad (\partial_y \hat{\varphi}_2(0, 0) = 1) \]
\[ \hat{\varphi}_3(x, y) = 0.5x^2 - 1.5x^3 + 1.5x^2y^2 - 0.5x^5 + 1.5x^3y^2 + x^2y^3, \quad (\partial_x^2 \hat{\varphi}_3(0, 0) = 1) \]
\[ \hat{\varphi}_4(x, y) = xy - 4x^2y - 4xy^2 + 5x^3y + 10x^2y^2 + 5xy^3 - 2x^4y - 6x^3y^2 - 6y^3x^2 - 2xy^4, \quad (\partial_x^2 \hat{\varphi}_4(0, 0) = 1) \]
\[ \hat{\varphi}_5(x, y) = 0.5y^2 - 1.5y^3 - 1.5x^2y^2 + 1.5y^4 + x^3y^2 + 1.5x^2y^3 - 0.5y^5, \quad (\partial_y \hat{\varphi}_5(0, 0) = 1) \]
\[ \hat{\varphi}_6(x, y) = 10x^3 - 15x^4 + 15x^2y^2 + 6y^3 - 15x^2y^2 - 15x^3y^2, \quad (\hat{\varphi}_6(1, 0) = 1) \]
\[ \hat{\varphi}_7(x, y) = -4x^3 + 7x^4 - 3.5x^2y^2 - 3x^3 + 3.5x^3y^2 + 3.5y^3, \quad (\partial_y \hat{\varphi}_7(1, 0) = 1) \]
\[ \hat{\varphi}_8(x, y) = -5x^2y + 14x^3y + 18.5x^2y^2 - 8x^4y - 18.5x^3y^2 - 13.5x^2y^3, \quad (\hat{\varphi}_8(1, 0) = 1) \]
\[ \hat{\varphi}_9(x, y) = 0.5x^3 - x^4 + 0.25x^2y^2 + 0.5x^5 - 0.25x^3y^2 - 0.25x^2y^3, \quad (\partial_x^2 \hat{\varphi}_9(1, 0) = 1) \]
\[ \hat{\varphi}_{10}(x, y) = x^2y - 3x^3y - 3.5x^2y^2 + 2x^4y + 3.5x^3y^2 + 2.5x^2y^3, \quad (\partial_x^2 \hat{\varphi}_{10}(1, 0) = 1) \]
\[ \hat{\varphi}_{11}(x, y) = 1.25x^2y^2 - 0.75x^3y^2 - 1.25x^2y^3, \quad (\hat{\varphi}_{11}(1, 0) = 1) \]
\[ \hat{\varphi}_{12}(x, y) = 10y^3 + 15x^2y^2 - 15x^4 - 15x^3y^2 - 15x^2y^3 + 6y^5, \quad (\hat{\varphi}_{12}(0, 1) = 1) \]
\[ \hat{\varphi}_{13}(x, y) = -5xy^2 + 18.5x^2y^2 + 14x^3y - 13.5x^3y^2 - 18.5x^2y^3 - 8x^4y, \quad (\hat{\varphi}_{13}(0, 1) = 1) \]
\[ \hat{\varphi}_{14}(x, y) = -4y^3 - 3.5x^3y^2 + 7y^4 + 3.5x^2y^2 + 3.5y^3 - 3y^5, \quad (\hat{\varphi}_{14}(0, 1) = 1) \]
\[ \hat{\varphi}_{15}(x, y) = 1.25x^3y^2 - 1.25x^3y^3 - 0.75x^3y^3, \quad (\partial_x \hat{\varphi}_{15}(0, 1) = 1) \]
\[ \hat{\varphi}_{16}(x, y) = xy^3 - 3.5x^2y^2 - 3x^3 - 2.5x^2y^3 + 3.5x^2y^2 + 2xy^4, \quad (\partial_y \hat{\varphi}_{16}(0, 1) = 1) \]
\[ \hat{\varphi}_{17}(x, y) = 0.5y^4 + 0.25x^2y^2 - y^4 - 0.25x^3y^2 - 0.25x^2y^3 + 0.5y^5, \quad (\partial_x \hat{\varphi}_{17}(0, 1) = 1) \]
\[ \hat{\varphi}_{18}(x, y) = \sqrt{2}(8x^2y^2 + 8x^2y^3 + 8x^2y^4), \quad (\partial_x \hat{\varphi}_{18}(0, 0.5) + \partial_y \hat{\varphi}_{18}(0.5, 0.5) = 1) \]
\[ \hat{\varphi}_{19}(x, y) = -16x^2y^2 + 32x^2y^3 + 32x^2y^4 - 16x^3y^3 - 32y^3 - 16x^2y, \quad (\hat{\varphi}_{19}(0.5, 0.5) = 1) \]
\[ \hat{\varphi}_{20}(x, y) = -16x^2y + 32x^3y + 32x^2y^2 - 16x^2y + 32x^2y^3 - 16x^2y, \quad (\hat{\varphi}_{20}(0.5, 0.5) = 1) \]

