User documentation

Nicolas KIELBASIEWICZ, Eric LUNÉVILLE

November 4, 2016
<table>
<thead>
<tr>
<th>7.2</th>
<th>Linear Solvers</th>
<th>159</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.1</td>
<td>Direct solvers</td>
<td>160</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Iterative solvers</td>
<td>161</td>
</tr>
<tr>
<td>7.3</td>
<td>Eigen solvers</td>
<td>163</td>
</tr>
<tr>
<td>7.3.1</td>
<td>How to call an eigen solver?</td>
<td>163</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Results</td>
<td>164</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Calling sequence</td>
<td>165</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Advanced usage of ARPACK</td>
<td>173</td>
</tr>
<tr>
<td>8</td>
<td>Post processing and outputs</td>
<td>184</td>
</tr>
<tr>
<td>8.1</td>
<td>Integral representation</td>
<td>184</td>
</tr>
<tr>
<td>8.2</td>
<td>Output functions</td>
<td>186</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Print objects</td>
<td>186</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Export TermMatrix and TermVector</td>
<td>186</td>
</tr>
<tr>
<td>A</td>
<td>External libraries</td>
<td>188</td>
</tr>
<tr>
<td>A.1</td>
<td>Installation and use of BLAS and LAPACK libraries</td>
<td>188</td>
</tr>
<tr>
<td>A.2</td>
<td>Installation and use of UmfPACK library</td>
<td>188</td>
</tr>
<tr>
<td>A.3</td>
<td>Installation and use of ARPACK++ library</td>
<td>189</td>
</tr>
<tr>
<td>B</td>
<td>Utility types in details</td>
<td>190</td>
</tr>
<tr>
<td>B.1</td>
<td>String, Strings</td>
<td>190</td>
</tr>
<tr>
<td>B.2</td>
<td>Int, Dimen, Number, Numbers</td>
<td>191</td>
</tr>
<tr>
<td>B.3</td>
<td>Real, Complex and Reals</td>
<td>192</td>
</tr>
<tr>
<td>B.4</td>
<td>Point</td>
<td>192</td>
</tr>
<tr>
<td>B.5</td>
<td>Vector</td>
<td>194</td>
</tr>
<tr>
<td>B.6</td>
<td>Matrix</td>
<td>196</td>
</tr>
<tr>
<td>B.7</td>
<td>Parameters</td>
<td>198</td>
</tr>
<tr>
<td>B.7.1</td>
<td>The Parameter object</td>
<td>198</td>
</tr>
<tr>
<td>B.7.2</td>
<td>The Parameters: list of Parameter</td>
<td>199</td>
</tr>
<tr>
<td>B.8</td>
<td>Function</td>
<td>201</td>
</tr>
<tr>
<td>B.8.1</td>
<td>User function and object function</td>
<td>201</td>
</tr>
<tr>
<td>B.8.2</td>
<td>Advanced user</td>
<td>204</td>
</tr>
<tr>
<td>B.9</td>
<td>Kernel</td>
<td>205</td>
</tr>
<tr>
<td>B.10</td>
<td>Timer</td>
<td>208</td>
</tr>
</tbody>
</table>
XLiFE++ is the heir of 2 main finite elements library developed in POEMS laboratory, namely MELINA (and its C++ avatar MELINA++) and MONTJOIE, respectively developed since 1989 and 2003. It is a C++ high level library devoted to extended finite elements methods. Writing programs using XLiFE++ needs only basic knowledge of C++ language, so that it can be used to teach finite elements methods, but it is quite perfect for research.

XLiFE++ is self-consistent. It provides advanced mesh tools, with refinement methods, has every kind of elements (including pyramids) needed by finite elements methods, boundary elements methods or discontinuous galerkin methods, direct/iterative solvers and eigen solvers. Next to this, it provides also a wide range of interfaces to well-known libraries or softwares, such that UmfPack, ARPACK++, and an advanced interface to the mesh generator GMSH, so that you can do everything needed in a single program.

This documentation is dedicated to students at Master level, to engineers and researchers at any level, in so far as partial differential equations are concerned.
Introduction

Partial differential equations (PDE hereafter) are the core of modelling. A wide range of problems in Physics, Engineering, Mathematics, Banking are modelled by PDEs.

XLiFE++ is a C++ library designed to solve these equations numerically. It is a free extended library based on finite elements methods. It is an autonomous library, providing everything you need for solving such problems, including interfaces to specific external libraries or softwares, such as Gmsh, Arpack++, UmfPack, ...

What does XLiFE++ do?

- Problem description (real or complex) by their variational formulations, with full access to the internal vectors or matrices;
- Multi-variables, multi-equations, 1D, 2D and 3D, linear or non linear coupled systems;
- Easy geometric input by composite description, to build meshes thanks to Gmsh;
- Easy automatic mesh generation on elementary geometries, based on refinement methods;
- Very high level user-friendly typed input language with full algebra of analytic and finite elements functions. Your main program will be very similar to the mathematical objects;
- A wide range of finite elements: segments, triangles, quadrangles, hexahedra, tetrahedra, prisms and pyramids;
- A wide set of internal linear direct and iterative solvers (LU, Cholesky, BiCG, BiCGStab, CG, CGS, GMRES, QMR, SOR, SSOR, ...) and internal eigenvalues and eigenvectors solvers, plus additional interfaces to external solvers (Arpack++, UmfPack,...);
- A full documentation suite: source documentation (online or inside sources), user documentation (pdf), developer documentation (pdf);
- An experimental parallel version using OpenMP.

1.1 Installation

1.1.1 Requirements


To use the full abilities of mesh definition in XLiFE++, you may also install Gmsh (http://gmsh.info).

To use the full abilities of XLiFE++, you may also install Arpack and/or UmfPack.

To plot solutions, you may install Gmsh, or Paraview (http://www.paraview.org).
1.1.2 Download

XLiFE++ is downloadable at the following url http://uma.ensta-paristech.fr/soft/XLiFE++/?module=main&action=dl. You can download releases and snapshots. Snapshots are supposed to be generated automatically every day when necessary.

There are 3 kinds of archives (snapshots or releases):

1. a "source" archive contains all XLIPE++ source files and tex/pdf documentation;
2. a "api" archive contains only source documentation generated by DOXYGEN;
3. a "full" archive contains everything.

What do you have to know about the content of the XLIPE++ directory?

- **bin** the binary directory contains the xlifepp_project_setup gui for every OS. Its use will be explained later.
- **doc** the doc directory contains the present user guide, the developer guide (also in pdf) and the source documentation (html)
- **src** the source directory contains all C++ sources of the XLIPE++ library
- **tests** the test directory contains all unitary and system tests to check your installation
- **lib** After the compilation, the lib directory will contain the static libraries of XLIPE++.
- **usr** This directory contains the user files to write and compile a C++ program using XLIPE++
- **CMakeLists.txt** the CMAKE compilation script.

1.1.3 Installation

You get the archive of XLIPE++ sources, you have to unzip everywhere you want in your hard drive.

Compilation of XLIPE++ is managed by CMAKE, a high-level cross-platform compilation tool, running on MacOS, Windows, and Linux. This application only needs a configuration file named CMakeLists.txt, at the root directory of the XLIPE++ archive. Whatever the OS, cmake also asks for another directory where to put generated files for compilation, called build directory hereafter. This directory can be everywhere. It will contains compilation files (objects files, ...), a Makefile or an IDE project file named XLIPE++ (for Eclipse, CodeBlocks, Visual Studio, XCode, ...). So we suggest you to set this directory as a subdirectory of XLIPE++ install directory, with the name "build" for instance.
Under MacOS and Linux

Figure 1.1: ccmake (MacOS, Unix)

When running ccmake, the default CMAKE GUI under MacOS or Linux, the build directory is given in argument.

    ccmake [options] path/to/CMakeLists.txt path/to/buildDirectory

When running CMAKE GUI application, you have to set the directory “Where is the source code” containing CMakeLists.txt you want to run CMAKE on, and to set the build directory: “Where to build the binaries”. Then, you click the “Configure” button. It will ask ou the generator you want. Then, you click the “Generate” button, to generate your IDE project file or your Makefile.

Figure 1.2: CMAKE application (MacOS)

When running CMAKE in command-line mode, the build directory is generally the directory in
which you are when calling the CMAKE command. If you want to know the general case, please take a look at CMAKE option -b.

To compile XLiFE++, you just have to run CMAKE on the CMakeLists.txt file:

```
cmake path/to/CMakeLists.txt
```

If you generated a project file, you launch your IDE on this file and can compile and run tests. The following command chooses for instance to use codeblocks on unix platform:

```
cmake path/to/CMakeLists.txt -G "CodeBlocks - Unix Makefiles"
```

If you generated a Makefile, you have to run the make command, to compile XLiFE++ sources and tests. If you want to compile XLiFE++ sources only, the target is "libs":

```
make libs
make
```

If you generated an IDE project file, you have to open it and compile target all to compile everything, sources and tests, or target libs, to compile sources only.

**Under Windows**

When running CMAKE GUI application, you have to set the directory “Where is the source code” containing CMakeLists.txt you want to run CMAKE on, and to set the build directory: “Where to build the binaries”. Then, you click the “Configure” button. It will ask you the generator you want. Then, you click the “Generate” button, to generate your IDE project file or your Makefile.

![CMake GUI](image)

Figure 1.3: `cmake` (Windows), after you clicked the configure button.
You can have the list of available generators inside CMake help. It does not tell that eclipse, codeblocks, ... are installed, but it tells that they can be installed on your computer and how you can tell CMake to use them as generators.

Useful general options to CMake

Four options may be very useful:

**CMAKE_CXX_COMPILER** This option is used to set the compiler CMake will use to compile your code:

```bash
cmake path/to/CMakeLists.txt -DCMAKE_CXX_COMPILER=g++-48
```

**CMAKE_BUILD_TYPE** This option is used to set the type of build (debug, release, ...). Possible values are essentially Debug and Release. The default value is Release.

```bash
cmake path/to/CMakeLists.txt -DCMAKE_BUILD_TYPE=Debug
```

**XLIFEPP_PRECISION** This option is used to set the precision of numerical types (int, ...). Possible values are STD, LONG and LONGLONG. The default value is LONG.

```bash
cmake path/to/CMakeLists.txt -DXLIFEPP_PRECISION=LONG
```

Except if you have a very good reason, you are not supposed to use this option.

**XLIFEPP_STRING_TYPE** This option is used to set string type: STD (for std::string) and WIDE (for std::wstring). The default value is STD.

```bash
cmake path/to/CMakeLists.txt -DXLIFEPP_STRING_TYPE=STD
```

Configuring XLiFE++ with Gmsh and/or Paraview

**XLIFEPP_GMSH_EXECUTABLE** to specify the Gmsh binary with full path. If Gmsh is in your PATH, it will be automatically detected, else you can use this option:

```bash
cmake path/to/CMakeLists.txt -DXLIFEPP_GMSH_EXECUTABLE=gmsh/exe/with/full/path
```

**XLIFEPP_PARAVIEW_EXECUTABLE** to specify the Paraview binary with full path. If Paraview is in your PATH, it will be automatically detected, else you can use this option:

```bash
cmake path/to/CMakeLists.txt
    -DXLIFEPP_PARAVIEW_EXECUTABLE=paraview/exe/with/full/path
```

Configuring XLiFE++ with OpenMP

**XLIFEPP_ENABLE_OMP** Activates/Deactivates use of OpenMP

```bash
cmake path/to/CMakeLists.txt -DXLIFEPP_ENABLE_OMP=ON
```
Configuring XLiFE++ with ARPACK

Configuring with ARPACK also means configuring with BLAS and LAPACK. When asked, these 3 libraries are automatically searched in standard paths. If these libraries are installed in other paths, you can give it directly. Otherwise, sources files of ARPACK inside XLiFE++ sources will be compiled and used.

**XLIFEPP_ENABLE_ARPACK** Activates/Deactivates configuration with ARPACK  
**XLIFEPP_BLAS_LIB_DIR** to specify the directory containing BLAS library  
**XLIFEPP_LAPACK_LIB_DIR** to specify the directory containing LAPACK library  
**XLIFEPP_ARPACK_LIB_DIR** to specify the directory containing ARPACK library  
**XLIFEPP_FORTRAN_LIB_DIR** to specify the directory containing FORTRAN library

See section A.3 for more details

Configuring XLiFE++ with UMFPACK

Configuring with UMFPACK also means configuring with BLAS and LAPACK. It can also mean configuring with other libraries provided by SUITESPARSE, insofar as UMFPACK is now distributed inside SUITESPARSE.

**XLIFEPP_ENABLE_UMFPACK** Activates/Deactivates configuration with UMFPACK  
**XLIFEPP_BLAS_LIB_DIR** to specify the directory containing BLAS library  
**XLIFEPP_LAPACK_LIB_DIR** to specify the directory containing LAPACK library  
**XLIFEPP_UMFPACK_INCLUDE_DIR** to specify the directory containing UMFPACK header  
**XLIFEPP_UMFPACK_LIB_DIR** to specify the directory containing UMFPACK header  
**XLIFEPP_SUITESPARSE_HOME_DIR** to specify the home directory of SUITESPARSE, containing UMFPACK. This option is to be used if you compiled SUITESPARSE by yourself. In this case, UMFPACK will be searched in the UMFPACK subdirectory.  
**XLIFEPP_FORTRAN_LIB_DIR** to specify the directory containing FORTRAN library

There are specific options to each library provided by SUITESPARSE UMFPACK may depend on. See section A.2 for more details

Using CMAKE generates a cache file (CMakeCache.txt) containing value of every option defined during execution of CMAKE, so that when reusing cmake on the same file resuses CMakeCache.txt, unless you change some options. This is why XLIFEPP_ENABLE_XXX options can take the value OFF.
1.2 How to use XLiFE++

1.2.1 Writing a program using XLiFE++

All the XLiFE++ library is defined in the namespace `xlifepp`. Then the users, if they refer to library objects, have to add once in their programs the command `using namespace xlifepp;`. Besides, they have to use the ”super” header file `xlife++.h` only in the main. A main program looks like, for instance:

```cpp
#include "xlife++.h"
using namespace xlifepp;

int main()
{
    init(en); // mandatory initialization of xlife++
    ...
}
```

1.2.2 Compiling a program using XLiFE++

The manual way

This way supposed that you know where XLiFE++ is installed.

1. You create your working directory.
2. You copy the content of XLiFE++ usr directory in your working directory (CMakeLists.txt file and/or main.cpp file)
3. You run CMAKE on the CMakelists.txt file to get your makefile or files for your IDE project (Eclipse, XCode, CodeBlocks, Visual C++, . . .)
4. You can now edit the main.cpp file to write your program and enjoy compilation with XLiFE++

The command-line way

This way is possible to make easier the manual way. In the bin directory of XLiFE++, you have shell script called `xlifepp.sh` for MacOS and Linux, and a batch script called `xlifepp.bat`. You can define a shortcut on it wherever you want.

Here is the list of options of both scripts:

**USAGE:**

```
xlifepp.sh --build [--interactive] [(-generate|--no-generate)]
xlifepp.sh --build --non-interactive [(-generate|--no-generate)]
    [-clean]
    [--compiler <compiler>] [--directory <dir>]
    [--generator-name <generator>]
    [--build-type <build-type>]
    [(--with-omp|--without-omp)]
```

```
xlifepp.sh --help
xlifepp.sh --version
```
MAIN OPTIONS:

--build, -b  
copy cmake files and eventually sample of main file
and run cmake on it to prepare your so-called project directory. This is the default

--clean, -cl  
clean project directory (remove CMake build files)

--generate, -g  
generate the project. Used with --build option.
This is the default.

--help, -help, -h  
show the current help

--interactive, -i  
runtime xlifepp in interactive mode. Used with --build option. This is the default

--non-interactive, -noi  
runtime xlifepp in non interactive mode. Used with --build option

--no-generate, -nog  
prevent generation of your project. You will do it yourself.

--version, -v  
print version number of XLiFE++ and its date

--verbose-level <value>, -vl <value>  
set the verbose level. Default value is 1

OPTIONS FOR BUILD IN NON INTERACTIVE MODE:

--build-type <value>, -bt <value>  
set cmake build type (Debug, Release, ...).

--cxx-compiler <value>, -cxx <value>  
set the C++ compiler to use.

--directory <dir>, -d <dir>  
set the directory where you want to build your project

--generator-name <name>, -gn <name>  
set the cmake generator.

--main-file <filename>, -nof  
copy <filename> as a main file for the user project.
do not copy the sample main.cpp file. This is the default.

--with-omp, -omp  
activates OpenMP mode

--without-omp, -nomp  
deactivates OpenMP mode

The graphical way

This way is possible to make easier the manual way and more pleasant than the command-line way. On the website, you have a GUI application called xlifepp-qt for MacOS, (Windows and Linux will come soon). You can define a shortcut on it wherever you want.
This application is a graphical user interface to the first 3 steps of the manual way.
1.3 Licence and credits

Licence

XLiFE++ is copyright (C) 2010-2014 by E. Lunéville and N. Kielbasiewicz and is distributed under the terms of the GNU General Public License (GPL) (Version 3 or later, see https://www.gnu.org/licenses/gpl-3.0.en.html). This means that everyone is free to use XLIIFE++ and to redistribute it on a free basis. XLIIFE++ is not in the public domain; it is copyrighted and there are restrictions on its distribution. You cannot integrate XLIIFE++ (in full or in parts) in any closed-source software you plan to distribute (commercially or not). If you want to integrate parts of XLIIFE++ into a closed-source software, or want to sell a modified closed-source version of XLIIFE++, you will need to obtain a different license. Please contact us directly for more information.

The developers do not assume any responsibility in the numerical results obtained using the XLIIFE++ library and are not responsible of bugs.

Credits

The XLIIFE++ library has been mainly developed by E. Lunéville and N. Kielbasiewicz of POEMS lab (UMR 7231, CNRS-ENSTA ParisTech-INRIA). Some parts are inherited from Melina++ library developed by D. Martin (IRMAR lab, Rennes University, now retired) and E. Lunéville. Other contributors are:

- Y. Lafranche (IRMAR lab), mesh tools using subdivision algorithms
- C. Chambeyron (POEMS lab), iterative solvers
- M.H N’Guyen (POEMS lab), eigen solvers and OpenMP implementation
- N. Salles (POEMS lab), boundary element methods
2.1 The variational approach

Before learning in details what XLiFE++ is able to do, let us explain the basics with an example, the **Helmholtz** equation:

For a given function \( f(x,y) \), find a function \( u(x,y) \) satisfying

\[
\begin{align*}
-\Delta u(x,y) + u(x,y) &= f(x,y) \quad \forall (x,y) \in \Omega \\
\frac{\partial u}{\partial n}(x,y) &= 0 \quad \forall (x,y) \in \partial \Omega
\end{align*}
\]  \tag{2.1}

To solve this problem by a finite element method, XLiFE++ is based on its variational formulation: find \( u \in H^1(\Omega) \) such that \( \forall v \in H^1(\Omega) \)

\[
\int_{\Omega} \nabla u \cdot \nabla v \, dx \, dy - \int_{\Omega} u \, v \, dx \, dy = \int_{\Omega} f \, v \, dx \, dy. \tag{2.2}
\]

All the mathematical objects involved in the variational formulation are described in XLiFE++. The following program solves the Helmholtz problem with \( f(x,y) = \cos \pi x \cos \pi y \) and \( \Omega \) is the unit square.

```cpp
#include "xlife++.h"
using namespace xliifepp;
using namespace std;

Real cosx2(const Point& P, Parameters& pa = defaultParameters) {
  Real x=P(1), y=P(2);
  return cos(pi_*x)*cos(pi_*y);
}

int main(int argc, char** argv) {
  init(_lang=fr);
  Square S(_origin=Point(0.,0.), _length=1, _nnodes=11, _domain_name="Omega");
  Mesh m(S, triangle, 1, structured, "triangle mesh of unit square");
  Domain omega=m.domain("Omega");
  Space sp(omega, P1, "Vh");
  Unknown u(sp, "u");
  TestFunction v(u, "v");
  BilinearForm a = intg(omega, grad(u) | grad(v)) + intg(omega, u * v);
  LinearForm l = intg(omega, cosx2 * v);
  TermMatrix A(a, "a(u,v)");
  TermVector b(l, "l(v)");
  TermVector U = directSolve(A, b);
}
Please notice how close to the Mathematics, XLiFE++ input language is.

2.2 How does it work ?

This first example shows how XLiFE++ executes all the usual steps required by the Finite Element Method. Let us walk through them one by one.

line 13 : every program using XLiFE++ begins by a call to the init function, taking up to 4 key/value arguments:

_lang enum to set the language for print and log messages. Possible values are en, fr, or de. Default value is en.
_verb integer to set the verbose level. Default value is 1.
_trackingMode boolean to set if in the log file, you have a backtrace of every call to a XLiFE++ routine. Default value is false.
_isLogged boolean to activate log. Default value is false.

Furthermore, the init function loads functionalities linked to the trace of where such messages come from. If this function is not called, XLiFE++ cannot work !!!

init(_lang=fr);

lines 14-15 : The mesh will be generated on the unit square geometry with 11 nodes per edge. Arguments of a geometry are given with a key/value system. _origin is the bottom left front vertex of Square. Next, we precise the mesh element type (here triangle), the mesh element order (here 1), and an optional description. See chapter 5 for more examples of mesh definitions.

Square S(_origin=Point(0,0), _length=1, _nnodes=11, _domain_name="Omega");
Mesh m(S, triangle, 1, structured, "triangle mesh of unit square");

line 16 : The main domain, named "Omega" in the mesh, is defined.

Domain omega=m.domain("Omega");

line 17 : A finite element space is generally a space of polynomial functions on elements, triangles here only. Here sp is defined as the space of continuous functions which are affine on each triangle $T_k$ of the domain $\Omega$, usually named $V_h$. The dimension of such a space is finite, so we can define a basis.

$$sp(\Omega, P1) = \left\{ w(x, y) \text{ such that } \exists (w_1, \ldots, w_N) \in \mathbb{R}^N, w(x, y) = \sum_{i=1}^{N} w_k \varphi_k(x, y) \right\}$$

where $N$ is the space dimension, i.e. the number of nodes, i.e. the number of vertices here. Currently, XLiFE++ implements the following elements : $P_k$ on segment, triangle and tetrahedron, $Q_k$ on quadrangle and hexahedron, $O_k$ on prism and pyramid (see Mesh chapter for more details).
lines 18-21: The unknown $u$ here is an approximation of the solution of the problem. $v$ is declared as test function. This comes from the variational formulation of Equation 2.1: multiplying both sides of equation and integrating over $\Omega$, we obtain:

$$-\int_{\Omega} v\Delta u dxdy + \int_{\Omega} v u dxdy = \int_{\Omega} f dxdy$$

Then, using Green’s formula, the problem is converted into finding $u$ such that:

$$a(u,v) = \int_{\Omega} \nabla u \cdot \nabla v dxdy + \int_{\Omega} u v dxdy = \int_{\Omega} f v dxdy = l(v) \quad (2.3)$$

The 4 next lines in the program declare $u$ and $v$ and define $a$ and $l$.

Please notice that:

- the test function is defined from the unknown. The reason is that the test function is dual to the unknown. Through the unknown, $v$ is also defined on the same space.
- the right hand side needs the definition of the function $f$. Such function can be defined as a classical C++ function, but with a particular prototype. In this example, $f$ (i.e. $\cos x^2$) is a scalar function. So it takes 2 arguments: the first one is a Point, containing coordinates $x$ and $y$. The second one is optional and contains parameters to use inside the function. Here, the Parameters object is not used. At last, as a scalar function, it returns a Real.

```cpp
Real cosx2(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2);
    return cos(pi_*x) * cos(pi_*y);
}
```

lines 22-23: The previous definitions are a description of the variational form. Now, we have to define the matrix and the right-hand side vector which are the algebraic representations of the linear forms in the finite element space. This is done by the first 2 following lines.

```cpp
TermMatrix A(a, "a(u,v)");
TermVector b(l, "l(v)");
```

line 24: Matrix and vector being assembled, you can now choose the solver you want. Here, the direct solver is automatically chosen.

XLiFE++ offers you a various choice of direct or iterative solvers:

- $LU$, $LDU$, $LL^t$, $LDL^t$, $LDL^*$ factorizations
• BICG, BiCGStab, CG, CGS, GMRES, QMR, Sor, SSor, solvers
• internal eigen solver
• interfaces to external packages such as UMFPACK, Arpack++

See chapter 7 for more details.

```cpp
TermVector U = directSolve(A, b);
```

**line 25**: To save the solution, XLiFE++ provides an export to Paraview format file (vtu).

```cpp
saveToFile("U", U, vtu);
```

**line 26**: This is the end of the program. A "main" function always ends with this line.

```cpp
return 0;
```
3.1 A 1D problem

Solving 1D problems seems to be out of interest. Anyway, most of FE software does not deal with. In fact, they are of interest, often as a part of more complex problems. XLiFE++ deals with 1D problem. As an example, we show how to solve the very simple problem:

\[
\begin{cases}
-u'' = f & \text{in } \Omega = [0, 1] \\
u(0) = u(1) = 0
\end{cases}
\]

Its variational formulation is

\[
\text{find } u \in V = \{ v \in L^2(\Omega), \ v' \in L^2(\Omega), \ u(0) = u(1) = 0 \} \text{ such that } \\
\int_0^1 u'(x)v'(x)dx = \int_0^1 f(x)v(x)dx \quad \forall v \in V.
\]

The following main program corresponds to solving this problem with \( f(x) = 1 \) using P1 Lagrange element (100 elements):

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
{return -1.;}

int main(int argc, char** argv)
{
    init(_lang=en);  // mandatory initialization of xlifepp

    // mesh and domains
    Strings sn("x=0", "x=1");
    Mesh mesh1d(Segment(_xmin=0, _xmax=1, _nnodes=11, _domain_name="Omega", _side_names=sn), 1, structured, "P1-mesh");
    Domain omega = mesh1d.domain("Omega");
    Domain sigmaL = mesh1d.domain("x=0"), sigmaR = mesh1d.domain("x=1");

    // space, unknowns, and test functions
    Space Vh(omega, P1, "Vh", true);
    Unknown u(Vh, "u");
    TestFunction v(u, "v");

    // define problem
    BilinearForm a = intg(omega, grad(u) | grad(v));
    LinearForm lf = intg(omega, f*v);
    EssentialConditions ecs = (u|sigmaL = 0) & (u|sigmaR = 0);

    // compute matrix and rhs
    TermMatrix A(a, ecs, "A");
    TermVector F(lf, "F");
```
3.2 Laplace Problems

We investigate here problems involving laplacian operator in a 2D bounded domain, say $\Omega$:

$$-\Delta u + a u = f \quad \text{in } \Omega \quad (a = -k^2 \text{ for Helmholtz equation})$$

and various essential conditions (Dirichlet, transmission, quasi periodic, average condition).

3.2.1 Neumann condition

First, let us consider the case of the homogeneous Neumann condition on $\partial \Omega$, the boundary of $\Omega$:

$$\frac{\partial u}{\partial n} = 0 \quad \text{on } \partial \Omega.$$

The variational formulation we deal with is

$$\text{find } u \in V = \{ v \in L^2(\Omega), \nabla v \in (L^2(\Omega))^2 \} \text{ such that } \int_{\Omega} \nabla u \cdot \nabla v + a \int_{\Omega} u v = \int_{\Omega} f v \quad \forall v \in V.$$

The following main program corresponds to solving this problem on unity square $\Omega = ]0,1[ \times ]0,1[$ with $f(x) = \cos \pi x \cos \pi y$ using P1 Lagrange element (20x20 elements):

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real cosx2(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2);
    // solve linear system and save solution
    TermVector U = directSolve(A, F);
    saveToFile("U_1d", U, vtu);
    return 0;
}
Viewing result using PARAVIEW:
```

![Graphical representation of a solution](image-url)
return cos(pi_ * x) * cos(pi_ * y);
}

int main(int argc, char** argv)
{
    init(_lang=en);

    //mesh square
    Square sq(_origin=Point(0.,0.), _length=1, _nnodes=21);
    Mesh mesh2d(sq, triangle, 1, structured);
    Domain omega = mesh2d.domain("Omega");

    //build space and unknown
    Interpolation& Pk=interpolation(Lagrange, standard, 1, H1);
    Space Vk(omega, Pk, "Vk", true);
    Unknown u(Vk, "u");
    TestFunction v(u, "v");

    //define variational formulation
    BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v);
    LinearForm fv=intg(omega, cosx2 * v);

    //compute matrix and right hand side
    TermMatrix A(auv, "a(u,v)");
    TermVector B(fv, "f(v)");

    //LLt factorize and solve
    TermMatrix LD;
    IdltFactorize(A, LD);
    TermVector U = factSolve(LD, B);

    saveToFile("U_LN.vtk", U, vtk);
    return 0;
}

Viewing result using PARAVIEW:

Solving this problem with P2 Lagrange interpolation should be the same except the line defining
the space:

Space Vh(omega, P2, "Vh", true);
Solving this problem in a 3D domain should be the same except the line defining the mesh and the right hand side function. For instance, on the unity cube, the mesh construction command using GMSH tool is:

```cpp
Real f(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2), z=P(3);
    return cos(pi*x) * cos(pi*y) * cos(pi*z);
}
```

Mesh mesh(Cube(0,1,0,1,0,1,10,10,10,sidenames),tetrahedron,1,gmsh,"P1 mesh");

### 3.2.2 Dirichlet condition

Let us consider now the case of non homogeneous Dirichlet condition on the boundaries $x = 0$ ($\Sigma^-$) and $x = 1$ ($\Sigma^+$):

$$u = 1 \text{ on } \Sigma^- \cup \Sigma^+.$$  

The variational formulation is now ($a = 0$)

$$\begin{vmatrix}
\int_{\Omega} \nabla u . \nabla v = \int_{\Omega} f v \\
 u = 1 \text{ on } \Sigma^- \cup \Sigma^+ 
\end{vmatrix} \forall v \in V, v = 0 \text{ on } \Sigma^- \cup \Sigma^+$$

Its approximation by P1 Lagrange finite element is implemented in XLIFE++ as follows

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
{ return -8.;}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp

    //create mesh of square
    Strings sn("y=0","x=1","y=1","x=0");
```

18
Square sq(_origin=Point(0.,0.), _length=1, _nnodes=20, _domain_name="Omega", _side_names=sn);
Mesh mesh2d(sq, triangle, 1, structured);
Domain omega=mesh2d.domain("Omega");
Domain sigmaM=mesh2d.domain("x=0"); sigmaP=mesh2d.domain("x=1");

// create interpolation
Space V(omega, P1, "V", true);
Unknown u(V, "u");
TestFunction v(u, "v");

// create bilinear form, linear form and their algebraic representation
BilinearForm auv=intg(omega, grad(u)|grad(v));
LinearForm fv=intg(omega, f*v);
EssentialConditions ecs= (u|sigmaM = 1) & (u|sigmaP = 1);
TermMatrix A(auv, ecs, "A");
TermVector B(fv, "B");

// solve linear system AX=B
TermVector U=directSolve(A, B);
saveToFile("U_LD", U, vtu);
return 0;
}

Viewing result using PARAVIEW:

Note how easy is to take into account essential conditions. Only two lines has to be modified!

3.2.3 Periodic condition

Now we consider the Laplace problem on the unit square \( \Omega = [0,1] \times [0,1] \) equipped with Dirichlet condition on and periodic condition:

\[
\begin{cases}
  -\Delta u = f \quad \text{in } \Omega \\
  u|_{\Gamma^-} = 0 \text{ and } u|_{\Gamma^+} = 0 \\
  u|_{\Sigma^-} = u|_{\Sigma^+} \quad \text{and } \partial_x u|_{\Sigma^-} = \partial_x u|_{\Sigma^+}
\end{cases}
\]
and its variational formulation in $V = \{ v \in H^1(\Omega), \; u_{\Gamma} = 0 \text{ and } u_{\Sigma-} = u_{\Sigma+} \}$:

$$
\begin{cases}
\text{find } u \in V \text{ such that } \\
\int_{\Omega} \nabla u \cdot \nabla v = \int_{\Omega} f v \quad \forall v \in V.
\end{cases}
$$

Its approximation by $P^2$ Lagrange finite element is implemented in XLiFE++ as follows

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2);
    return (4*pi*pi*y*(y-1)-2)*sin(2*pi*x);
}

Vector<Real> mapPM(const Point& P, Parameters& pa = defaultParameters)
{
    Vector<Real> Q(P);
    Q(1)-=1;
    return Q;
}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp

    // mesh square
    Strings sn("y=0","x=1","y=1","x=0");
    Square sq(_origin=Point(0.,0.), _length=1, _nnodes=20, _domain_name="Omega",
               _side_names=sn);
    Mesh mesh2d(sq, triangle, 1, structured);
    Domain omega=mesh2d. domain("Omega");
    Domain sigmaM=mesh2d. domain("x=0");
    Domain sigmaP=mesh2d. domain("x=1");
    Domain gammaM=mesh2d. domain("y=0");
    Domain gammaP=mesh2d. domain("y=1");
    defineMap(sigmaP, sigmaM, mapPM); // useful to periodic condition

    // create P2 Lagrange interpolation
    Space V(omega, P2, "$V\$, true);
    Unknown u(V, "$u\$);
    TestFunction v(u, "$v\$);

    //create bilinear form and linear form
    BilinearForm auv=intg(omega, grad(u)*grad(v));
    LinearForm fv=intg(omega, f*v);
    EssentialConditions ecs = (u|gammaM = 0) & (u|gammaP = 0)
                        & ((u|sigmaP) - (u|sigmaM) = 0); // EssentialConditions ecs

    TermMatrix A(auv, ecs, "A");
    TermVector B(fv, "B");

    // solve linear system AX=F using factorization
    TermVector U=directSolve(A, B);
    saveToFile("U_LP", U, vtu);
    return 0;
}
```
Note that at corners, periodic condition and Dirichlet condition are redundant. When executing, the following warning message is thrown:

```
Constraints::reduceConstraints() : in essential conditions
  Dirichlet condition u = 0 on y=0
  Dirichlet condition u = 0 on y=1
  periodic condition u|x=1 - u|x=0 = 0
2 redundant constraint row(s) detected and eliminated
```

### 3.2.4 Transmission condition

We turn to the Laplace problem with transmission condition:

\[
\begin{cases}
  -\Delta u^-= f & \text{in } \Omega^- \\
  -\Delta u^+= f & \text{in } \Omega^+ \\
  u^-|_{\Sigma^-} = u^+|_{\Sigma^+} = 1 \\
  u^-|_{\Gamma} = u^+|_{\Gamma} \text{ and } \partial_x u^-|_{\Gamma} = \partial_x u^+|_{\Gamma}
\end{cases}
\]

Its variational formulation in

\[ V = \{(v^-, v^+) \in H^1(\Omega^-) \times H^1(\Omega^+), \; v^-|_{\Sigma^-} = 0, v^+|_{\Sigma^-} = 0, \; v^-|_{\Gamma} = v^+|_{\Gamma} \} \]

is

\[
\begin{cases}
  \text{find } (u^-, u^+) \in H^1(\Omega^-) \times H^1(\Omega^+), \; u^-|_{\Sigma^-} = 1, u^+|_{\Sigma^-} = 1, \; u^-|_{\Gamma} = u^+|_{\Gamma} \text{ such that}
\end{cases}
\]

\[
\int_{\Omega^-} \nabla u^- \cdot \nabla v^- + \int_{\Omega^+} \nabla u^+ \cdot \nabla v^+ = \int_{\Omega^-} f v^- + \int_{\Omega^+} f v^+ \; \forall v \in V.
\]

Note that derivatives matching is taken into account in a weak sense. The implementation in XLiFE++, using $P^2$ Lagrange finite element, looks like:
```cpp
#include "xlife++\h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
{
    return -8.;
}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp
    //mesh domain
    Strings sn(4);
    sn[1] = "x=1/2-"; sn[3] = "x=0";
    Rectangle r1(_xmin=0, _xmax=0.5, _ymin=0, _ymax=1, _nnodes=Numbers(20,40),
        _domain_name="Omega-", _side_names=sn);
    Mesh mesh2d(r1, triangle, 1, _structured);
    sn[1] = "x=1"; sn[3] = "x=1/2+";
    Rectangle r2(_xmin=0.5, _xmax=1, _ymin=0, _ymax=1, _nnodes=Numbers(20,40),
        _domain_name="Omega+", _side_names=sn);
    Mesh mesh2d_p(r2, _triangle, 1, _structured);
    mesh2d.merge(mesh2d_p);
    Domain omegaM=mesh2d.domain("Omega-"), omegaP=mesh2d.domain("Omega+");
    Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");
    Domain gamma=mesh2d.domain("x=1/2- or x=1/2+");
    // create P2 interpolation
    Space VM(omegaM, P2, "VM", true);
    Unknown uM(VM, "u-"); TestFunction vM(uM, "v-");
    Space VP(omegaP, P2, "VP", true);
    Unknown uP(VP, "u+"); TestFunction vP(uP, "v+");
    //create bilinear form and linear form
    BilinearForm auv=intg(omegaM, grad(uM) | grad(vM)) + intg(omegaP, grad(uP) | grad(vP));
    LinearForm fv=intg(omegaM, f*vM) + intg(omegaP, f*vP);
    EssentialConditions ecs= (uM|sigmaM = 1) & (uP|sigmaP = 1) & ((uM|gamma) = (uP|gamma) = 0);
    TermMatrix A(auv, ecs, "A"); TermVector B(fv, "B");
    //solve linear system AX=B using LU factorization
    TermVector U=directSolve(A, B);
    saveToFile("U_LT", U, vtu);
    return 0;
}
```

Here, a tool merging mesh is used to create a two domains mesh. Gmsh should be used also. The picture below shows that the solution is continuous across the boundary $\Gamma$. 
3.2.5 Average condition

As a last example of essential condition, we consider average condition, for instance:

\[ \int_{\Sigma} u = 0. \]

Such condition is tricky to take into account in FE softwares. Generally, they do not! Because XLiFE++ uses a powerful process to deal with essential conditions, such condition can be easily addressed:

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters) {
    return -8.;}

int main(int argc, char** argv) {
    init(_lang=en); // mandatory initialization of xlifepp

    // create a mesh and Domains
    Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=10,
        _domain_name="Omega", _side_names=Strings("y=0", "x=1", "y=1", "x=0")),
        triangle, 1, structured);
    Domain omega=mesh2d.domain("Omega");
    Domain sigmaM=mesh2d.domain("x=0"), sigmaP=mesh2d.domain("x=1");

    // create interpolation
    Space V(omega, P2, "V", true);
    Unknown u(V, "u");
    TestFunction v(u, "v");

    // create bilinear form and linear form
    BilinearForm auv=intg(omega, grad(u) | grad(v));
    LinearForm fv=intg(omega, f*v);
    EssentialConditions ecs= (intg(sigmaM, u) = 0);
    TermMatrix A(auv, ecs, "A");
    TermVector F(fv ,"B");
```
### 3.3 Mixed formulation using P0 and Raviart-Thomas elements

Consider the Laplace problem with homogeneous Dirichlet condition:

\[ \begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial \Omega \end{cases} \]

Introducing \( p = \nabla u \), it is rewritten as a mixed problem in \((u, p)\):

\[ \begin{cases} -\text{div } p = f & \text{in } \Omega \\ p = \nabla u & \text{in } \Omega \\ u = 0 & \text{on } \partial \Omega \end{cases} \]

with the following variational formulation:

\[ \begin{cases} \text{find } (u, p) \in L^2(\Omega) \times H(div, \Omega) \text{ such that} \\ -\int_{\Omega} \text{div } p \, v = \int_{\Omega} f \, v & \forall v \in L^2(\Omega) \\ \int_{\Omega} u \text{div } q + \int_{\Omega} p \, q & \forall q \in H(div, \Omega). \end{cases} \]

Note that the Dirichlet boundary condition is a natural condition in this formulation.

The XLiFE++ implementation of this problem using P0 approximation for \( L^2(\Omega) \) and an approximation of \( H(div, \Omega) \) using Raviart-Thomas elements of order 1 is the following:
#include "xlife++.h"
using namespace xlifepp;

Real f(const Point& P, Parameters& pa = defaultParameters)
{  Real x=P(1), y=P(2);
   return 32*(x*(1-x)+y*(1-y));}

int main(int argc, char** argv)
{
   init(_lang=en);
   //mesh square
   Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=21), triangle, 1, structured);
   Domain omega=mesh2d.domain("Omega");
   //create approximation P0 and RT1
   Space H(omega, _P0, "H", false);
   Space V(omega, _RT1, "V", true);
   Unknown p(V,"p");
   TestFunction q(p,"q");  //p=grad(u)
   Unknown u(H,"u");
   TestFunction v(u,"v");
   //create problem (Poisson problem)
   TermMatrix A(intg(omega, p|q) + intg(omega, u*div(q)) − intg(omega, div(p)*v));
   TermVector b(intg(omega, f*v));
   //solve and save solution
   TermVector X=directSolve(A, b);
   saveToFile("u", X(u), vtk);
   return 0;
}

Using Paraview with the Cell data to point data filter that moves P0 data to P1 data and the Warp by scalar filter that produces elevation, the approximated field $u$ looks like:
3.4 2D Maxwell equations using Nedelec elements

XLiFE++ provides Nedelec elements (first and second family) that are $H(curl)$ conforming. Consider the following academic Maxwell problem:

\[
\begin{align*}
\text{curl curl } E - \omega^2 \mu \varepsilon E &= f \quad \text{in } \Omega \\
E \times n &= 0 \quad \text{on } \partial \Omega
\end{align*}
\]

with the following weak form:

\[
\begin{align*}
\text{find } E &\in V = \{ v \in H(curl, \Omega), \ v \times n = 0 \text{ on } \partial \Omega \} \text{ such that} \\
\int_{\Omega} \text{curl } E \text{ curl } v &= \int_{\Omega} E v \quad \forall v \in V.
\end{align*}
\]

Using first family Nedelec’s element, the XLiFE++ program looks like:

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real omg=1, eps=1, mu=1, a=pi, ome=omg*omg*mu*mu*eps;
Vector<Real> f (const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2);
    Vector<Real> res (2);
    Real c=2*a*a-ome;
    res (1)=-c*cos(a*x)*sin(a*y);
    res (2)= c*sin(a*x)*cos(a*y);
    return res;
}

Vector<Real> solex (const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), y=P(2);
    Vector<Real> res (2);
    res (1)=-cos(a*x)*sin(a*y);
    res (2)= sin(a*x)*cos(a*y);
    return res;
}

int main(int argc, char** argv)
{
    init (_lang=en);
    //mesh square using gmsh
    Strings sidenames (4, "Gamma");
    Mesh mesh2d (Rectangle (_xmin=0, _xmax=0.5, _ymin=0, _ymax=1, _nnodes=50,
                             _side_names=sidenames), triangle, 1, gmsh);
    Domain omega=mesh2d.domain ("Omega");
    Domain gamma=mesh2d.domain ("Gamma");
    //define space and unknown
    Space V(omega, interpolation (_Nedelec, _firstFamily, 1, Hrot), "V", true);
    Unknown e(V, "E");
    TestFunction q(e, "q");
    //define forms, matrices and vectors
    BilinearForm aev=intg (omega, curl(e)|curl(q)) - omegaintg (omega, e|q);
    LinearForm l=intg (omega, f|q);
    EssentialConditions ecs = ( ncross (e) | gamma)=0;
    //compute
    TermMatrix A(aev, ecs, "A");
    TermVector b(l, "B");
    //solve
```
As Nedelec finite elements approximation are not conforming in H1, the solution is not continuous across elements (only tangent component is continuous). So to represent the solution, it is projected on H1 as follows:

\[
\begin{aligned}
&\text{find } E_1 \in L^2(\Omega) \text{ such that} \\
&\int_{\Omega} E_1 w = \int_{\Omega} E w \quad \forall w \in L^2(\Omega).
\end{aligned}
\]

Using an H1 conforming approximation for \( E_1 \) leads to a continuous representation of the projection. We show on the next figure the \( E_x \) component field provided by this example.

The next figure shows the \( L^2 \) errors versus the step \( h \) for 1 and 2 order Nedelec approximation.
3.5 Eigenvalues and eigenvectors of Laplace operator

This example shows how to get eigen functions of Laplace operator equipped with homogeneous Neumann condition:

\[-\Delta u + u = \lambda u \quad \text{in } \Omega\]
\[\partial_n u = 0 \quad \text{on } \partial \Omega\]

and its variational formulation in \( V = H^1(\Omega) \):

\[\left\{ \begin{array}{l}
\text{find } (u, \lambda) \in V \times \mathbb{R} \text{ such that } \\
\int_{\Omega} \nabla u \cdot \nabla v + \int_{\Omega} uv = \lambda \int_{\Omega} uv \quad \forall v \in V.
\end{array} \right.\]

```cpp
#include "xlife++.h"
using namespace xlifepp;

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp

    //mesh square
    Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=20), triangle, 1, gmsh);
    Domain omega = mesh2d.domain("Omega");

    //build P2 interpolation
    Space Vk(omega, P2, "Vk", true);
    Unknown u(Vk, "u");
    TestFunction v(u, "v");

    //build eigen system
    BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v),
    muv = intg(omega, u * v);
    TermMatrix A(auv, "auv"), M(muv, "muv");

    //compute the 10 first smallest in magnitude
```
EigenElements  
\[
eigs = \text{eigenInternSolve}(\mathbf{A}, \mathbf{M}, \_\text{nev}=10, \_\text{mode}=_\text{krylovSchur}, \_\text{which}="\text{SM}");  \\
\text{// internal solver}
\]
thePrintStream << eigs.values;
saveToFile("eigs", eigs.vectors, vtu);
return 0;
}

The 9 first eigen functions are shown in the figure below:

![Eigenfunctions](image)

### 3.6 3D Helmholtz problem using single layer potential integral equation

XL\text{FE}++ is also able to deal with integral equation. This example illustrates the computation of the acoustic scattering by a sphere:

\[
\begin{aligned}
\Delta u + k^2 u &= 0 \quad \text{in } \Omega = \mathbb{R}^3 / B(0, R) \\
u &= -u_{inc} \quad \text{on } S
\end{aligned}
\]

Using single layer potential leads to the integral equation, :

\[
\int_S G(x, y)p(x)dx = -u_{inc} \quad \text{on } S
\]
where $G$ is the Green function of the Helmholtz equation:

$$G(x, y) = \frac{e^{ik|x-y|}}{4\pi|x-y|}.$$  

We deal with the variational formulation in $V = H^1(D)$:

$$\begin{cases} 
\text{find } p \in V \text{ such that } \\
\int_D \int_D p(x) G(x, y) \bar{q}(y) \, dx \, dy = -\int_D u_{\text{inc}} \bar{q} \quad \forall q \in V.
\end{cases}$$

The solution $u$ is get from potential $p$ from the integral representation:

$$u(x) = \int_D G(x, y) p(y) \, dy.$$  

This exemple has been implemented in XLiFE++ using a $P^0$ Lagrange interpolation:

```cpp
#include "xlife++.h"
using namespace xlifepp;

// incident plane wave
Complex uinc(const Point& p, Parameters& pa = defaultParameters)
{
    Real kx=pa("kx"), ky=pa("ky"), kz=pa("kz"),
    Real kp=kx*p(1)+ky*p(2);
    return exp(i*kp);
}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp
    numberOfThreads(2);
    // define parameters and functions
    Parameters pars;
    pars << Parameter(1., "k"); // wave number k
    pars << Parameter(1., "kx") << Parameter(0., "ky") << Parameter(0., "kz"); // kx, ky, kz
    pars << Parameter(1., "radius"); // disk radius
    Kernel G = Helmholtz2dKernel(pars); // load Helmholtz2D kernel
    Function finc(uinc, pars); // define right hand side function
    Function scatSol(scatteredFieldDiskDirichlet, pars); // exact solution

    // meshing the unit disk
    Number npa=16; // nb of points by diameter of disk
    Disk sp(_center=Point(0.,0.), _radius=1, _nnodes=npa, _domain_name="disk");
    Mesh mS(sp, _segment, 1, gmsh);
    Domain disk = mS.domain("disk");

    // Lagrange P0 space and unknown
    Space V1(disk, P1, "V1", false);
    Unknown u1(V1,"u1");
    TestFunction v1(u1, "v1");
    DuffyIM dufIM(5,5,4,3,1,2); // Duffy integration method
    BilinearForm b10=intg(disk,disk, u1*G*v1,dufIM);
    LinearForm f0v = -intg(disk, finc*v1);
}  
```
// compute matrix and right hand side and solve system
TermMatrix A0(blf0, _denseDualStorage, "A0");
TermVector B0(fv0, "B0");
TermVector U0 = directSolve(A0, B0);

// integral representation on x plane (far from disk), using P1 nodes
Number npp=20, npc=8*npp/10;
Real xm=4., eps=0.0001;
Point C1(0.,-xm) , C2(0.,xm) , C3(0.,-xm);
Square sqx(_center=Point(0.,0.),_length=4.,_nnodes=npp,_domain_name="Omega");
Disk dx(_center=Point(0.,0.),_radius=1.25,_nnodes=npc);
Mesh mx0(sqx-dx, triangle, 1,gmsh);
Domain planx0 = mx0.domain("Omega");
Space Wx(planx0,P1,"Wx",false);
Unknown wx(Wx,"wx");
TermVector U0x0=integralRepresentation(wx,planx0,intg(disk,G*u1),U0);
TermMatrix Mx0(intg(planx0,wx*wx), "Mx0");

// compare to exact solution
TermVector solx0(wx, planx0, scatSol);
TermVector ec0x0=U0x0-solx0;
thePrintStream<<"L2 error on x=0 plane : " << sqrt(abs((Mx0*ec0x0) | ec0x0)) << eol;

// export solution to file
saveToFile("U0", U0, vtk);
saveToFile("U0x0", U0x0, vtk);
return 0;
}

Viewing results using PARAVIEW:
3.7 2D Helmholtz problem coupling FEM and integral representation

We want to solve the acoustic diffraction of a plane wave on the disk of radius 1, with the boundary Γ:

\[
\begin{align*}
\Delta u + k^2 u &= 0 \quad \text{in } \mathbb{R}^2 / D \\
\partial_n u &= g \quad \text{on } \Gamma \quad (n \text{ the outward normal})
\end{align*}
\]

where \( g = \partial_n (e^{ikx}) \).

Let \( \Omega \) be a domain that strictly surrounding the disk \( D \) and \( \Sigma \) its boundary. We have to point out that in this case, we use normals going outside the domain of computation \( \Omega \) but then the normal on the obstacle (defined on \( \Gamma \)) is going inside the obstacle, that is opposite to usual case (see Figure 3.1). Then, because of the normal inverted, the solution \( u \) may be represented by the integral representation formula (\( G \) is the Green function related to the 2D Helmholtz equation in free space):

\[
\forall x \in \Sigma, \quad u(x) = -\int_{\Gamma} \partial_{n_y} G(x, y) u(y) \, dy + \int_{\Gamma} G(x, y) \partial_{n_y} u(y) \, dy \quad (3.1)
\]

say, because the boundary condition:

\[
u(x) = -\int_{\Gamma} \partial_{n_y} G(x, y) u(y) \, dy + \int_{\Gamma} G(x, y) g(y) \, dy.
\]

\( n_y \) is the outward normal (to \( \Omega \) not the obstacle) on \( \Gamma \) and \( n_x \) will denote the outward normal on \( \Sigma \). Now matching values and normal derivative on \( \Sigma \), we introduce the boundary condition:

\[
(\partial_{n_x} + \lambda)u(x) = - (\partial_{n_x} + \lambda) \int_{\Gamma} \partial_{n_y} G(x, y) u(y) \, dy + (\partial_{n_x} + \lambda) \int_{\Gamma} G(x, y) g(y) \, dy
\]

that reads, because \( G \) is not singular on \( \Gamma \times \Sigma \):

\[
(\partial_{n_x} + \lambda)u(x) = - \int_{\Gamma} \partial_{n_x} \partial_{n_y} G(x, y) u(y) \, dy - \lambda \int_{\Gamma} \partial_{n_y} G(x, y) u(y) \, dy + \int_{\Gamma} \partial_{n_x} G(x, y) g(y) \, dy + \lambda \int_{\Gamma} G(x, y) g(y) \, dy = \mathcal{R}_\lambda(u)(x)
\]

Using this exact boundary condition, if \( \text{Im}(\lambda) \neq 0 \) the initial problem is equivalent to:

\[
\begin{align*}
\Delta u + k^2 u &= 0 \quad \text{in } \Omega \\
\partial_n u &= g \quad \text{on } \Gamma \\
(\partial_{n_x} + \lambda)u &= \mathcal{R}_\lambda(u) \quad \text{on } \Sigma
\end{align*}
\]

Its variational formulation in \( V = H^1(\Omega) \) is:

\[
\begin{align*}
\text{find } u \in V \text{ such that } \forall v \in V \\
\int_{\Omega} \nabla u \cdot \nabla \bar{v} - k^2 \int_{\Omega} u \bar{v} + \lambda \int_{\Sigma} u \bar{v} + \int_{\Sigma} \int_{\Gamma} u(y) \partial_{n_x} \partial_{n_y} G(x, y) \bar{v}(x) + \lambda \int_{\Sigma} \int_{\Gamma} u(y) \partial_{n_x} G(x, y) \bar{v}(x) \\
= \int_{\Gamma} g \bar{v} + \int_{\Sigma} \int_{\Gamma} g(y) \partial_{n_x} G(x, y) \bar{v}(x) + \lambda \int_{\Sigma} \int_{\Gamma} g(y) G(x, y) \bar{v}(x)
\end{align*}
\]

Considering the geometrical configuration: the variational formulation is implemented as follows.
Figure 3.1: Geometrical configuration for the FEM-Integral Representation problem. The normal on $\Gamma$ is going inside the obstacle (to point outside $\Omega$).

```cpp
#include "xlife++.h"
using namespace xlifepp;

Complex data_g(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), k=pa(“k”);
    Vector<Complex> g(2,0.);
    g(1) = i_*k*exp(i_*k*x);
    return dot(g,P/norm2(P)); // dr \{e^{i k x}\}
}

Complex u_inc(const Point& P, Parameters& pa = defaultParameters)
{
    Real x=P(1), k=pa(“k”);
    return exp(i_*k*x);
}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp
    // parameters
    Number nh = 10; // number of elements on Gamma
    Real h=2*pi_/nh; // size of mesh
    Real re=1.2*h; // exterior radius
    Number ne=Number(2*pi_*re/h); // number of elements on Sigma
    Real l = 4*re; // length of exterior square
    Number nr=Number(4*l/h); // number of elements on exterior square
    Real k= 4, k2=k*k; // wavenumber
    Parameters pars;
    pars << Parameter(k,”k”) << Parameter(k2,”k2”);
    Kernel H=Helmholtz2dKernel(pars);
    Function g(data_g,pars);
    Function ui(u_inc,pars);
    //Mesh and domains definition
    Disk d1(_center=Point(0,0.),_radius=1.,_nnodes=nh,
            _side_names=Strings(4,”Gamma”));
    Disk d2(_center=Point(0,0.),_radius=re,_nnodes=ne,_domain_name=”Omega”,
            _side_names=Strings(4,”Sigma”));
```
In the beginning, some geometric parameters used to design crown surrounded by a square, are given. Next the mesh is generated using gmsh mode and the geometrical domains are get from the mesh. The normal orientations are chosen in order to have outwards normals to the crown omega.

Then a P2 Lagrange space over the elements of the crown omega is constructed and all bilinear and linear forms involved in variational form are defined. Then the TermMatrix and TermVector are computed and the problem is solved using a direct method (Umfpack if it is installed, LU factorization else), that leads to the solution U in the crown omega.

Finally, using integral representation formula 3.1, the solution is computed in the exterior domain omega_ext. The vectors U and Uext are diffracted fields. To get total field, the incident field has to be added to the diffracted filed. This is the final job that it is done.
The real part of the total field computed is presented on the figure 3.2.

Figure 3.2: 2D Helmholtz diffraction problem using FE-IR method: real part of the total field

3.8 2D Helmholtz problem coupling FEM and BEM

We want to solve the acoustic propagation of a plane wave in a heterogeneous medium. In order to do that, we distinguish a domain $\Omega$ that is heterogeneous, its boundary $\Gamma$ and the exterior domain $\Omega_{ext}$ that is homogeneous (see Figure 3.3).

Figure 3.3: Domains for computation: $\Omega$ the heterogeneous medium, $\Omega_{ext}$ the homogeneous exterior domain and $\Gamma = \partial \Omega$. 
We solve:
\[ \Delta u(x) + k^2 \eta^2(x) u(x) = 0 \quad \text{in } \mathbb{R}^2 \]
\[ u(x) = -u_i(x) \quad \text{on } \Gamma \]

with \( \eta(x) = 1 \) in \( \Omega_{\text{ext}} \), and \( \eta(x) \) that can vary in \( \Omega \), and finally with \( u_i = e^{ikx} \).

We will use: \( \Omega = [-0.5, 0.5]^2 \) and
\[ \eta(x) = \begin{cases} 
\exp(-x_1^2 - 0.25) * (x_2^2 - 0.25)/(2. * 0.05), & \text{when } \max(x_1, x_2) < 0.5. \\
1 & \text{otherwise.} 
\end{cases} \]

Index \( \eta \) is represented in \( \Omega \cup \Omega_{\text{ext}} \) in Figure 3.4.

![Figure 3.4: \( \eta(x) \) in \( \Omega \cup \Omega_{\text{ext}} \).](image)

We decompose the problem in a coupled system of two equations:

- in the FEM part, the solution solves the following equation:
  \[ \Delta u + k^2 \eta^2 u = 0 \]
  which gives the variational formulation: Find \( u \in H^1(\Omega) \) such that:
  \[ \int_{\Omega} \nabla u(x) \cdot \nabla \bar{v}(x) dx - k^2 \int_{\Omega} \eta^2(x) u(x) \bar{v}(x) dx - \int_{\Gamma} \lambda(x) \bar{v}(x) dx = 0, \forall v \in H^1(\Omega), \quad (3.2) \]
  with \( \lambda = \frac{\partial u}{\partial n} \) is the normal trace of \( u \) on \( \Gamma \).

- in the BEM part, we solve:
  \[ \Delta u + k^2 u = 0 \quad \text{in } \Omega_{\text{ext}} \]
  \[ u = -u_i \quad \text{on } \Gamma. \]
  \[ (3.3) \]
The scattered field verifies:

\[ u_s(x) = -S_\Gamma \lambda(x) + K_\Gamma u(x), \quad x \in \Omega_{\text{ext}}, \tag{3.4} \]

with \( u \) the total field solution of the equation and \( \lambda \) the normal trace of \( u \) on \( \Gamma \). \( S_\Gamma \) and \( K_\Gamma \) are the single and double layer boundary potentials:

\[ S_\Gamma \phi(x) = \int_\Gamma G(x, y)\phi(y)dy, \]
\[ K_\Gamma \phi(x) = \int_\Gamma \frac{\partial G(x, y)}{\partial n_y}\phi(y)dy, \]

and \( G(x, y) = \frac{e^{ik\|x-y\|}}{4\pi\|x-y\|} \).

Since \( u_s = u - u_i \), and taking the limit when \( x \) goes to \( \Gamma \), we obtain the integral equation:

\[ \left( I_2 - K_\Gamma \right) u(x) + S_\Gamma \lambda(x) = u_i(x), \quad x \in \Gamma. \tag{3.5} \]

The resulting variational formulation for the BEM part is then: Find \( u \in H^{1/2}(\Gamma) \) and \( \lambda \in H^{-1/2}(\Gamma) \) such that

\[
\int_\Gamma u(x)\bar{\tau}(x)dx - \int_{\Gamma \times \Gamma} u(y)\frac{\partial G(x, y)}{\partial n_y}\bar{\tau}(x)dydx + \int_{\Gamma \times \Gamma} \lambda(y)G(x, y)\bar{\tau}(x)dydx \\
= \int_\Gamma u_i(x)\bar{\tau}(x)dx, \quad \forall \tau \in H^{1/2}(\Gamma). \tag{3.6} 
\]

By adding the variational formulations relatives to the two linked problems, we obtain the final variational formulation.

Finally, the solution is obtained directly from \( u \) for the FEM part and we need to compute the integral representation to obtain \( u_s \), the scattered field, and then to add the incident field to obtain the total field for this problem.

The last step is to merge the FEM solution in \( \Omega \) and the BEM solution in \( \Omega_{\text{ext}} \) to obtain a solution on the whole domain \( \Omega \cup \Omega_{\text{ext}} \) to simplify the visualisation. Solution of this FEM-BEM problem is shown in Figure 3.5.

The code of this example follows:

```cpp
#include "xlife++h"
using namespace xlifepp;

// find = eta(x)
Real find(const Point & M, Parameters & pa = defaultParameters) {
    Real res=1.;
    if((std::max(std::abs(M[0]), std::abs(M[1]))<0.5)
        res=std::exp(-(M[0]*M[0]-0.25)*(M[1]*M[1]-0.25))/(2.*0.05));
    return res;
}

Real eta2(const Point & M, Parameters & pa = defaultParameters) {
    Real tmp=find(M);
    return tmp*tmp;
}
```

37
Complex g1(const Point& M, Parameters& pa = defaultParameters)
{
    Real k = real(pa("k"));
    Point d(1.,0.);
    return exp(i_*(k*dot(M,d))); // i is the imaginary unit
}

int main(int argc, char** argv)
{
    Real k=10.;
    init(\_lang=en); // mandatory initialization of xlifepp
    verboseLevel(10);
    Real hsize=(2*pi_/k)/15.;
    std::cout << "Mesh size = " << hsize << eol;
    Square sp(_center=Point(0.,0.),
              _length=1.,_hsteps=hsize,_domain_name="Omega",_side_names="Gamma");
    Mesh ml=Mesh(sp,_triangle,1,gmsh);
    std::cout << "Number of Triangles = " << ml.nbOfElements() << eol;

    Parameters pars;
    pars << Parameter(k,"k");
    Vector<Real> nv(2);
    pars << Parameter(&nv,"n");
    Kernel G=Helmholtz2dKernel(pars);
    Function finc(g1,pars);

    Domain omega = ml.domain("Omega");
    Domain gamma = ml.domain("Gamma");
Space V1(omega,P1,"V1",false);
Space V0(gamma,P1,"V0",false);

std::cout << "Number of Dofs BEM= " << V0.nbDofs() << eol;
std::cout << "Number of Dofs FEM= " << V1.nbDofs() << eol;

// Unknown, TestFunction for FEM
Unknown u1(V1,"u1");
TestFunction v1(u1,"v1");

// Unknown, TestFunction for BEM
Unknown l0(V0,"l0");
TestFunction l0t0(l0,"l0t0");

std::cout << "Generating TermMatrix ... " << eol;
DuffyIM ssim(15,12,10,8,1,2);
BilinearForm blf=intg(omega,grad(u1)|grad(v1))−k∗k∗intg(omega,eta2∗u1∗v1)−
    intg(gamma,10∗v1) + 0.5∗intg(gamma,u1∗l0t0)−
    intg(gamma,gamma,u1∗ndotgrad_y(G)∗l0t0,ssim) +
    intg(gamma,gamma,10∗G∗l0t0,ssim);
elapsedTime();
TermMatrix lhs(blf,"lhs");
elapsedTime("Time to assemble lhs =");

LinearForm lf=intg(gamma,finc∗l0t0);
TermVector rhs(lf);

elapsedTime();
TermVector sol=directSolve(lhs,rhs);
elapsedTime("Time to solve the system =");

// Representation of the solution FEM and BEM
Square Sint(_center=Point(0.,0.),_length=1._hsteps=hsize,_domain_name="S_int");
Square Sext(_center=Point(0.,0.),_length=3._hsteps=1.5*hsize,_domain_name="S_ext");

Mesh mrep(Sext+Sint,_triangle,1.,gmsh);
Domain S_ext=mrep.domain("S_ext"), S_int=mrep.domain("S_int");
Domain S=merge(S_ext,S_int,"S");
Space Vrep(S,P1,"Vrep",false);
Unknown ur(Vrep,"ur");

// Representation of eta
TermVector findex(ur,S,find);
saveToFile("findex",findex,.vtu);

// Representation of the FEM solution (total field)
TermVector Uint=interpolate(ur,S_int,sol(u1));
saveToFile("Uint",Uint,.vtu);

// Representation of the BEM part
TermVector Uext = − integralRepresentation(ur,S_ext,intg(gamma, G∗sol(10),_GaussLegendreRule,25)) +
    integralRepresentation(ur,S_ext,intg(gamma, ndotgrad_y(G)∗sol(u1),_GaussLegendreRule,25));
3.9 Elasticity problem

The elasticity problem illustrates how to use vector unknown in XLiFE++:

\[
\begin{align*}
- \text{div}(\sigma(u)) - \omega^2 u &= f \quad \text{in } \Omega \\
\sigma(u)n &= 0 \quad \text{on } \partial \Omega
\end{align*}
\]

For homogeneous isotropic material:

\[
\sigma(u) = \lambda \text{div}(u) I + 2\mu \varepsilon(u) \varepsilon_{ij}(u) = \partial_i u_j.
\]

The variational formulation in \( V = (H^1(\Omega))^3 \) is:

\[
\begin{cases}
\text{find } u \in V \text{ such that } \\
\lambda \int_{\Omega} \varepsilon(u) : \varepsilon(\bar{v}) + 2\mu \int_{\Omega} \text{div}(u) \text{div}(\bar{v}) - \omega^2 \int_{\Omega} u \bar{v} = \int_{\Omega} \bar{f} \cdot \bar{v} \quad \forall \bar{v} \in V.
\end{cases}
\]

This is implemented as follows

```cpp
#include "xlife++.h"
using namespace xlifepp;

// data function
Vector<Real> f(const Point& P, Parameters& pa = defaultParameters)
{
    return Vector<Real>(2, -1.);
}

int main(int argc, char** argv)
{
    init(_lang=en); // mandatory initialization of xlifepp

    // mesh square
    Square sq(_origin=Point(0,0), _length=1, _nnodes=20);
    Mesh mesh2d(sq, triangle, 1, structured);
    ...
The displacement and its modulus are represented on the next figure.

3.10 Solving wave equation

So far, only the harmonic problems were considered. Time problem may also be solved using XLiFE++. But there is no specific tools dedicated to. Users have to implement the time loop related to the finite difference time scheme they choose.

As an example, consider the wave equation:

\[
\begin{align*}
\frac{\partial^2 u}{\partial t^2} - c^2 \Delta u &= f & \text{in } \Omega \times ]0, T[ \\
\frac{\partial u}{\partial n} &= 0 & \text{in } \partial \Omega \times ]0, T[ \\
u(x, 0) = \frac{\partial u}{\partial t}(x, 0) &= 0 & \text{in } \Omega
\end{align*}
\]
Using classical leap-frog scheme with time discretization $t^n = n\Delta t$, leads to ($u^n$ approximates $u(x,t^n)$):

$$
\begin{align*}
\begin{cases}
u^{n+1} &= 2u^n - u^{n-1} + (c\Delta t)^2 \Delta u^n + (\Delta t)^2 f^n \quad \text{in } \Omega, \forall n \geq 1 \\
\frac{\partial u^n}{\partial n} &= 0 \quad \text{in } \partial \Omega, \forall n > 1 \\
u^n &= u^1 = 0 \quad \text{in } \Omega
\end{cases}
\end{align*}
$$

or, in variational form, $\forall v \in V = H^1(\Omega)$:

$$
\begin{align*}
\begin{cases}
\int_{\Omega} u^{n+1} v &= 2 \int_{\Omega} u^n v - \int_{\Omega} u^{n-1} v - (c\Delta t)^2 \int_{\Omega} \nabla u^n \cdot \nabla v + (\Delta t)^2 \int_{\Omega} f^n v \quad \forall n > 1 \\
u^0 &= u^1 = 0 \quad \text{in } \Omega
\end{cases}
\end{align*}
$$

When approximating space $V$ by a finite dimension space $V_h$ with basis $(w_i)_{i=1,p}$, the variational formulation is reinterpreted in terms of matrices and vectors as follows:

$$
\begin{align*}
\begin{cases}
U^{n+1} &= 2U^n - U^{n-1} - M^{-1} \left((c\Delta t)^2 \mathbb{K} U^n - (\Delta t)^2 F^n\right) \quad \forall n > 1 \\
U^0 &= U^1 = 0 \quad \text{in } \Omega
\end{cases}
\end{align*}
$$

where

$$
M_{ij} = \int_{\Omega} w_i w_j, \quad \mathbb{K}_{ij} = \int_{\Omega} \nabla w_i \cdot \nabla w_j, \quad (F^n)_i = \int_{\Omega} f^n w_i.
$$

The XLiFE++ implementation of this scheme on the unity square when using P1 Lagrange interpolation looks like ($f(x,t) = h(t)g(x)$):

```cpp
#include "xlife++.h"
using namespace xlifepp;

Real g(const Point& P, Parameters& pa = defaultParameters)
{
    Real d=P.distance(Point(0.5,0.5));
    Real R = 0.02; // source radius
    Real amp= 1./(pi*R*R); // source amplitude (constant power)
    if (d<=0.02) return amp; else return 0.;
}

Real h(const Real& t)
{
    Real a=10000, t0=0.04; // gaussian slope and center
    return exp(-a*(t-t0)*(t-t0));
}

int main()
{
    init(_lang=en);
    // create a mesh and domain omega
    Square sq(_origin=Point(0.,0.), _length=1, _nnodes=70);
    Mesh mesh2d(sq, triangle, 1, structured);
    Domain omega=mesh2d.domain("Omega");
    // create interpolation
    Space V(omega, P1, "V", true);
    Unknown u(V, "u");
    TestFunction v(u, "v");
    // define FE terms
    TermMatrix A(intg(omega, grad(u)|grad(v)), "A") , M(intg(omega, u*v), "M");
    TermVector G(intg(omega, g*v), "G");
}
```

42
TermMatrix L; ldltFactorize(M,L);

// leap-frog scheme
Real c=1, dt=0.004, dt2=dt*dt, cdt2=c*c*dt2;
Number nbt=200;
TermVectors U(nbt); // to store solution at t=ndt
TermVector zeros(u, omega, 0.); U(1)=zeros; U(2)=zeros;
Real t=dt;
for (Number n=2; n<nbt; n++, t+=dt)
{
    U(n+1)=2.*U(n)-U(n-1)-factSolve(L,cdt2*(A*U(n))-dt2*h(t)*G);
}
saveToFile("U", U, vtu);
return 0;

Note the very simple syntax taken into account the leap-frog scheme. The pictures below
represents the solution at different instants for a constant source localized in disk with center
(0.5,0.5), radius \( R = 0.02 \) and time excitation that is a Gaussian function. For chosen parameter
\( dt = 0.04 \), the leap-frog scheme is stable (it satisfies the CFL condition) but dispersion effects
obviously appear.
This chapter is devoted to the basics of C++ language required to use XLiFE++. It is addressed to people who do not know C++.

### 4.1 Instruction sequence

All C++ instructions (ending by semicolon) are defined in block delimited by braces:

```cpp
{ 
  instruction ;
  instruction ;
  ... 
}
```

Instruction block may be nested in other one:

```cpp
{ 
  instruction ;
  {
    instruction ;
    instruction ;
  }
  ...
}
```

and are naturally involved in tests, loops, ... and functions.

A function is defined by its name, a list of input argument types, an output argument type and an instruction sequence in an instruction block:

```cpp
argout name_of_function (argin1 , argin2 , ...) 
{ 
  instruction ;
  instruction ;
  ...
  return something ;
}
```

The main program is a particular function returning an error code:

```cpp
int main () 
{ 
  ...
  return 0; //no error
}
```
4.2 Variables

In C++, any variable has to be declared, say defined by specifying its type. The fundamental types are:

- integer number: `int` (Int type in XLIFE++), `unsigned int` (Number type in XLIFE++)
  and `short unsigned int` (Dimen type in XLIFE++)
- real number: `float` for single precision (32bits) or `double` (64bits) for double precision (Real type in XLIFE++)
- boolean: `bool` that takes `true` (1) or `false` (0) as value
- character: `char` containing one of the standard ASCII character

All other types are derived types (pointer, reference) or classes (Complex, String for instance).

All variable names must begin with a letter of the alphabet. Do not begin by underscore (_) because it is used by XLIFE++. After the first initial letter, variable names can also contain letters and numbers. No spaces or special characters, however, are allowed. Upper-case characters are distinct from lower-case characters.

A variable may be declared any where. When they are declared before the beginning of the main, they are available anywhere in the file where they are declared.

⚠️ All variables declared in an instruction block are deleted at the end of the block.

4.3 Basic operations

The C++ provides a lot of operators. The main ones are:

- `=` : assignment
- `+`, `-`, `*`, `/` : usual algebraic operators
- `+=`, `-=` , `*=` , `/=` : operation on left variable
- `++`, `--`: to increment by 1 (`+=1`) and decrement by 1 (`-=1`)
- `==` , `!=` , `<` , `<=` , `>=` , `!` : comparaison operators and negation
- `&`, `|`, `&&` : logical and, or
- `<<`, `>>`: to insert in a stream (read, write)

All these operators may work on object of a class if they have been defined for this class. See documentation of a class to know what operators it supports.

The operators `+=`, `-=`, `*=` , `/=` may be useful when they act on large structure because they, generally, do not modify their representation and avoid copy.
4.4 if, switch, for and while

The syntax of test is the following:

```c
if (predicate)
{
...
}
else if (predicate2)
{
...
}
else
{
...
}
```

else if and else blocks are optional and you can have as many else if blocks as you want. predicate is a boolean or an expression returning a boolean (true or false):

```c
if ((x==3 && y<=2) || (!a>b))
{
...
}
```

For multiple choice, use the switch instruction:

```c
switch (i)
{
    case 0:
    {
        ...
        break;
    }
    case 1:
    {
        ...
        break;
    }
    ...
default:
    {
        ....
    }
}
```

The switch variable has to be of enumeration type (integer or explicit enumeration).

The syntax of the for loop is the following:

```c
for (initialization; end_test; incrementing_sequence)
{
...
}
```

The simplest loop is:
```cpp
for ( int i = 0; i < n; i++)
{
    ...
}
```

An other example with no initializer and two incrementing values:

```cpp
int i = 1, j = 10;
for ( ; i < n && j > 0; i++, j --)
{
    ...
}
```

A for loop may be replaced by a while loop:

```cpp
int i = 0;
while ( i < n)
{
    ...
    i ++;
}
```

### 4.5 In/out operations

The simplest way to print something on screen is to used the predefined output stream `cout` with operator `<<`:

```cpp
Real x = 2.25;
Number i = 3;
String msg = " Xlfe++ : ";
cout << msg << " x=" << x << " i=" << i << eol;
```

`eol` is the `Xlfe++` end of line. You can insert in output stream any object of a class as the operator `<<` is defined for this class. Almost all `Xlfe++` classes offer this feature.

To read information from keyboard, you have to use the predefined input stream `cin` with operator `>>`:

```cpp
Real x;
Number i = 3;
cin >> i >> x;
```

The program waits for an input of a real value, then for an an input of integer value.

To print on a file, the method is the same except that you have to define an output stream on a file:

```cpp
ofstream out ;
out.open ("myfile");
Real x = 2.25;
Number i = 3;
String msg = " Xlfe++ : ";
out << msg << " x=" << x << " i=" << i << eol;
out.close();
```
To read from a file:

```cpp
#include <iostream>
#include <fstream>
...
using namespace std;
```

```cpp
Real x;
Number i=3;
in >> i >> x;
in . close();
```

The file has to be compliant with data to read. The default separators are white space and carriage return (end of line).

To read and write on file in a same time, use `fstream`.

### 4.6 Using standard functions

The STL library provides some fundamental functions such as `abs`, `sqrt`, `power`, `exp`, `sin`, .... To use it, you have to include the `cmath` header file:

```cpp
#include <cmath>
using namespace std;
```

```cpp
double pi=4*atan(1);
double y=sqrt(x);
```

### 4.7 Use of classes

The C++ allows to define new types of variable embedding complex structure: say class. A class may handle some data (member) and functions (member functions). A variable of a class is called an object.

In XLiFE++, you will have only to use it, not to define new one. The main questions are: how to create an object of a class, how to access to its members and how to apply operations on it. To illustrate concepts, we will use the very simple Complex class:

```cpp
class Complex {
    public:
        float x, y;
```
Classes have special functions, called constructors, to create objects. They have the name of the class and are invoked at the declaration of the object:

```cpp
int main()
{
    Complex z1;          // default constructor
    Complex z2(1,0);     // explicit constructor
    Complex z4(z2);      // copy constructor
    Complex z5=z3;       // use copy constructor

    Complex z4=Complex(0,1);
}
```

Copy constructor and operator = are always defined. When operator = is used in a declaration, the copy constructor is invoked. The last instruction uses the explicit constructor and the copy constructor. In practice, compilers are optimized to avoid copies.

To address a member or a member function, you have to use the operator point (.) :

```cpp
int main()
{
    Complex z(0,1);
    float r=z.x;
    float i=z.y;
    float a=z.abs();
}
```

and to use operators, use it as usual:

```cpp
int main()
{
    Complex z1(0,1), z2(1,0);
    z1+=z2;
}
```

Most of XLiFE++ user’s classes have been developed to be close to natural usage.

### 4.8 Understanding memory usage

In scientific computing, the computer memory is often asked intensively. So its usage has to be well managed:

- avoid copy of large structures (mainly TermMatrix)
- clear large objects (generally it exists a clear function). You do not have to delete objects, they are automatically destroyed at the end of the blocks where they have been declared!
- when it is possible, use +=, -=, *=, /= operators instead of +, -, *, / operators which induce some copies
• in large linear combination of TermMatrix, do not use partial combinations which also
induce unnecessary copies and more computation time

4.9 Main user’s classes of XLiFE++

For sake of simplicity, the developers choose to limit the number of user’s classes and to restrict
the use of template paradigm. Up to now the only template objects are Vector and Matrix to
deal with real or complex vectors/matrices. The name of every XLiFE++ class begins with a
capital letter.

XLiFE++ provides some utility classes (see Appendix B for details):

String to deal with character string

Strings to deal with vector of character strings

Number to deal with unsigned (positive) integers

Numbers to deal with vector of unsigned (positive) integers

Real to deal with floats, whatever the precision.

Reals to deal with vector of floats, whatever the precision.