This element is not \( \tau \)-equivalent (The matrix \( M \) is not equal to identity). On the real element linear combinations of the transformed base functions \( \hat{\varphi}_i \) are used to match the gradient, the second derivatives and the normal derivatives on the faces. Note that the use of the matrix \( M \) allows to define Argyris element even with nonlinear geometric transformations (for instance to treat curved boundaries).

**Table 33: Argyris element on a triangle "FEM_ARGYRIS"**

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomia</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>21</td>
<td>( C^1 )</td>
<td>No ((Q = 1))</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**28.6.6 Hsieh-Clough-Tocher element**

This element is not \( \tau \)-equivalent. This is a composite element. Polynomial of degree 3 on each of the three sub-triangles (see figure Hsieh-Clough-Tocher (HCT) element, P.3. 12 d.o.f., \( C^1 \) and [ciarlet1978]). It is strongly advised to use a "IM_HCT_COMPOSITE" integration method with this finite element. The numbering of the dof is the following: 0, 3 and 6 for the lagrange dof on the first second and third vertex respectively; 1, 4, 7 for the derivative with respects to the first variable; 2, 5, 8 for the derivative with respects to the second variable and 9, 10, 11 for the normal derivatives on face 0, 1, 2 respectively.

**Table 34: HCT element on a triangle "FEM_HCT_TRIANGLE"**

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomia</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
<td>( C^1 )</td>
<td>No ((Q = 1))</td>
<td>No</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

**28.6. Specific elements in dimension 2**
Fig. 18: Hsieh-Clough-Tocher (HCT) element, $P_3$, 12 d.o.f., $C^1$
This element exists also in its reduced form, where the normal derivatives are assumed to be polynomial of degree one on each edge (see figure Reduced Hsieh-Clough-Tocher (reduced HCT) element, \( P_3 \), 9 d.o.f., \( C^1 \)).

### Table 35: Reduced HCT element on a triangle

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>( \tau )-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
<td>( C^1 )</td>
<td>No ( (Q = 1) )</td>
<td>No</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

**28.6.7 A composite \( C^1 \) element on quadrilaterals**

This element is not \( \tau \)-equivalent. This is a composite element. Polynomial of degree 3 on each of the four sub-triangles (see figure Composite element on quadrilaterals, piecewise \( P_3 \), 16 d.o.f., \( C^1 \)). At least on the reference element it corresponds to the Fraeijs de Veubeke-Sander element (see [ciarlet1978]). It is strongly advised to use a "IM_QUADC1_COMPOSITE" integration method with this finite element.
Fig. 20: Composite element on quadrilaterals, piecewise $P_3$, 16 d.o.f., $C^1$
Table 36: $C^1$ composite element on a quadrilateral (FVS) "FEM_QUADC1_COMPOSITE"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>16</td>
<td>$C^1$</td>
<td>No $(Q = 1)$</td>
<td>No</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

Fig. 21: Reduced composite element on quadrilaterals, piecewise $P_3$, 12 d.o.f., $C^1$

This element exists also in its reduced form, where the normal derivatives are assumed to be polynomial of degree one on each edge (see figure "Reduced composite element on quadrilaterals, piecewise $P_3$, 12 d.o.f., $C^1$")

28.6. Specific elements in dimension 2
Table 37: Reduced $C^1$ composite element on a quadrilateral (reduced FVS) "FEM_REDUCE_QUADC1_COMPOSITE"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>12</td>
<td>$C^1$</td>
<td>No ($Q = 1$)</td>
<td>No</td>
<td>piecewise</td>
</tr>
</tbody>
</table>

### 28.7 Specific elements in dimension 3

#### 28.7.1 Lagrange elements on 3D pyramid

*GetFEM++* proposes some Lagrange pyramidal elements of degree 0, 1 and two based on [GR-GH1999] and [BE-CO-DU2010]. See these references for more details. The proposed element can be raccorded to standard $P_1$ or $P_2$ Lagrange fem on the triangular faces and to a standard $Q_1$ or $Q_2$ Lagrange fem on the quadrilateral face.

Table 38: Lagrange element on a pyramidal element of order 0, 1 and 2

<table>
<thead>
<tr>
<th>Degree 0 pyramidal element with 1 dof</th>
<th>Degree 1 pyramidal element with 5 dof</th>
<th>Degree 2 pyramidal element with 14 dof</th>
</tr>
</thead>
</table>

The associated geometric transformations are "GT_PYRAMID(K)" for $K = 1$ or 2. The associated integration methods "IM_PYRAMID(im)" where $im$ is an integration method on a hexahedron (or alternatively "IM_PYRAMID_COMPOSITE(im)" where $im$ is an integration method on a tetrahedron, but it is theoretically less accurate) The shape functions are not polynomial ones but rational fractions. For the first degree the shape functions read:

\[
\hat{\varphi}_0(x, y, z) = \frac{1}{4} \left( 1 - x - y - z + \frac{xy}{1 - z} \right), \\
\hat{\varphi}_1(x, y, z) = \frac{1}{4} \left( 1 + x - y - z - \frac{xy}{1 - z} \right), \\
\hat{\varphi}_2(x, y, z) = \frac{1}{4} \left( 1 - x + y - z - \frac{xy}{1 - z} \right), \\
\hat{\varphi}_3(x, y, z) = \frac{1}{4} \left( 1 + x + y - z + \frac{xy}{1 - z} \right), \\
\hat{\varphi}_4(x, y, z) = z.
\]

For the second degree, setting

\[
\xi_0 = \frac{1 - z - x}{2}, \quad \xi_1 = \frac{1 - z - y}{2}, \quad \xi_2 = \frac{1 - z + x}{2}, \quad \xi_3 = \frac{1 - z + y}{2}, \quad \xi_4 = z,
\]
the shape functions read:

\[ \hat{\phi}_0(x, y, z) = \frac{\xi_0 \xi_1}{(1 - \xi_4)^2} (1 - \xi_4 - 2\xi_0)(1 - \xi_4 - 2\xi_1) - \xi_4(1 - \xi_4), \]

\[ \hat{\phi}_1(x, y, z) = 4\frac{\xi_0 \xi_1 \xi_2}{(1 - \xi_4)^2} (2\xi_1 - (1 - \xi_4)), \]

\[ \hat{\phi}_2(x, y, \xi_4) = \frac{\xi_1 \xi_2}{(1 - \xi_4)^2} ((1 - \xi_4 - 2\xi_1)(1 - \xi_4 - 2\xi_2) - \xi_4(1 - \xi_4)), \]

\[ \hat{\phi}_3(x, y, z) = 4\frac{\xi_0 \xi_4}{(1 - \xi_4)^2} (2\xi_0 - (1 - \xi_4)), \]

\[ \hat{\phi}_4(x, y, z) = 16\frac{\xi_0 \xi_1 \xi_2 \xi_3}{(1 - \xi_4)^2}, \]

\[ \hat{\phi}_5(x, y, z) = 4\frac{\xi_1 \xi_2 \xi_3}{(1 - \xi_4)^2} (2\xi_2 - (1 - \xi_4)), \]

\[ \hat{\phi}_6(x, y, z) = \frac{\xi_3 \xi_0}{(1 - \xi_4)^2} ((1 - \xi_4 - 2\xi_3)(1 - \xi_4 - 2\xi_0) - \xi_4(1 - \xi_4)), \]

\[ \hat{\phi}_7(x, y, z) = 4\frac{\xi_2 \xi_3}{(1 - \xi_4)^2} (2\xi_3 - (1 - \xi_4)), \]

\[ \hat{\phi}_8(x, y, z) = \frac{\xi_2 \xi_3}{(1 - \xi_4)^2} ((1 - \xi_4 - 2\xi_2)(1 - \xi_4 - 2\xi_3) - \xi_4(1 - \xi_4)), \]