Complex to deal with complexes

Vector<T> to deal with numerical vectors (T is a real/complex scalar or real/complex Vector)

Matrix<T> to deal with numerical matrices (T is a real/complex scalar or real/complex Matrix)

RealVector, RealVectors, RealMatrix, RealMatrices are aliases of previous real vectors and
matrices

Complexes, ComplexVector, ComplexVectors, ComplexMatrix, ComplexMatrices are aliases
of previous complex vectors and matrices

Point to deal with Point in 1D, 2D, 3D

Parameter structure to deal with named parameter of type Real, Complex, Integer, String

Parameters a list of parameters

Function generalized function handling a c++ function and a list of parameters

Kernel generalized kernel managing a Function (the kernel) and some additional data
(singularity type, singularity order, ...)

TensorKernel a special form of kernel useful to DtN map

XLiFE++ also provides the main user’s modelling classes:

Geometry to describe geometric objects (segment, rectangle, ellipse, ball, cylinder, ...). Each
geometry has its own modelling class (Segment, Rectangle, Ellipse, Ball, Cylinder, ...)

50
Mesh  mesh structure containing nodes, geometric elements, ...

Domain alias of geometric domains describing part of the mesh, in particular boundaries

Space  class handles discrete spaces (FE space or spectral space)

Unknown abstract element of space (TestFunction is an alias of Unknown)

LinearForm symbolic representation of a linear form

BiLinearForm symbolic representation of a bilinear form

EssentialCondition symbolic representation of an essential condition on a geometric domain

EssentialConditions list of essential conditions

TermVector algebraic representation of a linear form or element of space as vector

TermVectors list of TermVector’s

TermMatrix algebraic representation of a bilinear form

EigenElements list of eigen pairs (eigen value, eigen vector)

These classes will be shown in more details in the following sections.
5 Mesh definition

The geometry library collects all the general classes and functionalities about geometries, meshes, geometrical domains and geometrical elements.

In order to handle a finite element mesh, XLiFE++ provides the class Mesh. Thus, the user must first of all create an object of this type, which can be done mainly in two ways:

- reading a file containing the description of the geometry,
- or using XLiFE++ internal (simple) meshing tools.

The internal tools are designed to provide the user with a mesh in a straightforward way. They only deal with simple geometries. Complicated geometries need to use a specific software that stores the geometrical description of the mesh into a file.

In this section we will see:

1. How to define geometries, canonical ones and more complicated ones: section 5.1
2. How to apply transformations on geometries (rotations, translations, extrusions, ...): section 5.2
3. How to define a mesh from a geometry: section 5.3
4. How to transform a mesh: section 5.5
5. How to define a mesh from a file: section 5.4
6. How to use geometrical domains: section 5.6

5.1 Defining geometries

To define a geometry object, you will use a constructor:

```
Pyramid pyr(key1 = val1, key2 = val2, ...);
```

There is a lot of available parameters (or keys) for each geometry object. You can give them in any order. Some keys are parts of a group of keys. When you use a group of keys, you have to set every key of the group. For instance, in the following example, to define a triangle, you have to give the three vertices of the triangles with the keys _v1, _v2, _v3. You must not forget one of them.

```
Triangle tri(_v1 = Point(0,0), _v2 = Point(1,0), _v3 = Point(0,1), ...);
```

There are 3 kind of parameters (plus 1 single parameter):

- First, you have parameters dedicated to geometry definition. This part is different for each geometry and will be explained in following subsections.
• Secondly you have 2 parameters dedicated to mesh parameters such as the number of nodes on each edge of the geometry (always greater than 2) or the local mesh step on each vertex of the geometry (fitted to the gmsh mesh generator). For this 2 kinds of arguments, you will have the choice to give a value per edge (or vertex), or a smaller number of values according to properties of symmetry of the geometry, or a common value for each edge (or vertex). To set the number of nodes on each edge, you will have to use the \texttt{nnodes} key. To set the local mesh step on each vertex, you will have to use the \texttt{hsteps} key. These parameters are optional and only one of them is to be used.

\begin{verbatim}
Triangle  tri(_v1 = Point(0.,0.), _v2 = Point(1.,0.), _v3 = Point(0.,1.), _nnodes = Numbers(11,15,11));
\end{verbatim}

What is the difference between \texttt{nnodes} and \texttt{hsteps}? It is as in the GMSH documentation.

\texttt{nnodes} When you use this parameter, you set the number of nodes of a regular mesh on an edge. As a result, the mesh step is constant on the edge. Using this parameter, you can refine a mesh near an edge.

\texttt{hsteps} When you use this parameter, you set the value of the mesh step near a vertex. If the mesh step is the same for both vertices of the edge, then this is a regular mesh (equivalent to define the number of nodes in this case). If the mesh step is different on vertices of an edge, it varies progressively to fit the expected value on vertices. Using this parameter, you can refine a mesh near a vertex.

• Thirdly you have parameters dedicated to definition of geometrical domains. These keys are all optional:

\texttt{domain name} is used to set the name of the main domain of the geometry. The main domain depends on the type of mesh (if you mesh a cube with triangles, the main domain will be the whole border, whereas with tetrahedra, it is the cube itself).

\texttt{side names} is used to set the names of every side domain. You can give a vector of strings (\texttt{Strings} object) or a single \texttt{String} if it is the same name for every side domain.

Default values are empty strings. When a domain has an empty name, it is not built. For some geometries (cylinders and cones), there is an additional parameter.

• At last you have \texttt{type}, for geometries fitted to the subdivision mesh generator (See subsection 5.3.2 for details).

Let’s summarize information about these keys:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{domain name}</td>
<td>\texttt{String} or const char*</td>
<td>\texttt{domain name}=&quot;Omega&quot;</td>
</tr>
<tr>
<td>\texttt{hsteps}</td>
<td>single real value, std::vector of real values or \texttt{Reals}</td>
<td>\texttt{hsteps}=0.5, \texttt{hsteps}=\texttt{Reals}(0.5, 0.2)</td>
</tr>
<tr>
<td>\texttt{nnodes}</td>
<td>single (unsigned) integer value, std::vector of integer values, \texttt{Number} or \texttt{Numbers}</td>
<td>\texttt{nnodes}=11, \texttt{hsteps}=\texttt{Numbers}(11, 22)</td>
</tr>
<tr>
<td>\texttt{side names}</td>
<td>single string, std::vector of string, \texttt{String} or \texttt{Strings}</td>
<td>\texttt{side names}=&quot;Gamma&quot;, \texttt{side names}=\texttt{Strings}(&quot;Gamma1&quot;, &quot;Gamma2&quot;, &quot;Gamma2&quot;)</td>
</tr>
<tr>
<td>\texttt{type}</td>
<td>single (unsigned) integer value, or \texttt{Number}</td>
<td>\texttt{type}=1</td>
</tr>
</tbody>
</table>
In the following, we will see how to define each canonical geometries, before showing how to define more complicated ones.

### 5.1.1 Segments

A segment is just a straight line between 2 points.

![Segment diagram](image)

The general case is to give points through parameters \( v_1 \) and \( v_2 \), but when 1D, you can give directly the real coordinate.

```plaintext
Segment s1 (v1=Point(0.,0.,0.), v2=Point(0.,1.,-1.), nnodes=11, _domain_name="Omega");
Segment s2 (v1=Point(0.,0.), v2=Point(0.,1.), hsteps=0.1, _domain_name="Omega");
Segment s3 (v1=Point(0.), v2=Point(1.), hsteps=Reals(0.1,0.2), _domain_name="Omega");
Segment s4 (v1=0., v2=1., nnodes=11, _domain_name="Omega");
```

In previous examples \( s_3 \) and \( s_4 \) are identical. A better comprehensive way for \( s_4 \) is to use parameters \( \text{xmin} \) and \( \text{xmax} \) instead of \( v_1 \) and \( v_2 \).

```plaintext
Segment s4 (_xmin=0., _xmax=1., _nnodes=11, _domain_name="Omega");
```

In previous examples, you can notice that \( \text{nnodes} \) take only a single integer value and \( \text{hsteps} \) can take one real value or a vector of 2 real values (\text{Reals} object).

One of the combination \( \text{xmin} \) and \( \text{xmax} \) or \( v_1 \) and \( v_2 \) is needed.

After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

**Examples.**

```plaintext
// segment \([-2,5]\) with 50 points when meshing
Segment s1 (_xmin=-2, _xmax=5, _nnodes=50);

// segment linking A(1,2,3) and B(-2,5,0) with 20 points when meshing and domain is "Omega1"
Point a(1.,2.,3.);
Point b(-2.,5.,0.)
Segment s2 (_v1=a, _v2=b, _nnodes=20, _domain_name="Omega1");

// segment \([0,1]\) with 20 points when meshing and side domains are "Gamma1" and "Gamma2"
Segment s3 (_xmin=0., _xmax=1., _nnodes=20, _side_names=Strings("Gamma1","Gamma2"));

// segment \([0,1]\) with 10 points when meshing and domain is "Omega" and side domains are "Gamma1" and "Gamma2"
Segment s4 (_xmin=0., _xmax=1., _nnodes=10, _domain_name="Omega", _side_names=Strings("Gamma1","Gamma2"));

// segment \([0,1]\) with 10 points when meshing and domain is "Omega" and side domain is "Gamma"
Segment s4 (_xmin=0., _xmax=1., _nnodes=10, _domain_name="Omega", _side_names="Gamma");
```

You can reverse the orientation of a segment by using one of the following:
When defining composite or loop geometries, you shall not use the `reverse` method, but only the \( \sim \) operator

Let’s summarize information about geometrical keys for segments:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>single integer or real value, or <code>Point</code></td>
<td>( v_1=Point(0.), ) ( v_1=Point(0.,0.), ) ( v_1=Point(0.,0.,0.), ) ( v_1=0. )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>single integer or real value, or <code>Point</code></td>
<td>( v_2=Point(0.), ) ( v_2=Point(0.,0.), ) ( v_2=Point(0.,0.,0.), ) ( v_2=0. )</td>
</tr>
<tr>
<td>( x_{\text{min}} )</td>
<td>single integer or real value</td>
<td>( x_{\text{min}}=1, ) ( x_{\text{min}}=-2.5 )</td>
</tr>
<tr>
<td>( x_{\text{max}} )</td>
<td>single integer or real value</td>
<td>( x_{\text{max}}=1, ) ( x_{\text{max}}=-2.5 )</td>
</tr>
</tbody>
</table>

5.1.2 Elliptic and circular arcs

**Elliptic arcs**

To define an elliptic arc, you need 4 points: the center of the ellipse, the apogee of the ellipse and the bounds of the arc.

There is a parameter for each of them: \( \text{center}, \text{apogee}, \text{v}_1 \) and \( \text{v}_2 \). These parameters take 2D or 3D points. When omitted, the apogee point is defined as the first bound of the arc. An elliptic arc must be smaller than a half-ellipse, to be defined correctly.

\( \text{nnodes} \) take only one single value and \( \text{hsteps} \) can take one real value or a vector of 2 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.

```c
Point c(0.,0.,0.);
Point a(2.,0.,0.);
```
Point p1(0.,1.,1.);
Point p2(-1.,2.,0.)

// whole side domain will be "Gamma"
EllArc e1(_center=c, _apogee=a, _v1=p1, _v2=p2, _nnodes=20,
    _domain_name="Omega", _side_names="Gamma");

You can reverse the orientation of an elliptic arc by using one of the following:

EllArc e1(_center=c, _apogee=a, _v1=p1, _v2=p2, _nnodes=20,
    _domain_name="Omega", _side_names="Gamma"); e1.reverse(); // e1 is modified
EllArc e2=˜e1; // e1 is not modified

When defining composite or loop geometries, you shall not use the reverse method, but only the ~ operator.

Let’s summarize information about geometrical keys on elliptic arcs:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_apogee</td>
<td>Point</td>
<td>_apogee=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_apogee=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Circular arcs

To define a circular arc, you need 3 points: the center of the circle and the bounds of the arc.

There is a parameter for each of them: _center, _v1 and _v2. These parameters take 2D or 3D points. A circular arc must be smaller than a half-circle, to be defined correctly.

_nnodes take only one single value and _hsteps can take one real value or a vector of 2 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.
You can reverse the orientation of a circular arc by using one of the following:

```plaintext
CircArc c1(_center=Point(0.,0.) , _v1=Point(1.0.,0.) , _v2=Point(0.,1.) , _nnodes=30, _domain_name="Omega");
```

```plaintext
CircArc c1(_center=Point(0.,0.) , _v1=Point(1.0.,0.) , _v2=Point(0.,1.) , _nnodes=30, _domain_name="Omega")
```

```plaintext
// c1 is modified
```n
c1.reverse();

```plaintext
// c1 is not modified
CircArc c2=c1;
```

When defining composite or loop geometries, you shall not use the `reverse` method, but only the ~ operator.

Let’s summarize information about geometrical keys on circular arcs:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

### 5.1.3 Polygons and polygon-likes

**Polygons**

A polygon is defined by its ordered list of vertices.

![Polygon diagram](image)

To do so, you will use the parameter `vertices`.

- `_nnodes` can take one single value or a vector of values (`Numbers` object) and `_hsteps` can take one real value or a vector of real values (`Reals` object). The vector sizes are the number of vertices (same as the number of edges for a polygon). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.
Let's summarize information about geometrical keys on polygons:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_vertices</td>
<td>vector of Point</td>
<td>(std::vector&lt;Point&gt; vp; ...)</td>
</tr>
</tbody>
</table>

**Triangles**

To define a triangle, you give the 3 vertices.

There is a parameter for each of them: _v1, _v2 and _v3. These parameters take 2D or 3D points. _nnodes can take one single value or a vector of 3 values (Numbers object) and _hsteps can take one real value or a vector of 3 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.

```cpp
Point a(-1., 2., 0.);
Point b(1., -4., 2.);
Point c(2., 3., 1.);
Triangle t1(_v1=a, _v2=b, _v3=c, _nnodes=Numbers(10, 15, 20),
            _domain_name="Omega", _side_names="Gamma");
```

Let’s summarize information about geometrical keys on triangles:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0., 0., 0.),</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0., 0., 0.)</td>
</tr>
<tr>
<td>_v3</td>
<td>Point</td>
<td>_v3=Point(0., 0., 0.)</td>
</tr>
</tbody>
</table>
Quadrangles

To define a quadrangle, you give the 4 vertices.

There is a parameter for each of them: \texttt{_v1}, \texttt{_v2}, \texttt{_v3} and \texttt{_v4}. These parameters take 2D or 3D points. \texttt{_nnodes} can take one single value or a vector of 4 values (\texttt{Numbers} object) and \texttt{_hsteps} can take one real value or a vector of 4 real values (\texttt{Reals} object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.

\begin{verbatim}
Quadrangle q1(_v1=Point(0.,0.), _v2=Point(2.,0.), _v3=Point(2.,1.), _v4=Point(0.,1.), _nnodes=Numbers(20, 10, 20, 10), _domain_name="Omega", _side_names="Gamma");
\end{verbatim}

Let’s summarize information about geometrical keys on quadrangles:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.), _v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.), _v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v3</td>
<td>Point</td>
<td>_v3=Point(0.,0.), _v3=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.), _v4=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Parallelograms

To define a parallelogram, you give 3 vertices. If you refer to the following figure, \( p_3 \) is unnecessary.
There is a parameter for each of them: \(_{v1}\), \(_{v2}\), and \(_{v4}\). These parameters take 2D or 3D points.

\(_{nnodes}\) can take one single value or a vector of 2 or 4 values (\(\text{Numbers}\) object) and \(_{hsteps}\) can take one real value or a vector of 4 real values (\(\text{Reals}\) object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

\begin{Verbatim}
Parallelogram p1(_{v1}=Point(0.,0.), _{v2}=Point(2.,0.), _{v4}=Point(0.,0.),
_{nnodes}=Numbers(20, 10, 20, 10), _domain_name="Omega", _side_names="Gamma")
Parallelogram p2(_{v1}=Point(0.,0.), _{v2}=Point(2.,0.), _{v4}=Point(0.,1.),
_{nnodes}=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma")
\end{Verbatim}

Both parallelograms of previous examples are identical. This explains the ability to give 2 values for \(_{nnodes}\).

Let’s summarize information about geometrical keys on parallelograms:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
</table>
| \(_{v1}\) | Point           | \(_{v1}=\text{Point}(0.,0.), \(_{v1}=\text{Point}(0.,0.,0.)
| \(_{v2}\) | Point           | \(_{v2}=\text{Point}(0.,0.), \(_{v2}=\text{Point}(0.,0.,0.)
| \(_{v4}\) | Point           | \(_{v4}=\text{Point}(0.,0.), \(_{v4}=\text{Point}(0.,0.,0.)

Rectangles

To define a rectangle, you give 3 vertices, as for parallelograms.

There is a parameter for each of them: \(_{v1}\), \(_{v2}\), and \(_{v4}\), as for Parallelogram. These parameters take 2D or 3D points.

For rectangles in plane \(z=0\), where sides are parallel to x-axis and y-axis, you can define the rectangle by its center (\(c\) in the figure) and its lengths or \(p_1\) (recalled origin in this case) and its lengths. You may use \(_{center}\), \(_{xlength}\) and \(_{ylength}\) or \(_{origin}\), \(_{xlength}\) and \(_{ylength}\) to do so. \(_{origin}\) and \(_{center}\) take 2D or 3D points. \(_{xlength}\) and \(_{ylength}\) take one single positive value.

There is another possibility : defining the rectangle by its bounds : parameters \(_{xmin}\), \(_{xmax}\), \(_{ymin}\) and \(_{ymax}\). These parameters take one single value.

\(_{nnodes}\) can take one single value or a vector of 2 or 4 values (\(\text{Numbers}\) object) and \(_{hsteps}\) can take one real value or a vector of 4 real values (\(\text{Reals}\) object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.
Rectangle r1(_v1=Point(0.,0.), _v2=Point(2.,0.), _v4=Point(0.,1.), _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));
Rectangle r2(_center=Point(1.,0.5), _xlength=2., _ylength=1., _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));
Rectangle r3(_origin=Point(0.,0.), _xlength=2., _ylength=1., _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));
Rectangle r3(_xmin=0., _xmax=2., _ymin=0., _ymax=1., _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));

This is 4 definitions of the same Rectangle object.
Let’s summarize information about geometrical keys on rectangles:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.), _center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_origin</td>
<td>Point</td>
<td>_origin=Point(0.,0.), _origin=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.), _v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.), _v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.), _v4=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_xlength</td>
<td>single unsigned integer or real positive value</td>
<td>_xlength=1, _xlength=2.5</td>
</tr>
<tr>
<td>_ylength</td>
<td>single unsigned integer or real positive value</td>
<td>_ylength=1, _ylength=2.5</td>
</tr>
<tr>
<td>_xmin</td>
<td>single integer or real value</td>
<td>_xmin=1, _xmin=-2.5</td>
</tr>
<tr>
<td>_xmax</td>
<td>single integer or real value</td>
<td>_xmax=1, _xmax=-2.5</td>
</tr>
<tr>
<td>_ymin</td>
<td>single integer or real value</td>
<td>_ymin=1, _ymin=-2.5</td>
</tr>
<tr>
<td>_ymax</td>
<td>single integer or real value</td>
<td>_ymax=1, _ymax=-2.5</td>
</tr>
</tbody>
</table>

Squares
To define a square, you give 3 vertices, as for rectangles and parallelograms.
There is a parameter for each of them: \( v_1 \), \( v_2 \), and \( v_4 \), as for Parallelogram and Rectangle. These parameters take 2D or 3D points.

For squares in plane \( z=0 \), where sides are parallel to x-axis and y-axis, you can define the square by its center \((c \text{ in the figure})\) and its length or \( p_1 \) (recalled origin in this case) and its length. You may use \( \_\text{center} \) and \( \_\text{length} \) or \( \_\text{origin} \) and \( \_\text{length} \) to do so. \( \_\text{origin} \) and \( \_\text{center} \) take 2D or 3D points. \( \_\text{length} \) takes one single positive value.

\( \_\text{nnodes} \) can take one single value or a vector of 2 or 4 values (\text{Numbers} object) and \( \_\text{hsteps} \) can take one real value or a vector of 4 real values (\text{Reals} object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

**Examples.**

\[
\text{Square } s1(\_v1=\text{Point}(0.,1.), \_v2=\text{Point}(1.,1.), \_v4=\text{Point}(0.,2.), \_\text{nnodes}=\text{Numbers}(20, 10), \_\text{domain_name}="\Omega") ;
\]

\[
\text{Square } s2(\_\text{center}=\text{Point}(0.5,1.5), \_\text{length}=1., \_\text{nnodes}=\text{Numbers}(20, 10), \_\text{domain_name}="\Omega", \_\text{side_names}=\text{Strings}("\Gamma_1", "\Gamma_2", "\Gamma_1", "\Gamma_2") ) ;
\]

\[
\text{Square } s3(\_\text{origin}=\text{Point}(0.,1.), \_\text{length}=1., \_\text{nnodes}=\text{Numbers}(20, 10), \_\text{domain_name}="\Omega", \_\text{side_names}=\text{Strings}("\Gamma_1", "\Gamma_2", "\Gamma_1", "\Gamma_2") ) ;
\]

This is 3 definitions of the same \text{Square} object.

Let’s summarize information about geometrical keys on squares:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_\text{center}</td>
<td>Point</td>
<td>_\text{center}=Point(0.,0.), _\text{center}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_\text{origin}</td>
<td>Point</td>
<td>_\text{origin}=Point(0.,0.), _\text{origin}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_\text{v1}</td>
<td>Point</td>
<td>_\text{v1}=Point(0.,0.), _\text{v1}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_\text{v2}</td>
<td>Point</td>
<td>_\text{v2}=Point(0.,0.), _\text{v2}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_\text{v4}</td>
<td>Point</td>
<td>_\text{v4}=Point(0.,0.), _\text{v4}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_\text{length}</td>
<td>single unsigned integer or real positive value</td>
<td>_\text{length}=1, _\text{length}=2.5</td>
</tr>
</tbody>
</table>

**5.1.4 Ellipses and disks**

**Ellipses**

To define an elliptic surface, you have to precise the plane where it is and the axis parameters. To define the plane, you just have to give the center point (parameter \_\text{center}) and 2 other points, in order to have 3 unaligned points. These points are supposed to be both apogees of the ellipse (parameters \_\text{v1} and \_\text{v2}), namely \( c \), \( p_1 \) and \( p_2 \) in the following figure:
These parameters take 2D or 3D points.
When apogees are along x-axis and y-axis respectively, you can give semi-axes lengths by using _xlength and _ylength.

_nnodes can take one single value or a vector of 4 values (Numbers object), one for each quarter of ellipse. _hsteps can take one real value or a vector of 4 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```plaintext
Ellipse e1(_center=Point(0.,0.), _v1=Point(2.,0.), _v2=Point(0.,1.), _nnodes=Numbers(5, 10, 5, 10), _domain_name="Omega", _side_names=Strings("Gamma5", "Gamma10", "Gamma5", "Gamma10"));
Ellipses e2(_center=Point(0.,0.,0.), _v1=Point(1.,0.,1.), _v2=Point(0.,1.,1.), _nnodes=40, _domain_name="Omega", _side_names="Gamma" ;
Ellipses e3(_center=Point(0.,0.,_xlength=2, _ylength =3.5, _nnodes=40, _domain_name="Omega", _side_names="Gamma" ;
```

Let's summarize information about geometrical keys on ellipses:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.), _center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.), _v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.), _v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_xlength</td>
<td>single unsigned integer or real positive value</td>
<td>_xlength=1, _xlength=2.5</td>
</tr>
<tr>
<td>_ylength</td>
<td>single unsigned integer or real positive value</td>
<td>_ylength=1, _ylength=2.5</td>
</tr>
</tbody>
</table>

Disks

To define an disk, you have to precise the plane where it is and the radius parameters. To define the plane, you just have to give the center point and 2 other points, in order to have 3 unaligned points. These points are supposed to be doing a right angle with the center of the disk (as if they were apogees of an ellipse).
To do so, you will use parameters \texttt{center}, \texttt{v1} and \texttt{v2}, as for an ellipse. These parameters take 2D or 3D points. Furthermore, you can define disk sectors with two additional parameters: \texttt{angle1} and \texttt{angle2}. Values of angles are given in degree and between 0 and 360.

\texttt{nnodes} can take one single value or a vector of 4 values (\texttt{Numbers} object), one for each quarter of ellipse. \texttt{hsteps} can take one real value or a vector of 4 real values (\texttt{Reals} object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

\begin{verbatim}
Disk d1(_center=Point(0.,0.), _v1=Point(1.,0.), _v2=Point(0.,1.),
    _nnodes=Numbers(5., 10., 5., 10.), _domain_name="Omega",
    _side_names=Strings("Gamma5", "Gamma10", "Gamma5", "Gamma10"));
Disk d2(_center=Point(0.,0.,0.), _v1=Point(1.,0.,1.), _v2=Point(0.,1.,1.),
    _nnodes=40., _domain_name="Omega", _side_names="Gamma"));
Disk d3(_center=Point(0.,0.), _radius=2.5., _nnodes=40., _domain_name="Omega",
    _side_names="Gamma");
\end{verbatim}

The \texttt{Disk} object has another name: \texttt{Circle}

Let’s summarize information about geometrical keys on disks:
5.1.5 Polyhedra and polyhedron-likes

Polyhedra

A polyhedron is defined by its faces. The list of faces is a vector of polygons (See subsection 5.1.3 for details).

To do so, you will use the parameter _faces. The only other parameter you may use is _domain_name, to set the name of the polyhedral main domain. Everything else is defined by the faces.

Example.

```cpp
std::vector<Point> v(5), v2(4);
v[0]=Point(0.,0.,0.); v[1]=Point(2.,0.,0.); v[2]=Point(3.,1.,0.);
   v[3]=Point(1.,4.,0.); v[4]=Point(-1.,-2.,0.);
Polygon pg1(_vertices=v, _nnodes="Gamma1");
v[0]=Point(0.,0.,1.); v[1]=Point(2.,0.,1.); v[2]=Point(3.,1.,1.);
   v[3]=Point(1.,4.,1.); v[4]=Point(-1.,-2.,1.);
Polygon pg2(_vertices=v, _nnodes="Gamma2");
v2[0]=Point(0.,0.,0.); v2[1]=Point(2.,0.,0.); v2[2]=Point(2.,0.,1.);
   vs2[3]=Point(0.,0.,1.);
```
Let’s summarize information about geometrical keys on polyhedra:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_faces</td>
<td>vector of Polygon</td>
<td>(std::vector&lt;Polygon&gt; vp; ...)</td>
</tr>
<tr>
<td></td>
<td>_faces=vp</td>
<td></td>
</tr>
</tbody>
</table>

**Tetrahedra**

To define a tetrahedron, you give the 4 vertices.

There is a parameter for each of them: _v1, _v2, _v3 and _v4. These parameters take 3D points. _nnodes can take one single value or a vector of 6 values (Numbers object) and _hsteps can take one real value or a vector of 4 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Example.

```cpp
Point a(1.0,0.0), b(0.1,0.0), c(0.0,0.1), d(0.0,0.0);
Tetrahedron t1(_v1=a, _v2=b, _v3=c, _v4=d, _nnodes=10, _domain_name="Omega", _side_names="Gamma")
```
Let’s summarize information about geometrical keys on tetrahedra:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v3</td>
<td>Point</td>
<td>_v3=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Hexahedra

To define a hexahedron, you just have to give the 8 vertices, defined as in the following figure.

There is a parameter for each of them: _v1, _v2, _v3, _v4, _v5, _v6, _v7 and _v8. These parameters take points or a single value (in this case, it is like a 1D point). _nnodes can take one single value or a vector of 12 values (Numbers object) and _nsteps can take one real value or a vector of 8 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```plaintext
Point a ( 0 . , 0 . , 0 . ) , b ( 4 . , 0 . , 0 . ) , c ( 4 . , 2 . , 0 . ) , d ( 0 . , 2 . , 0 . ) ;
Point aa ( 0 . , 0 . , 1 . ) , bb ( 4 . , 0 . , 1 . ) , cc ( 4 . , 2 . , 1 . ) , dd ( 0 . , 2 . , 1 . ) ;
Hexahedron h1(_v1=a, _v2=b, _v3=c, _v4=d, _v5=aa, _v6=bb, _v7=cc, _v8=dd, _nnodes=Numbers(40, 20, 40, 20, 40, 20, 40, 20), _domain_name="Omega") ;
```

Let’s summarize information about geometrical keys on hexahedra:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v3</td>
<td>Point</td>
<td>_v3=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v5</td>
<td>Point</td>
<td>_v5=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v6</td>
<td>Point</td>
<td>_v6=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v7</td>
<td>Point</td>
<td>_v7=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v8</td>
<td>Point</td>
<td>_v8=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>
Parallelepipeds

To define a parallelepiped, you just have to give 4 vertices (namely $p_1$, $p_2$, $p_4$ and $p_5$), defined as in the following figure:

![Parallelepiped Diagram]

There is a parameter for each of them: \texttt{v1}, \texttt{v2}, \texttt{v4}, and \texttt{v5}. These parameters take points or a single value (in this case, it is like a 1D point). \texttt{nnodes} can take one single value or a vector of 3 or 12 values (\texttt{Numbers} object) and \texttt{hsteps} can take one real value or a vector of 8 real values (\texttt{Reals} object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

\begin{verbatim}
Point a(0.,0.,0.), b(4.,0.,0.), c(4.,2.,0.), d(0.,2.,0.);
Point aa(0.,0.,1.), bb(4.,0.,1.), cc(4.,2.,1.), dd(0.,2.,1.);
Parallelepiped p1(v1=a, v2=b, v4=d, v5=aa, nnodes=Numbers(40, 20, 40, 20, 40, 20, 40, 20, 10, 10, 10, 10), domain_name="Omega");
Parallelepiped p2(v1=a, v2=b, v4=d, v5=aa, nnodes=Numbers(40, 20, 10), domain_name="Omega");
\end{verbatim}

Both parallelepipeds of previous examples are identical. This explains the ability to give 3 values for \texttt{nnodes}.

Let’s summarize information about geometrical keys on parallelepipeds:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{v1}</td>
<td>Point</td>
<td>\texttt{v1}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>\texttt{v2}</td>
<td>Point</td>
<td>\texttt{v2}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>\texttt{v4}</td>
<td>Point</td>
<td>\texttt{v4}=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>\texttt{v5}</td>
<td>Point</td>
<td>\texttt{v5}=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Cuboids

To define a cuboid, you give 4 vertices, as for parallelepipeds.
There is a parameter for each of them: \texttt{\_v1}, \texttt{\_v2}, \texttt{\_v4}, and \texttt{\_v5}. These parameters take points or a single value (in this case, it is like a 1D point). For cuboids where faces are parallel to planes \(x=0, y=0\) and \(z=0\), you can define the cuboid by its center (\(c\) in the figure) and its lengths or \(p_1\) (recalled origin in this case) and its lengths. You may use \texttt{\_center}, \texttt{\_xlength}, \texttt{\_ylength} and \texttt{\_zlength} or \texttt{\_origin}, \texttt{\_xlength}, \texttt{\_ylength} and \texttt{\_zlength} to do so. \texttt{\_origin} and \texttt{\_center} take points or a single value (in this case, it is like a 1D point). \texttt{\_xlength}, \texttt{\_ylength} and \texttt{\_zlength} take one single positive value. There is another possibility: defining the rectangle by its bounds: parameters \texttt{\_xmin}, \texttt{\_xmax}, \texttt{\_ymin}, \texttt{\_ymax}, \texttt{\_zmin} and \texttt{\_zmax}. These parameters take one single value.

\texttt{\_nnodes} can take one single value or a vector of 3 or 12 values (\texttt{Numbers} object) and \texttt{\_hsteps} can take one real value or a vector of 8 real values (\texttt{Reals} object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

\begin{verbatim}
Cuboid c1(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v4=Point(0.,3.,0.),
           _v5=Point(0.,0.,4.), _nnodes=40, _domain_name="Omega");
Cuboid c2(_origin=Point(0.,0.,0.), _xlength=2., _ylength=3., _zlength=4,
           _nnodes=40, _domain_name="Omega");
Cuboid c3(_center=Point(1.,1.5.,2.), _xlength=2., _ylength=3., _zlength=4,
           _nnodes=40, _domain_name="Omega");
Cuboid c4(_xmin=0., _xmax=2, _ymin=0, _ymax=3, _zmin=0, _zmax=4, _nnodes=40,
           _domain_name="Omega");
\end{verbatim}

This is 4 definitions of the same \texttt{Cuboid} object.

Let’s summarize information about geometrical keys on cuboids:
<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_origin</td>
<td>Point</td>
<td>_origin=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v5</td>
<td>Point</td>
<td>_v5=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_xlength</td>
<td>single unsigned integer or real positive value</td>
<td>_xlength=1, _xlength=2.5</td>
</tr>
<tr>
<td>_ylength</td>
<td>single unsigned integer or real positive value</td>
<td>_ylength=1, _ylength=2.5</td>
</tr>
<tr>
<td>_zlength</td>
<td>single unsigned integer or real positive value</td>
<td>_zlength=1, _zlength=2.5</td>
</tr>
<tr>
<td>_xmin</td>
<td>single integer or real value</td>
<td>_xmin=1, _xmin=-2.5</td>
</tr>
<tr>
<td>_xmax</td>
<td>single integer or real value</td>
<td>_xmax=1, _xmax=-2.5</td>
</tr>
<tr>
<td>_ymin</td>
<td>single integer or real value</td>
<td>_ymin=1, _ymin=-2.5</td>
</tr>
<tr>
<td>_ymax</td>
<td>single integer or real value</td>
<td>_ymax=1, _ymax=-2.5</td>
</tr>
<tr>
<td>_zmin</td>
<td>single integer or real value</td>
<td>_zmin=1, _zmin=-2.5</td>
</tr>
<tr>
<td>_zmax</td>
<td>single integer or real value</td>
<td>_zmax=1, _zmax=-2.5</td>
</tr>
</tbody>
</table>

**Cubes**

To define a cube, you give 4 vertices, as for parallelepipeds and cuboids.

There is a parameter for each of them: _v1, _v2, _v4, and _v5. These parameters take points or a single value (in this case, it is like a 1D point). For cuboids where faces are parallel to planes x=0, y=0 and z=0, you can define the cuboid by its center (c in the figure) and its lengths or p1 (recalled origin in this case) and its lengths. You may use _center and _length or _origin and _length to do so. _origin and _center take points or a single value (in this case, it is like a 1D point). _length take one single positive value.
At last, you can give an additional argument: the number of octants to deal with (parameter `nboctants`). Let us explain this with the following figure:

![Diagram of octants]

Considering the center of the cube, and the associated trihedron, symbolized by black dashed arrows, the cube can be splitted into 8 cubic parts, corresponding to one octant. Octants having a numbering convention. When giving the number of octants he asked, for instance 5, the user wants to build intersection of the cube with octants 1 to 5. This is a way to define the Fichera Cube (7 octants) or the L-shape (3 or 6 octants). The default value is 8, so that the whole cube is considered.

`nnodes` can take one single value or a vector of 3 or 12 values (`Numbers` object) and `hsteps` can take one real value or a vector of 8 real values (`Reals` object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```plaintext
Cube c1(_v1=Point(0.,0.,0.), _v2=Point(4.,0.,0.), _v4=Point(0.,4.,0.), _v5=Point(0.,0.,4.), _nnodes=40, _domain_name="Omega");
Cube c2(_origin=Point(0.,0.,0.), _length=2., _nnodes=40, _domain_name="Omega");
Cube c3(_center=Point(1.,1.,1.), _length=2., _nnodes=40, _domain_name="Omega");
```

This is 3 definitions of the same `Cube` object.

Let’s summarize information about geometrical keys on cubes:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td><code>Point</code></td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_origin</td>
<td><code>Point</code></td>
<td>_origin=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td><code>Point</code></td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td><code>Point</code></td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td><code>Point</code></td>
<td>_v4=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v5</td>
<td><code>Point</code></td>
<td>_v5=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_length</td>
<td>single unsigned integer or real positive value</td>
<td>_length=1, _length=2.5</td>
</tr>
<tr>
<td>_nboctants</td>
<td>single unsigned integer value between 1 and 8</td>
<td>_nboctants=3</td>
</tr>
</tbody>
</table>
5.1.6 Ellipsoids and balls

Ellipsoids

To define an ellipsoidal volume, you do the same way as for an ellipse or a disk (See section 5.1.4 or section 5.1.4), namely using 4 points $c, p_1, p_2, p_6$, defined as in the following figure:

There is a parameter for each of them: $\_center$, $\_v1$, $\_v2$, and $\_v6$. These parameters take points or a single value (in this case, it is like a 1D point). For ellipsoidal volumes where main axes are parallel to x-axis, y-axis and z-axis, you can define the ellipsoid with the center and 3 axis lengths. For this purpose, use $\_xlength$, $\_ylength$ and $\_zlength$, taking one single positive value.

$\_nnodes$ can take one single value or a vector of 3 or 12 values ($Numbers$ object), one for each quarter of ellipse. $\_hsteps$ can take one real value or a vector of 6 real values ($Reals$ object).

After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```plaintext
Ellipsoid e1(\_center=Point(0.0,0.0), \_v1=Point(3.0,0.0), \_v2=Point(0.2,0.0), \
    \_v6=Point(0.0,1.0), \_nnodes=Numbers(35,30,25), \_domain_name="Omega1", \
    \_side_names="Gamma");
Ellipsoid e2(\_center=Point(0.0,0.0), \_v1=Point(3.0,0.0), \_v2=Point(0.2,0.0), \
    \_v6=Point(0.0,1.0), \_nnodes=Numbers(35,35,35,35,30,30,30,30,25, \
    25,25,25), \_domain_name="Omega1", \_side_names="Gamma");
Ellipsoid e3(\_center=Point(0.0,0.0), \_xlength=6, \_ylength=4, \_zlength=2, \
    \_nnodes=Numbers(35,30,25), \_domain_name="Omega1", \_side_names="Gamma");
```

This is 3 definitions of the same $Ellipsoid$ object. The difference between $e_1$ and $e_2$ explains the ability to give 3 values for $\_nnodes$.

Let’s summarize information about geometrical keys on ellipsoids:
<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v6</td>
<td>Point</td>
<td>_v6=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>xlength</td>
<td>single unsigned integer or real positive value</td>
<td>xlength=1, xlength=2.5</td>
</tr>
<tr>
<td>ylength</td>
<td>single unsigned integer or real positive value</td>
<td>ylength=1, ylength=2.5</td>
</tr>
<tr>
<td>zlength</td>
<td>single unsigned integer or real positive value</td>
<td>zlength=1, zlength=2.5</td>
</tr>
</tbody>
</table>

**Balls**

To define a ball, you do the same way as for an ellipsoid (See section 5.1.6), namely using 4 points \( c, p_1, p_2, p_6 \), defined as in the following figure:

![Diagram of a ball with labeled points](image)

There is a parameter for each of them: _center, _v1, _v2, and _v6. These parameters take points or a single value (in this case, it is like a 1D point). For balls where main axes are parallel to x-axis, y-axis and z-axis, you can define the ellipsoid with the center and the radius. For this purpose, use _radius, taking one single positive value.

_nnodes can take one single value or a vector of 3 or 12 values (Numbers object), one for each quarter of ellipse. _hsteps can take one real value or a vector of 6 real values (Reals object). After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

At last, you can give an additional argument: the number of octants to deal with (parameter _nboctants). See section 5.1.5 for details.