\[ \hat{\phi}_9(x, y, z) = 4\frac{\xi_4}{1 - \xi_4} \xi_0 \xi_1, \]

\[ \hat{\phi}_{10}(x, y, z) = 4\frac{\xi_4}{1 - \xi_4} \xi_1 \xi_2, \]

\[ \hat{\phi}_{11}(x, y, z) = 4\frac{\xi_4}{1 - \xi_4} \xi_4 \xi_0, \]

\[ \hat{\phi}_{12}(x, y, z) = 4\frac{\xi_4}{1 - \xi_4} \xi_4 \xi_3, \]

\[ \hat{\phi}_{13}(x, y, z) = \xi_4(2\xi_4 - 1). \]

Table 39: Continuous Lagrange element of order 0, 1 or 2

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>(\tau)-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>(C^0)</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>14</td>
<td>(C^0)</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 40: Discontinuous Lagrange element of order 0, 1 or 2

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>(\tau)-equivalent</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>14</td>
<td>discontinuous</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
28.7.2 Elements with additional bubble functions

Table 41: Lagrange element on a tetrahedron with additional internal bubble function

<table>
<thead>
<tr>
<th>$P_1$ with additional bubble function, 5 d.o.f., $C^0$</th>
<th>$P_2$ with additional bubble function, 11 d.o.f., $C^0$</th>
<th>$P_3$ with additional bubble function, 21 d.o.f., $C^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 42: $P_K$ Lagrange element with an additional internal bubble function "FEM_PK_WITH_CUBIC_BUBBLE(3, K)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomi</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>5, 11 or 21</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 43: Lagrange $P_1$ element with an additional bubble function on face 0 "FEM_P1_BUBBLE_FACE(3)"

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>$\tau$-equivalent</th>
<th>Polynomi</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>$C^0$</td>
<td>No ($Q = 1$)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Fig. 22: $P_1$ Lagrange element on a tetrahedron with additional bubble function on face 0, 5 d.o.f., $C^0$
Fig. 23: Hermite element on a tetrahedron, $P_3$, 20 d.o.f., $C^0$
28.7.3 Hermite element

Base functions on the reference element:

\[ \hat{\phi}_0(x, y) = 1 - 3x^2 - 13xy - 13xz - 3y^2 - 13yz - 3z^2 + 2x^3 + 13x^2y + 13xy^2 + 13xyz + 13xz^2 + 2y^3 + 13yz^2 + 13y^3z + 13y^2z + 2z^3, \]  
\[ \hat{\phi}_1(x, y) = x - 2x^2 - 3xy - 3xz + x^3 + 3x^2y + 3x^2z + 2xy^2 + 4xyz + 2xz^2, \]  
\[ \hat{\phi}_2(x, y) = y - 3xy - 2y^2 - 3yz + 2x^2y + 3xy^2 + 4xyz + y^3 + 3y^2z + 2yz^2, \]  
\[ \hat{\phi}_3(x, y) = z - 3xz - 3yz - 2z^2 + 2x^2z + 4xyz + 3x^2z + 2y^2z + 3yz^2 + z^3, \]  
\[ \hat{\phi}_4(x, y) = 3x^3 - 7xy - 7xz - 2x^3 + 7x^2y + 7xz^2 + 7y^2z + 7x^2z, \]  
\[ \hat{\phi}_5(x, y) = -x^2 + 2xy + 2xz + x^3 - 2x^2y - 2xz^2 - 2y^2x - 2y^2z - 2x^2y - 2xz^2, \]  
\[ \hat{\phi}_6(x, y) = -xy + 2x^2y + xy^2, \]  
\[ \hat{\phi}_7(x, y) = -xz + 2x^2z + xz^2, \]  
\[ \hat{\phi}_8(x, y) = -7xy + 3y^2 - 7yz + 7x^2y + 7xy^2 + 7xyz - 2y^3 + 7y^2z + 7yz^2, \]  
\[ \hat{\phi}_9(x, y) = -xy + x^2y + 2xy^2, \]  
\[ \hat{\phi}_{10}(x, y) = 2xy - y^2 + 2yz - 2x^2y - 2xy^2 - 2xyz + y^3 - 2yz^2 - 2y^2z, \]  
\[ \hat{\phi}_{11}(x, y) = -yz + 2y^2z + yz^2, \]  
\[ \hat{\phi}_{12}(x, y) = -7xz - 7yz + 3x^2 + 7x^2z + 7xyz - 2x^2z + 7y^2z - 2z^3, \]  
\[ \hat{\phi}_{13}(x, y) = -xz + x^2z + 2xz^2, \]  
\[ \hat{\phi}_{14}(x, y) = -yz + y^2z + 2yz^2, \]  
\[ \hat{\phi}_{15}(x, y) = 2xz + 2yz - z^2 - 2x^2z - 2xy^2 - 2x^2y - 2yz^2 - 2y^2z + z^3, \]  
\[ \hat{\phi}_{16}(x, y) = 27xyz, \]  
\[ \hat{\phi}_{17}(x, y) = 27yz - 27xyz - 27y^2z - 27yz^2, \]  
\[ \hat{\phi}_{18}(x, y) = 27xz - 27x^2z - 27xyz - 27xz^2, \]  
\[ \hat{\phi}_{19}(x, y) = 27xy - 27x^2y - 27xy^2 - 27xyz, \]

This element is not \(\tau\)-equivalent (The matrix \(M\) is not equal to identity). On the real element linear combinations of \(\hat{\phi}_8, \hat{\phi}_{12}\) and \(\hat{\phi}_{16}\) are used to match the gradient on the corresponding vertex. Idem on the other vertices.

<table>
<thead>
<tr>
<th>degree</th>
<th>dimension</th>
<th>d.o.f. number</th>
<th>class</th>
<th>vector</th>
<th>(\tau)-equivalent</th>
<th>Polynomials</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>20</td>
<td>(C^0)</td>
<td>No ((Q = 1))</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 44: Hermite element on a tetrahedron

"FEM_HERMITE(3)"
Appendix B. Cubature method list

For more information on cubature formulas, the reader is referred to [EncyclopCubature] and [so-se-do2004] for instance. The integration methods are of two kinds. Exact integrations of polynomials and approximated integrations (cubature formulas) of any function. The exact integration can only be used if all the elements are polynomial and if the geometric transformation is linear.

A descriptor on an integration method is given by the function:

\[
\text{ppi} = \text{getfem::int_method_descriptor("name of method");}
\]

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

The program integration located in the tests directory lists and checks the degree of each integration method.

### 29.1 Exact Integration methods

*GetFEM++* furnishes a set of exact integration methods. This means that polynomials are integrated exactly. However, their use is (very) limited and not recommended. The use of exact integration methods is limited to the low-level generic assembly for polynomial \( \tau \)-equivalent elements with linear transformations and for linear terms. It is not possible to use them in the high-level generic assembly.

The list of available exact integration methods is the following
Table 1: Exact Integration Methods

<table>
<thead>
<tr>
<th>Method Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IM_NONE()&quot;</td>
</tr>
<tr>
<td>&quot;IM_EXACT_SIMPLEX(n)&quot;</td>
</tr>
<tr>
<td>&quot;IM_PRODUCT(a, b)&quot;</td>
</tr>
<tr>
<td>&quot;IM_EXACT_PARALLELEPIPED(n)&quot;</td>
</tr>
<tr>
<td>&quot;IM_EXACT_PRISM(n)&quot;</td>
</tr>
</tbody>
</table>

Even though a description of exact integration method exists on parallelepipeds or prisms, most of the time the geometric transformations on such elements are nonlinear and the exact integration cannot be used.

29.2 Newton cotes Integration methods

Newton cotes integration of order \( K \) on simplices, parallelepipeds and prisms are denoted by "IM_NC(N,K)", "IM_NC_PARALLELEPIPED(N,K)" and "IM_NC_PRISM(N,K)" respectively.

29.3 Gauss Integration methods on dimension 1

Gauss-Legendre integration on the segment of order \( K \) (with \( K/2+1 \) points) are denoted by "IM_GAUSS1D(K)". Gauss-Lobatto-Legendre integration on the segment of order \( K \) (with \( K/2+1 \) points) are denoted by "IM_GAUSSLOBATTO1D(K)". It is only available for odd values of \( K \). The Gauss-Lobatto integration method can be used in conjunction with "FEM_PK_GAUSSLOBATTO1D(K/2)" to perform mass-lumping.
### 29.4 Gauss Integration methods on dimension 2

#### Table 2: Integration methods on dimension 2

<table>
<thead>
<tr>
<th>graphic</th>
<th>coordinates (x, y)</th>
<th>weights</th>
<th>function to call / order</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1/3, 1/3)</td>
<td>1/2</td>
<td>&quot;IM_TRIANGLE(1)&quot;</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1/6, 1/6)</td>
<td>1/6</td>
<td>&quot;IM_TRIANGLE(2)&quot;</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(2/3, 1/6)</td>
<td>1/6</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1/6, 2/3)</td>
<td>1/6</td>
<td></td>
</tr>
</tbody>
</table>

1 point, order 1.