Examples.

```python
Ball b1(_center=Point(0.,0.,0.), _v1=Point(3.,0.,0.), _v2=Point(0.,3.,0.), _v6=Point(0.,0.,3.), _nnodes=Numbers(35, 30, 25), _domain_name="Omega1", _side_names="Gamma")
Ball b2(_center=Point(0.,0.,0.), _v1=Point(3.,0.,0.), _v2=Point(0.,3.,0.), _v6=Point(0.,0.,3.), _nnodes=Numbers(35, 35, 35, 35, 30, 30, 30, 30, 25, 25, 25, 25), _domain_name="Omega1", _side_names="Gamma")
```
This is 3 definitions of the same Ball object. The difference between \( b_1 \) and \( b_2 \) explains the ability to give 3 values for \( _\text{nnodes} \).

Let’s summarize information about geometrical keys on balls:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v6</td>
<td>Point</td>
<td>_v6=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_radius</td>
<td>single unsigned integer or real positive value</td>
<td>_radius=1, _radius=2.5</td>
</tr>
<tr>
<td>_nboctants</td>
<td>single unsigned integer value between 0 and 8</td>
<td>_nboctants=3</td>
</tr>
</tbody>
</table>

5.1.7 Trunks and trunk-likes

Trunks

A trunk is a generalized truncated cone. To define a trunk, you need to give a surface, namely a polygonal surface (Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, or Square), or a elliptical surface (Ellipse or Disk). To define the other surface, you just need to give a point of this surface (origin), and the scale factor according to the first surface.

For a trunk with polygonal basis, origin is the equivalent of the first vertex of the surface you give, as you can see on the following figure of a trunk with triangular basis. The triangle being defined by its vertices \( p_1, p_2 \) and \( p_3 \), origin is the equivalent of \( p_1 \):

To do so, you will use parameter _basis to define the basis, parameter _origin to define origin, and parameter _scale to define the scale factor.
_basis_ takes any surface object: Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse or Disk. _origin_ takes a point or a single value (in this case, it is like a 1D point). _scale_ takes one single positive value.

For a trunk with elliptical basis, _origin_ is the center of the second basis, as you can see on the following figure of a trunk with elliptical basis.

To do so, you will use parameters _center1, v1, v2, center2_ and _scale_ to define such a trunk. _center1, v1, v2_ and _center2_ take a point or a single value (in this case, it is like a 1D point). _center1, v1_ and _v2_ are used as for a Ellipse or Disk object (see section 5.1.4 or section 5.1.4 for details). _center2_ is used in this case instead of _origin_, as it is the center of the second basis.

_nnodes_ can take one single value or a vector of 3 or n values (Numbers object), where n is 3 times the number of edges of the basis. _hsteps_ can take one real value or a vector of p real values (Reals object), where p is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```
Trunk t1(_basis=Triangle(_v1=Point(0.,0.,0.), _v2=Point(3.,0.,0.),
    _v3=Point(0.,2.,0.)), _origin=Point(0.,2.,1.), _scale=0.5,
    _nnodes=Numbers(10, 10, 10, 5, 5, 5, 20, 20, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma"));
Trunk t2(_basis=Triangle(_v1=Point(0.,0.,0.), _v2=Point(3.,0.,0.),
    _v3=Point(0.,2.,0.)), _origin=Point(0.,2.,1.), _scale=0.5,
    _nnodes=Numbers(10, 5, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma"));
```

This is 2 definitions of the same Trunk object, explaining the ability to give 3 values for _nnodes_, instead of 9.

Let’s summarize information about geometrical keys on trunks:
<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_basis</td>
<td>Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse, Disk</td>
<td>_basis=Triangle(...)</td>
</tr>
<tr>
<td>_origin</td>
<td>Point</td>
<td>_origin=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_scale</td>
<td>single unsigned integer or real positive value</td>
<td>_scale=2, _scale=0.5</td>
</tr>
<tr>
<td>_center1</td>
<td>Point</td>
<td>_center1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_center2</td>
<td>Point</td>
<td>_center2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

**Cylinders**

A cylinder is a truncated cone whose apex is at infinite distance. So it is the geometry defined by the extrusion of a surface by translation.

To do so, you have to use parameters _basis and _direction. _basis, as for trunks, take any surface object: Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse or Disk. _direction takes a vector of real numbers (Point or Reals objects) or a single value (in this case, it is like a direction parallel to x-axis).

As for a trunk, a cylinder with elliptical basis can be defined by another way, using parameters _center1, _v1, _v2 and _center2, taking a point or a single value (in this case, it is like a 1D point). _center1, _v1 and _v2 are used as for a Ellipse or Disk object (see section 5.1.4 or section 5.1.4 for details). _center2 is used in this case instead of _direction, as it is easier to give the center of the second basis, instead of the direction vector.

_mnodes can take one single value or a vector of 3 or n values (Numbers object), where n is 3 times the number of edges of the basis. _lnsteps can take one real value or a vector of p real values (Reals object), where p is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.
This is 3 definitions of the same **Cylinder** object, explaining the ability to give 3 values for **nnodes**, instead of 12.

Let’s summarize information about geometrical keys on cylinders:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_basis</td>
<td>Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse, Disk</td>
<td>_basis=Triangle(...)</td>
</tr>
<tr>
<td>_direction</td>
<td>std::vector of real values, Reals or Point</td>
<td>_direction=Reals(0,0,1.), _direction=Point(0,0,1)</td>
</tr>
<tr>
<td>_center1</td>
<td>Point</td>
<td>_center1=Point(0,0,0)</td>
</tr>
<tr>
<td>_center2</td>
<td>Point</td>
<td>_center2=Point(0,0,0)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0,0,0)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0,0,0)</td>
</tr>
</tbody>
</table>

**Prisms**

A prism is by definition a cylinder whose basis is a polygonal surface (**Polygon**, **Triangle**, **Quadrangle**, **Parallelogram**, **Rectangle**, or **Square**).

As for cylinder, you will use parameters **_basis** and **_direction**. **_basis**, as for trunks, take any polygonal object: **Polygon**, **Triangle**, **Quadrangle**, **Parallelogram**, **Rectangle** or **Square**.
_direction_ takes a vector of real numbers (Point or Reals objects) or a single value (in this case, it is like a direction parallel to x-axis).

Often a prism refers to a cylinder with triangular basis (as the finite element cell). So you can also define a prism from 3 points (for triangular basis), using parameters _v1_, _v2_, _v3_ instead of _basis_, taking a point or a single value (in this case, it is like a 1D point).

_nnodes_ can take one single value or a vector of 3 or n values (Numbers object), where n is 3 times the number of edges of the basis. _hsteps_ can take one real value or a vector of p real values (Reals object), where p is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```
Prism p1(_basis=Triangle(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.),
                      _v3=Point(0.,1.,0.)), _direction=Reals(0.,2.,1.), _nnodes=Numbers(10, 10,
                      5, 5, 5, 20, 20, 20), _domain_name="Omega",
                      _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma"));
Prism p2(_basis=Triangle(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.),
                      _v3=Point(0.,1.,0.)), _direction=Reals(0.,2.,1.), _nnodes=Numbers(10, 5,
                      20), _domain_name="Omega", _side_names=Strings("Gamma", "Gamma", "Sigma",
                      "Sigma", "Sigma"));
Prism p3(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v3=Point(0.,1.,0.),
                      _direction=Reals(0.,2.,1.), _nnodes=Numbers(10, 5, 20),
                      _domain_name="Omega", _side_names=Strings("Gamma", "Gamma", "Sigma",
                      "Sigma", "Sigma"));
```

This is 3 definitions of the same Prism object, explaining the ability to give 3 values for _nnodes_, instead of 9.

Let’s summarize information about geometrical keys on prisms:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>basis</em></td>
<td>Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square</td>
<td><em>basis</em>=Triangle(...)</td>
</tr>
<tr>
<td><em>direction</em></td>
<td>std::vector of real values, Reals or Point</td>
<td><em>direction</em>=Reals(0.,0.,1.), <em>direction</em>=Point(0.,0.,1.)</td>
</tr>
<tr>
<td><em>v1</em></td>
<td>Point</td>
<td><em>v1</em>=Point(0.,0.,0.)</td>
</tr>
<tr>
<td><em>v2</em></td>
<td>Point</td>
<td><em>v2</em>=Point(0.,0.,0.)</td>
</tr>
<tr>
<td><em>v3</em></td>
<td>Point</td>
<td><em>v3</em>=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Cones

A cone is defined by a surface and an apex.
To do so, you will use parameters _basis and _apex. _basis, as for trunks, take any surface object: Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse or Disk. _apex takes a point or a single value (in this case, it is like a 1D point).

As for trunks and cylinders, you can also define directly a cone with elliptical basis, with parameters _center1, _v1, _v2 (and _apex). These parameters take a point or a single value (in this case, it is like a 1D point).

_nnodes can take one single value or a vector of 2 or n values (Numbers object), where n is twice the number of edges of the basis. _hsteps can take one real value or a vector of p real values (Reals object), where p is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

Cone c1(_basis=Disk(_center=Point(0.,0.,0.), _v1=Point(2.,0.,0.), _v2=Point(0.,2.,0.)), _apex=Point(0.,0.,1.), _nnodes=Numbers(20, 20, 20, 20, 10, 10, 10, 10), _domain_name="Omega", _side_names="Gamma");
Cone c2(_basis=Disk(_center=Point(0.,0.,0.), _v1=Point(2.,0.,0.), _v2=Point(0.,2.,0.)), _apex=Point(0.,0.,1.), _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma");
Cone c3(_center1=Point(0.,0.,0.), _v1=Point(2.,0.,0.), _v2=Point(0.,2.,0.), _apex=Point(0.,0.,1.), _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma");

This is 3 definitions of the same Cone object, explaining the ability to give 2 values for _nnodes, instead of 8.

Actually, this geometry cannot be meshed. Please use Pyramid for cones with polygonal basis, or RevCone for revolution cones.

Let’s summarize information about geometrical keys on cones:
<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_apex</td>
<td>Point</td>
<td>_apex=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_basis</td>
<td>Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse, Disk</td>
<td>_basis=Triangle(…)</td>
</tr>
<tr>
<td>_center1</td>
<td>Point</td>
<td>_center1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Pyramids

A pyramid is a cone with a polygonal basis (Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, or Square).

As for cones, you will use parameters _basis and _apex. _basis takes any polygonal object: Polygon, Triangle, Quadrangle, Parallelogram, Rectangle or Square. _apex takes a point or a single value (in this case, it is like a 1D point).

Often a pyramid refers to a cone with quadrangular basis (as the finite element cell). So you can also define a pyramid from 4 points (for quadrangular basis), using parameters _v1, _v2, _v3, _v4 instead of _basis, taking a point or a single value (in this case, it is like a 1D point).

_nnodes can take one single value or a vector of 2 or n values (Numbers object), where n is twice the number of edges of the basis. _nsteps can take one real value or a vector of p real values (Reals object), where p is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

Pyramid p1(_basis=Quadrangle(_v1=Point (0.,0.,0.), _v2=Point (2.,0.,0.),
_v3=Point (1.,1.,0.), _v4=Point (-1.,2.,0.), _apex=Point (0.,0.,1.),
_nnodes=Numbers(20, 20, 20, 20, 10, 10, 10, 10), _domain_name="Omega",
_side_names="Gamma"));
Pyramid p2(_basis=Quadrangle(_v1=Point (0.,0.,0.), _v2=Point (2.,0.,0.),
_v3=Point (1.,1.,0.), _v4=Point (-1.,2.,0.), _apex=Point (0.,0.,1.),
_nnodes=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma");
Pyramid p3(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v3=Point(1.,1.,0.), _v4=Point(-1.,2.,0.), _apex=Point(0.,0.,1.), _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma");

This is 3 definitions of the same Pyramid object, explaining the ability to give 2 values for _nnodes, instead of 8.

Let’s summarize information about geometrical keys on pyramids:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_apex</td>
<td>Point</td>
<td>_apex=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_basis</td>
<td>Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square</td>
<td>_basis=Triangle(…)</td>
</tr>
<tr>
<td>_v1</td>
<td>Point</td>
<td>_v1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v2</td>
<td>Point</td>
<td>_v2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v3</td>
<td>Point</td>
<td>_v3=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_v4</td>
<td>Point</td>
<td>_v4=Point(0.,0.,0.)</td>
</tr>
</tbody>
</table>

Revolution trunks

A revolution trunk is a right trunk with circular basis.

So, to define a revolution trunk, you just need to give centers and radiiuses of bases, using dedicated parameters _center1, _center2, taking a point or a single value (in this case, it is a 1D point), and _radius1 and _radius2, taking one single positive value. RevTrunk offers you more geometry abilities. Indeed, you can decide to add extensions at ends of the revolution trunk. Extensions can be: none, flat, ellipsoid, or cone. To define an extension, you just have to give its shape (type GeometricEndShape, values: _gesNone, _gesFlat, _gesEllipsoid or _gesCone) and its height (called distance, as it is the distance of the apex/apogee from the corresponding basis of the trunk). Default values are flat with no height. Please also note that any extension means 4 additional edges and 4 additional side domains.
To do so, you will use parameters _end1_shape and _end2_shape, taking a GeometricEndShape, and _end1_distance and _end2_distance, taking one single positive value.

_nnodes_ can take one single value or a vector of 3 or _n_ values (Numbers object), where _n_ is 3 times the number of edges of the basis. _hsteps_ can take one real value or a vector of _p_ real values (Reals object), where _p_ is the number of points defining the trunk. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

You also have an additional parameter, _nbsubdomains_, enabling you to slice the main trunk (without its extensions) in as many domains as you want.

Let’s summarize information about geometrical keys on revolution trunks:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>center1</em></td>
<td>Point</td>
<td>_center1=Point(0.,0.,0.)</td>
</tr>
<tr>
<td><em>center2</em></td>
<td>Point</td>
<td>_center2=Point(0.,0.,0.)</td>
</tr>
<tr>
<td><em>radius1</em></td>
<td>single unsigned integer or real positive value</td>
<td>_radius1=1, _radius1=2.5</td>
</tr>
<tr>
<td><em>radius2</em></td>
<td>single unsigned integer or real positive value</td>
<td>_radius2=1, _radius2=2.5</td>
</tr>
<tr>
<td><em>end1_shape</em></td>
<td>enum GeometricEndShape</td>
<td>_end1_shape=gesNone,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end1_shape=gesFlat,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end1_shape=gesCone,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end1_shape=gesEllipsoid,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end1_shape=gesSphere</td>
</tr>
<tr>
<td><em>end2_shape</em></td>
<td>enum GeometricEndShape</td>
<td>_end2_shape=gesNone,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end2_shape=gesFlat,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end2_shape=gesCone,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end2_shape=gesEllipsoid,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_end2_shape=gesSphere</td>
</tr>
<tr>
<td><em>end1_distance</em></td>
<td>single unsigned integer or real positive value</td>
<td>_end1_distance=1, _end1_distance=2.5</td>
</tr>
<tr>
<td><em>end2_distance</em></td>
<td>single unsigned integer or real positive value</td>
<td>_end2_distance=1, _end2_distance=2.5</td>
</tr>
<tr>
<td><em>nbsubdomains</em></td>
<td>single unsigned integer value</td>
<td>_nbsubdomains=2</td>
</tr>
</tbody>
</table>

Revolution cylinders

A revolution cylinder is a revolution trunk where both radiuses are equal. So, we need centers of both bases, and the radius.
To do so, you just have to give centers and radius of bases, using dedicated parameters _center1, _center2, taking a point or a single value (in this case, it is a 1D point), and _radius, taking one single positive value.

As RevTrunk, RevCylinder offers you the ability to add extensions at ends of the revolution cylinder. See section 5.1.7 for how to define these extensions. To do so, you will use parameters _end1_shape and _end2_shape, taking a GeometricEndShape, and _end1_distance and _end2_distance, taking one single positive value.

_nnodes can take one single value or a vector of 3 or n values (Numbers object), where n is 3 times the number of edges of the basis. _hsteps can take one real value or a vector of p real values (Reals object), where p is the number of points defining the cylinder. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

You also have an additional parameter, _nsubdomains, enabling you to slice the main cylinder (without its extensions) in as many domains as you want.

Examples.

```plaintext
RevCylinder r1(_center1=Point(0.,0.,0.), _center2=Point(5.,0.,0.), _radius=1, _nnodes=Numbers(10, 10, 10, 10, 10, 10, 10, 10, 10, 50, 50, 50, 50), _domain_name="Omega", _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma", "Sigma"));
RevCylinder r2(_center1=Point(0.,0.,0.), _center2=Point(5.,0.,0.), _radius=1, _nnodes=Numbers(10, 10, 50), _domain_name="Omega", _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma", "Sigma", "Sigma"));
```

Let’s summarize information about geometrical keys on revolution cylinders:
**key** | **authorized types** | **examples**
---|---|---
`center1` | `Point` | `center1=Point(0.,0.,0.)`
`center2` | `Point` | `center2=Point(0.,0.,0.)`
`radius` | `single unsigned integer or real positive value` | `radius=1, _radius=2.5`

`end1_shape` | `enum GeometricEndShape` | `end1_shape=gesNone, end1_shape=gesFlat, end1_shape=gesCone, end1_shape=gesEllipsoid, end1_shape=gesSphere`

`end2_shape` | `enum GeometricEndShape` | `end2_shape=gesNone, end2_shape=gesFlat, end2_shape=gesCone, end2_shape=gesEllipsoid, end2_shape=gesSphere`

`end1_distance` | `single unsigned integer or real positive value` | `end1_distance=1, _end1_distance=2.5`

`end2_distance` | `single unsigned integer or real positive value` | `end2_distance=1, _end2_distance=2.5`

`nbsubdomains` | `single unsigned integer value` | `nbsubdomains=2`

**Revolution cones**

A revolution cone is a revolution trunk where second radius is equal to 0.

To define a revolution cone, you need to give a center, a radius, and an apex, through parameters `center`, `radius` ans `apex`. `center` and `apex` take a point or a single value (in this case, it is like a 1D point), whereas `radius` takes a single positive value.

As `RevTrunk, RevCone` offers you more the ability to add an extension to the basis of a revolution cone. See section 5.1.7 for how to define this extension. To do so, you will use parameters `end_shape`, taking a `GeometricEndShape`, and `end_distance`, taking one single positive value. `nnodes` can take one single value or a vector of 2 or `n` values (`Numbers` object), where `n` is twice the number of edges of the basis. `hsteps` can take one real value or a vector of `p` real values.
Let’s summarize information about geometrical keys on revolution cones:

<table>
<thead>
<tr>
<th>key</th>
<th>authorized types</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>_apex</td>
<td>Point</td>
<td>_apex=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_center</td>
<td>Point</td>
<td>_center=Point(0.,0.,0.)</td>
</tr>
<tr>
<td>_radius</td>
<td>single unsigned integer or real positive value</td>
<td>_radius=1, _radius=2.5</td>
</tr>
<tr>
<td>_end_shape</td>
<td>enum GeometricEndShape</td>
<td>_end_shape=gesNone, _end_shape=gesFlat, _end_shape=gesCone, _end_shape=gesEllipsoid, _end_shape=gesSphere</td>
</tr>
<tr>
<td>_end_distance</td>
<td>single unsigned integer or real positive value</td>
<td>_end_distance=1, _end_distance=2.5</td>
</tr>
<tr>
<td>_nsubdomains</td>
<td>single unsigned integer value</td>
<td>_nsubdomains=2</td>
</tr>
</tbody>
</table>

5.1.8 Definition of a geometry from its boundary

A loop geometry is a geometry defined by its boundaries. For example, instead of defining a triangle, you will define here the surface inside the closed boundary composed of 3 segments.

With XLiFE++ geometry engine, you can define 2D or 3D geometries, thanks to the following routines:

```cpp
Geometry planeSurfaceFrom(const Geometry& boundary, String domName = String());
Geometry ruledSurfaceFrom(const Geometry& boundary, String domName = String());
Geometry volumeFrom(const Geometry& boundary, String domName = String());
```

The first argument must be a "composite" geometry defined from curve boundaries (2D case) or surface boundaries (3D case) such that the result is closed.

Let’s see an example using segments and circular arcs to define a mesh on a rectangle with rounded corners:

```cpp
Point a(-1.5,-4.); Point b(1.5,-4.); Point c(2.,-3.5); Point d(2.,3.5);
Point e(1.5,4.); Point f(-2.3.5); Point g(-2.,3.5);
Segment s1(_v1=a, _v2=b, _nnodes=21, _domain_name="AB");
CircArc c1(_center=Point(3.5,0.5), _v1=b, _v2=c, _nnodes=5, _domain_name="BC");
Segment s2(_v1=c, _v2=d, _nnodes=11, _domain_name="CD");
CircArc c2(_center=Point(3.5,1.5), _v1=d, _v2=e, _nnodes=5, _domain_name="DE");
Segment s3(_v1=e, _v2=f, _nnodes=21, _domain_name="EF");
CircArc c3(_center=Point(0.5,1.5), _v1=f, _v2=g, _nnodes=5, _domain_name="FG");
Segment s4(_v1=g, _v2=h, _nnodes=11, _domain_name="GH");
CircArc c4(_center=Point(0.5,0.5), _v1=h, _v2=a, _nnodes=5, _domain_name="HA");
Geometry g=planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4,"Omega");
```

The `surfaceFrom` routine is devoted to define surfaces from their boundaries. Segments and circular arcs must be defined with the same orientation (clockwise or counter-clockwise).

With such definitions of segments $s_1, s_2, s_3$ and $s_4$ and circular arcs $c_1, c_2, c_3$ and $c_4$, in previous example, the following definitions are right:

(Reals object), where $p$ is the number of points defining the cone. After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

You also have an additional parameter, `nsubdomains`, enabling you to slice the main cone (without its extension) in as many domains as you want.
The order of components here, and also the first component, has no meaning, but they all are oriented in the same way.

Figure 5.1: Rectangular geometry with rounded corners, defined with the `surfaceFrom` routine

We tell you that it is also possible for 3D case. Here is an example of a geometry basically composed of a cube and a pyramid sharing one face:

```plaintext
Point a(0,0,0); Point b(2,0,0); Point c(2,2,0); Point d(0,2,0);
Point e(0,0,2); Point f(2,0,2); Point g(2,2,2); Point h(0,2,2)
Point i (4,1,1);
Square s1(_v1=a, _v2=b, _v4=e, _nnodes=11, _domain_name="S1");
Square s2(_v1=d, _v2=c, _v4=h, _nnodes=11, _domain_name="S2");
Square s3(_v1=a, _v2=b, _v4=d, _nnodes=11, _domain_name="S3");
Square s4(_v1=e, _v2=f, _v4=h, _nnodes=11, _domain_name="S4");
Square s5(_v1=a, _v2=d, _v4=e, _nnodes=11, _domain_name="S5");
Triangle t1(_v1=b, _v2=c, _v3=i, _nnodes=11, _domain_name="T1");
Triangle t2(_v1=c, _v2=g, _v3=i, _nnodes=11, _domain_name="T2");
Triangle t3(_v1=g, _v2=f, _v4=i, _nnodes=11, _domain_name="T3");
Triangle t4(_v1=f, _v2=b, _v4=i, _nnodes=11, _domain_name="T4");
Geometry vf=volumeFrom(s1+s2+s3+s4+s5+t1+t2+t3);
```

Figure 5.2: 3D geometry defined with the `volumeFrom` routine
3D loop geometries can be defined by a mix between 2D loop geometries and 2D canonical geometries.

Although C++ authorizes it, do not write loop geometries as follows:

```cpp
volumeFrom(Rectangle(a,b,d,11,11,"R1")+...);
```

You have to define the rectangle \( r_1 \) instead, as in the previous example.

### 5.1.9 Combining geometries

A composite geometry is a geometry defined from a list of canonical or loop geometries. It is for example the right way to define holes in your mesh, or to define multi-domains geometries.

How to define composite geometries? It’s easy, you just have to use the operators + and -.

Let’s see a first example:

```cpp
Rectangle r (xmin=-3, xmax=3, ymin=-2, ymax=2, nnodes=Numbers(33,22), _domain_name="Omega");
Ellipse e (_center=Point(0,0), _xlength=1, _ylength=0.5, _nnodes=11);
Geometry gm=r-e;
Geometry gp=r+e;
```

![Figure 5.3: Composite geometry of an ellipse inside the rectangle](image)

In both cases, the ellipse is geometrically inside the rectangle. This hole will be meshed if you used the operator +, and not meshed if you use the operator -. Both operators can detect if a geometry is inside another geometry, in most of the cases.

If you forgot to give a domain name for the right hand side of the operator +, it will not be stored, so that you still will have a hole.

These operators work with any geometries as far as geometrical inclusion is easy enough to detect. Using operators + and - to define composite geometries is not restricted to 2 components. You can define composite geometries with any number of components, and some of them can be loop geometries:
Figure 5.4: Composite geometry with multiple components and inclusions between components

When at least 2 components share several vertices, several edges and/or several surfaces, everything works fine, shared geometrical entities are not duplicated.
As far as composite geometries are concerned, XLiFE++ detects inclusions between canonical components. It is not always the case if components are loop geometries. Let’s take the previous example, but this time, we want to mesh every domain.

\[
\text{Geometry } \text{gmulti2}=(e1+sf1)+(e2+e3)+r2+sf2; \]

You can see that both holes of the rounded rectangle are not taken into account, whereas the half disk is correctly managed. Indeed, XLiFE++ can in most of the cases determine if a loop geometry is inside a canonical geometry but it can’t determine if a canonical geometry is inside a loop geometry.

How to solve this problem? By forcing it with the unary + operator, and rewriting the composite expression if necessary, as in the following:

\[
\text{Geometry } \text{gmulti3}=e1+(sf1+((e2+e3))+r2+sf2; \]

Figure 5.5: Composite geometry with edges shared by components.

Figure 5.6: Composite geometry with multiple components and inclusions between components. Some inclusions are not detected correctly.
When you write \((sf1+(+(e2+e3))\), you tell explicitly that the right operand \(+(e2+e3)\) is forced be inside the left operand \((sf1)\).

When at least two components intersect and the intersection has same dimension (2 surfaces whose intersection is a surface, for instance), the resulting mesh will not be generated properly. In this case, you must reconsider how to define your geometry.

5.2 Transformations on geometries

XLtFE++ allows you to apply geometrical transformations on Mesh, Geometry and Geometry children objects. The main type is Transformation. It can be a canonical transformation or a composition of transformations.

5.2.1 Canonical transformations

In the following, we will consider straight lines and planes.
A straight line is fully defined by a point and a direction. The latter is a vector of components (2 or 3). This is a reason why we will write a straight line as follows: \((\Omega, \vec{d})\)

A plane is fully defined by a point and a normal vector. This is a reason why we will write a plane as follows: \([\Omega, \vec{n}]\)

**Translations**

Point B is the image of point A by a translation of vector \(\vec{u}\) if and only if

\[
\vec{AB} = \vec{u}
\]

A translation can be defined by a STL vector (size 2 or 3) or its components:

```cpp
Vector<Real> u;
Real ux, uy, uz;
Translation t1(u), t2(ux, uy), t3(ux, uy, uz);
```

\(\vec{u}\) can be omitted. If so, its default value is the 3d zero vector. \(uy\) and \(uz\) can be omitted too. If so, their default value is 0.

As the `Vector` class inherits from `std::vector` you can use it in place of `Vector` because all prototypes are based on `std::vector`.

**2d rotations**

Point B is the image of point A by the 2d rotation of center \(\Omega\) and of angle \(\theta\) if and only if

\[
\vec{\Omega B} = \left( \begin{array}{cc} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{array} \right) \vec{\Omega A}
\]

A 2d rotation is defined by a point and an angle:

```cpp
Point c;
Real angle;
Rotation2d r(c, angle);
```

\(\text{angle}\) can be omitted. If so, its default value is 0 and \(c\) can be omitted too. If so, its default value is the 3d zero point.

**3d rotations**

Point B is the image of point A by the 3d rotation of axis \((\Omega, \vec{d})\) and of angle \(\theta\) if and only if

\[
\vec{\Omega B} = \cos \theta \vec{\Omega A} + (1 - \cos \theta) \vec{\Omega A} \cdot \vec{n} + \sin \theta \vec{n} \wedge \vec{\Omega A} \quad \text{(Rodrigues' rotation formulae)}
\]
where \( \hat{n} = \frac{\vec{u}}{||\vec{u}||} \) (the unitary direction).

The direction can be defined by a STL vector or by its components:

```cpp
Point c;
Vector<Real> d;
Real dx, dy, dz;
Real angle;
Rotation3d r1(c, d, angle), r2(c, dx, dy, dz, angle);
```

In the first syntax, `angle` can be omitted. If so, its default value is 0. and `d` can also be omitted. If so, its default value is the 3d zero vector.
In the second syntax, `dz` can be omitted too. If so, its default value is 0. .

**Homotheties**

Point B is the image of point A by the homothety of center \( \Omega \) and of factor \( k \) if and only if

\[
\overrightarrow{\Omega B} = k \overrightarrow{\Omega A}
\]

```cpp
Point c;
Real factor;
Homothety h(c, factor);
```

`factor` can be omitted. If so, its default value is 0. and `c` can also be omitted. If so, its default value is the 3d zero vector.

**Point reflections**

Point B is the image of point A by the point reflection of center \( \Omega \) if and only if

\[
\overrightarrow{\Omega B} = -\overrightarrow{\Omega A}
\]

It is an homothety of factor -1 and same center.

```cpp
Point c;
PointReflection h(c);
```

`c` can also be omitted. If so, its default value is the 3d zero vector.
2d reflections

Point B is the image of point A by the 2d reflection of axis \((\Omega, \vec{d})\) if and only if

\[
\overrightarrow{AB} = 2\overrightarrow{AH}
\]

where \(H\) is the orthogonal projection of \(A\) on \((\Omega, \vec{d})\).

3d reflections

Point B is the image of point A by the 2d reflection of plane \([\Omega, \vec{n}]\) if and only if

\[
\overrightarrow{AB} = 2\overrightarrow{AH}
\]

where \(H\) is the orthogonal projection of \(A\) on \([\Omega, \vec{n}]\).

5.2.2 Composition of transformations

To define a composition of transformations, you can use the operator * between canonical transformations, as in the following example:

\[
\begin{align*}
\text{Rotation2d} & \quad \text{r1(Point}(0.,0.), 120.) ; \\
\text{Reflection2d} & \quad \text{r2(} \text{Point}(1., -1.), 1., 2.5, -3.) ; \\
\text{Translation} & \quad \text{t1(} -1., 4.) ; \\
\text{Homothety} & \quad \text{h(} \text{Point}(-1., 0.), -3.2) ; \\
\text{Transformation} & \quad \text{t} = \text{r1*h*r2*t1 ;}
\end{align*}
\]

Composition * has to be understood as usual composition operator \(\circ\) : \(t(P)=r1(h(r2(t1(P))))\).

5.2.3 Geometrical extrusions

This is another way to define geometries: by extrusion of geometries of lesser dimension. Extruded geometries can be surfaces or volumes, defined by a geometry (the section of the extruded geometry) and a geometrical transformation. This feature can be used to generate meshes with the
Gmsh interface with some restrictions about the transformation: only translations or rotations are authorized. There is also another parameter: the number of layers. Let’s see the following figures:

Figure 5.9: On the left, extrusion of a disk by a translation, with 3 layers. On the right, extrusion of a circular arc by rotation, with 4 layers

XLtFE++ offers 4 variants of the same function to define a Geometry by extrusion, enabling to give the domain name to the extruded geometry and to its sides. Sides numbering is as follows: first, the geometry used as section of the extrusion, second, the other section, and next the lateral surfaces generated by the extrusion.

```
Geometry extrude(const Geometry& g, const Transformation& t, Number layers);
Geometry extrude(const Geometry& g, const Transformation& t, Number layers, String domName);
Geometry extrude(const Geometry& g, const Transformation& t, Number layers, Strings sidenames);
Geometry extrude(const Geometry& g, const Transformation& t, Number layers, String domName, Strings sidenames);
```

The Geometry given to the `extrude` function can be:

- a canonical one (1D or 2D). Here, a CircArc.

Figure 5.10: Extrusion of a circular arc by translation, with 5 layers

- A loop geometry (1D or 2D). Here, a rounded rectangle defines as in Figure 5.32
• Every composite geometry composed exclusively of a geometry and its holes (1D or 2D). That is to say only operator- or operator-= is used to define the geometry

Figure 5.11: Extrusion of a rounded rectangle (loop geometry) by a rotation, with 10 layers

Figure 5.12: Extrusion of an ellipse with an elliptic hole by rotation, with 10 layers

5.2.4 Applying transformations

How to apply a transformation?

In this paragraph, we will look at the Cube object, but you have same functions for any canonical or composite Geometry.

If you want to apply a transformation and modify the input object, you can use one of the following functions:

```cpp
// ! apply a geometrical transformation on a Cube
Cube& Cube::transform(const Transformation& t);
// ! apply a translation on a Cube
Cube& Cube::translate(std::vector<Real> u = std::vector<Real>(3, 0.));
Cube& Cube::translate(Real ux, Real uy = 0., Real uz = 0.);
// ! apply a rotation 2d on a Cube
Cube& Cube::rotate2d(const Point& c = Point(0., 0.), Real angle = 0.);
// ! apply a rotation 3d on a Cube
Cube& Cube::rotate3d(const Point& c = Point(0., 0., 0.), std::vector<Real> u = std::vector<Real>(3, 0.), Real angle = 0.);
Cube& Cube::rotate3d(Real ux, Real uy, Real uz, Real angle);
Cube& Cube::rotate3d(Real ux, Real uy, Real uz, Real angle);
Cube& Cube::rotate3d(const Point& c, Real ux, Real uy, Real uz, Real angle);
Cube& Cube::rotate3d(const Point& c, Real ux, Real uy, Real uz, Real angle);
```
apply a homothety on a Cube
Cube& Cube::homothetize(const Point& c = Point(0.,0.,0.), Real factor = 1.);
Cube& Cube::homothetize(Real factor);
apply a point reflection on a Cube
Cube& Cube::pointReflect(const Point& c = Point(0.,0.,0.));
apply a reflection 2d on a Cube
Cube& Cube::reflect2d(const Point& c = Point(0.,0.), std::vector<Real> u = std::vector<Real>(2,0.));
Cube& Cube::reflect2d(const Point& c, Real ux, Real uy = 0.);
apply a reflection 3d on a Cube
Cube& Cube::reflect3d(const Point& c = Point(0.,0.,0.), std::vector<Real> u = std::vector<Real>(3,0.));
Cube& Cube::reflect3d(const Point& c, Real ux, Real uy, Real uz = 0.);

For instance,
Cube c;
c . translate(0.,0.,1.);

If you want now to create a new Cube by applying a transformation on a Cube, you should use one of the following functions instead :

apply a geometrical transformation on a Cube (external)
Cube transform(const Cube& m, const Transformation& t);
apply a translation on a Cube (external)
Cube translate(const Cube& m, std::vector<Real> u = std::vector<Real>(3,0.));
Cube translate(const Cube& m, Real ux, Real uy = 0., Real uz = 0.);
apply a rotation 2d on a Cube (external)
Cube rotate2d(const Cube& m, const Point& c = Point(0.,0.), Real angle = 0.);
apply a rotation 3d on a Cube (external)
Cube rotate3d(const Cube& m, const Point& c = Point(0.,0.,0.),
             std::vector<Real> u = std::vector<Real>(3,0.), Real angle = 0.);
Cube rotate3d(const Cube& m, const Point& c, Real ux, Real uy, Real uz, Real angle);
Cube rotate3d(const Cube& m, const Point& c, Real ux, Real uy, Real uz, Real angle);
apply a homothety on a Cube (external)
Cube homothetize(const Cube& m, const Point& c = Point(0.,0.,0.), Real factor = 1.);
Cube homothetize(const Cube& m, Real factor);
apply a point reflection on a Cube (external)
Cube pointReflect(const Cube& m, const Point& c = Point(0.,0.,0.));
apply a reflection 2d on a Cube (external)
Cube reflect2d(const Cube& m, const Point& c = Point(0.,0.), std::vector<Real> u = std::vector<Real>(2,0.));
Cube reflect2d(const Cube& m, const Point& c, Real ux, Real uy = 0.);
apply a reflection 3d on a Cube (external)
Cube reflect3d(const Cube& m, const Point& c = Point(0.,0.,0.),
                std::vector<Real> u = std::vector<Real>(3,0.));
Cube reflect3d(const Cube& m, const Point& c, Real ux, Real uy, Real uz = 0.);

For instance,
Cube c1;
Cube c2 = translate(c1,0.,0.,1.);
Of course, you can not apply a 2d rotation or a 2d reflection for geometries defined by 3d points!

What does a transformation really do?

Applying a transformation on an object means computing the image of each point defining the object. But it can also change names.

When you create a new object by applying a transformation on a object, names are modified. Indeed, the transformation add a suffix ”_prime”. It concerns geometry names and sidenames.

When you transform a Geometry, it also apply the transformation on the underlying bounding box.

5.3 Defining a mesh from a geometry

XLiFE++ owns some constructors that allow to create meshes based on simple geometries in one, two or three dimensions. The constructors to use are defined as follows:

```c++
// ! constructor from 1D geometries
Mesh (const Geometry& g, Number order = 1, MeshGenerator mg =
   _defaultGenerator, const String& name = "");
// ! constructor from 2D or 3D geometries
Mesh (const Geometry& g, ShapeType sh, Number order = 1, MeshGenerator mg =
   _defaultGenerator, const String& name = "");
```

The arguments are:

- `g` is the geometrical object to be meshed (such as Segment, Quadrangle, Hexahedron, ..., all of them being declared in the file geometries.hpp),
- `sh` is the shape of the mesh elements (_segment, _triangle, _quadrangle, _tetrahedron, _hexahedron),
- `order` is the interpolation order of the mesh elements ; it depends on the way the mesh is generated (see below),
- `mg` defines the way the object is computed:
  - _structured : a structured mesh can be built for canonical geometries only (Segment, Parallelogram, Rectangle, Square, Parallelepiped, Cuboid and Cube); the order of the mesh is necessarily one,
  - _subdiv : a unstructured mesh can be built using the so-called subdivision basic algorithm for the following geometries : Cube, Ball, RevTrunk, RevCone, RevCylinder, Disk and SetOfElems; the order can be any integer \( k > 0 \),
  - _gmsh : for more complicated geometries, with a nested call of the Gmsh software; the order depends on the chosen shape (refer to Gmsh documentation).
name defines the mesh name.

Examples.

```cpp
// P1 structured mesh of segment [0,1] with 10 nodes. Domain is Omega
Mesh m1D(Segment(_xmin=0., _xmax=1., _nnodes=10, _domain_name="Omega"), 1, _structured);
// P1 unstructured mesh of disk of center (0,0,1) and radius 2.5 with 40 nodes. Domain is Omega and side domain is Gamma
Mesh m2D(Disk(_center=Point(0.,0.,1.), _radius=2.5, _nnodes=40, _domain_name="Omega" _side_names="Gamma"), _triangle, 1, _subdiv);
// Q2 unstructured mesh (using gmsh) of cube [0,2] x [0,1] x [0,4] with 20 nodes on x edges, 10 nodes on y edges and 40 nodes on z edges
Mesh m3D(Cube(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v4=Point(0.,1.,0.), _v5=Point(0.,0.,4.), _nnodes=Numbers(20,10,40), _domain_name="Omega"), _hexahedron, 2, _gmsh);
```

This is described in more detail in next paragraph.