3 points, order 2.
### Table 3: Integration methods on dimension 2

<table>
<thead>
<tr>
<th>graphic</th>
<th>coordinates (x, y)</th>
<th>weights</th>
<th>function to call / order</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1/3, 1/3)</td>
<td>-27/96</td>
<td>&quot;IM_TRIANGLE(3)&quot;</td>
</tr>
<tr>
<td></td>
<td>(1/5, 1/5)</td>
<td>25/96</td>
<td>4 points, order 3.</td>
</tr>
<tr>
<td></td>
<td>(3/5, 1/5)</td>
<td>25/96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1/5, 3/5)</td>
<td>25/96</td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(a, a)</td>
<td>c</td>
<td>&quot;IM_TRIANGLE(4)&quot;</td>
</tr>
<tr>
<td></td>
<td>(1-2a, a)</td>
<td>c</td>
<td>6 points, order 4</td>
</tr>
<tr>
<td></td>
<td>(a, 1-2a)</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b, b)</td>
<td>d</td>
<td>a = 0.445948490915965</td>
</tr>
<tr>
<td></td>
<td>(1-2b, b)</td>
<td>d</td>
<td>b = 0.091576213509771</td>
</tr>
<tr>
<td></td>
<td>(b, 1-2b)</td>
<td>d</td>
<td>c = 0.111690794839005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>d = 0.054975871827661</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1/3, 1/3)</td>
<td>9/80</td>
<td>&quot;IM_TRIANGLE(5)&quot;</td>
</tr>
<tr>
<td></td>
<td>(a, a)</td>
<td>c</td>
<td>7 points, order 5</td>
</tr>
<tr>
<td></td>
<td>(1-2a, a)</td>
<td>c</td>
<td>a = b = 4/7 - a c =</td>
</tr>
<tr>
<td></td>
<td>(a, 1-2a)</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b, b)</td>
<td>d</td>
<td>b = 155 + \sqrt{15}</td>
</tr>
<tr>
<td></td>
<td>(1-2b, b)</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b, 1-2b)</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>
Table 4: Integration methods on dimension 2

<table>
<thead>
<tr>
<th>graphic</th>
<th>coordinates (x, y)</th>
<th>weights</th>
<th>function to call / order</th>
</tr>
</thead>
</table>
| ![Diagram](image) | (a, a) | h | "IM_TRIANGLE(7)"
| | (b, a) | h | 13 points, order 7 |
| | (a, b) | h | a = |
| | (c, e) | i | b = |
| | (d, c) | i | c = |
| | (e, d) | i | d = |
| | (d, e) | i | e = |
| | (c, d) | i | f = |
| | (e, c) | i | g = |
| | (f, f) | j | g = |
| | (g, f) | j | h = |
| | (1/3, 1/3) | k | k = |

(1/2 + \sqrt{1/6}, 1/2)
(1/2 - \sqrt{1/24}, 1/2 ± \sqrt{1/8})
(1/3, 1/3)

"IM_QUAD(2)"
3 points, order 2

(see [EncyclopCubature])

29.4. Gauss Integration methods on dimension 2
There is also the "IM_GAUSS_PARALLELEPIPED(n, k)" which is a direct product of 1D gauss integrations.

**Important note:** do not forget that IM_QUAD(k) is exact for polynomials up to degree k, and that a $Q_k$ polynomial has a degree of $2 \times k$. For example, IM_QUAD(7) cannot integrate exactly the product of two $Q_2$ polynomials. On the other hand, IM_GAUSS_PARALLELEPIPED(2, 4) can integrate exactly that product …
### 29.5 Gauss Integration methods on dimension 3

#### Table 6: Integration methods on dimension 3

<table>
<thead>
<tr>
<th>Graphic</th>
<th>Coordinates (x, y)</th>
<th>Weights</th>
<th>Function to call / Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Graphic 1" /></td>
<td>(1/4, 1/4, 1/4)</td>
<td>1/6</td>
<td>&quot;IM_TETRAHEDRON(1)&quot; 1 point, order 1</td>
</tr>
<tr>
<td><img src="image2.png" alt="Graphic 2" /></td>
<td>(a, a, a) (a, b, a) (a, a, b) (b, a, a)</td>
<td>1/24</td>
<td>&quot;IM_TETRAHEDRON(2)&quot; 4 points, order 2</td>
</tr>
</tbody>
</table>

- \( a = \frac{5 - \sqrt{5}}{20} \)
- \( b = \frac{5 + 3\sqrt{5}}{20} \)
Table 7: Integration methods on dimension 3

<table>
<thead>
<tr>
<th>graphic</th>
<th>coordinates (x, y)</th>
<th>weights</th>
<th>function to call / order</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td>(1/4, 1/4, 1/4)</td>
<td>-2/15</td>
<td>&quot;IM_TETRAHEDRON (3) &quot; 5 points, order 3</td>
</tr>
<tr>
<td><img src="image2" alt="Diagram" /></td>
<td>(1/4, 1/4, 1/4)</td>
<td>8/405</td>
<td>&quot;IM_TETRAHEDRON (5) &quot; 15 points, order 5</td>
</tr>
</tbody>
</table>

### Cubature method list
Others methods are:

<table>
<thead>
<tr>
<th>name</th>
<th>element type</th>
<th>number of points</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IM_TETRAHEDRON(6)&quot;</td>
<td>tetrahedron</td>
<td>24</td>
</tr>
<tr>
<td>&quot;IM_TETRAHEDRON(8)&quot;</td>
<td>tetrahedron</td>
<td>43</td>
</tr>
<tr>
<td>&quot;IM_SIMPLEX4D(3)&quot;</td>
<td>4D simplex</td>
<td>6</td>
</tr>
<tr>
<td>&quot;IM_HEXAHEDRON(5)&quot;</td>
<td>3D hexahedron</td>
<td>14</td>
</tr>
<tr>
<td>&quot;IM_HEXAHEDRON(9)&quot;</td>
<td>3D hexahedron</td>
<td>58</td>
</tr>
<tr>
<td>&quot;IM_HEXAHEDRON(11)&quot;</td>
<td>3D hexahedron</td>
<td>90</td>
</tr>
<tr>
<td>&quot;IM_CUBE4D(5)&quot;</td>
<td>4D parallelepiped</td>
<td>24</td>
</tr>
<tr>
<td>&quot;IM_CUBE4D(9)&quot;</td>
<td>4D parallelepiped</td>
<td>145</td>
</tr>
</tbody>
</table>

29.6 Direct product of integration methods

You can use "IM_PRODUCT(IM1, IM2)" to produce integration methods on quadrilateral or prisms. It gives the direct product of two integration methods. For instance "IM_GAUSS_PARALLELEPIPED(2,k)" is an alias for "IM_PRODUCT(IM_GAUSS1D(2,k), IM_GAUSS1D(2,k))" and can be use instead of the "IM_QUAD" integrations.

29.7 Specific integration methods

For pyramidal elements, "IM_PYRAMID(im)" provides an integration method corresponding to the transformation of an integration im from a hexahedron (for instance "IM_GAUSS_PARALLELEPIPED(3,5)") onto a pyramid. It is a singular integration method specically adapted to rational fraction shape functions of the pyramidal elements.

29.8 Composite integration methods

Use "IM_STRUCTURED_COMPOSITE(IM1, S)" to copy IM1 on an element with S subdivisions. The resulting integration method has the same order but with more points. It could be more stable to use a composite method rather than to improve the order of the method. Those methods have to be used also with composite elements. Most of the time for composite element, it is preferable to choose the basic method IM1 with no points on the boundary (because the gradient could be not defined on the boundary of sub-elements).

For the HCT element, it is advised to use the "IM_HCT_COMPOSITE(im)" composite integration (which split the original triangle into 3 sub-triangles).