Moreover, it is possible to subdivide an existing mesh of order 1: a new mesh is created using the subdivision algorithm mentioned above. The corresponding constructor is defined as follows:

```cpp
// constructor from a mesh to be subdivided
Mesh (const Mesh & msh, Number nbsubdiv, Number order = 1);
```

The arguments are:

- `msh` is the input mesh object, i.e. the given mesh to be subdivided; it should consist of triangles, quadrangles, tetrahedra or hexahedra;
- `nbsubdiv` is the number of subdivisions to be performed;
- `order` is the order of the final mesh; its default value is 1.

Example.

```cpp
Mesh m1("mesh.msh", "My Mesh", msh);
Mesh subm1(m1, 2);
```

This builds a mesh `subm1` which is obtained by subdividing twice the mesh `m1`, itself read from the file “mesh.msh”.

Once a mesh is created, it is possible to get information about what it is made of using the function `printInfo`, which displays on the terminal general information about the mesh: characteristic data, domains, etc.:

```cpp
Mesh m1("mesh.msh", "My Mesh", msh);
m1.printInfo();
```

If you want to mesh a 2D geometry with segment elements, only borders will be meshed. The same goes for 3D geometries mesh with triangles or quadrangles.
5.3.1 Structured internal meshing tools: structured generator

When the **structured mesh generator** is chosen \((mg=_\text{structured})\), one can create a mesh of order 1:

- of a segment,
- of a parallelogram with triangles or quadrangles,
- of a parallelepiped with hexahedra.

One has to declare an object of type \textit{Geometry}, more precisely of one of its derived type \textit{Segment}, \textit{Parallelogram}, \textit{Rectangle}, \textit{Square}, \textit{Parallelepiped}, \textit{Cuboid} or \textit{Cube} using one of these constructors, that allow in particular to specify the mesh refinement by setting the number of points (nodes) on each edge, including the two endpoints.

**Example 1.**

```
Strings sn(2);
sn[0] = "Sigma_1";
Mesh mesh1dP1(Segment(_xmin=0, _xmax=1, _nnodes=11, _side_names=sn), 1,
             _structured, "P1 mesh of [0,1], step=0.1");
```

This builds a mesh of the interval \([0,1]\) with 10 subintervals. The boundary domain \(\Sigma_1\), corresponding to the lower bound 0 of the interval, will be created; the other one will not be created since it has no name. The second argument is the mesh order; in the case of a structured mesh, the only possible value is 1.

It can be noticed that the segment may have been defined by two points in the plane or in the space as well.

**Example 2.**

```
Strings sn(4);
sn[0] = "Gamma_1"; sn[2] = "Gamma_2";
Mesh mesh2dP1(Rectangle(_xmin=0, _xmax=1, _ymin=1, _ymax=3, _nnodes=Numbers(3, 5), _side_names=sn), _triangle, 1, _structured, "P1 mesh of [0,1] \times [1,3]");
```

This builds a mesh of the rectangle \([0,1] \times [1,3]\) with triangles. The interval \([0,1]\) is subdivided into 2 subintervals; the interval \([1,3]\) is subdivided into 4 subintervals. Only the two domains \(\Gamma_1\) and \(\Gamma_2\) will be created. For a rectangle \([a,b] \times [c,d]\), the correspondence of the sidenames is the following:

- \(\text{sideNames}[0]\) is \([a,b] \times c\),
- \(\text{sideNames}[1]\) is \(b \times [c,d]\),
- \(\text{sideNames}[2]\) is \([a,b] \times d\),
- \(\text{sideNames}[3]\) is \(a \times [c,d]\).

**Example 3.**

```
Strings sn(4);
sn[0] = "Sigma_1"; sn[1] = "Sigma_2";
Mesh mesh2dQ1(Rectangle(_xmin=1, _xmax=2, _ymin=1, _ymax=3, _nnodes=Numbers(3, 5), _side_names=sn), _quadrangle, 1, _structured, "Q1 mesh of [1,2] \times [1,3]");
```

This builds a mesh of the rectangle \([1,2] \times [1,3]\) with quadrangles. Only the two domains \(\Sigma_1\) and \(\Sigma_2\) will be created. See example 2 for other commentaries.

**Example 4.**

```
Strings sn("z=1", "z=5", "y=1", "y=3", "x=0", "x=1");
Mesh mesh3dQ1(Cuboid(_xmin=0, _xmax=1, _ymin=1, _ymax=3, _zmin=1, _zmax=5, _nnodes=Numbers(3, 5, 9), _side_names=sn), _hexahedron, 1, _structured, "Q1 mesh of [0,1] \times [1,3] \times [1,5]");
```

99
This builds a mesh of the parallelepiped \([0, 1] \times [1, 3] \times [1, 5]\) with hexahedra. The interval \([0, 1]\) is subdivided into 2 subintervals; the interval \([1, 3]\) is subdivided into 4 subintervals; the interval \([1, 5]\) is subdivided into 8 subintervals. The 6 boundary domains will be created with their corresponding names. For a parallelepiped \([a, b] \times [c, d] \times [e, f]\), the correspondence of the sidenames is the following:

- \(\text{sideNames}[0] = [a, b] \times [c, d] \times e, \text{sideNames}[1] = [a, b] \times [c, d] \times f,\)
- \(\text{sideNames}[2] = [a, b] \times c \times [e, f], \text{sideNames}[3] = [a, b] \times d \times [e, f],\)
- \(\text{sideNames}[4] = a \times [c, d] \times [e, f], \text{sideNames}[5] = b \times [c, d] \times [e, f].\)

### 5.3.2 Unstructured internal meshing tools: subdivision generator

When the **subdivision algorithm** is chosen \((\text{mg} = \_\text{subdiv})\), one can create a mesh of any order based on the following volumetric geometries:

- the sphere meshed by tetrahedra,
- the cube meshed by tetrahedra or hexahedra,
- the cone or truncated cone, which may be a cylinder, meshed by tetrahedra or hexahedra.

The following surface geometries are also handled:

- the boundary of the sphere meshed by triangles,
- the boundary of the cube meshed by quadrangles,
- the boundary of the cone or truncated cone meshed by triangles or quadrangles,
- the disk meshed by triangles or quadrangles,
- mesh built from an initial set of triangles or quadrangles in 2D or 3D.

The principle is to start from an initial mesh, a kind of “seed” mesh, consisting of a set of elements, and build the mesh by subdividing each of them by cutting each edge in the middle until a prescribed so-called “subdivision level” is reached. A subdivision level equal to 0 gives a mesh reduced to the initial mesh. A triangle and a quadrangle is subdivided into 4 pieces; a tetrahedron and a hexahedron is subdivided into 8 pieces. Thus, at each subdivision, the number of elements of the mesh is multiplied by 4 for a surface mesh, by 8 for a volumetric one, and the characteristic dimension of the elements is halved.

One has to declare an object of type **Geometry**, more precisely of type **Sphere**, **Ball**, **Cube**, **RevTrunk**, **RevCone**, **RevCylinder**, **Disk** or **SetOfElems** using one of the available constructors, e.g.:

```cpp
SetOfElems(const std::vector<Point>& pts, const std::vector< std::vector< number_t > >& elems, const std::vector< std::vector< number_t > >& bounds, const ShapeType esh, const number_t nbsubdiv=1);
```
Mesh of a ball (or sphere) with tetrahedra

The seed of the mesh consists of a unique tetrahedron inside each octant of the cartesian axes. We can choose the number of octants to be taken into account, from 1 to 8, to mesh different portions of the sphere. The figure 5.13 shows this in the case of a subdivision level equal to 2. Each figure corresponds to the result of the following code, using the unit sphere and \textit{nboctants} varying from 1 to 8:

\begin{verbatim}
  order = 1, nbpts=5, meshType = 1;
  Ball sph(_center=Point(0.,0.,0.), _radius=1., _nboctants=nboctants,
           _nnodes=nbpts, _type=meshType);
  Mesh m(sph, _tetrahedron, order, _subdiv);
\end{verbatim}

Each color corresponds to a different boundary domain. The default value of the argument meshType is 1; setting it to 0 leads to a so-called flat mesh, where the points created during the algorithm are not projected onto the sphere, thus keeping the shape of the initial mesh. We can see the effect of this choice on figure 5.14.

![Volumic meshes of the different portions of the ball according to the number of octants.](image)

Figure 5.13: Volumic meshes of the different portions of the ball according to the number of octants.
Figure 5.14: Volumic meshes of the different portions of the “flat ball” according to the number of octants.

If we specifically want the mesh inside the whole sphere, it can be useful to start from an icosahedron because of its geometric properties, which lead to a more isotropic mesh than the one based on the 8 octants of the space. On the other hand, there will be no interface planes defined. In order to activate this option, the argument nboctants should simply be set to 0. The following code gives the figure 5.15, which shows both the round and flat results of the algorithm:

```plaintext
nboctants = 0, nbpts=5; meshType = 1;
Ball sph(_center=Point(0.,0.,0.), _radius=1., _nboctants=nboctants, _nnodes=nbpts, _type=meshType);
order = 1;
Mesh m(sph, _tetrahedron, order, _subdiv);

meshType = 0;
Ball sph2(_center=Point(0.,0.,0.), _radius=1., _nboctants=nboctants, _nnodes=nbpts, _type=meshType);
Mesh m2(sph2, _tetrahedron, order, _subdiv);
```

Figure 5.15: Volumic meshes starting from an icosahedron.
Mesh of a cube with tetrahedra or hexahedra

The selection of the octants is also used in the case of the cube as shown on the figure 5.16, which is the result of the following code, with no subdivision and \( \textit{nboctants} \) varying from 1 to 8:

```
order = 1, nbpts=2;
Cube cube(_center=Point(0.,0.,0.), _length=2., _nboctants=nboctants, _nnodes=nbpts);
Mesh m(cube, _hexahedron, order, _subdiv);
```

Notice that the edge length of the total cube is 2, so that the cube in the first octant is the so-called unit cube. Apart the choice of the mesh element (tetrahedron or hexahedron), the main interest of this case is the easy creation of a L-shape domain (3 octants) and the Fichera corner (7 octants), classical benchmark problem in the analysis of corner and edge singularities. It is shown on figure 5.16 in an unusual position; in order to put the missing cube in the first octant, one must apply a rotation, which is done by the following code:

```
order = 1, nboctants=7, nbpts=2;
Cube cube(_center=Point(0.,0.,0.), _length=2., _nboctants=nboctants, _nnodes=nbpts); cube.rotate3d(Point(0.,0.,0.), 1.,0.,0., pi);
Mesh m(cube, _hexahedron, order, _subdiv);
```

The two additional arguments define the rotation of angle 180 degrees around the first axis (X-axis); the result is shown on figure 5.16, at the last position (bottom right). If necessary, one can specify one or two more rotations in the form (angle, naxis). The angle is to be given in degrees and naxis defines the rotation axis: it is the number of the absolute axis, thus 1, 2 or 3.
Figure 5.16: Volumic meshes of the different portions of the cube according to the number of octants.

Mesh of a cylinder with tetrahedra or hexahedra

The subdivision algorithm can handle the case of a cylinder of revolution, whose axis is defined by two points P1 and P2, and delimited by the two planes containing the two points and orthogonal to the axis. As an example, we consider the “unit” cylinder of radius 1 and height 1. The following code produces the first two meshes shown on figure 5.17:

```plaintext
radius = 1.;
nbpts = 3;
Point P1(0., 0., 0.), P2(0., 0., 1.);
RevCylinder cyl1(_center1=P1, _center2=P2, _radius=radius, _nnodes=nbpts);
order = 1;
Mesh mT(cyl, _tetrahedron, order, _subdiv);
Mesh mH(cyl, _hexahedron, order, _subdiv);
```
Figure 5.17: Volumic meshes of the “unit” cylinder with tetrahedra and hexahedra.

Obviously, this is a poor approximation of the cylinder. To get a more accurate description, the user can then increase the number of elements (greater value of nbsubdiv) or increase the approximation order (or both).

In the case of a tetrahedron mesh, each end-plane may be covered by a “hat”, that is to say a solid whose shape may be a cone or an ellipsoid. The last drawing of figure 5.17 shows such a configuration, with an ellipsoid on the side of P1 (keyword _gesEllipsoid) whose apex is at radius/2 from the basis of the cylinder, and a cone on the side of P2 (keyword _gesCone) whose apex is at radius from the other basis of the cylinder. It is obtained by the following code:

```
radius=1.;
nbpts=5;
RevCylinder cyl2e(_center1=Point(0.,0.,0.) , _center2=Point(0.,0.,1.) ,
    _radius=radius , _end1_shape=_gesEllipsoid , _end1_distance=radius/2 ,
    _end2_shape=_gesCone , _end2_distance=radius , _nnodes=nbpts);
order=1;
Mesh P1VolMeshTetCylE(cyl2e , _tetrahedron , order , _subdiv);
```

Two other keywords exist: _gesNone and _gesFlat. They have an equivalent meaning in the case of a solid body. They are the default value and indicate that no “hat” should be added at the corresponding end.

**Mesh of a cone or a truncated cone with tetrahedra or hexahedra**

A truncated cone of revolution is defined by an axis, given by two points P1 and P2, delimited by the two planes containing the two points and orthogonal to the axis. The two circular sections are defined by two radii. The following code produces the first two meshes shown on figure 5.18:

```
nbpts=5;
radius1=0. , radius2=1.;
Point P1(-1.,-1.,0.) , P2(0.,0.,2.) ;
RevCone cone(_center=P2, _radius=radius2 , _apex=P1 , _nnodes=nbpts);
order=1;
Mesh mT(cone , _tetrahedron , order , _subdiv);
```

```
radius1=0.5;
RevTrunk cone2(_center1=P1, _radius1=radius1 , _center2=P2 , _radius2=radius2 ,
    _nnodes=nbpts);
Mesh mH(cone2 , _hexahedron , order , _subdiv);
```
The number of slices is 0, which means that a suitable default value is automatically computed from the length of the axis and the radii. The first object is a "true" cone since one radius is 0; it can be meshed exactly with tetrahedra. Using hexahedra for this geometry is not advised since the elements will be degenerated at the apex of the cone. Moreover, the radius cannot be 0, it should be at least 1.e-15, leading to a "near true" cone, but with highly degenerated hexahedra close to the apex. Hexahedra are more suitable to build a truncated cone; an example is shown on the middle drawing of the figure 5.18.

Figure 5.18: Volumic meshes of the cone and truncated cone with tetrahedra and hexahedra.

The following code produces the last mesh shown on figure 5.18:

```plaintext
nbpts = 5;
radius1 = 0.6, radius2 = 1;
RevTrunk conel(_center1=Point(-1, -1, 0), _radius1=radius1,
    _center2=Point(0, 0, 2), _radius2=radius2, _end1_shape=gesCone,
    _end1_distance=1.5, _end2_shape=gesEllipsoid, _end2_distance=0.7,
    _nnodes=nbpts);
order = 1;
Mesh mTE(conel, _tetrahedron, order, _subdiv);
```

In the same way as for the cylinder, the truncated cone can be "covered" with a solid. This is only available for a mesh made of tetrahedra. We show a cone and an ellipsoid put at each end of a truncated cone, respectively on the side of P1 and on the side of P2.

**Mesh of a sphere with triangles**

The same logic described previously for a mesh of tetrahedra apply here for a mesh of triangles. The following code leads to meshes of the boundary of the unit sphere, and the result is shown on figure 5.19:

```plaintext
order = 1, nbpts=5;
Ball sph(_center=Point(0, 0, 0), _radius=1, _nboctants=nboctants,
    _nnodes=nbpts);
Mesh m(sph, _triangle, order, _subdiv);
```
Figure 5.19: Surfacic meshes of the different portions of the boundary of the sphere according to the number of octants.

Again, if the argument meshType is set to 0, we get the “flat” version of the meshes, i.e. the meshes obtained from the subdivision of the nboctants initial triangles (see figure 5.20).

Figure 5.20: Surfacic meshes of the different portions of the “flat sphere” according to the number of octants.

If we specifically want the mesh of the whole sphere, it can be useful to start from an icosahedron because of its geometric properties, which lead to a more isotropic mesh than the one based on the 8 octants of the space. On the other hand, there will be no interface planes defined.

In order to activate this option, the argument nboctants should simply be set to 0. The following code gives the figure 5.21, which shows both the round and flat results of the algorithm:

```plaintext
nboctants = 0, nbpts=5; meshType = 1;
```
Ball sph(_center=Point(0.,0.,0.), _radius=1., _nboctants=nboctants,
    _nnodes=nbpts, _type=meshType);
order = 1;
Mesh m(sph, _triangle, order, _subdiv);

meshType = 0;
Ball sph2(_center=Point(0.,0.,0.), _radius=1., _nboctants=nboctants,
    _nnodes=nbpts, _type=meshType);
Mesh m2(sph2, _triangle, order, _subdiv);

Figure 5.21: Surfacic meshes starting from an icosahedron.

**Mesh of a cube with quadrangles**

We can obtain the mesh of the surface of a cube, or part of it, with quadrangles, by using the same logic described just above for the sphere. Consider the following code:

```cpp
order = 1, nbpts=2;
Cube cube(_center=Point(0.,0.,0.), _length=2., _nboctants=nboctants,
    _nnodes=nbpts);
Mesh m(cube, _quadrangle, order, _subdiv);
```

Letting nboctants vary from 1 to 8, then 0, lead to the objects shown on figure 5.22 below.
Figure 5.22: Surfacic meshes of the different portions of the cube according to the number of octants.

The last object (nboctants = 0) is the simplest mesh of a cube made of 6 quadrangles (squares here). By subdividing it once, we get the previous yellow object (nboctants = 8) made of 24 quadrangles.

**Mesh of a cone or a truncated cone with triangles**

We can build a mesh of the surface of a truncated cone with triangles. The following code produces the first two drawings of the figure 5.23:

```plaintext
radius = 1.;
nbslices = 1, nbpts = 3;
Point P1(0., 0., 0.), P2(0., 0., 1.);
RevCylinder cyll(_center1=P1, _center2=P2, _radius=radius, _nnodes=nbpts);
order = 1;
Mesh mT(cyl, _triangle, order, _subdiv, fname);

nbpts = 5;
RevCylinder cylE(_center1=P1, _center2=P2, _radius=radius,
    _end1_shape=gesFlat, _end1_distance=0., _end2_shape=gesNone,
    _end2_distance=0., _nnodes=nbpts);
Mesh mTE(cylE, _triangle, order, _subdiv);
```

109
Figure 5.23: Surfacic meshes of a cylinder.

```plaintext
nbpts = 5;
radius1 = 0.5, radius2 = 1.;
Point P1(-1.,-1.,0.), P2(0.,0.,2.);
RevTrunk cone3(_center1=P1, _radius1=radius1, _center2=P2, _radius2=radius2,
   _end1_shape=gesNone, _end1_distance=0., _end2_shape=gesFlat,
   _end2_distance=0., _nnodes=nbpts);
order=1;
Mesh mT(cone3, _triangle, order, _subdiv);

RevTrunk cone1(_center1=P1, _radius1=radius1, _center2=P2, _radius2=radius2,
   _end1_shape=gesCone, _end1_distance=1.5, _end2_shape=gesEllipsoid,
   _end2_distance=0.7, _nnodes=nbpts);
Mesh mTE(cone1, _triangle, order, _subdiv);
```

Figure 5.24: Surfacic meshes of a truncated cone.

**Mesh of a cone or a truncated cone with quadrangles**

We can build a mesh of the surface of a truncated cone with quadrangles. The following code produces the first two drawings of the figure 5.25:

```plaintext
nbpts=5;
radius1 = 0.5, radius2 = 1.;
Point P1(-1.,-1.,0.), P2(0.,0.,2.);
RevTrunk cone2(_center1=P1, _radius1=radius1, _center2=P2, _radius2=radius2,
   _nnodes=nbpts);
```
order = 1;
.Mesh mQ(cone2, _quadrangle, order, _subdiv);
.RevTrunk cone3(_center1=P1, _radius1=radius1, _center2=P2, _radius2=radius2, _end1_shape=gesNone, _end1_distance=0., _end2_shape=gesFlat, _end2_distance=0., _nnodes=nbpts);
.Mesh mQF(cone3, _quadrangle, order, _subdiv, "mQF");
mQF.printInfo();

The left truncated cone is opened at both ends (this is the default); thus it has two boundaries shown in green (bottom) and orange (top). The object cone2 is the same as the one used previously to make the mesh of hexahedra (see figure 5.18).

The second drawing shows the same object bearing a “lid” on its top (on the side of P2). Indeed, such a truncated cone may be closed at one or both ends by a plane “lid”. This is obtained by specifying the geometric end shape to be used at each end: _gesNone means that the cone is left opened, which is the default behaviour, and _gesFlat means that a plane “lid” is requested. The object proposed in this example has thus one boundary at its other end (on the side of P1), shown as an orange line.

In both cases, the requested number of slices is 0; thus, the algorithm decided to create two slices displayed in magenta and yellow. The result of the instruction mQF.printInfo(); is the following:

Mesh ‘mQF’ (cone - Quadrangle mesh)
space dimension : 3, element dimension : 2
Geometry of shape type revolution volume based on cone of dimension 3,
BoundingBox [-1.45412,0.908248]x[-1.45412,0.908248]x[-0.288675,2.57735], names of variable : x, y, z
number of elements : 208, number of vertices : 217, number of nodes : 217, number of domains : 5
domain number 0: Omega (whole domain)
domain number 1: Sigma_1 (End subdomain on the side of end point 2)
domain number 2: Sigma_2 (Slice 1)
domain number 3: Sigma_3 (Slice 2)
domain number 4: Kappa_1 (Boundary: End curve on the side of end point 1)

Figure 5.25: Surfacic meshes of a cone and a cylinder.

The last drawing shows the surfacic mesh of a cylinder, since it is a particular kind of cone. The cylinder is the same as the one shown on figure 5.17. The code that produces this mesh is:

.radius = 1.;
.nbpts = 3;
.Point P1(0.,0.,0.), P2(0.,0.,1.);
.RevCylinder cyl(_center1=P1, _center2=P2, _radius=radius, _nnodes=nbpts);
.order = 1;
.Mesh mQCyl(cyl, _quadrangle, order, _subdiv);
The cylinder is opened at both ends and thus has two boundaries shown as a green line and an orange line.

**Mesh of a disk or a part of a disk**

We can build the mesh of a disk or a portion of a disk, with triangles or quadrangles. Using the following code, we get the result shown on figure 5.26.

```plaintext
radius = 2.;
nbpts = 5, order = 1;

Disk pdisk(_center=Point(0.,1.), _radius=radius, _angle1=10., _angle2=300.,
_nnodes=nbpts);

Mesh meshTriDisk(pdisk, _triangle, order, _subdiv);
Mesh meshQuaDisk(pdisk, _quadrangle, order, _subdiv);
```

![Figure 5.26: Meshes of a portion of disk with triangles and quadrangles.](image)

**Mesh from a set of triangles or quadrangles**

This possibility is designed to build a mesh starting from an elementary set of elements. Generally, this initial mesh is build “manually”. This gives a flexible mean to create a mesh which cannot be obtained with another constructor, but without having to resort to the help of a more complicated solution (like an external mesh generator in particular).

Such meshes can be made in 2D or in 3D with triangles or quadrangles. Figure 5.27 shows two examples, in 3D with triangles, in 2D with quadrangles on a domain with a hole.
The program that produces it looks like the following:

```c
order = 1, nbsubdiv = 1;
SetOfElems sot(tpts, telems, tbounds, _triangle, nbsubdiv);
Mesh mT(sot, _triangle, order, _subdiv);
SetOfElems soq(qpts, qelems, qbounds, _quadrangle, nbsubdiv);
Mesh mQ(soq, _quadrangle, order, _subdiv);
```

The mesh of triangle is based on an initial set of 2 triangles \{1,2,3\} and \{1,4,2\}, stored in the vector elems. The 4 points are Point(0.,0.,0.), Point(1.,0.,0.), Point(0.,1.,0.3), Point(0.,-1.,0.3) stored in the vector tpts. Four boundaries are defined. A boundary is simply defined by the list of point numbers lying on it, in any order. Thus, here, the four boundaries are \{1,4\}, \{4,2\}, \{2,3\} and \{1,3\}; they are stored in the vector tbounds. The same apply for the set of quadrangles.

### 5.3.3 Meshing tool with nested call to GMSH: gmsh generator

Using the GMSH interface to define meshes allows you to define more canonical geometries than both previous generators:

- segments, ellipses, circles, elliptic or circular arcs as 1D geometries
- quadrangles, rectangles, squares, disks, elliptical surfaces, spheres, ellipsoids, triangles as 2D geometries with either triangular or quadrangular mesh elements.
- hexahedron, parallelepipeds, cubes, balls, tetrahedron, cylinders, prisms, pyramids as 3D geometries with either tetrahedral or hexahedral mesh elements.

When you use it, 2 files will be generated in your directory:

**xlifepp_script.geo** This is the input file of GMSH. To simplify its write, we developed a macro file includes in this one. If you look at this file, you will find a very elegant way to define meshes with GMSH.

**xlifepp_script.msh** This is the real mesh file, generated by a system call to GMSH from the .geo file. This file is loaded by XLiFE++. 

Figure 5.27: Meshes from initial set of triangles and quadrangles.
Next to this, you can define 2 types of complicated geometries: the so-called "composite" and "loop" geometries.

If you want to define a geometry that XLiFE++ can not directly handle, you can use GMSH directly.

**Examples of composite and loop geometries**

Please see subsection 5.1.9 for definition of composite geometries and the use of operators + and -, and see subsection 5.1.8 for definition of loop geometries and the use of `surfaceFrom` and `volumeFrom` routines.

Let’s see a first example of an ellipse inside a rectangle:

```plaintext
Rectangle  r( _xmin=-3, _xmax=3, _ymin=-2, _ymax=2, _nnodes=Numbers(33,22),
           _domain_name="Omega" );
Ellipse  e( _center=Point( 0, 0 ), _xlength=1, _ylength=0.5, _nnodes=11 );
Mesh  m1(r-e, _triangle, 1, _gmsh);
Mesh  m2(r+e, _triangle, 1, _gmsh);
```

![Figure 5.28: GMSH view of m1 and m2](image-url)

114
The following example shows how it works in 3D with a parallelepiped hole inside an ellipsoid:

```plaintext
Ellipsoid ed1(_center=Point(0.,0.,0.), _v1=Point(3.,0.,0.),
    _v2=Point(0.,2.,0.), _v3=Point(0.,0.,1.), _nnodes=16);
Parallelepiped pal(_v1=Point(-0.5,-0.5,-0.5), _v2=Point(0.5,-0.5,-0.5),
    _v4=Point(-0.5,0.5,-0.5), _v5=Point(-0.5,-0.5,0.5), _nnodes=3);
Mesh mesh3dP1Composite(ed1=pal, _tetrahedron, 1, _gmsh);
```

Figure 5.29: Gmsh view of a 3d composite geometry (ellipsoid - parallelepiped)

Figure 5.30: Gmsh view of the hole of a 3d composite geometry (ellipsoid - parallelepiped)

Let’s see now an example with more than 2 components:
Strings sn ("Gamma_1", "Gamma_2", "Gamma_3", "Gamma_4");

Ellipse e1 (_center=Point(0.,0.), _v1=Point(4.,0.), _v2=Point(0.,5.), _nnodes=12, _domain_name="Omega1", _side_names=sn); sn[0]="Gamma_5"; sn[1]="Gamma_6"; sn[2]="Gamma_7"; sn[3]="Gamma_8";


Ellipse e2 (_center=Point(1.,2.), _v1=Point(1.5,2.), _v2=Point(1.,3.), _nnodes=12, _side_names=sn); sn[0]="Gamma_13"; sn[1]="Gamma_14"; sn[2]="Gamma_15"; sn[3]="Gamma_16";


Disk d1 (_center=Point(5.5,0.5,0.), _v1=Point(5.7,0.5,0.), _v2=Point(5.5,0.7,0.), _nnodes=12, _side_names=sn); Mesh mesh2dP1Composite((e1+r1)-(e2+e3)+r2-d1, _triangle, 1, _gmsh);

Figure 5.31: Gmsh view of complex 2d composite geometry

Let’s take an example using segments and circle arcs to define a mesh on a rectangle with rounded corners:

Point a(-1.5,-4.); Point b(1.5,-4.); Point c(2.,-3.5); Point d(2.,3.5);
Point e(1.5,4.); Point f(-1.5,4.); Point g(-2.,3.5); Point h(-2.,-3.5);
Segment s1 (_v1=a, _v2=b, _nnodes=21, _domain_name="AB");
CircArc c1 (_center=Point(3.5,0.5), _v1=b, _v2=c, _nnodes=5, _domain_name="BC");
Segment s2 (_v1=c, _v2=d, _nnodes=11, _domain_name="CD");
CircArc c2 (_center=Point(3.5,1.5), _v1=d, _v2=e, _nnodes=5, _domain_name="DE");
Segment s3 (_v1=e, _v2=f, _nnodes=21, _domain_name="EF");
CircArc c3 (_center=Point(0.5,1.5), _v1=f, _v2=g, _nnodes=5, _domain_name="FG");
Segment s4 (_v1=g, _v2=h, _nnodes=11, _domain_name="GH");
CircArc c4 (_center=Point(0.5,0.5), _v1=h, _v2=a, _nnodes=5, _domain_name="HA");
Mesh mesh2dP1Loop(planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4), _triangle, 1, _gmsh);

Figure 5.32: Gmsh view of rectangular geometry with rounded corners, defined with the surfaceFrom routine

You can define composite geometries using loop geometries. Let’s take the complex 2d composite example in which we will replace the first rectangle by a rounded rectangle and the disk by a half disk, both defined by the surfaceFrom routine.

Ellipse e1(_center=Point(0.,0.), _v1=Point(4.,0.), _v2=Point(0.,5.), _nnodes=12, _domain_name="Omega1");
Point a(-1.5,-4.); Point b(1.5,-4.); Point c(2.,-3.5); Point d(2.,3.5);
Point e(1.5,4.); Point f(-1.5,4.); Point g(-2.,3.5);
Segment s1(_v1=a, _v2=b, _nnodes=21, _domain_name="AB");
CircArc c1(_center=Point(3.5,0.5), _v1=b, _v2=c, _nnodes=5, _domain_name="BC");
Segment s2(_v1=c, _v2=d, _nnodes=11, _domain_name="CD");
CircArc c2(_center=Point(3.5,1.5), _v1=d, _v2=e, _nnodes=5, _domain_name="DE");
Segment s3(_v1=e, _v2=f, _nnodes=21, _domain_name="EF");
CircArc c3(_center=Point(0.5,0.5), _v1=f, _v2=g, _nnodes=5, _domain_name="FG");
Segment s4(_v1=g, _v2=h, _nnodes=11, _domain_name="GH");
CircArc c4(_center=Point(0.5,0.5), _v1=h, _v2=a, _nnodes=5, _domain_name="HA");
Geometry sf1=(surfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4,"Omega2");
Ellipse e2(_center=Point(1.,2.), _v1=Point(1.5,2.), _v2=Point(1.,3.), _nnodes=12, _domain_name="Omega3");
Ellipse e3(_center=Point(0.,0.), _v1=Point(0.5,0.), _v2=Point(0.,1.), _nnodes=12, _domain_name="Omega4");
Rectangle r2(_xmin=5., _xmax=6., _ymin=0., _ymax=1., _nnodes=6, _domain_name="Omega5");
Segment s5(_v1=Point(5.3,0.5), _v2=Point(5.7,0.5), _nnodes=5);
CircArc c5(_center=Point(5.5,0.5), _v1=Point(5.7,0.5), _v2=Point(5.5,0.7), _nnodes=5);
CircArc c6(_center=Point(5.5,0.5), _v1=Point(5.5,0.7), _v2=Point(5.3,0.5), _nnodes=5);
Geometry sf2=planeSurfaceFrom(s5+c5+c6,"Omega6");
Mesh mesh2dP1Composite1((e1+sf1)-(e2+e3)+r2-sf2, _triangle, 1, _gmsh);
Finally, let’s see now an example of complex composite geometry with use of forcing inclusion. Components are defined in example above:

```cpp
Mesh mesh2dP1Composite3(e1+(sf1+((e2+e3))+r2+sf2 , _triangle , 1 , _gmsh);
```

5.4 Loading a mesh from a file

XLiFE++ allows you to read various mesh file formats. The constructor to use is defined as follows:

```cpp
// Constructor from a file
Mesh (const String& filename , const String& meshname , IOFormat mft , Number nodesDim);
```

The arguments are:

- `filename` is the name of the mesh file,
- `meshname` is the name of the mesh, for log purpose. Default value is empty string.
- `mft` defines the mesh format. It can take four values as we can see on the examples hereafter.
nodesDim defines minimal number of coordinates of each vertex. Default is 0 for automatic behavior. Normally, you should not have to use this argument.

```cpp
// loading a VTK mesh file
Mesh m1("mesh.vtk", "My Mesh M1", vtk);
// loading a VTU mesh file
Mesh m2("mesh.vtu", "My Mesh M2", vtu);
// loading a GMSH mesh file
Mesh m3("mesh.msh", "My Mesh M3", msh);
// loading a GMSH script file
Mesh m4("mesh.geo", "My Mesh M4", geo);
// loading a MELINA mesh file
Mesh m5("mesh.mel", "My Mesh M5", mel);
// loading a PLY mesh file
Mesh m6("mesh.ply", "My Mesh M6", ply);
```

which create six `Mesh` objects called m1, m2, m3, m4, m5 and m6.

- To have more information about the VTK and VTU file formats, please go to [http://www.paraview.org](http://www.paraview.org)
- The MELINA file format is the input format of the MELINA finite element library, ancestor of XLiFE++. For more information, please go to [http://anum-maths.univ-rennes1.fr/melina/danielmartin/melina/](http://anum-maths.univ-rennes1.fr/melina/danielmartin/melina/)
- To have more information on the PLY file format, please go to [http://paulbourke.net/dataformats/ply](http://paulbourke.net/dataformats/ply).
- To have more information about Gmsh, please go to [http://geuz.org/gmsh/](http://geuz.org/gmsh/). You will have everything you need about the msh format and about the geo scripts.

If you load a geo file, XLiFE++ will call Gmsh to create the corresponding msh file, which is then read. Consequently, Gmsh needs to be installed on your computer and the executable file, called `gmsh`, should be found through your PATH environment variable. If Gmsh is installed after XLiFE++, XLiFE++ needs to be reinstalled.

### 5.5 Transformations on meshes

Geometrical transformations on meshes work as on geometries. Please see section 5.2 for definition and use of transformations routines.

Then, if you want to apply a transformation and modify the input object, you can use one of the following functions:

```cpp
// apply a geometrical transformation on a Mesh
Mesh::transform(const Transformation& t);
// apply a translation on a Mesh
Mesh::translate(std::vector<Real> u = std::vector<Real>(3,0.));
Mesh::translate(Real ux, Real uy = 0., Real uz = 0.);
```
However, if you want now to create a new Mesh by applying a transformation on a Mesh, you should use one of the following functions instead:

For instance:

```cpp
Mesh m;
  m.translate(0.,0.,1.);
```

However, if you want now to create a new Mesh by applying a transformation on a Mesh, you should use one of the following functions instead:

```cpp
// apply a rotation 2d on a Mesh
Mesh::rotate2d(const Point& c = Point(0.,0.), Real angle = 0.);
// apply a rotation 3d on a Mesh
Mesh::rotate3d(const Point& c = Point(0.,0.,0.), std::vector<Real> u = std::vector<Real>(3.,0.), Real angle = 0.);
Mesh::rotate3d(Real ux, Real uy, Real angle);
Mesh::rotate3d(const Point& c, Real ux, Real uy, Real uz, Real angle);
Mesh::rotate3d(const Point& c, Real ux, Real uy, Real uz, Real angle);
// apply a homothety on a Mesh
Mesh::homothetize(const Point& c = Point(0.,0.,0.), Real factor = 1.);
Mesh::homothetize(Real factor);
// apply a point reflection on a Mesh
Mesh::pointReflect(const Point& c = Point(0.,0.,0.));
// apply a reflection 2d on a Mesh
Mesh::reflect2d(const Point& c = Point(0.,0.), std::vector<Real> u = std::vector<Real>(2.,0.));
Mesh::reflect2d(const Point& c, Real ux, Real uy = 0.);
// apply a reflection 3d on a Mesh
Mesh::reflect3d(const Point& c = Point(0.,0.,0.), std::vector<Real> u = std::vector<Real>(3.,0.));
Mesh::reflect3d(const Point& c, Real ux, Real uy, Real uz = 0.);
```
Mesh reflect3d(const Mesh & m, const Point & c, Real ux, Real uy, Real uz = 0.);

For instance:

Mesh m1;
Mesh m2 = translate(m1, 0.0, 0.0, 1.0);

Applying a transformation on a Mesh object means applying the transformation on the underlying Geometry object and adding the suffix "_prime" to the mesh name and the domain names.

5.6 Using geometrical domain

Related to mesh, the geometric domains are fundamental objects because they are the support of integrals involved in variational problem. These domains are defined by mesh tools, using names and sidenames in definition of geometries or given as physical domain in 'geo' file.

5.6.1 Retrieving domains

In order to be used in program, the domains have to be 'retrieved' as Domain object from mesh:

Strings sn("y=0", "y=1", "x=0", "x=1");
Mesh mesh2d(Square(_origin=Point(0.0,0.), _length=1, _nnodes=20,
                  _side_names=sn), _triangle, 1, _structured);

Domain omega=mesh2d.domain("Omega");
Domain sigmaM=mesh2d.domain("x=0");
Domain sigmaP=mesh2d.domain("x=1");
Domain gammaM=mesh2d.domain("y=0");
Domain gammaP=mesh2d.domain("y=1");

By default, "Omega" is the string name of the main domain of mesh.

It is possible to rename a domain of a mesh:

Strings sn("", "", "x=0", "x=1/2-");
Mesh mesh2d(Rectangle(_origin=Point(0.0,0.), _xlength=0.5, _ylength=1,
                       _nnodes=Numbers(20,40), _side_names=sn), _triangle, 1, _structured);
mesh2d.renameDomain("Omega", "Omega-");

mesh2d.renameDomain("Omega", "Omega+");

In this example, the unit square is split in two domains $\Omega^+$ and $\Omega^-$ using the merge and renameDomain functions. Note that the merging process of meshes concatenates 'same' domain in a new one named "name1 or name2".

5.6.2 Dealing with normals of a domain

Normal vectors may be required in many variationnal forms, in particular when using BEM like methods. In (bi)linear forms, they appears with symbolic names \_n, \_nx, \_ny that corresponds
to real normal vectors. The question is which normal vectors are selected by XLiFE++.

Note that the normal vectors of a domain, say $\Gamma$, are computed only if the domain is a manifold, say a surface/curve domain in a 3d/2d space. If they are required, they are automatically computed with the following default rules:

- if $\Sigma$ is a boundary (or a part of) of a unique domain $\Omega$, the selected normals are the outwards vectors to $\Omega$
- if $\Sigma$ is a boundary between two domains (an interface), the selected normals are the towards infinite vectors
- if $\Sigma$ is not a boundary, say an immersed manifold, the selected normals are the towards infinite vectors

![Diagram](image)

Manifold: towards infinite normal  Boundary: outwards $\Omega$ normal  Interface: towards infinite normal

When the boundary or the manifold is not closed, the normal orientations are consistent but the selected orientation is not predictable.