For pyramidal elements, "IM_PYRAMID_COMPOSITE(im)" provides an integration method ase on the decomposition of the pyramid into two tetrahedrons (im should be an integration method on a tetrahedron). Note that the integraton method "IM_PYRAMID(im)" where im is an integration method on an hexahedron, should be prefered.
Fig. 1: Composite method "IM_STRUCTURED_COMPOSITE(IM_TRIANGLE(2), 3)"
References


Index

A
add() (built-in function), 18
add_fem_data (C function), 105
add_fem_variable (C function), 105
add_filtered_fem_variable (C function), 105
add_fixed_size_data (C function), 105
add_fixed_size_variable (C function), 104, 105
add_initialized_fem_data (C function), 105
add_initialized_fixed_size_data (C function), 105
add_initialized_scalar_data (C function), 105
add_multiplier (C function), 105
asm, 34, 59

C
complex_rhs (C function), 106
complex_tangent_matrix (C function), 106
complex_variable (C function), 105

F
fem, 22

G
generic assembly, 34, 59
getfem::slicer_apply_deformation (C function), 96
getfem::slicer_boundary (C function), 96
getfem::slicer_build_edges_mesh (C function), 97
getfem::slicer_build_mesh (C function), 97
getfem::slicer_build_stored_mesh_slice (C function), 97

getfem::slicer_complementary (C function), 96
getfem::slicer_cylinder (C function), 96
getfem::slicer_explode (C function), 97
getfem::slicer_half_space (C function), 96
getfem::slicer_intersect (C function), 96
getfem::slicer_isovalues (C function), 96
getfem::slicer_mesh_with_mesh (C function), 96
getfem::slicer_none (C function), 96
getfem::slicer_sphere (C function), 96
getfem::slicer_union (C function), 96

I
index() (built-in function), 19
is_in() (built-in function), 18
is_only_convexes() (built-in function), 19
is_only_faces() (built-in function), 18

M
m.is_complex (C function), 104
mesh, 22
mesh_fem, 22
mesh_fem_of_variable (C function), 106
mf.basic_dof_on_region() (built-in function), 27
mf.clear() (built-in function), 25
mf.convex_index() (built-in function), 25
mf.dof_on_region() (built-in function), 27
mf.extension_matrix() (built-in function), 27
mf.fem_of_element() (built-in function), 25
mf.first_convex_of_basic_dof() (built-in function), 27
mf.get_gdim() (built-in function), 27
mf.ind_basic_dof_of_element() (built-in function), 26
mf.is_reduced() (built-in function), 27
mf.linked_mesh() (built-in function), 25
mf.nb_basic_dof() (built-in function), 27
mf.nb_basic_dof_of_element() (built-in function), 26
mf.nb_dof() (built-in function), 27
mf.point_of_basic_dof() (built-in function), 26
mf.reduce_to_basic_dof() (built-in function), 27
mf.reduction_matrix() (built-in function), 27
mf.reference_point_of_basic_dof() (built-in function), 26
mf.set_reduction() (built-in function), 27
mf.set_reduction_matrices() (built-in function), 27
mim.clear() (built-in function), 31
mim.convex_index() (built-in function), 31
mim.int_method_of_element() (built-in function), 31
mim.linked_mesh() (built-in function), 31
mymesh.clear() (built-in function), 18
mymesh.convex_area_estimate() (built-in function), 18
mymesh.convex_index() (built-in function), 17
mymesh.convex_quality_estimate() (built-in function), 18
mymesh.convex_radius_estimate() (built-in function), 18
mymesh.convex_to_point() (built-in function), 17
mymesh.dim() (built-in function), 17
mymesh.has_region() (built-in function), 18
mymesh.ind_points_of_convex() (built-in function), 17
mymesh.is_convex_having_neighbour() (built-in function), 18
mymesh.neighbour_of_convex() (built-in function), 17
mymesh.neighbours_of_convex() (built-in function), 17
mymesh.normal_of_face_of_convex() (built-in function), 18
mymesh.optimize_structure() (built-in function), 18
mymesh.points_index() (built-in function), 17
mymesh.points_of_convex() (built-in function), 17
mymesh.region() (built-in function), 18
mymesh.structure_of_convex() (built-in function), 17
mymesh.trans_of_convex() (built-in function), 18
mymesh.write_to_file() (built-in function), 20
N
Nitsche’s method, 119
R
real_rhs (C function), 106
real_tangent_matrix (C function), 106
real_variable (C function), 105
S
sup() (built-in function), 18