The user can modify the normal orientation by using the function `setNormalOrientation(OrientationType,[Domain])` where `OrientationType` has one of the following values:

- `_undefOrientationType`  // default rules are applied
- `_towardsInfinite, _outwardsInfinite`  // for any side domain
- `_towardsDomain, _outwardsDomain`  // for any boundary or interface

To change the normal orientation of a side domain $\Sigma$, write for instance

```
Sigma.setNormalOrientation(_towardsInfinite);  //towards infinite normals
Sigma.setNormalOrientation(_outwardsDomain);  //outwards normals, //UNSAFE for an interface!
Sigma.setNormalOrientation(_outwardsDomain, Omega); //outwards normals to Omega
```
5.6.3 Map of domains

Some processes require a geometric map between two domains. For instance, to deal with periodic condition related to two side domains:

\[ u|_{\Sigma^+} = u|_{\Sigma^-} \]

the elimination process uses the geometric map \( F: \Sigma^+ \rightarrow \Sigma^- \). The simple way to define such map is the following:

```cpp
Reals mapPM(const Point& P, Parameters& pa=defaultParameters) {
    Point Q(P);
    Q(1)=1;
    return Q;
}
```

```cpp
defineMap(sigmaP, sigmaM, mapPM);
```

Note that the `mapPm` function returns a `Vector<Real>` which is more general than a `Point`. Respect this prototype!

It is currently not possible to define two different maps for a pair of domains.

5.6.4 Assign properties to domains

In some problems, physical properties may be different from a domain to other one. This may be managed by differentiating integrals in variational formulation:

\[
\int_{\Omega_1} \rho_1(x)u(x)v(x) + \int_{\Omega_2} \rho_2(x)u(x)v(x).
\]

But it may be too intricate if there are a lot of domains or integrals. So there is an alternative method consisting in defining a unique function \( \rho \) and deal with a unique integral:

\[
\int_{\Omega} \rho(x)u(x)v(x)
\]

and assign id to domains that are subdomains:

```cpp
Real rho(const Point&P, Parameters& pars=defaultParameters) {
    Number mat=materialIdFromParameters(pars);
    if(mat==1) return ... else return ... };
```

```cpp
Domain omega=mesh2d.domain("omega"); // whole domain
Domain omega1=mesh2d.domain("omega_1"); // subdomain
Domain omega2=mesh2d.domain("omega_2"); // subdomain
omega1.setMaterialId(1);
omega2.setMaterialId(2);
```
### 5.6.5 Cracking a Domain

Theoretically, **Gmsh** allows you to crack domains (1D cracks in 2D meshes, 1D or 2D cracks in 3D meshes). Cracks can be opened or not. A crack is opened when some boundary nodes of the domain to crack are duplicated as the other nodes, else it will be a closed crack.

To notify that the segment has to be cracked, you just call the `crack` routine on it. This is a general routine defining both opened and closed cracks through 2 additional optional arguments. Default behavior is closed cracks. You can call the routine `closedCrack` (only the geometry in argument) to define a closed crack. You can call the routine `openCrack` (the geometry and a domain name) to define an opened crack. In this case, the domain name is the boundary domain of the geometry you want to crack that will be opened. Let’s see following examples to understand this.

There are two ways to define a geometry with a crack inside it: the direct one and the indirect one.

#### Defining cracks directly

This way is the way you should always do to define a crack. A crack is a geometry inside a geometry of bigger dimension. So the geometry to be cracked must be defined as a meshed "hole" inside the container geometry.

```plaintext
Point x1 (0, 0, 0), x2 (1, 0, 0), x3 (1, 1, 0), x4 (0, 1, 0),
    x5 (0.2, 0.2, 0), x6 (0.8, 0.8, 0), x7 (0.2, 0, 0),
    x8 (0.8, 1, 0);
Rectangle rect8 (v1=x1, v2=x2, v4=x4,
    _domain_name=“Omega”, _side_names=“Gamma”);
Segment scrack (v1=x5, v2=x6, _nnodes=3,
    _domain_name=“Crack”, _side_names=“Sigma”);
openCrack(scrack, “Sigma”);
Mesh m(rect8+scrack, _triangle, 1, _gmsh);
```

Here, it is an opened crack. A side name is given to both ends of the segment. This name will be given to the routine `openCrack` to tell which ends are to be opened. Here, it is both.

#### Defining cracks indirectly

This way is called indirect, compared to the previous one, insofar as you have to link the geometry you want to crack to the boundaries of the parent geometry and define surfaces from their boundaries:
Point \( x_1(0,0,0), x_2(1,0,0), x_3(1,1,0), x_4(0,1,0), \)
\( x_5(0.2,0.2,0), x_6(0.8,0.8,0), x_7(0.2,0.8,0), \)
\( x_8(0.8,1,0); \)
Segment \( s_1(\_v1=x_1, \_v2=x_7, \_domain\_name="Gamma"); \)
Segment \( s_2(\_v1=x_2, \_v2=x_7, \_domain\_name="Gamma"); \)
Segment \( s_3(\_v1=x_3, \_v2=x_2, \_domain\_name="Gamma"); \)
Segment \( s_4(\_v1=x_8, \_v2=x_3, \_domain\_name="Gamma"); \)
Segment \( s_5(\_v1=x_8, \_v2=x_4, \_domain\_name="Gamma"); \)
Segment \( s_6(\_v1=x_4, \_v2=x_1, \_domain\_name="Gamma"); \)
Segment \( s_7(\_v1=x_7, \_v2=x_5); \)
Segment \( s_8(\_v1=x_5, \_v2=x_6, \_nnodes=3, \_domain\_name="Crack"); \)
Segment \( s_9(\_v1=x_6, \_v2=x_8); \)
\texttt{crack(s8);} \)
Geometry sf1=\texttt{surfaceFrom}(s7+s8+s9+s5+s6+s1,"Omega1");
Geometry sf2=\texttt{surfaceFrom}(s7+s8+s9+s4+s3+s2,"Omega2");
Mesh m(sf1+sf2, _triangle, 1, _gmsh);

Here, it is a closed crack.

\begin{itemize}
  \item In this example, surfaces have different domain names. You can also give the same domain name
\end{itemize}

Which way is better ?

<table>
<thead>
<tr>
<th></th>
<th>direct way</th>
<th>indirect way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D crack in 2D mesh</td>
<td>100% safe</td>
<td>100% safe</td>
</tr>
<tr>
<td>2D crack in 3D mesh</td>
<td>not 100% safe</td>
<td>100% safe</td>
</tr>
<tr>
<td>1D crack in 3D mesh</td>
<td>to be tested</td>
<td>to be tested</td>
</tr>
</tbody>
</table>

\begin{itemize}
  \item Gmsh team is currently working on improving their crack engine to be 100% whatever the case.
\end{itemize}

A look at the mesh file

Let’s see the resulting mesh file for the indirect example above:

\begin{verbatim}
$MeshFormat
2 2 0 8
$EndMeshFormat
$PhysicalNames
4
1 1 "Crack"
1 2 "Gamma"
2 3 "Omega1"
2 4 "Omega2"
$EndPhysicalNames
$Nodes
18
\end{verbatim}

125
Bounds of the cracked domain are not duplicated (nodes 1 and 2), whereas the middle node of the cracked domain is duplicated (nodes 3 and 10)
Segments are also duplicated. If you’re familiar with the msh file format, by reading elements 11 and 24 for instance, we can deduce that domain "Omega1" has geometrical reference 7 and domain "Omega2" has geometrical reference 11. These references will be used with the cracked domain name to name sides of the crack, namely "Crack_7" and "Crack_11".

### 5.7 A full example with periodic cavities

You want to mesh a rectangular domain, but on the bottom side of the rectangle, you want to have periodic cavities with the following pattern:
First, you have to define the first cavity, according to the previous figure:

```plaintext
// Definition of the cavity
Real l1=0.06, l2=0.04, l3=0.08, h1=0.05, h2=0.1, s=0.01;
Point po(1.,0.);
Point pa=po+Point(l1,0.);
Point pb=pa+Point(0.,h1);
Point pc=pb+Point(-l2,0.);
Point pd=pc+Point(0.,h2);
Point pe=pd+Point(2.*l2+l3,0.);
Point pf=pe+Point(0.,-h2);
Point pg=pf+Point(-l2,0.);
Point ph=pg+Point(0.,-h1);
Point pi=ph+Point(l1,0.);
Segment s1(_v1=po, _v2=pa, _hsteps=s), s2(_v1=pa, _v2=pb, _hsteps=s),
s3(_v1=pb, _v2=pc, _hsteps=s), s4(_v1=pc, _v2=pd, _hsteps=s),
s5(_v1=pd, _v2=pe, _hsteps=s), s6(_v1=pe, _v2=pf, _hsteps=s),
s7(_v1=pf, _v2=pg, _hsteps=s), s8(_v1=pg, _v2=ph, _hsteps=s),
s9(_v1=ph, _v2=pi, _hsteps=s);
Geometry cavity=s1+s2+s3+s4+s5+s6+s7+s8+s9;
```

When done, you can define the other cavities as results of translations of the first cavity:

```plaintext
// Definition of the cavities
Real cL=2.*l1+l3; // cavity length
Number nbcav=10; // number of cavities
Geometry cavities=cavity;
for (Number n=1;n<nbcav; n++) { cavities+=translate(cavity,n*cL,0.); }
```

Figure 5.35: Definition of the cavity and the mesh you want
Finally, we define borders of the main domain, and mesh the resulting `Geometry` defined with `surfaceFrom`.

```plaintext
// full geometry
Real sb = 0.05;
Point p1 (0., 0.), p2 (2. + nb cav * cL, 0.), p3 (2. + nb cav * cL, 1.), p4 (0., 1.);
Segment s0 (_v1=p1, _v2=p0, _hsteps=Reals(sb, s)),
    s10 (_v1=Point(1. + nb cav * cL, 0.), _v2=p2, hsteps=Reals(s, sb)),
    s11 (_v1=p2, _v2=p3, _hsteps=sb, _domain_name='SigmaP'),
    s12 (_v1=p3, _v2=p4, _hsteps=sb),
    s13 (_v1=p4, _v2=p1, _hsteps=sb, _domain_name='SigmaM');
Geometry borders=s0+cavities+s10+s11+s12+s13;

// create mesh
Mesh mesh2d (surfaceFrom(borders, "Omega"), triangle, 1, _gmsh);
Domain omega=mesh2d.domain("Omega");
Domain sigmaP=mesh2d.domain("SigmaP");
Domain sigmaM=mesh2d.domain("SigmaM");
```
6.1 Domains, spaces, unknowns and test functions

XLiFE++ allows you to solve PDE with the finite elements method, and the spectral elements method. Both methods approximate all functions $w$ as follows:

Given a basis of $n$ functions $\varphi_i(x, y, z)$ and $w_i = (w, \varphi_i)$, then $w(x, y, z) \approx \sum_{i=0}^{n} w_i \varphi_i(x, y, z)$.

The basis functions define the so-called approximation space:

$$V_h = \left\{ w, \text{ such as } w = \sum_{i=0}^{n} w_i \varphi_i(x, y, z) \right\}.$$  

We will now see how to define spaces, dealing with finite element spaces and spectral spaces. XLiFE++ is built so that just one object is concerned. Note that XLiFE++ does not declare essential conditions in space whereas mathematics requires it!

6.1.1 Domains and finite element spaces

With the finite element method, the basis function $\varphi_i$ is constructed from the elements having $i$ as a degree of freedom (dof). It is a shape function.

What do we need to define a finite element space?

- a geometrical definition of the domain where the problem is to be solved,
- a finite element interpolation, such as $P_k$, $k \in \mathbb{N}$ for instance.

The geometrical definition of the domain consists in a mesh, which is a set of geometrical elements (such as triangles, hexahedra, prisms, etc) whose union describes the domain. Inside the program, this description is handled through an object of type Mesh. The definition of such an object is the very first step of the resolution process to the problem.

The different ways XLiFE++ provides to define a Mesh are detailed in chapter 5. In order to prepare the second step, namely the construction of the finite element space, we have to declare variables to handle the main domain and eventually the subdomains needed by the problem. This can be seen as an extraction from the mesh of the right information. This is done by means of strings which are the names of the subdomains. Consequently, the user has to know in advance the names of the subdomains.

When the mesh comes from a file, the names of the domains are generally written inside the file. In the case of a GMSH mesh file, a default name is automatically generated by XLiFE++ for each domain whose name is not specified in the file. When the mesh is built by an internal meshing tool provided by XLiFE++, the name of the main domain is always “Omega”. Moreover, if the mesh is built by a structured generator, the boundary names have to be given by the user, in a specific order defined in the documentation of
the constructor. But if the mesh is built by a subdivision generator, the names of the subdomains are also automatically generated. So, in any case, the best way for the user to make sure he uses the right names is to run the short following program, which is in fact the minimum mandatory program for XLiFE++ usage:

```cpp
#include "xlife++.h"
using namespace xlifepp;

int main() {
    init(en); // initialisation
    Number order = 1;
    Mesh m(Ball(), _tetrahedron, order, _subdiv, "test"); // for example
    m.printInfo(); // prints mesh information on the terminal
}
```

The output is:

```
Mesh 'test' (Ball - Tetrahedron mesh over 8 octants)
  space dimension : 3, element dimension : 3
  Geometry ball (center = (0,0,0), radius = 1) of shape type ball of dimension 3, BoundingBox [-1,1]x[-1,1]x[-1,1],
  MinimalBox [(-1, -1, -1), (1, -1, -1), (-1, 1, -1), (-1, -1, 1)], names of variable : x, y, z
  number of elements : 8, number of vertices : 7, number of nodes : 7, number of domains : 5
    domain number 0: Omega (Interior of the domain)
    domain number 1: Sigma_1 (Boundary: The sphere centered at vertex 4)
    domain number 2: Sigma_2 (Interface: YZ plane)
    domain number 3: Sigma_3 (Interface: XZ plane)
    domain number 4: Sigma_4 (Interface: XY plane)
```

Now, we can declare the handle variables that will be used just afterwards:

```cpp
Domain omega = m.domain("Omega");
Domain gamma = m.domain("Sigma_1");
```

The definition of a finite element space is done by using one the constructors:

- Space(const GeomDomain& g, PolynomType pt, const String& name, bool opt = true);
- Space(const GeomDomain& g, QolynomType qt, const String& name, bool opt = true);
- Space(const GeomDomain& g, FeFaceType ft, const String& name, bool opt = true);
- Space(const GeomDomain& g, FeEdgeType fe, const String& name, bool opt = true);

The first argument g corresponds to the handle variables (omega, gamma) just defined whose type, Domain, is an alias for GeomDomain.

The second argument is the type of the elements in the space, to be chosen in the following list:

- P0, P1, P2…, P10 for standard Lagrange element on segment, triangle or tetrahedron
- Q0, Q1, Q2…, Q10 for standard Lagrange element on quadrangle or hexahedron
- NF1_1, NF1_2, …, NF1_5 for Raviart-Thomas element on triangle or Nedelec Face first family element on tetrahedron. It is also possible to use RT_k instead of NF1_k!
- NE1_1, NE1_2, …, NE1_5 for Nedelec edge first family element on tetrahedron. It is also possible to use N_k instead of NF1_k!
The second family edge or face elements are not yet available on triangle and tetrahedron. Edge or face elements on quadrangle and hexahedron are not yet available.

The last argument can be used to deactivate numbering optimisation (reduction of the band width of the matrix) if its value is false.

Here are some examples of FE space construction:

```cpp
// 2D examples
Mesh mesh2d(Rectangle(_xmin=0.,_xmax=1.,_ymin=0.,ymax=1.,_nnodes=10), _triangle,1.,_gmsh);
Domain omega=mesh2d.domain("Omega");

// Lagrange Finite Element spaces
Space V1(omega,P1,"P1",true); // with numbering optimisation
Space V2(omega,P2,"P2",false); // no numbering optimisation
Space V3(omega,interpolation(_Lagrange,_standard,20,H1),"P20");

// Hdiv Finite Element spaces
Space W1(omega,RT1,"RT1");
Space W3(omega,interpolation(_RaviartThomas,_standard,3,Hdiv),"RT3");

// Hrot Finite Element spaces
Space R1(omega,N1,"Ned1");
Space R2(omega,N2,"Ned2");
Space R4(omega,interpolation(_Nedelec,_firstFamily,4,Hrot),"Ned4");

// 3D examples
Mesh mesh3d(Cube(_origin=Point(0.,0.,0.),_length=1.,_nnodes=n), _tetrahedron,1.,_gmsh);
Domain omega=mesh3d.domain("Omega");
Space V1(omega,P1,"P1",true);
Space V3(omega,interpolation(_Lagrange,_standard,3,H1),"P3");
Space W1(omega,NF1,"Hdiv_Ned1");
Space R2(omega,NE1,"Hrot_Ned2");
```

Note that when dealing with problem with vector unknown where each component is approximated in the same space (for instance $P_1 \times P_1 \times P_1$ for the displacement field in elasticity problem), you have to build ‘vector’ unknown on a ‘scalar’ space; see the Unknown section.

### 6.1.2 Spectral spaces

Spectral spaces are spaces defined from basis functions defined on a mesh domain. Contrary to finite element basis function given by a local definition on elements, the spectral basis functions are given as global functions either by their analytic forms or by a set of interpolated functions (say vectors related to an other space).

#### Analytic spectral space

The following declaration instanciate spectral space from analytic basis functions:

```cpp
Space(const GeomDomain& g, Function f, Number n, Dimen d, const String& name);
Space(const GeomDomain& g, Function f, Number n, const String& name);
Space(const GeomDomain& g, Function f, Number n, const String& name);
```

Let us see an example:
To define a spectral basis, you need to define a function of space coordinates with at least one parameter: the basis index. To do so, you have to define a standard C++ function, taking a `Point` and a `Parameters`. The first one contains the space coordinates. The second one contains all parameters needed to define the function. The return type of such function is `Real`. In the example, you can notice how to define and use the parameter "h".

Once you have defined your C++ function, you have to pass it to the list of arguments of the `Space` constructor. To do so, you have to use the `Function` object, taking the name of the function, a string, and the `Parameters` object.

### Interpolated spectral space

An interpolated spectral space is defined from a set of interpolated functions, say vectors of another space (`TermVectors`, see `Terms` chapter). The following example shows how it works:

```cpp
int main(int argc, char** argv)
{
    Mesh m(...);
    Domain omega=m.domain("Omega");
    Parameters ps(1., "h");
    Number n=10;
    Space sp(omega, Function(sin_n, "sin_n", ps), n, "sinus basis");
...
}
```

The `Unknown` object is described in the next section.

### Advance usage

It is possible to manipulate spectral basis by instanciate such objects:

```cpp
SpectralBasisFun sbFun(omega, sinBasis, N, 1);
```
Thus it is possible to evaluate basis functions at a point:

```cpp
Real r;
Point P(1.,0.);
sbFun.function(u,P,r);
sbInt.function(u,P,r);
```

Be cautious, the type of returned argument `r` has to be consistent with the type of basis functions.

### 6.1.3 Unknowns and test functions

Once you have defined the space, the next step is to define unknowns and test functions on this space.

```cpp
Unknown(Space& sp, const String& name, Dimen d=1);
TestFunction(Space& sp, const String& name, Dimen d=1);
```

According to the problem, you may want to define scalar or vector unknowns or test functions. The third argument is dedicated to this.

In case of a multiple unknowns problem, the order of unknowns may be sensitive. By default, they are sorted by the construction order, using the rank property of unknown:

```cpp
Unknown u(V, "u", 2);
Unknown p(V, "p");
TestFunction v("v",u);
TestFunction q("q",p);
cout<<u.rank()<<"\n"<<v.rank()<<"\n"<<p.rank()<<"\n"<<q.rank();
```

This exemple gives 1 3 2 4.

It is possible to assign the rank of an unknown at the construction:

```cpp
Unknown u(V, "u", 2, 2); //rank 2
Unknown p(V, "p", 1, 1); //rank 1
TestFunction v(u, "v", 4); //rank 4
TestFunction q(p, "q", 3); //rank 3
cout<<u.rank()<<"\n"<<v.rank()<<"\n"<<p.rank()<<"\n"<<q.rank();
```

Be cautious, rank has to be unique! It is not mandatory that ranks follow.

The `setRanks` function may be used to change the ranks of a collection of unknowns:

```cpp
... setRanks(u,1,p,2,v,11,q,12);
```

### 6.2 Forms

Given a PDE, you have to write a variational formulation. As a result, you have an equality between 2 forms: a bilinear form on the unknown `u` and the tests function `v`, generally called `a`, and a linear form on the test function `v`, generally called `l`. Both are defined as linear combination of single or double integrals on operators on unknowns and, in the bilinear case, test functions, and an integration method or a quadrature rule.
BilinearForm intg(const GeomDomain& dom, const OperatorOnUnknowns& opus, QuadRule qr=defaultRule, Number qro=0);
BilinearForm intg(const GeomDomain& domx, const GeomDomain& domy, const OperatorOnUnknowns& opus, QuadRule qr=defaultRule, Number qro=0);
BilinearForm intg(const GeomDomain& domx, const GeomDomain& domy, const KernelOperatorOnUnknowns& kopus, QuadRule qr=defaultRule, Number qro=0);
LinearForm intg(const GeomDomain& dom, const OperatorOnUnknown& opu, QuadRule qr=defaultRule, Number qro=0);
LinearForm intg(const GeomDomain& dom, const Unknown& u, QuadRule qr=defaultRule, Number qro=0);
LinearForm intg(const GeomDomain& domx, const GeomDomain& domy, const OperatorOnUnknown& opu, QuadRule qr=defaultRule, Number qro=0);
LinearForm intg(const GeomDomain& domx, const GeomDomain& domy, const Unknown& u, QuadRule qr=defaultRule, Number qro=0);

In simple case, symmetry property of a bilinear form may be deduced from its definition. In some cases, the analysis being to intricate, the symmetry property is not deduced. It is the reason why it is possible to enforce this property in the definition of bilinear form by specifying as last argument one of the symmetry keywords:

_noSymmetry, _symmetric, _skewSymmetric, _selfAdjoint, _skewAdjoint

For instance, if $A$ is a symmetric matrix:

BilinearForm $b = \intg(S, u | v)$; // implicit symmetry
BilinearForm $b = \intg(S, (A \ast u) | v, _symmetric)$; // explicit symmetry

Keep in mind that $u$ and $v$ represent shape functions of the space which are real functions!

Thus, there is no reason to conjugate test functions.

6.2.1 Operators on unknowns

XLiFE++ management of operators on unknowns is as close as possible to the mathematical description, few operators are overloaded and a lot of possibilities are offered, For instance:

<table>
<thead>
<tr>
<th>mathematical expression</th>
<th>XLiFE++ translation</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nabla(u)$</td>
<td>grad(u)</td>
<td>u unknown</td>
</tr>
<tr>
<td>$\nabla(u) \cdot \nabla(v)$</td>
<td>grad(u)</td>
<td>grad(v)</td>
</tr>
<tr>
<td>$(A \ast \nabla(u)) \cdot \nabla(v)$</td>
<td>(A*grad(u))</td>
<td>grad(v)</td>
</tr>
<tr>
<td>$(F(x) \ast \nabla(u)) \cdot \nabla(\sigma)$</td>
<td>(F*grad(u))</td>
<td>grad(conj(v))</td>
</tr>
<tr>
<td>$u(x) \ast G(x,y) \ast v(y)$</td>
<td>$u\ast G\ast v$</td>
<td>u unknown, v test function, G a kernel</td>
</tr>
</tbody>
</table>

Defining functions needs C++ functions with specific prototypes:

```
OUT f1 (const Point& P, Parameters& pa = defaultParameters); // scalar form
Vector<OUT> f2 (const Vector<Point>& Ps, Parameters& pa = defaultParameters); // vector form
```

The return type OUT can be one of the following: Real, Complex, Vector<Real>, Vector<Complex>, Matrix<Real> or Matrix<Complex>.

You can use the function directly in your integral, or define Function object such as
You can optionally give a name to the Function object and a Parameters object when needed. Defining kernels needs also C++ functions with specific prototypes:

```cpp
OUT f1(const Point& P, const Point & MParameters& pa = defaultParameters); // scalar form
Vector<OUT> f2(const Vector<Point>& Ps, const Vector<Point>& Ms, Parameters& pa = defaultParameters); // vector form
```

You can use the function directly in your integral, or define Kernel object such as:

```cpp
Kernel F(f1, "name", params);
```

You can optionally give a name to the Kernel object and a Parameters object when needed.

The complete list of operators is in the following, where u is either a scalar or vector unknown, x, y and z are the cartesian coordinates and n is the normal:

<table>
<thead>
<tr>
<th>mathematical</th>
<th>built in functions</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity</td>
<td>id(u) or u</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\partial t</td>
<td>d0(u) or dt(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\partial x</td>
<td>d1(u) or dx(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\partial y</td>
<td>d2(u) or dy(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\partial z</td>
<td>d3(u) or dz(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\nabla</td>
<td>grad(u) or nabla(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>div</td>
<td>div(u)</td>
<td>vector</td>
</tr>
<tr>
<td>curl</td>
<td>curl(u) or rot(u)</td>
<td>vector</td>
</tr>
<tr>
<td>\nabla (surf)</td>
<td>gradS(u) or nablaS(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>div (surf)</td>
<td>divS(u)</td>
<td>vector</td>
</tr>
<tr>
<td>curl (surf)</td>
<td>curlS(u) or rotS(u)</td>
<td>vector</td>
</tr>
<tr>
<td>\nabla abc</td>
<td>gradG(u,a,b,c) or nablaG(u,a,b,c)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>\nabla abc.</td>
<td>divG(u,a,b,c)</td>
<td>vector</td>
</tr>
<tr>
<td>\nabla abc×</td>
<td>curlG(u,a,b,c) or rotS(u,a,b,c)</td>
<td>vector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mathematical</th>
<th>built in functions</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>\epsilon</td>
<td>epsilon(u)</td>
<td>vector</td>
</tr>
<tr>
<td>\epsilon_{iabc}</td>
<td>epsilonG(u,i,a,b,c)</td>
<td>vector</td>
</tr>
<tr>
<td>\epsilon_R</td>
<td>epsilonR(u) = (\epsilon_{11},\epsilon_{11},\epsilon_{22},\epsilon_{33},\epsilon_{32},\epsilon_{31})</td>
<td>vector</td>
</tr>
<tr>
<td>voigtToM</td>
<td>voigtToM(u)= [u1 u6 u5; u6 u2 u4; u5 u4 u3]</td>
<td>vector</td>
</tr>
<tr>
<td>n*</td>
<td>nx(u) or _n*u</td>
<td>scalar</td>
</tr>
<tr>
<td>n.</td>
<td>ndot(u) or _n.u</td>
<td>vector</td>
</tr>
<tr>
<td>n×</td>
<td>ncross(u) or _n`u</td>
<td>vector</td>
</tr>
<tr>
<td>n×n×</td>
<td>ncrossncross(u) or _n<code>_n</code>u</td>
<td>vector</td>
</tr>
<tr>
<td>n.\nabla</td>
<td>ndotgrad(u) or _n.grad(u)</td>
<td>scalar</td>
</tr>
<tr>
<td>n×\nabla</td>
<td>ncrossgrad(u) or _n`.grad(u)</td>
<td>scalar</td>
</tr>
<tr>
<td>ndiv</td>
<td>ndiv(u) or _n*div(u)</td>
<td>vector</td>
</tr>
<tr>
<td>n×curl</td>
<td>ncrosscurl(u) or _n`curl(u)</td>
<td>vector</td>
</tr>
<tr>
<td>[ ] (jump across)</td>
<td>jump(u)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>{ } (mean across)</td>
<td>mean(u)</td>
<td>scalar or vector</td>
</tr>
</tbody>
</table>

### 6.2.2 Operators on kernel

Similar to operator on unknowns, XLTFe++ allows to apply some operators on kernel (say \(k(x,y)\)). The complete list of operators is the following:
In operators, the normal vectors \( n, nx, ny \) are symbolic ones. They refer to real normal vectors related to the domain involved in integrals where operators appear. See the section 5.6.2 to know how normal vectors are oriented.

When one of the argument is a complex, the "inner product" means a hermitian product.

### 6.2.3 Kernels available

There are currently 2 kernels available in XLiFE++ for 2D and 3D problems: Laplace and Helmholtz Green functions:

- The Laplace kernel used for 2D problems is

  \[
  G(x, y) = -\frac{1}{2\pi} \log (\|x - y\|) ,
  \]

  and for 3D we use

  \[
  G(x, y) = \frac{1}{4\pi \|x - y\|}
  \]

<table>
<thead>
<tr>
<th>mathematical</th>
<th>built in functions</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>identity</td>
<td>id(k) or k</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>( \nabla_x )</td>
<td>grad_x(k) or nbla_x(k)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>( \nabla_y )</td>
<td>grad_y(k) or nbla_y(k)</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>( \text{div}_x )</td>
<td>div_x(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( \text{div}_y )</td>
<td>div_y(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( \text{curl}_x )</td>
<td>curl_x(k) or rot_x(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( \text{curl}_y )</td>
<td>curl_y(k) or rot_y(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( n_x \times )</td>
<td>ncross_x(k) or _nx_k</td>
<td>vector</td>
</tr>
<tr>
<td>( n_y \times )</td>
<td>ncross_y(k) or _ny_k</td>
<td>vector</td>
</tr>
<tr>
<td>( n_x \times (n_x \times) )</td>
<td>ncrossncross_x(k) or _nx_k</td>
<td>vector</td>
</tr>
<tr>
<td>( n_y \times (n_y \times) )</td>
<td>ncrossncross_y(k) or _ny_k</td>
<td>vector</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mathematical</th>
<th>built in functions</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_x \nabla_x )</td>
<td>ndotgrad_x(k) or _nx_grad_x(k)</td>
<td>scalar</td>
</tr>
<tr>
<td>( n_y \nabla_y )</td>
<td>ndotgrad_y(k) or _ny_grad_y(k)</td>
<td>scalar</td>
</tr>
<tr>
<td>( n_x \text{div}_x )</td>
<td>ndiv_x(k) or _nx_div_x(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( n_y \text{div}_y )</td>
<td>ndiv_y(k) or _ny_div_y(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( n_x \times \text{curl}_x )</td>
<td>ncrosscurl_x(u) or _nx_curl_x(k)</td>
<td>vector</td>
</tr>
<tr>
<td>( n_x \times n_y )</td>
<td>nxdotny_times(k) or (_nx_ny)*k</td>
<td>scalar or vector</td>
</tr>
<tr>
<td>( (n_x \times n_y) \times )</td>
<td>ncrossny_cross(k) or (_nx_ny_nx)*k</td>
<td>vector</td>
</tr>
<tr>
<td>( (n_y \times n_x) \times )</td>
<td>nycrossnx_cross(k) or (_ny_nx_nx)*k</td>
<td>vector</td>
</tr>
</tbody>
</table>

137
• The Helmholtz Green function for 2D problems used is

\[ G(x, y) = \frac{i}{4} H_0^{(1)}(k \| x - y \|) , \]

with \( H_0^{(1)} \) then Hankel function. Finally, for 3D problems we use

\[ G(x, y) = \frac{e^{ik\| x - y \|}}{4\pi \| x - y \|} . \]

Examples of declaration of these kernels follow:

```cpp
Kernel GLap2D=Laplace2dKernel();
Kernel GLap3D=Laplace3dKernel();
Parameters pars(k,"k"); // We provide the wavenumber k using a Parameters
Kernel G=Helmholtz2dKernel(pars);
Kernel GHelm3D=Helmholtz3dKernel(pars);
```

### 6.2.4 Integration method

When defining a linear or a bilinear form, the user may specify the integration method (IM) to use. Currently the following objects are available:

- **QuadratureIM** : quadrature methods based on quadrature points and weights, see quadrature rule in the next section
- **SauterSchwabIM, LenoirSallesIM, DuffyIM**: specific methods to integrate singular kernel in bilinear form, see details in next section
- **IntgRepresentationIM** : specific method to integrate singular kernel in integral representation, see details in next section

To use it in a computation, specify an IM object in the definition of form:

```cpp
QuadratureIM quadIM(Gauss_Legendre,2); //standard quadrature method
BilinearForm blf=intg(omega,uv,quadIM);
BilinearForm blf=intg(omega,uv,Gauss_Legendre,2); //shortcut syntax
SauterSchwabIM ssIM(3,3); //Sauter−Schwab integration method
BilinearForm blf=intg(sigma,sigma,G*v,ssIM);
IntgRepresentationIM irIM(Gauss_Legendre,3,Gauss_Legendre,2,0.1);
LinearForm lf=intg(sigma,G*u,irIM,U);
```

Note that integration method is attached to the integral definition. So you can mix different integration methods in a bilinear form:

```cpp
BilinearForm blf = intg(omega,grad(u)|grad(v),Gauss_Lobatto,1) + intg(omega,u*v,Gauss_Legendre,2);
```

Using mixed integration methods is generally slower than using the same integration method!
It is not mandatory to specify an integration method in form; a default one is chosen according to the order of unknown interpolations, the order of differential operators involved and the fact that there are functions in operator on unknowns. For FE form, we use the minimal quadrature rule for shapes involved in domain which integrates exactly the polynomials of order

\[ k = (\text{deg}(u) - \text{order}(\text{dif}(u))) \times \text{deg}(u) \times [(\text{deg}(v) - \text{order}(\text{dif}(v))) \times \text{deg}(v)] \]

where \( \text{deg}(u) \) (resp. \( \text{deg}(v) \)) is the degree of polynomials used by \( u \)-interpolation (resp. \( v \)-interpolation), \( \text{order}(\text{dif}(u)) \) (resp. \( \text{order}(\text{dif}(v)) \)) is the order of differential operator applied to \( u \) (resp. \( v \)). \( \square \) means an optional coefficient.

The table of best rules is given in the developer’s documentation.

Quadrature rules

To perform computation of integrals over reference elements, XLiFE++ provides a lot of quadrature formulae of the form:

\[
\int_{\hat{E}} f(\hat{x}) d\hat{x} \approx \sum_{i=1,q} \omega_i f(\hat{x}_i)
\]

where \((\hat{x}_i)_{i=1,q}\) are quadrature points belonging to reference element \( \hat{E} \) and \((\omega_i)_{i=1,q}\) are quadrature weights.

Up to now, there exist quadrature formulae for unit segment \([0,1]\\), for unit triangle, for unit quadrangle (square), for unit tetrahedron, for unit hexahedron (cube), for unit prism and for unit pyramid. The following tables gives the list of quadrature rule available:

<table>
<thead>
<tr>
<th>General rules</th>
<th>Gauss-Legendre</th>
<th>Gauss-Lobatto</th>
<th>Grundmann-Muller</th>
<th>symmetrical Gauss</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>quadrangle</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>odd degree up to 21</td>
</tr>
<tr>
<td>triangle</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>degree up to 10</td>
</tr>
<tr>
<td>hexahedron</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>odd degree up to 11</td>
</tr>
<tr>
<td>tetrahedron</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td>degree up to 10</td>
</tr>
<tr>
<td>prism</td>
<td></td>
<td></td>
<td></td>
<td>degree up to 10</td>
</tr>
<tr>
<td>pyramid</td>
<td>any odd degree</td>
<td>any odd degree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Particular rules</th>
<th>nodal</th>
<th>miscellaneous</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>P1 to P4</td>
<td></td>
</tr>
<tr>
<td>quadrangle</td>
<td>Q1 to Q4</td>
<td></td>
</tr>
<tr>
<td>triangle</td>
<td>P1 to P3</td>
<td>Hammer-Stroud 1 to 6</td>
</tr>
<tr>
<td>hexahedron</td>
<td>Q1 to Q4</td>
<td></td>
</tr>
<tr>
<td>tetrahedron</td>
<td>P1, P3</td>
<td>Stroud 1 to 5</td>
</tr>
<tr>
<td>prism</td>
<td>P1</td>
<td>centroid 1, tensor product 1,3,5</td>
</tr>
<tr>
<td>pyramid</td>
<td>P1</td>
<td>centroid 1, Stroud 7</td>
</tr>
</tbody>
</table>

139
The developer documentation gives more details on quadrature rules and indicates what best rules (in terms of number of quadrature points) are selected when only shape and degree are specified. Generally for low degree \( (d \leq 3) \) a specific rule is selected, for intermediate degree \( (4 \leq d \leq 10) \) a symmetrical Gauss rule is selected and for high degree a quadrature rule working at any degree (Gauss-Legendre or Grundman-Muller) is chosen.

**How to choose the quadrature rule?**

By using \texttt{intg} with a specific pair of arguments:

- \texttt{QuadRule qr}, to give the quadrature rule formulae (possible values are \texttt{Gauss\_Legendre}, \texttt{Gauss\_Lobatto}, \texttt{nodalQuadrature}, \texttt{miscQuadrature}, \texttt{Grundmann\_Moller} or \texttt{symmetrical\_Gauss}),

- \texttt{Number qro}, to give the quadrature rule degree:

```plaintext
BilinearForm a=intg(Omega,u*v,Gauss\_Legendre,4);
```

When no rule and degree are given, the degree is determined by looking the degree of polynomials involved in the (bi)linear form taking into account derivative operators and the existence of an additional user function. For instance, the following bilinear form in \( P^k \) finite element space

\[
\int_{\Omega} f uv
\]

will ask for a quadrature rule of degree \( d = 3k \) while the following bilinear form

\[
\int_{\Omega} \nabla u.\nabla v
\]

will ask for a quadrature rule of degree \( d = 2(k - 1) \).

Once the degree \( d \) is determined, XLiFE++ chooses the best quadrature rule available for degree \( d \) and element shapes involved in the mesh domain.

\[\ddagger\] The user choice is always a priority even his choice leads to under integration. In doubt, let XLiFE++ work for you!

**Integration methods for integral equation**

Integral equation involves singular kernels. To deal with the singularity in integrals, some particular methods are proposed to users. \texttt{LenoirSallesIM} works for 2D and 3D problems but with low order finite elements (piecewise constant basis functions) whereas there are methods for any finite element order but specific problem dimension such as \texttt{SauterSchwabIM} for 3D problems and \texttt{DuffyIM} for 2D problems. Currently, only \( \log(r) \) in 2D and \( r^{-1} \) in 3D singularities are addressed.
- **SauterSchwabIM**\(^1\) class addresses computation of integral

\[
\int_\Gamma \int_\Sigma K(x,y) \, dx \, dy
\]

where \(\Gamma\) and \(\Sigma\) are 2D domains (in 3D) (partition of triangles) and \(K(x,y)\) a kernel having possibly a singularity of type \(1/||x - y||\). This technique is well adapted for most of second order PDE in 3D. It uses Gauss-Legendre quadrature on segment and triangle. When creating such method, you may specify the quadrature order on segment (\(oS\)) and the quadrature order on triangle (\(oT\)):

```
SauterSchwabIM ssIM(oS,oT);
```

By default \(oS = 4\), \(oT = 3\). Sauter-Schwab method works for any finite element on triangle.

In order to speed-up the computation, it is possible to define 3 quadrature orders instead of one for the non-singular cases. We distinguish the relative distance between elements in order to use a higher order for close elements than for further elements. Let’s define \(dR\) the relative distance between the two elements, then we define \(dM\) and \(dF\) and 3 quadrature orders: \(oN\) (near case), \(oM\) (medium relative distance) and \(oF\) (further elements) and to select the quadrature order we use the rule: when \(dR \leq dM\), quadrature \(oN\) is used, for \(dM < dR \leq dF\) quadrature \(oM\) is used and finally, when \(dR > dF\), quadrature \(oF\) is used.

```
SauterSchwabIM ssIM(oS, oN, oM, oF, dM, dF);
```

- **DuffyIM**\(^2\) class addresses computation of integral

\[
\int_\Gamma \int_\Sigma K(x,y) \, dx \, dy
\]

where \(\Gamma\) and \(\Sigma\) are 1D domains (in 2D) (partition of segments) and \(K(x,y)\) a kernel having possibly a singularity of type \(\log(||x - y||)\). This technique is well adapted for most of second order PDE in 2D. It uses Gauss-Legendre quadrature on segment.

When creating such method, you may at least specify 2 quadrature order on segment, the first \(oS\) for singular integrals (identical segments and segments with a common vertex) and the second \(oR\) for non-singular integrals:

```
DuffyIM dufIM(oS, oR);
```

By default \(oS = 6\), \(oR = 4\). Duffy method works for any finite element on segment.

In order to speed-up the computation, it is possible to define 3 quadrature orders instead of one for the non-singular cases. We distinguish the relative distance between elements in order to use a higher order for near element than for further elements. Let’s define \(dR\) the relative distance between the two elements, then, we define \(dM\) and \(dF\) and 3 quadrature orders \(oN\) (near case), \(oM\) (medium relative distance) and \(oF\) (further elements) and to select the quadrature order used we use the rule: when \(dR \leq dM\), quadrature \(oN\) is used, for \(dM < dR \leq dF\) quadrature \(oM\) is used and finally, when \(dR > dF\), quadrature \(oF\) is used.

---

\(^1\)Integral over a product of geometric elements with singularity using Sauter-Schwab technique, Stefan A. Sauter, Christoph Schwab, "Boundary Element Methods”, Springer, 2010

\(^2\)Integral over a product of segments with singularity using the Duffy transformation
• **LenoirSallesIM** \(^{3}\) class addresses computation of integral

\[
\int_{\Gamma} \int_{\Sigma} K(x, y) \, dx \, dy
\]

where \(\Gamma\) and \(\Sigma\) are 2D domains (in 3D) (partition of triangles) or 1D domaines (in 2D) (partition of segments) and \(K(x, y)\) a kernel having a singularity of type \(1/||x - y||\). It deals with analytical techniques for the singular part \(1/||x - y||\). This concerns only elements close. Otherwise, it computes integral using quadrature rules. As this new method is still experimental, see the code to use it.

• **IntgRepresentationIM** class addresses computation of single integral used in integral representation

\[
f(x) = \int_{\Gamma} K(x, y) \, dy
\]

where \(\Gamma\) is any domain. Using a distance factor \(d\) from \(x\) to element, integral is computed by different method:
- if \(\text{dist}(x, E) \geq d\), a given quadrature rule is applied to \(K(x, y)\)
- if \(\text{dist}(x, E) < d\), a given quadrature rule is applied to the regular part \(K_r(x, y)\) and a singular method is used to compute the singular part \(K_s(x, y)\)

The singular and regular parts have to be provided in the definition of \(\text{Kernel}\).

To define such a method, user has to specify the quadrature rules and the distance factor:

\[
\text{IntgRepresentationIM} \ irIM(\text{quadruleFar}, \text{orderFar}, \text{quadruleReg}, \text{orderReg}, \ d ) ;
\]

### 6.3 Essential conditions

Essential conditions are conditions that appear in spaces involved in variational problem. The most common one is the Dirichlet condition on a boundary : \(u = 0\) on \(\Gamma\) (homogeneous) or \(u = g\) on \(\Gamma\) (non homogeneous). But there are others : transmission condition on a boundary, periodic condition between two boundaries, null average on a domain, ... XLtFE++ provides a symbolic description of such conditions based on operator’s stuff already described.

The general syntax of an essential condition is the following

\[
( a_1 \otimes \text{op1}(u_1) )|D1 \pm ( a_2 \otimes \text{op2}(u_2) )|D2 = f
\]

where

- \(a_1, a_2\) are some constants
- \(\otimes\) is any algebraic operator (\(*, \|, \%\), \(^\wedge\) )

---

• op1, op2 are some operators on unknown
• u1, u2 are some unknowns
• D1, D2 are some domains
• f is a constant or a function

Some classic scalar expressions are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u</td>
<td>D = 0 )</td>
</tr>
<tr>
<td>( u</td>
<td>D = f )</td>
</tr>
<tr>
<td>( u1</td>
<td>D - u2</td>
</tr>
<tr>
<td>( u</td>
<td>D1 - u</td>
</tr>
<tr>
<td>( u</td>
<td>D1 - g \ast u</td>
</tr>
</tbody>
</table>

Obviously, syntax supports more than conditions that the program can really deal with!

⚠️ As the operator priority rules are the C++ rules, omitted parenthesis may induce some hazardous compilation errors. In doubt, use parenthesis.

To declare essential condition, users have to instantiate `EssentialConditions` object, which handles a set of conditions:

```cpp
Strings sn("y=0", "y=1", "x=0", "x=1");
Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=10,
    _side_names=sn), _triangle, 1, _structured);
Domain omega=mesh2d.domain("Omega");
Domain sigmaM=mesh2d.domain("x=0");
Domain sigmaP=mesh2d.domain("x=1");
Space V(omega,P1,"V",true);
Unknown u(V, "u");
EssentialConditions ecs = (u|sigmaM = 1);
```

or using a function:

```cpp
Real un(const Point& P, Parameters& pa = defaultParameters)
{
  return 1.;
}
EssentialConditions ecs = (u|sigmaM = un);
```

To concatenate conditions, use the operator &:

```cpp
EssentialConditions ecs = (u|sigmaM = 1) & (u|sigmaP = 1) ;
```

It is possible to mix conditions. Here is a case with two unknowns related by a transmission condition:
To deal with periodic condition, the map related to the two domains involved is required:

```cpp
Vector<Real> mapPM(const Point& P, Parameters& pa = defaultParameters)
{
    Point Q(P);
    Q(1) -= 1;
    return Q;
}
```

XLiFE++ uses a very powerful process to deal with essential condition: all constraints are merged in a unique linear constraints system which is reduced using a QR algorithm. This process is able to detect redundant or conflicting constraints. When some are redundant, they are deleted. When some are in conflict, they are also deleted but the right hand side related components are averaged. For instance, this occurs when two Dirichlet conditions are not compatible at the intersection of two boundaries. In both cases a warning message is handled. It is the responsibility of user to check possible conflict.

Contrary to the mathematical point of view, in XLiFE++ the essential conditions are NOT attached to spaces but to algebraic representation of bilinear forms (see next section). This choice avoid to define multiple spaces.
Now, from the previous symbolic representation, we go to the algebraic representation of the problem, that is to say the representation of the problem in terms of matrices and vectors.

### 7.1 Algebraic representation

The algebraic representation consists in representation in terms of vectors and matrices of linear and bilinear forms, say:

\[ L_i = l(\tau_i) \quad \text{and} \quad A_{ij} = a(w_j, \tau_i) \]

where \((w_j)_{j=1,n}\) and \((\tau_i)_{i=1,m}\) are respectively the basis of finite space \(V\) (unknown space) and \(W\) (test function space).

XLiFE++ provides two fundamental classes to deal with such vectors and matrices:

- **TermVector** class which handles vector and space stuff (linear form, unknowns, dof numbering, ...)
- **TermMatrix** class which handles matrix and spaces stuff (bilinear form, unknowns, dof numbering, ...)

These two classes support either single unknown or multiple unknowns representation. Multiple unknowns vector or matrix are represented by single unknown blocks:

\[
L = \begin{bmatrix} L_{v_1} \\ L_{v_2} \\ \vdots \end{bmatrix} \quad \text{and} \quad A = \begin{bmatrix} A_{v_1 u_1} & A_{v_1 u_2} \\ A_{v_2 u_1} & A_{v_2 u_2} \\ \vdots & \vdots \end{bmatrix}
\]

where \(u_1, u_2\) stands for unknowns and \(v_1, v_2\) stands for test functions.

Ununknowns correspond to matrix columns and test functions to matrix rows!

The algebraic representation of a linear form or a bilinear form is simply done by specifying forms in **TermVector** or **TermMatrix**:

```markdown
Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _unnodes=20), _triangle, 1, _structured);
Domain omega=mesh2d . domain("Omega");
Space V(omega, P1, "V", true);
Unknown u(V, "u");
TestFunction v(u,"v");
LinearForm fv=intg(omega, f*v);
TermVector F(fv,"F");
```
BilinearForm \( auv=\intg(\omega, \grad(u) | \grad(v)) \);
TermMatrix \( A(auv,"A") \);

Naming them using a string is more convenient for printing purpose.

For multiple unknowns forms, the syntax is the same:

```
...  
Domain \( \omega_1 = \text{mesh2d} . \text{domain("Omega1"}) ;
Space \ V1(\omega_1, P1, "V", true) ;
Unknown u1(V1,"u1") ;
TestFunction v1(u1,"v1") ;

LinearForm \( fv = \intg(\omega_1, f*v1) + \intg(\omega_2, f*v2) \);
TermVector \( F(fv,"F") \);

BilinearForm \( auv=\intg(\omega_1, \grad(u1) | \grad(v1)) + \intg(\omega_2, \grad(u2) | \grad(v2)) \);
TermMatrix \( A(auv,"A") \);
```

As mentioned before, essential conditions are not attached, neither to space nor to bilinear form, but directly to TermMatrix. You have to specify them when you construct a TermMatrix from bilinear form:

```
BilinearForm \( auv=\intg(\omega, \grad(u) | \grad(v)) \);
EssentialConditions \( ecs = (u|\sigmaM = 1) \& (u|\sigmaP = 1) \);
TermMatrix \( A(auv,ecs,"A") \);
```

⚠️ Essential conditions are never attached to a TermVector! When solving a system involving essential conditions, the TermVector representing the right hand side of the system is automatically corrected to take into account essential conditions effects.

When defined, TermVector and TermMatrix are automatically computed, except if the option _notCompute is set in definition of TermMatrix or TermVector.

⚠️ The computation algorithms find the minimal representation of matrices. It means that the size of matrix is equal to the number of unknown dofs (and test function dofs) involved in the computation. For instance, a mass matrix on a boundary involve only dofs supported by the boundary.

⚠️ The value type of matrix (real or complex) is managed by TermMatrix and TermVector. The user has not to deal with that, except in an advanced usage.

Definition of TermMatrix or TermVector supports some optional arguments to be inserted in any order before the optional name argument :

```
TermMatrix(bf,[ecs_u],[ecs_v],[option],[option], ..., [name])
```
where \(bf\) is the bilinear form, \(ecs_u\) and \(ecs_v\) possible essential conditions, and \(option\) any of:

- \(_\text{compute}, \_\text{notCompute} : to manage the automatic computation of the TermMatrix\)
- \(_\text{assembled}, \_\text{unassembled} : to manage the automatic assembling of the matrix; not assembled implies not computed\)
- \(_\text{nonSymmetricMatrix}, \_\text{symmetricMatrix}, \_\text{selfAdjointMatrix}, \_\text{skewSymmetricMatrix}, \_\text{skewAdjointMatrix} : to enforce symmetry property when bilinear form has such symmetry and XLiFE++ has not detected it.\)
- \(_\text{csRowStorage}, \_\text{csColStorage}, \_\text{csDualStorage}, \_\text{csSymStorage}, \_\text{denseRowStorage}, \_\text{denseColStorage}, \_\text{denseDualStorage}, \_\text{skylineSymStorage}, \_\text{skylineDualStorage} : to enforce the storage if the default one chosen by XLiFE++ is not well suited\)
- \(_\text{pseudoReductionMethod}, \_\text{realReductionMethod}, \_\text{penalizationReductionMethod} : to indicate the method to deal with essential condition.\)

Up to now, only pseudo reduction method is available.

If necessary, it is possible to change the diagonal coefficient (by default 1) of the pseudo eliminated block matrix by invoking the \texttt{ReductionMethod} object:

\[
\text{BilinearForm} \ \texttt{auv} = \int \text{g} (\omega, \text{grad}(u) | \text{grad}(v));
\]

\[
\text{EssentialConditions} \ \texttt{ecs} = (u | \sigma M = 0);
\]

\[
\text{TermMatrix} \ A(\texttt{auv}, \texttt{ecs}, \texttt{ReductionMethod(\_pseudoReduction,10.)}, "A");
\]

If you choose to declare the TermMatrix with the \_notCompute option, its computation may be done later using the \texttt{compute} command:

\[
\ldots
\]

\[
\text{BilinearForm} \ \texttt{auv} = \int \text{g} (\omega, \text{grad}(u) | \text{grad}(v));
\]

\[
\text{TermMatrix} \ A(\texttt{auv}, \_\text{notCompute}, "A");
\]

\[
\ldots
\]

\[
\texttt{compute}(A);
\]

\texttt{TermMatrix} and \texttt{TermVector} manages some additional parameters and a lot of facilities are provided. Let us go to details.

### 7.1.1 TermVector in details

\texttt{TermVector} represents either a linear form on discrete space or any element of space as vector of components on the space basis.

It has a default constructor and one from linear form with options:

\[
\texttt{TermVector(name)};
\]

\[
\texttt{TermVector(LinearForm, opt1, opt2, opt3, name)};
\]

\(opt1, opt2, opt3, name\) are optional arguments.
Reals f(const Point& P, Parameters& pa = defaultParameters) {
  return Reals(2,-1.);
}

Strings sn(4, "");

Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=4, _side_names=sn),
  _triangle,1,_structured);
Domain omega=mesh2d.domain("Omega");
Space V(omega,P1,"V",true);
Unknown u(V,"u",2);
TestFunction v(u,"v");
LinearForm fv=intg(omega,f|v);
TermVector B(fv,"B");

In the previous example, the TermVector B is derived from a linear form defined on a vector test function.

A TermVector may be constructed from values of a function $f$ on a geometric domain. It is available only for FE Lagrange unknown: the vector is built with components $f(M_i)$ for any node $M_i$ in the domain:

```
TermVector(UUnknown, GeomDomain, T, String name)
```

The T argument may be a function (standard or Function object) or a constant value:

```
Reals f(const Point& P, Parameters& pa = defaultParameters) {
  return P;
}

Space V(omega,P1,"V",true);
Unknown u(V,"u",2);
TermVector B(u, omega, f, "B");
```

There are also a copy constructor and a constructor assigning a constant value from another TermVector:

```
TermVector(TermVector, name);
template <typename T> TermVector(TermVector, T, name);
```

The constructors of TermVector compute automatically the algebraic representation except if the option _notCompute is specified. In that case, the TermVector object may be computed later using the compute function:

```
Reals f(const Point& P, Parameters& pa = defaultParameters) {
  return Reals(2,-1.);
}

Strings sn(4, "");
Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1, _nnodes=4,
  _side_names=sn),_triangle,1,_structured);
Domain omega=mesh2d.domain("Omega");
Space V(omega,P1,"V",true);
Unknown u(V,"u",2);
TestFunction v(u,"v");
LinearForm fv=intg(omega,f|v);
TermVector B(fv,_notCompute,"B"); //do not compute B
... compute(B); //now compute B
```
If a TermVector object is already computed, the compute function does not re-compute it! If you want to re-compute it, you have to change its computation status:

```cpp
B.compute() = false;
```

In case of a multiple unknowns vector, a unknown block may be extracted as follows:

```cpp
...  
Space V(omega, P1, "V", true), H(omega, P0, "V", true);  
Unknown u(V, "u", 2), p(W, "p");  
LinearForm fv=intg(omega, f|u)+intg(omega, p);  
TermVector B(fv);  
TermVector B_u = B(u); // extract u part
```

Unknown is used as index and the returned TermVector is a copy of the extracted block.

It is possible to do algebraic operations (+=, -=, *=, /=, +, -, *, /) on TermVector:

```cpp
...  
Space V(omega, P1, "V", true);  
Unknown u(V, "u");  
LinearForm fvo=intg(omega, f*u);  
LinearForm fsv=intg(sigma, g*u);  
TermVector Bo(fvo, "Bo");  
TermVector Bs(fsv, "Bs");  
TermVector B = 2*Bo + 3*Bs;
```

If TermVector’s have not been computed, the operations have no effect!

In order to be more efficient, the linear combination of TermVector’s is delayed up to the assign (=) operation or a constructor operation. It means that some expression may not be evaluated and produce warning/error message related to LcTerm class.

```cpp
...  
TermVector W = U + V; // Ok  
cout<<U+V; // NOT EVALUATED
```

Besides, it is possible to convert TermVector:

```cpp
...  
Space V(omega, P1, "V", true);  
Unknown u(V, "u");  
LinearForm fv=intg(omega, f*u);  
TermVector B(fv, "B");  
B.toAbs();
```
Be cautious, once it is converted it is not possible to go back.

Inner/hermitian product and standard norms are provided:

```cpp
Complex innerProduct(TermVector, TermVector);
Complex hermitianProduct(TermVector, TermVector);
Complex operator|(TermVector, TermVector);

Real norm(TermVector, Number l = 2);
Real norm1(TermVector);
Real norm2(TermVector);
Real norminfty(TermVector);
```

Notice that inner and hermitian product return always a complex even if vectors are real!

Some general informations may be retrieved, using the following member functions:

```cpp
TV.valueType() // value type (_real or _complex)
TV.size() // size counted in scalar
TV.nbDofs() // size counted in dofs
TV.nbDofs(u) // number of dofs related to unknown u
```

Some member functions give useful access to part of a TermVector object:

```cpp
TermVector U=TV(u); // access to u part as a TermVector
Reals V; TV.asVector(V); // reinterpret TermVector as a raw Vector
TermVector W=TV.onDomain(Sigma); // restrict to domain Sigma
W=TV[Sigma]; // restrict to domain Sigma (same as OnDomain)
Value val=TV.getValue(u, n); // access to n-th component of unknown u (n>=1)
TV.setValue(u, n, 3.); // set value of n-th component of unknown u
Value val=TV.evaluate(u, P); // evaluate at point P
Real v;
TV(P, v); // evaluate at point P
TV(u, P, v); // evaluate at point P, specifying unknown
```

Finally, a TermVector may be printed or saved into a file:

```cpp
...
LinearForm fv=intg(omega, f*u);
TermVector B(fv, "B");

cout<<"vector B "<<B;
B.print(cout);
saveToFile("file.dat", B, _vtk);
```

In this example, B is saved to a file in vtk format (format of paraview software). Other available formats are _vtu (paraview xml format) and _raw (only values are saved);

### 7.1.2 TermMatrix in details

TermMatrix is the algebraic representation of a bilinear form, say a matrix. It supports different types of storage and possibly, has to take into account essential conditions. So there are different constructors of TermMatrix from bilinear forms:
opt1, opt2, opt3 are any options picked in the list

- \_compute, \_notCompute, \_assembled, \_unassembled
- \_nonSymmetricMatrix, \_symmetricMatrix, \_selfAdjointMatrix, \_skewSymmetricMatrix, \_skewAdjointMatrix
- \_csRowStorage, \_csColStorage, \_csDualStorage, \_csSymStorage, \_denseRowStorage, \_denseColStorage, \_denseDualStorage, \_skylineSymStorage, \_skylineDualStorage
- \_pseudoReductionMethod, \_realReductionMethod, \_penalizationReductionMethod

and name is an optional string used for printing purpose.

Some examples of TermMatrix construction:

Strings sidenames ("y=0","y=1","x=0","x=1");
Square sq (_origin=Point(0.,0.), _length=1, _nnode=20, _side_names=sidenames);
Mesh mesh2d (sq, _triangle, 1, _structured);
Domain omega=mesh2d.domain("Omega"); gamma=mesh2d.domain("x=0");
Space V(omega, P1, "V", true);
Unknown u(V,"u"); TestFunction v(u,"v");
BilinearForm auv=intg(omega, grad(u)|grad(v)), muv=intg(omega, u*v);

TermMatrix A(auv); //simplest constructor
EssentialConditions ecs=(u|gamma=0);
TermMatrix A0(auv,ecs,"A0"); //with ess. condition and naming
TermMatrix M(muv,notCompute); //defined but not computed

The computation algorithm chooses the well adapted matrix storage, generally compressed sparse storage or dense storage, taking into account symmetry property of the matrix. Using option, the storage method may be imposed at construction:

BilinearForm auv=intg(omega,grad(u)|grad(v));
TermMatrix A(auv,ecs,_skylineSymStorage,"A");

The available matrix storage are:

- the compressed sparse storage \_cs, generally the best one in terms of memory size
- the skyline storage \_skyline, required by direct solvers
- the dense storage \_dense

Each of these storages have different internal storages (say access) : \_row, \_col, \_dual, \_sym.
The storage may be changed after computation by using the `setStorage` function. Be cautious, some storage conversions may be time expansive.

It is also possible to construct void matrix, copy of matrix, diagonal matrix from `TermVector` or standard vector and matrix of the form $G(M_i, P_j)$ where $G$ is a kernel and $M_i$ and $P_j$ belongs to some geometrical domains:

```
TermMatrix(name);
TermMatrix(TermMatrix, name);
TermMatrix(TermVector, name);
TermMatrix(Unknown, Domain, Vector<T>, name);
TermMatrix(Unknown, Domain, Unknown, Domain, OperatorOnKernel, name);
```

In case of multiple unknowns bilinear form, block matrix may be extracted using unknowns as index:

```
BilinearForm auv=intg(omega1, grad(u1) | grad(v1)) + intg(omega2, grad(u2) | grad(v2));
TermMatrix A(auv);
TermMatrix A11=A(u1,v1);
```

The `TermMatrix` result is a copy of the extracted block!

The users can print the matrix and its storage, and save it to file in dense (`_dense`) or coordinate format (`_coor`). The coordinate format $(i,j,val)$ is well adapted to export sparse matrix to Matlab.

```
...  
BilinearForm auv=intg(omega1,grad(u1)|grad(v1)) + intg(omega2,grad(u2)|grad(v2));
TermMatrix A(auv, "A");

verboseLevel(30);
A.print(out);
out<<A;
A.viewStorage(out);
A.saveToFile("matA.dat", _coor);
```

As matrices are memory consuming, it is possible at any time to deallocate the memory allocated by a matrix:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
BilinearForm muv=intg(omega,u*v);
TermMatrix A(auv,"A"), M(muv,"M");
compute(A,M);
...
clear(A,M);
```

Only memory used to store matrix values is deallocated. It means that `clear` has no effect on a matrix that has not be computed!

`TermMatrices` may be combined using standard algebraic operators (`+=, -=, *=, /=, +, -, *, /`)

```
BilinearForm kuv=intg(omega,grad(u)|grad(v)),
     muv=intg(omega,u*v),
```

152
The sum (resp. the difference) of TermMatrixs involves a complex algorithm: sum is done by unknown blocks (nothing is done with a void block) and for each block, the common dofs numbering is searched, then the sum is performed. This process may induce the construction of a new matrix storage. When combining more than two matrices, it is better to write the summation in one step rather than in several steps.

Summing TermMatrixs is equivalent to sum bilinear forms in a new one and compute it:

\[
\text{BilinearForm} \quad auv=kuv-3*muv+mguv; \\
\text{TermMatrix} \quad A(auv,"A");
\]

Regarding memory consuming and time performance, this method is better.

Product of TermMatrix and TermVector are provided using the * operator:

\[
\begin{align*}
\text{TermMatrix} \quad A(auv,"A"); \\
\text{TermVector} \quad L(\text{uv},"L"); \\
\text{compute}(A,L); \\
\text{TermVector} \quad AL=A*L;
\end{align*}
\]

Matrix and vector must have compatible unknowns but some may be omitted (void blocks are ignored).

To conclude this section we give the example of the Helmholtz problem in waveguide using Dirichlet to Neuman map as transparent boundary condition. This exemple illustrates multiple uses of algebraic operators on TermMatrix.

```cpp
// define spectral space to deal with DtN
Number N=10;
Space Sp(sigmaP, Function(cosny,params), N, "cos(n*pi*y)");
Unknown phiP(Sp,"phiP");
Complexes lambda(N);
for(Number n=0; n<N; n++) lambda[n]=sqrt(Complex(k*k-n*n*pi*pi/(h*h)));
// define TermMatrix with no DtN
BilinearForm auv=intg(omega,grad(u)|grad(v)) - k*k*intg(omega,u*v);
TermMatrix A(auv,-csDualStorage,"A");
// contract DtN TermMatrix using matrix product
BilinearForm euv=intg(sigmaP,phiP*v),
```
This DtN approach using product of matrices was the MELINA approach. In XLife++ it is better to use TensorKernel approach:

```
...  
Number N=10;  
Space Sp(sigmaP, Function(cosny,params), N, "cos(n*pi*y)");  
Unknown phiP(Sp,"phiP");  
Complexes lambda(N);  
for(Number n=0; n<N; n++) lambda[n]=sqrt(Complex(k*k-n*n*pi*pi/(h*h))));  
TensorKernel tkp(phiP,lambda);  
BilinearForm auv = intg(omega,grad(u)|grad(v)) - k*k*intg(omega,u*v)  
    - i*intg(sigmaP,sigmaP,u*tkp*v);  
TermMatrix A(auv, "A");  
```

Advanced usage

When computing eigen values of a TermMatrix with essential conditions that have been reduced using the pseudo reduction method, you may be annoyed by spurious eigen values corresponding to the residual diagonal block. These eigen values may be shifted by modifying the diagonal coefficient of this residual block:

```
...  
EssentialConditions ecs = (u|sigma=0);  
TermMatrix A(auv,ecs,ReductionMethod(_pseudoReduction,100.) "A"); // shift by 100  
```

7.1.3 HMatrix

In the context of integral equation, HMatrix method consists in using a hierarchical representation (tree) of the BEM matrix, each tree node being either a real submatrix (leaf) or a virtual submatrix addressing up to four nodes:

```
Figure 7.1: Hierarchical matrix
```

154
Then some sub-matrices may be "compressed" to save memory and time computation. The matrix may be not squared and therefore the sub-matrices too.

The hierarchical structure of Hmatrix is based on clusters of row indices or column indices (in FE context, clusters of dofs supported by mesh domains):

![Cluster of points](image1.png) ![Clustering of triangles using centroids](image2.png)

When building the tree structure of the HMatrix by a recursive division algorithm that travels the row and column clusters, some a priori geometrical rules are used to know if a sub-matrix will be compressed later, say admissible sub-matrix. When a sub-matrix is admissible, it is not divided. Up to now only the following boxes rule is available:

**Admissibility rule** A sub-matrix is admissible if the bounding box \( B_r \) of the row cluster node and the bounding box \( B_c \) of the column cluster node satisfy:

\[
\text{diam}(B_r) \leq 2 \eta \text{dist}(B_r, B_c).
\]

Default value of \( \eta \) is 1.

If a sub-matrix is admissible then it can be compressed using several methods. All the methods proposed try to get a low rank approximation of the original sub-matrix of the form:

\[
UDV^*
\]

where \( U \) is a \( m \times r \) matrix, \( V \) is a \( n \times r \) matrix and \( D \) is a \( r \times r \) diagonal matrix. The rank of a such matrix is at most \( r \). So it is a low rank representation of a \( m \times n \) matrix if \( r \) is small compared to \( m, n \).

The Singular Value Decomposition (SVD) gives an exact "approximation" of a \( m \times n \) matrix. So, from the Eckart–Young–Mirsky theorem, approximate matrix of low rank can be produced by keeping a small number of the largest singular values. Because, this approach requires a full SVD computation that is expansive, alternative methods are based on random SVD which consists in capturing the matrix range using only few gaussian random vectors and doing a SVD on a smaller matrix. Nevertheless, these approximate SVD methods still require the full computation of the matrix, so adaptative cross approximation (ACA) methods computing only some rows and columns of the matrix are faster but less robust.
The figure 7.2 shows an example of structure of a BEM HMatrix (2500 × 2500) computed by XLiFE++:

![HMatrix with a sphere cluster, non admissible blocks are in red](image)

**Figure 7.2: HMatrix with a sphere cluster, non admissible blocks are in red**

**How to involve HMatrix computation?**

By default, XLiFE++ build BEM matrix in dense storage. To involve HMatrix storage (and computation) XLiFE++ uses a special integration method in the bilinear form describing the BEM term: the HMatrixIM object that can be handled as follows

```cpp
HMatrixIM him(clmeth, minrow, mincol, hmapp, rank, im);
```

or

```cpp
HMatrixIM him(clmeth, minrow, mincol, hmapp, eps, im);
```

where

- **clmeth**: the clustering method, one of `_regularBisection`, `_boundingBoxBisection`, `_cardinalityBisection`, `_uniformKdtree`, `_nonuniformKdtree`
- **minrow, mincol**: the minimum size of sub-matrix, more precisely a sub-matrix is divided if it is not admissible and if its number of rows/columns is greater than `minrow/mincol`
- **hmapp**: the approximate matrix methods, one of `_noHMApproximation`, `_svdCompression`, `_rsvdCompression`, `_r3svdCompression`, `_acaFull`, `_acaPartial`, `_acaPlus`
- **rank**: the desired rank of approximate matrix
- **eps**: the desired precision of approximated matrix
- **im**: an integration method for double integral
The figure 7.3 illustrates the difference between clustering bisection methods with a disk mesh of 1673 points and at most 10 points by box; different colors correspond to different tree node levels.

Figure 7.3: clustering of a disk mesh using regular (left), bounding box (middle), cardinality (right) bisection methods.

and the figure 7.4 shows the cluster get when using the kdtree (quadtree in 2D) methods with at most 5 nodes by box.

Figure 7.4: clustering of a disk mesh using kdtree methods - _uniformKdtree (left) and _nonuniformKdtree(right).

When specifying _noHMApproximation all the sub-matrices (admissible and not admissible) are computed and not compressed. There is no real advantage, except that the assembly appears to be faster than the standard assembly in dense matrix, in particular when multi-threading is enable.

_svdCompression corresponds to some truncated svd either a rank truncature when rank is given or a eps truncature (keep all singular values greater than eps).

_rsvdCompression uses random svd methods that are faster than full svd methods. As svd methods, user can choose either a rank truncature or a eps truncature. _r3svdCompression is a more sophisticated random svd that does only eps truncature; do not choose _r3svdCompression with a rank parameter!

_acaFull, _acaPartial, _acaPlus are adaptative cross approximation methods that use some rows and columns of the matrix to build a low rank matrix. _acaFull method requires all the row and the columns of the matrix, so it gives good approximates but it is not of a great interest compared to the random svd methods. _acaPartial and _acaPlus (an improvement of _acaPartial ) are more interesting because they use only several rows and columns of matrix,
saving time computation of BEM coefficients. But they are less robust!

The following XLiFE++ gives some examples of how to compute BEM HMatrix:

```cpp
// mesh sphere and define domain, space, unknown
Mesh meshd(Sphere(_center=Point(0.,0.,0.),_radius=1.,_nnodes=9,
                  _domain_name="Omega"), _triangle,1,_subdiv);
Domain omega=meshd.domain("Omega");
Space W(omega,P0,"V",false);
Unknown u(W,"u"); TestFunction v(u,"v");
// define kKernel, integration method and HMatrix parameters
Kernel G=Laplace3dKernel();
SauterSchwabIM ssim(5,5,4,3,2,4.);
HMatrixIM him(_cardinalityBisection, 20, 20, acaaplus, 0.00001, ssim);
// compute single layer matrix
BilinearForm alff=intg(omega,omega,u*Gl*v,him);
TermMatrix A(alf,"A"); //compute double double layer matrix
BilinearForm blf=intg(omega,omega,u*ndotgrad_y(G)*v,him)−0.5*intg(omega,u*v);
TermMatrix B(blf,"B");
```

Note that when adding a sparse FE matrix and a HMatrix, the result is a HMatrix. This is only possible if integrals are supported by the same domain. Indeed, FE matrix addresses non-admissible blocks of the HMatrix. Be care when combining some matrices.

⚠️ HMatrix integration method (HMatrixIM) is only available for double integral (BEM). Do not use withe a single integral bilinearform!

⚠️⚠️ When building HMatrixIM object passing clustering parameters, the row and column clusters will be computed when computing HMatrix, and referenced by your HMatrixIM object. As the row and column clusters are not re-computed if they have been built, do not re-use your HMatrixIM object in a bilinear form having an other domain support than this you have first involved. If you want re-use it on a different domain, call the clear method that frees the row and column cluster:

```cpp
HMatrixIM him(_cardinalityBisection, 20, 20, acaaplus, 0.00001, ssim);
BilinearForm alff=intg(omega,omega,u*Gl*v,him);
TermMatrix A(alf,"A"); //row and column clusters are built
him.clear(); //deallocates row and column clusters
```

⚠️⚠️⚠️ HMatrix does not work yet for problems with vector unknown!

The figure 7.5 and 7.6 give an idea of the efficiency of the aca+ and r3svd methods compared to the computations done with a dense matrix.
We note that the assembly computation time with Hmatrix and no approximation is better than the computation time get with dense matrix. Is due to a better parallelization of HMatrix assembly. The ACA+ methods is significantly faster than the other methods!

HMatrix supports the matrix/vector product so it may be used with iterative methods but it is not supported by direct linear solvers!

### 7.2 Linear Solvers

After a problem is well-defined in the form of Term: TermMatrix and Vector, it can be easily solved with a direct solver or with an iterative solver. XLiFE++ provides a wide set of linear
equation solvers. The following section explains some simple steps to make use of these solvers.

### 7.2.1 Direct solvers

Because direct solvers involve some complicated algorithms to solve very large linear systems through LDLt or LU factorization, a prerequisite to call them is to have TermMatrix factorized. It can be done like below:

```latex
\text{TermMatrix} \text{ LD}; // Create a new TermMatrix to store factorized result
\text{ldltFactorize}(A, \text{LD}); // LDLt-Factorize the TermMatrix
```

Then the linear system is solved with a very simple code:

```latex
\text{TermVector} X = \text{factSolve} (\text{LD}, B);
```

The TermVector U is returned as a solution of the solver. Or a TermMatrix can be factorized into LU before being solved:

```latex
\text{TermMatrix} \text{ LD}; // Create a new TermMatrix to store factorized result
\text{luFactorize}(A, \text{LD}); // LU-Factorize the TermMatrix
\text{TermVector} X = \text{factSolve} (\text{LD}, B);
```

Factorisation and solving may be called in one time:

```latex
\text{TermVector} X = \text{luSolve} (\text{LD}, B);
```

The available factorisations and direct solvers are:

- LDLt factorisation and solver (for symmetric matrix): routines \text{ldltFactorize}, \text{ldltSolve}
- LDLstar factorisation and solver (for self-adjoint matrix): routines \text{ldtstarFactorize}, \text{ldlstarSolve}
- LU factorisation and solver (for any matrix): routines \text{luFactorize}, \text{luSolve}
- umfpack factorisation and solver (for any matrix) if umfpack is installed and configured: \text{umfpackFactorize}, \text{umfpackSolve}
- gauss elimination with pivoting for matrix stored in dense format: \text{gaussSolve}

LDLt, LDLstar and LU move matrix to skyline storage and may fail even if the matrix is invertible (no pivoting strategy)! Umfpack is most powerful because it works with compressed sparse storage and has pivoting strategy.

Be sure of symmetry property of your matrix before calling LDLt or LDLstar methods. If you are not, call generic direct solver \text{directSolve} which performs tests before calling the well adapted method:

```latex
\text{TermMatrix} \text{ Af}; // create a new TermMatrix to store factorized result
\text{factorize}(A, \text{ Af}); // factorize the TermMatrix
\text{TermVector} X = \text{factSolve} (\text{Af}, B); //solve factorized linear system
\text{TermVector} X = \text{directSolve} (A, B); //same in one call
```

All solvers support multiple right hand sides given as a TermVectors or a std::vector<TermVector>: 160
TermVectors Bs;
...
TermVectors Xs = directSolve(A, Bs);

7.2.2 Iterative solvers

Unlike direct solvers, the iterative ones are delivered with very simple interface. In contrast to direct solvers, iterative methods approach the solution gradually, rather than in one large computational step. Up to now, there are several built-in iterative solvers of XLiFE++:

- Conjugate Gradient (CG, CGS, BiCG, BiCGStab)
- Generalized Minimal RESidual (GMRes)
- Quasi Minimal Residual (QMR)

These methods can be called with a preconditioner (SOR and SSOR excepted)

How to define a preconditioner?

To define a preconditionner, use the class PreconditionerTerm:

```c++
TermMatrix A;
real_t omega;
PreconditionerTerm precond(A, _ssorPrec, omega);
```

The PreconditionerTerm constructor takes 3 arguments:

- the matrix used to build the precondition matrix.
- the type of preconditioner. possible values are:
  - _luPrec the precondition matrix will be a LU precondition of the input matrix given
  - _ldltPrec the precondition matrix will be a LDLt precondition of the input matrix given
  - _ldlstarPrec the precondition matrix will be a LDL* precondition of the input matrix given
  - _ssorPrec the precondition matrix will be a SSOR precondition of the input matrix given
  - _diagPrec the precondition matrix will be a diagonal precondition of the input matrix given
  - _embeddedPrec the precondition matrix will be the input matrix given, with no transform

  Its default value is _embeddedPrec

- the relaxation parameter when SSOR precondition. It is optional, and its default value is 1.0
How to call an iterative solver?

To invoke an iterative solver and make use of it, the easiest way is to call external functions:

\[
\text{TermVector } U = \text{cgSolve}(A, B); // Solve with default initial guess } X0=0
\]

The available functions are \text{bicgSolve}, \text{bicgStabSolve}, \text{cgSolve}, \text{cgsSolve}, \text{gmresSolve}, and \text{qmrSolve}. They all call a general function \text{iterativeSolve}. All these functions take parameters in the following orders:

1. the matrix A (\text{TermMatrix})
2. the right hand side B (\text{TermVector})
3. optionally the initial guess X0 (\text{TermVector})
4. optionally the preconditioner P (\text{PreconditionerTerm})
5. optionally one or more keyvalue parameters, among the following:
   - \_solver Only for routine \text{iterativeSolve}, it needs one item among \_cg, \_cgs, \_qmr, \_bicg, \_bicgstab, \_gmres. This parameter is not optional.
   - \_tolerance tolerance of the iterative solver. Default value is 1e-6
   - \_maxIt Maximum number of iterations. Default value is ten times the number of unknowns
   - \_verbose verbose level. Default value is 0.
   - \_krylovDim Only for routine \text{gmresSolve}, the krylov dimension.

An advanced use of solvers would be to instantiate an iterative solver object and call it using operator ():

\[
\text{CgSolver } mySolver ; \quad // \text{Define an iterative solver object} \\
\text{TermVector } U = mySolver(A, B, X0) ; \quad // \text{Solve the system with initial guess } X0
\]

For objects \text{BicgSolver}, \text{BicgStabSolver}, \text{CgSolver}, \text{CgsSolver}, \text{QmrSolver}, parameters of the constructor are respectively the tolerance (default value is 1e-6), the maximum number of iterations (default value is 10000) and the verbose level (default value is 0). For object \text{GmresSolver}, parameters of the constructor are respectively the krylov dimension (default value is 20), the tolerance (default value is 1e-6), the maximum number of iterations (default value is 10000) and the verbose level (default value is 0).

\[
\text{CgSolver } mySolver(1.e-04, 20); \quad // \text{Solve the linear problem} \\
\text{TermVector } U = mySolver(A, B, X0); \quad // \text{Solve the linear problem}
\]

In the code above with double precision, the tolerance is made looser than default, for a faster solution with a convergence error being \(10^{-4}\). Nevertheless, the solver will cease after 20 iterations even if the solution has not been converged. It is a big disadvantage of the iterative solvers: they do not always “just work”. Different problems do require different iterative solver settings, depending on the nature of the governing equation being solved. However, the advantage of the iterative solvers is their memory usage, which is significantly less than a direct solver for the same sized problems. Look at the example “Helmholtz problem with CG solver” to know more how to write code with iterative solvers.
7.3 Eigen solvers

XLiFE++ currently provides a built-in solver which targets Hermitian and non-Hermitian eigenvalue problems, standard or generalized, and a wrapper to the well known external library ARPACK via its companion package ARPACK++. The internal solver is provided in case ARPACK is not available (see subsection 1.1.3); as far as possible, the latter should be preferred.

In the following, we will denote the problems using the generic form:

- \( Ax = \lambda x \), for a standard eigenvalue problem,
- \( Ax = \lambda B x \), for a generalized eigenvalue problem.

The couple \((\lambda, x)\) is called an eigen pair, that consists of an eigenvalue \(\lambda\) and the corresponding eigenvector \(x\). The nature of the problem to be solved is determined by the matrix \(A\): real or complex, symmetric or not, etc.

Both solvers can be used in a rather uniform way, although some parameters may be specific to one package or the other. The calling sequence requires a few mandatory arguments; some optional ones are provided by the user in the form “\_key = value”.

In the following, we describe some features common to both solvers, targeting in particular the result object. Then the parameters governing the computation are described for the built-in solver, followed by those related to ARPACK, including a special help paragraph that is worth to be mentioned right now. At last, two special sections are devoted to post-computation information retrieving and an advanced usage of ARPACK.

7.3.1 How to call an eigen solver?

Given two suitable TermMatrix objects \(A\) and \(B\), corresponding to the mathematical operators \(A\) and \(B\) above, a few eigen elements can be computed as the result of one of the generic calling sequence:

\[
\begin{align*}
\text{EigenElements } \text{ees} &= \text{eigenSolve}(A); & \quad \text{// standard eigenvalue problem} \\
\text{EigenElements } \text{eeg} &= \text{eigenSolve}(A, B); & \quad \text{// generalized eigenvalue problem}
\end{align*}
\]

In this example, 10 (the default number) eigen pairs are computed from the unique knowledge of the mandatory arguments \(A\), or \(A\) and \(B\); the other parameters are left to their default values. The function \text{eigenSolve} automatically selects ARPACK if it is available, or the internal solver otherwise.

\text{Remark.}

The user may choose himself by adding the argument \_solver=_intern or \_solver=_arpack.

If present, the other parameters, given in the form “\_key = value”, are filtered and passed to the specific solver; indeed some of them may be amended to benefit from experience feedback, in which case an information message is printed in the main print file of XLiFE++.

The user has always direct access and full control over the parameters by using the specific functions: the function \text{eigenInternSolve} provides a direct access to the built-in solver, while \text{arpackSolve} uses ARPACK. With these functions, the statements in the previous example would become:

\[
\begin{align*}
\text{// specific call to internal engine} \\
\text{EigenElements } \text{eesi} &= \text{eigenInternSolve}(A); & \quad \text{// standard eigenvalue problem} \\
\text{EigenElements } \text{eegi} &= \text{eigenInternSolve}(A, B); & \quad \text{// generalized eigenvalue problem}
\end{align*}
\]
7.3.2 Results

All these functions store their result in an `EigenElements` object that holds two containers:

- **values**, containing the list of the found eigenvalues,
- **vectors**, containing the list of the corresponding eigenvectors.

The eigenvectors are always computed together with the eigenvalues. Both containers have the same size which can be obtained with the member function `numberOfEigenValues()`. The list `values` is in fact a vector of complex numbers (even if the problem is real symmetric), and `vectors` is a vector of `TermVector` objects. Given an `EigenElements` object `eeg`, these containers can be used directly by `eeg.values` and `eeg.vectors` with the standard C++ syntax; alternatively, some member functions are available to extract the eigen pairs using their number, starting at 1. Their names are simply `value` and `vector`. For example, the following code prints the computed eigenvalues stored in `eeg`, the real part and the imaginary part being separated with a white space if they are complex:

```cpp
if (eeg.isReal()) {
    // eigenvalues and eigenvectors are real
    for (int i = 1; i < eeg.numberOfEigenValues(); i++) {
        cout << eeg.value(i).real() << endl;
    }
} else {
    // eigenvalues or eigenvectors may not be real
    for (int i = 1; i < eeg.numberOfEigenValues(); i++) {
        cout << eeg.value(i).real() << " " << eeg.value(i).imag() << endl;
    }
}
```

The type of the eigenvalues depend on the problem. It can be retrieved by the member function `isReal()`, as shown above, which returns `true` if the problem is real symmetric, `false` otherwise. The eigenvalues are always returned as complex numbers, even if the problem is real symmetric in which case the imaginary parts are set to 0. The eigenvectors are real if the problem is real symmetric, complex otherwise.

By default, the eigenvalues are sorted by increasing module; the eigen pairs are internally stored according to this order. There are several sorting possibilities which can be specified by the `sort` key (see below).

The eigenvectors can be easily used in the following of the program, since they are available as `TermVector` objects. They can also be saved individually into a file using one of the output format, in order to be plotted afterwards. The statement:

```cpp
saveToFile("V1", eeg.vector(1), _vtk);
```

creates the file `V1_Omega.vtk`, whose name is build from the prefix given by the user and the name of the domain where the solution is computed; the suffix is automatically appended according to the output format (here `.vtk`).

Moreover, an `EigenElements` object can be saved in multiple files in a single statement:
The names of all the created files will begin with the same prefix given as first argument (here EV). The eigenvalues will be written in the file EV_eigenvalues and the $i^{th}$ eigenvector will be written in the file EV_i_DomainName.ext, where DomainName will be replaced with the domain name and the extension depend on the chosen format (here .vtk). The eigenvalues are printed in the file EV_eigenvalues from the first one to the last one according to the chosen sorting criterion; the eigenvectors are printed in files whose numbers follow the same order.

### 7.3.3 Calling sequence

Let’s recall that the functions `eigenSolve`, `eigenInternSolve` and `arpackSolve` have two main calling sequences according to the kind of problem to define. The arguments can be:

- **A, _key1=value1, . . . ,keyN=valueN**, in the case of a *standard* eigenvalue problem,
- **A, B, _key1=value1, . . . ,keyN=valueN**, in the case of a *generalized* eigenvalue problem.

The arguments **A** and **B** are `TermMatrix` objects. The others (key, value) pairs are not mandatory. They are used to specify some particular settings. They can be given in any order and their list is given in the corresponding sections below.

**Optional parameters for the built-in eigen solver in details**

The built-in eigen solver accepts the following keys:

- **nev** (integer) number of eigen elements to be computed. The default value is 10.
- **which** (string) specifies which part of the spectrum is to be scanned. The default value is “LM”, for largest magnitude. The other possible value is “SM”, for smallest magnitude.
- **sigma** (real or complex) shift value $\sigma$ used in the spectral transformation in order to scan a portion of the spectrum around $\sigma$.
- **mode** (enumeration) Two computational modes are implemented:
  - the block Krylov-Schur method, based on Krylov decomposition with Rayleigh quotient ably reduced to Schur form, and suitable for hermitian and non hermitian eigenvalue problems. To call it, use the value `_krylovSchur`. This is the default.
  - the block Davidson method, suited only for hermitian problems and sometimes faster than the block Krylov-Schur algorithm. To call it, use the value `_davidson`.
- **tolerance** (real) precision of the computation. The default value is 1e-6.
- **maxIt** (integer) maximum number of iterations. The default value is 10000.
- **verbose** (integer) verbosity level. The default value is 0.
- **sort** (enumeration) sort criterion. The default value is `_incr_module`, which means “by increasing module”. One can also sort by increasing real part (_incr_realpart) and by increasing imaginary part (_incr_imagpart); conversely, one can sort by decreasing order by selecting one of _decr_module, _decr_realpart or _decr_imagpart.
Examples:
The following call computes the 20 eigenvalues of largest magnitude (and the corresponding
eigenvectors) of a generalized eigenvalue problem using the block Davidson method:

\[
\text{Number nev = 20;}
\]
\[
\text{EigenElements ee = } \text{eigenInternSolve}(A, B, \_nev=nev, \_which="LM", \
\_mode=_davidson);
\]

The following call computes nev eigenvalues around the complex shift value 2.5 + i (and the
 corresponding eigenvectors) of a generalized eigenvalue problem using the block Krylov-Schur
 method:

\[
\text{Complex sig = (2.5, 1.)};
\]
\[
\text{EigenElements ee = } \text{eigenInternSolve}(A, B, \_nev=nev, \_sigma=sig);
\]

Optional parameters for Arpack solver in details
The Arpack solver accepts the following keys:

\_nev (integer) number of eigen elements to be computed. The default value is 10.

\_which (string) specifies which part of the spectrum is to be scanned. The default value is "LM",
for largest magnitude. The possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE</td>
<td>eigenvalues from both ends of the spectrum</td>
</tr>
<tr>
<td>LA</td>
<td>eigenvalues with largest algebraic value</td>
</tr>
<tr>
<td>SA</td>
<td>eigenvalues with smallest algebraic value</td>
</tr>
<tr>
<td>LM</td>
<td>eigenvalues with largest magnitude</td>
</tr>
<tr>
<td>SM</td>
<td>eigenvalues with smallest magnitude</td>
</tr>
<tr>
<td>LR</td>
<td>eigenvalues with largest real part</td>
</tr>
<tr>
<td>SR</td>
<td>eigenvalues with smallest real part</td>
</tr>
<tr>
<td>LI</td>
<td>eigenvalues with largest imaginary part</td>
</tr>
<tr>
<td>SI</td>
<td>eigenvalues with smallest imaginary part</td>
</tr>
</tbody>
</table>

For symmetric problems, \_which must set to be one of LA, SA, LM, SM or BE. For real
nonsymmetric and complex problems, the alternatives are LM, SM, LR, SR, LI and SI.

\_sigma (real or complex) shift value \( \sigma \) used in the spectral transformation in order to scan a
portion of the spectrum around \( \sigma \).

\_mode (enuotation) when a shift is specified, some additional computational modes are
available. In order to activate one of them, one of the following keywords should be specified:

- \_buckling or \_cayley for the Buckling mode or the Cayley mode, for a generalized
  real symmetric problem,
• _cshiftRe or _cshiftIm for the complex shift invert mode, for a generalized real
nonsymmetric problem.

_tolerance_ (real) precision of the computation. The default value is set by ARPACK to the
machine epsilon.

_maxIt_ (integer) maximum number of iterations. The default value is computed by ARPACK.

_ncv_ (integer) number of Arnoldi vectors to be computed. It must be less than the dimension of
the problem. The default value is computed by ARPACK.

_forceNonSym_ parameter specified to force to use a nonsymmetric computational mode
although the problem is symmetric. This key does not take any value: it is present or
absent.

_verbos e_ (integer) verbosity level. The possible values are 0 or 1, and the default value is 0,
which means no output trace.

_sort_ (enumeration) sort criterion. The default value is _incr_module_, which means “by
increasing module”. One can also sort by increasing real part (_incr_realpart_) and by
increasing imaginary part (_incr_imagpart_); conversely, one can sort by decreasing order
by selecting one of _decr_module_, _decr_realpart_ or _decr_imagpart_.

For a better understanding of all those parameters, one should know that ARPACK classifies
the eigenvalue problems first as standard or generalized problems, and second according to the
matrix _A_ which can be real symmetric, real nonsymmetric or complex. This makes six categories
that all own at least two main computational modes called “regular” and “shift and invert”. The
“regular” mode is automatically selected if no shift is given. Some additional particular shifted
computational modes exist for generalized problems; this is summarized in the following table:

<table>
<thead>
<tr>
<th>Kind of problem</th>
<th>Computational mode</th>
<th>Relevant parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard, real symmetric,</td>
<td>Regular</td>
<td>_which</td>
</tr>
<tr>
<td>nonsymmetric or complex</td>
<td>Shift and invert</td>
<td>_sigma</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generalized</td>
<td>Regular</td>
<td>_which</td>
</tr>
<tr>
<td>real symmetric</td>
<td>Shift and invert</td>
<td>_sigma</td>
</tr>
<tr>
<td></td>
<td>Buckling</td>
<td>_sigma, _mode=_buckling</td>
</tr>
<tr>
<td></td>
<td>Cayley</td>
<td>_sigma, _mode=_cayley</td>
</tr>
<tr>
<td>Generalized</td>
<td>Regular</td>
<td>_which</td>
</tr>
<tr>
<td>real nonsymmetric</td>
<td>Real shift and invert</td>
<td>_sigma</td>
</tr>
<tr>
<td></td>
<td>Complex shift and invert (Re)</td>
<td>_sigma, _mode=_cshiftRe</td>
</tr>
<tr>
<td></td>
<td>Complex shift and invert (Im)</td>
<td>_sigma, _mode=_cshiftIm</td>
</tr>
<tr>
<td>Generalized</td>
<td>Regular</td>
<td>_which</td>
</tr>
<tr>
<td>complex</td>
<td>Shift and invert</td>
<td>_sigma</td>
</tr>
</tbody>
</table>
For a generalized eigenvalue problem:

- if $A$ is real, the matrix $B$ is required to be real symmetric positive semi-definite, except in regular mode where it should be real symmetric positive definite. In buckling mode, the real symmetric matrix $A$ is required to be positive semi-definite while $B$ is only required to be real symmetric indefinite;
- if $A$ is complex, the matrix $B$ is required to be hermitian positive semi-definite, except in regular mode where it should be positive definite. Notice that $B$ may still be real and symmetric.

It should be noticed that the parameters `nev`, `tolerance`, `maxIt`, `ncv` and `verbose` can be used in any case. On the contrary, `_which` and `_sigma` are mutually exclusive; the latter takes precedence over the former. Moreover, `_mode` indicates a particular shifted computational mode, and as such is ignored if it is used without `_sigma`.

For generalized problems, the shifted modes require the computation of $(A - \sigma B)^{-1}x$ (see the table in the Advanced usage of ARPACK section below). When $A$ is real nonsymmetric and $\sigma$ is complex, $(A - \sigma B)^{-1}$ is complex, but the internal computation steps of the algorithm are performed in real arithmetic (the vector $x$ is real). This saves memory requirements and computation time. This is the reason why the user should specify which part of the operator $(A - \sigma B)^{-1}$, real or imaginary, must be taken into account. Both strategies lead to comparable results (see ARPACK’s documentation). The parameter `_mode` should be set to `_cshiftRe` to select $\Re((A - \sigma B)^{-1})$; it should be set to `_cshiftIm` to select $\Im((A - \sigma B)^{-1})$, and in this case obviously, the imaginary part of $\sigma$ should not be nil.

At last, `_forceNonSym` is a switch useful to allow the computational modes of the real nonsymmetric case to be used for a real symmetric problem; indeed, it may happen that the symmetric algorithms fail, and using the nonsymmetric algorithms can be helpful to obtain a solution. As a last resort, we can also use the complex algorithms, but the entries of the matrix $A$ should be first converted to complex (to achieve that, see the second example in the section Advanced usage below).

Example:
The following call computes the 20 eigenvalues of smallest magnitude (and the corresponding eigenvectors) of a standard eigenvalue problem using the regular mode, with a prescribed tolerance:

```plaintext
Number nev = 20;
EigenElements ee = arpackSolve(A, _nev=nev, _which="SM", _tolerance=1.e-12);
```

Some hints about the parameters.
The convergence of the algorithms highly depends on the data. The ideal situation is when there is no multiplicity and the eigenvalues are well separated, which is rarely the case in practice. Here are some hints to help convergence to occur.

In regular mode, ARPACK is better used to search for eigenvalues of largest magnitude (this is why “LM” is the default value of the parameter `_which`). Thus, as far as possible, the problem should be written to use this mode. It may happen that the eigenvalues of smallest magnitude are hard to compute; in this case, try using the shifted mode which is generally very powerful.
If there is multiplicity or the eigenvalues are clustered, consider decreasing or on the contrary increasing the number of requested eigenvalues.

The number of iterations is by default computed by ARPACK and is generally large enough; if convergence is not attained, the tolerance (\_tolerance) or the number of Arnoldi vectors (\_ncv) should be modified in priority. By default, ARPACK sets the tolerance parameter to the machine epsilon which insures the computation to be performed with the highest possible precision. This represents the relative precision on the computed eigenvalues. It sometimes happens that this stopping criterion is unattainable and the tolerance value should be increased. On the other hand, a too loose value may lead the algorithm to miss some eigenvalues.

The last parameter that can be tuned is the number of Arnoldi vectors computed by the algorithm at each iteration. This parameter can greatly influence the convergence of the algorithm. It must be greater than the number of wanted eigenvalues nev and less than the problem dimension n. By default, ARPACK sets it to min(2*nev + 1, n-1). Increasing this value may facilitate the convergence; on the other hand, this increases the computational time and the memory consumption.

**Retrieving post-computation informations**

After a call to arpackSolve, one can inquire about informations related to the last computation using functions whose names are the names of the true ARPACK++ function names prefixed with ar (the true ARPACK++ functions are member functions that should be used in conjunction with an ARPACK object; these ones are external functions that can be directly used alone). The available functions are the following:

```cpp
bool arParametersDefined() ;
```

which returns true if all internal variables were correctly defined, false otherwise.

```cpp
int arConvergedEigenvalues() ;
```

which returns the number of eigenvalues found. This is the same value as the one provided by the member function numberOfEigenValues() already seen in the Results section above.

```cpp
int arGetMaxit() ;
```

which returns the maximum number of Arnoldi update iterations allowed.

```cpp
int arGetMode() ;
```

which returns the computational mode used as described in the following table:

<table>
<thead>
<tr>
<th>Value</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>regular mode (standard problems)</td>
</tr>
<tr>
<td>2</td>
<td>regular inverse mode (generalized problems)</td>
</tr>
<tr>
<td>3</td>
<td>shift and invert mode. For real nonsymmetric generalized problems, this option can also mean that a complex shift is being used but, in this case the operator is ( \Re((A - \sigma B)^{-1}) )</td>
</tr>
<tr>
<td>4</td>
<td>buckling mode (real symmetric generalized problems) or shift and invert mode with complex shift and the operator is ( \Im((A - \sigma B)^{-1}) ) (real nonsymmetric generalized problems)</td>
</tr>
<tr>
<td>5</td>
<td>Cayley mode (real symmetric generalized problems)</td>
</tr>
</tbody>
</table>
std::string arGetModeStr();

which returns a user friendly string describing the computational mode used. This function does not exist in ARPACK and has been written in complement to the previous one which gives a rather raw information.

int arGetIter();

which returns the number of Arnoldi update iterations actually taken by ARPACK to solve the eigenvalue problem.

int arGetN();

which returns the dimension of the eigenvalue problem.

int arGetNcv();

which returns the number of Arnoldi vectors generated at each iteration.

int arGetNev();

which returns the number of required eigenvalues. The number of eigenvalues actually found, however, is given by the function arConvergedEigenvalues.

std::complex<double> arGetShift();

which returns the shift $\sigma$ used to define the spectral transformation. This one is a slightly modified version of the original ARPACK’s one in that it returns a complex value; so if the problem is real symmetric, only the real part is relevant. If the problem is being solved in regular mode, this function will return 0.0. To avoid any confusion in this case, the user should call the function arGetMode before this one.

double arGetShiftImag();

which returns the imaginary part of the shift when the shift and invert mode is being used to solve a real nonsymmetric problem. This value is also returned as the imaginary part of the previous function.

double arGetTol();

which returns the stopping tolerance used to find the eigenvalues. It corresponds to the relative accuracy of the computed eigenvalues.

std::string arGetWhich();

which returns the part of the spectrum the user is seeking for. The returned string is one of those used in conjunction with the parameter _which above.

♠ A full example.
The following program computes the smallest eigenvalues of the Laplace operator on a segment with Neumann conditions. The approximation is made with the finite element method using a single element, the segment $[0, \pi]$, using an interpolation degree $k = 60$. The quadrature rule has degree $2k + 1$. The expected eigenvalues are the square of the integers, i.e. 0, 1, 4, 9, 16, 25, 36, 49, etc. After the call to arpackSolve, informations about the computation done are retrieved using most of the above functions and printed.
```cpp
#include "xlife++.h"
using namespace xlifepp;

int main() {
    using std::cout;
    using std::endl;
    init(); // mandatory initialization of XLiFE++

    int nbint=1; // number of intervals
    int dk=60;  // interpolation degree
    cout << "Interpolation degree = " << dk << endl;

    // mesh : segment [0, pi]
    Mesh zeroPi(Segment(_xmin=0, _xmax=pi, _nnodes=nbint+1), 1);
    Domain omega = zeroPi.domain("Omega");

    Interpolation& interp=interpolation(Lagrange, GaussLobatto, dk, H1);
    Space Vk(omega, interp, "Vk");
    Unknown u(Vk, "u");
    TestFunction v(u, "v");

    int qrodeg = 2*dk+1;
    BilinearForm muv = intg(omega, u * v, defaultQuadrature, qrodeg),
                    auv = intg(omega, grad(u) | grad(v), defaultQuadrature, qrodeg);

    // Compute the Stiffness and Mass matrices
    TermMatrix S(auv,"auv"), M(muv,"muv");

    // The eigenvalue problem writes S x = l M x
    // Compute the new first eigenvalues with smallest magnitude with Arpack
    int nev = 8;
    EigenElements areigs = arpackSolve(S,M, _nev=nev, _which="SM");

    cout.precision(6);
    cout << " Number of converged eigenvalues: " << arConvergedEigenValues() << endl;
    cout << " Computational mode: " << arGetModeStr() << endl;
    int mode = arGetMode();
    if (mode < 3) {
        cout << " Part of the spectrum requested: " << arGetWhich() << endl;
    } else {
        cout << " Shift used: " << arGetShift() + complex_t(0., arGetShiftImag()) << endl;
    }
    cout << " Problem dimension = " << arGetN() << endl;
    cout << " Nb_iter / Nb_iter_Max = " << arGetIter() << " / " << arGetMaxit() << endl;
    cout << " Tolerance = " << arGetTol() << endl;
    cout << " Number of Arnoldi vectors = " << arGetNcv() << endl;
    cout.precision(17);
    cout << "Eigenvalues: " << endl;
    int nconv = areigs.numberOfEigenValues();
    for (int i = 0; i < nconv; i++) { cout << areigs.value(i+1).real() << endl; }
    saveToFile("Sy", areigs, matlab);
}
```
The output produced by this program is the following:

Interpolation degree = 60
computing FE term intg_Omega grad(u) | grad(v), using 1 threads : done
computing FE term intg_Omega u * v, using 1 threads : done
Number of converged eigenvalues: 8
Computational mode: regular inverse mode (generalized problem)
Part of the spectrum requested: SM
Problem dimension = 61
Nb_iter / Nb_iter_Max = 443 / 800
Tolerance = 1.11022e-16
Number of Arnoldi vectors = 17
Eigenvalues :
1.6569854567517169e-10
0.99999999999869593
4.0000000000087512
9.00000000000560245
16.000000000016872
25.00000000006708
36.000000000077279
48.999999999728566

Moreover, the last statement produces nine files; their names are Sy_eigenvalues, which contains the eigenvalues, and Sy_i_Omega.m with i equal to 1, 2...8, which contain the components of the eigenvectors, one eigenvector per file. They are shown on the following figure:
The way the function \texttt{arpackSolve} is to be used, as presented so far, is sufficient if the operator $A$ can be expressed as a linear combination of \texttt{TermMatrix} objects, or more frequently as a unique \texttt{TermMatrix} object built from a combination of bilinear forms. However, when the definition of the operator involves other algebraic operations, like the computation of the inverse of a matrix for example, the above process cannot be used.

Indeed, it requires the creation of a specific object containing the sequence of operations needed to define the operator. In other words, the user has to write a class describing the way the operator can be computed. To do that, the user should know some fundamentals of the C++ programming language.

In the following, we give some general guidelines, followed by additional technical features shown on a first example; two other examples complete the description of the practical implementation.

\section*{General guidelines.}

In order to get rid of the technicalities of ARPACK++ and to help to the creation of the user class, a general frame has been prepared that involves the creation of an intermediate placeholder object whose type is \texttt{ArpackProb}, designed to hold the characteristics of the true ARPACK object.
internally created. The definition of the user class should be made in coherence with the ARPACK computational mode chosen, hold in the ArpackProb object. To summarize this, the user has to:

1. create a so-called user class to define the operators of the problem (this requires some programming work),

2. use it to create an ArpackProb intermediate object (this is straightforward),

3. call arpackSolve with this object as unique argument.

ARPACK++’s usage made here imposes the user class to define some matrix-vector products related to the computational mode chosen. The required products are described in the documentation of ARPACK++ and are summarized in the following table. The name of the member functions, MultOPx, MultBx and MultAx, are the generic names used in the documentation of ARPACK++. They have been kept here to make things easier and should be left unchanged in our context:
<table>
<thead>
<tr>
<th>Kind of problem</th>
<th>Computational mode</th>
<th>Matrix-vector products</th>
<th>Name of member functions to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard, real symmetric, nonsymmetric or complex</td>
<td>Regular</td>
<td>$y \leftarrow Ax$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Shift and invert</td>
<td>$y \leftarrow (A - \sigma I)\cdot x$</td>
<td>MultOPx</td>
</tr>
<tr>
<td>Generalized</td>
<td>Regular</td>
<td>$y \leftarrow B^{-1} Ax$ ; $x \leftarrow Ax$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Shift and invert</td>
<td>$y \leftarrow (A - \sigma B)^{-1} x$</td>
<td>MultOPx</td>
</tr>
<tr>
<td>real symmetric</td>
<td>Buckling</td>
<td>$y \leftarrow (A - \sigma B)^{-1} x$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Cayley</td>
<td>$y \leftarrow (A - \sigma B)^{-1} x$</td>
<td>MultOPx</td>
</tr>
<tr>
<td>Generalized</td>
<td>Regular</td>
<td>$y \leftarrow B^{-1} Ax$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Real shift and invert</td>
<td>$y \leftarrow (A - \sigma B)^{-1} x$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Complex shift and invert (Real part)</td>
<td>$y \leftarrow \Re((A - \sigma B)^{-1} x)$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Complex shift and invert (Imaginary part)</td>
<td>$y \leftarrow \Im((A - \sigma B)^{-1} x)$</td>
<td>MultOPx</td>
</tr>
<tr>
<td>complex</td>
<td>Regular</td>
<td>$y \leftarrow B^{-1} Ax$</td>
<td>MultOPx</td>
</tr>
<tr>
<td></td>
<td>Shift and invert</td>
<td>$y \leftarrow (A - \sigma B)^{-1} x$</td>
<td>MultOPx</td>
</tr>
</tbody>
</table>

In order to help the definition of the user class, we have found convenient to create it as a derived class of `ARStdFrame<real_t>`, `ARStdFrame<complex_t>`, `ARGenFrame<real_t>` or `ARGenFrame<complex_t>`, depending on the nature of the problem, `standard` or `generalized`, and its type, `real` or `complex`. As their name suggests, these classes are frames prepared to accommodate the definition of the matrix-vector product(s) required by the computational mode chosen. They are abstract classes that declare the matrix-vector products `MultOPx`, `MultBx` and `MultAx` as virtual functions that the user must provide. They have a unique constructor whose prototype is:

```cpp
template<class K> ARStdFrame(const TermMatrix& charMat);
template<class K> ARGenFrame(const TermMatrix& charMat);
```
The unique argument is a so-called characteristic matrix that allows to retrieve informations about the context of the problem such as its dimension and the associated unknowns. In practice, it is one of the matrices involved in the definition of the operator $A$. Those two classes derive themselves from the class ARInterfaceFrame that provides, in addition, the member function

```cpp
int GetN();
```

which returns the dimension of the problem to be solved.

We will now show how all this takes place and give further details on an example.

♠ User class example 1.

We consider again the problem of the Laplace operator on a segment with Neumann conditions (see the previous section). This problem is written and solved there as a generalized eigenvalue problem $Sx = \lambda Mx$. Now, assume we want to write this problem as a standard eigenvalue problem $M^{-1}Sx = \lambda x$, which is correct since the mass matrix $M$ is invertible. We are facing to the operator $A = M^{-1}S$ that cannot be handled in the framework presented in the previous sections. Thus, we write a special class StdNonSym whose definition is the following:

```cpp
class StdNonSym: public ARStdFrame<real_t> {
    public:
        // constructor
        StdNonSym(TermMatrix& S, TermMatrix& M);

        // destructor
        ~StdNonSym(){ delete fact_p; }

        // matrix-vector product required: $y = \text{inv}(M)Sx$  
        void MultOPx(real_t *x, real_t *y);

    private:
        // pointers to internal data objects
        const LargeMatrix<real_t> *matS_p, *matM_p;
        // pointer to temporary factorized matrix $M$
        LargeMatrix<real_t>* fact_p;
};
```

This class contains three member functions: a constructor (with two arguments which are the two matrices needed to define the problem), the destructor and the matrix-vector product required, whose name is MultOPx, as mentioned in the previous table.

Since we plan to use the regular computational mode, the function MultOPx should compute the result $y$ of $M^{-1}Sx$ for a given $x$. This is done in two steps: first compute $z = Sx$, second solve $My = z$ using a Cholesky factorization of $M$ which is symmetric positive definite. The function MultOPx may be called many times during the computation, so the Cholesky factorization of $M$ has to be computed once and stored. This is done in the constructor through the initialization of the pointer fact_p, along with the initialization of the two other pointers matS_p and matM_p (see below).

The destructor’s unique role is to free the memory allocated to store the Cholesky factorization. Let’s now describe the constructor and the matrix-vector product in more details. The implementation is the following:

```cpp
/*!
 * Assumptions (not checked) :  
 * $S$ real
 */
```
Since `StdNonSym` derives from `ARStdFrame`, the `ARStdFrame` constructor is first called, passing `S` as the characteristic matrix, and the pointers `matS_p` and `matM_p` are initialized. They hold the addresses of the low level `LargeMatrix` objects containing the effective real data values. The reason is that the algebraic operations are attached to the `LargeMatrix` class with the adequate storage type. Then the Cholesky factorization of `M` is computed in two steps: first record in the pointer `fact_p` the result of `newSkyline`, i.e. the address of a copy of the matrix `M` stored in skyline storage type, second call the function `ldltFactorize` to compute the Cholesky factorization.

The prototype of the function `MultOPx` is imposed by ARPACK. Each of the two arguments is the address of a C-style array. But the algebraic operations provided by XLiFE++ require operands whose type are `std::vector`. Thus, the data values should be copied in and out using the two utilitary functions `array2Vector` and `vector2Array`. The two vectors `lx` and `ly` are local buffers with the right size prepared for this purpose; they are members of `ARInterfaceFrame` and are ready to use. Thus, the input array `x` is first copied into the local vector `lx`, then the matrix-vector product `Sx` is computed by the function `multMatrixVector` and stored into the local vector `Sx`. Then comes the resolution of the linear system `M y = Sx` by the function `ldltSolve` which uses the precomputed Cholesky factorization through the pointer `fact_p`. At last, the result is copied from the local vector `ly` into the output array `y`.

The following action is to use this user class to create an `ArpackProb` intermediate object which will set the computational mode to be used by ARPACK. For this purpose, five constructors are available. In their list of arguments, `usrcl` denotes the user class, which bears the kind of problem to be solved, standard or generalized, and `nev` is the number of desired eigenvalues:

- Constructors for regular mode (for standard or generalized eigenvalue problems)

  The argument `which` defines the part of the spectrum to be scanned. The possible values are described in a previous section (see optional parameter `_which`).

  1. real case

```cpp
ArpackProb(const ARInterfaceFrame<real_t>& usrcl, int nev, const char* which, bool sym = true);
```

The last argument `sym` specifies by default to use the algorithm designed for a symmetric operator; if it takes the value `false`, then the algorithm designed for a nonsymmetric operator will be used.
2. complex case

\texttt{ArpackProb(const ARInterfaceFrame<complex_t>& usrc1, int nev, const char* which);}  

- Constructors for \textit{shifted} computational mode (for standard or generalized problems)
  Note: for standard eigenvalue problems, the last argument (computational mode \texttt{cMode}) is irrelevant and thus has not to be specified.

1. real symmetric case:
   - for standard problems: shift and invert mode (default)
   - for generalized problems: shift and invert mode (default), buckling and Cayley mode
2. real nonsymmetric case:
   - for standard or generalized problems: (real) shift and invert mode

\texttt{ArpackProb(const ARInterfaceFrame<real_t>& usrc1, int nev, bool sym, double sigma, char cMode = ’S’);}  

As above, the argument \texttt{sym} tells if the algorithm designed for the symmetric case (\texttt{true}) or nonsymmetric case (\texttt{false}) should be used. The argument \texttt{sigma} is the value of the shift (real number here). The buckling and Cayley modes can be selected by giving the argument \texttt{cMode} the character value ‘B’ or ‘C’ respectively.

2. real nonsymmetric case, for generalized problems only: complex shift and invert mode

\texttt{ArpackProb(const ARInterfaceFrame<real_t>& usrc1, int nev, double sigmaR, double sigmaI, char cMode = ’R’);}  

The shift is given by both its real part, \texttt{sigmaR}, and its imaginary part, \texttt{sigmaI}. The argument \texttt{cMode} should be set to ‘R’ to select $\Re((A - \sigma B)^{-1})$; it should be set to ‘T’ to select $\Im((A - \sigma B)^{-1})$ (for more explanations, see the description of the parameter \_mode in the previous section).

3. complex case, for standard or generalized problems: shift and invert mode

\texttt{ArpackProb(const ARInterfaceFrame<complex_t>& usrc1, int nev, double sigmaR, double sigmaI = 0.0);}  

The shift is given by both its real part, \texttt{sigmaR}, and its imaginary part, \texttt{sigmaI}.

In order to terminate this illustration, the last thing to do is to call the solver. Technically, we can reuse the program shown at the end of the previous section and called \textit{A full example.} and do the following:

1. copy the declaration of the user class \texttt{StdNonSym}, followed by its implementation as given above, just before the \texttt{main} function,

2. replace the call to \texttt{arpackSolve}:

\texttt{EigenElements areigs = arpackSolve(S,M,_nev=nev, _which=”SM”);}  

by the three lines:

\texttt{StdNonSym usrc1(S,M);}  
\texttt{ArpackProb Arpb(usrc1,nev,”SM”,false);}  
\texttt{EigenElements areigs = arpackSolve(Arpb);}
The first line creates an object called `usrcl` by calling the constructor of the user class to which the stiffness and mass matrices are passed. Then, the intermediate object `Arpb` is created using the first constructor in the list just above. This completely defines the ARPACK problem: the user class derives from `ARStdFrame<real_t>`, so it is a standard real problem; the eigenvalues of smallest magnitude are requested and the nonsymmetric algorithm is chosen. Indeed, the operator $A = M^{-1}S$ is not symmetric, so we should select the corresponding computational mode (this justifies the name given to the user class).

3. print the eigenvalues as complex numbers by removing the call to `real()` in the last line of the program:

```cpp
    for (int i = 0; i < nconv; i++) { cout << areigs.value(i+1) << endl; }
```

The output produced by this new program is the following:

```plaintext
Interpolation degree = 60
computing FE term intg_Omega grad(u) | grad(v), using 1 threads : done
computing FE term intg_Omega u * v, using 1 threads : done
Number of converged eigenvalues: 8
Computational mode: regular mode (standard problem)
Part of the spectrum requested: SM
Problem dimension = 61
Nb_iter / Nb_iter_Max = 266 / 800
Tolerance = 1.11022e-16
Number of Arnoldi vectors = 17
Eigenvalues :
(-3.9936942641816131e-12,0)
(0.99999999998896483,0)
(3.9999999999884972,0)
(8.9999999999974278,0)
(15.999999999996687,0)
(25.000000000000899,0)
(35.999999999997002,0)
(63.999999992827938,0)
```

We can observe that the last eigenvalue is close to 64 instead of 49 which has been missed. Inserting the line

```cpp
    Arpb.ChangeTol(1.e-15);
```

between the declaration of `Arpb` and the call to `arpackSolve` sets a slightly relaxed value of the tolerance that suffices to obtain the expected result:

```plaintext
Interpolation degree = 60
computing FE term intg_Omega grad(u) | grad(v), using 1 threads : done
computing FE term intg_Omega u * v, using 1 threads : done
Number of converged eigenvalues: 8
Computational mode: regular mode (standard problem)
Part of the spectrum requested: SM
Problem dimension = 61
Nb_iter / Nb_iter_Max = 252 / 800
Tolerance = 1e-15
Number of Arnoldi vectors = 17
```
Eigenvalues:
(-3.9815928332131989e-12,0)
(0.99999999998896794,0)
(3.9999999999885101,0)
(8.999999999974225,0)
(15.999999999966653,0)
(25.00000000000909,0)
(35.99999999996525,0)
(48.99999999971514,0)

♠ Other tuning functions
This gives the opportunity to mention that the parameters governing the computation can be set using exactly the same function names as the ones defined in ARPACK++. Besides the function ChangeTol just seen, we can use the following functions:

- **ChangeMaxit(int)** to change the maximum number of iterations,
- **ChangeNcv(int)** to change the number of Arnoldi vectors to be computed.
- **Trace()** to activate the output of statistics related to the computation.

For more information, see the description of the parameters _maxIt, _ncv and _verbose in the previous section.

♠ User class example 2.
As a second example, we can use the complex algorithm to solve the same problem. The operator $A = M^{-1}S$ should be complex; for this, we choose to convert the matrix $S$ to complex. The corresponding user class **StdComp** is a slight modification of the class **StdNonSym**:

```cpp
class StdComp : public ARStdFrame<complex_t> {
public:
    // constructor
    StdComp(TermMatrix& S, TermMatrix& M);

    // destructor
    ~StdComp(){ delete fact_p; }

    // matrix-vector product required : $y = \text{inv}(M)Sx$
    void MultOPx (complex_t *x, complex_t *y);

private:
    // pointers to internal data objects
    const LargeMatrix<complex_t> *matS_p;
    const LargeMatrix<real_t> *matM_p;
    // pointer to temporary factorized matrix $M$
    LargeMatrix<real_t> *fact_p;
}; // end of Class StdComp
```

The modifications concern the type of $S$ and the operands $x$ and $y$ changed to complex. The implementation is thus quite similar to the **StdNonSym** one:

```cpp
/*! Assumptions (not checked) :
    S complex
    M real symmetric positive definite
```
One can just mention the initialization of the pointer \texttt{matS\_p} to the complex data values. The factorization of \texttt{M} is unchanged and it should be noticed that here the Cholesky solver handles complex data. The final step consists in modifying the previous program by replacing the declaration and the implementation of \texttt{StdNonSym} by those of the class \texttt{StdComp} as given above, and replace the three lines:

\begin{verbatim}
StdNonSym usrc1(S,M);
ArpackProb Arpb(usrc1, nev, "SM", false);
EigenElements areigs = arpackSolve(Arpb);
\end{verbatim}

by the four lines:

\begin{verbatim}
TermMatrix Sc = toComplex(S);
StdComp usrc1(Sc,M);
ArpackProb Arpb(usrc1, nev, "SM");
EigenElements areigs = arpackSolve(Arpb);
\end{verbatim}

The first statement converts the matrix \texttt{S} to the complex one \texttt{Sc}. Then the object corresponding to the user class \texttt{usrc1} is created from the complex stiffness matrix and the real mass matrix. The intermediate object \texttt{Arpb} is created using the second constructor in the list given above. This completely defines the \texttt{ARPACK} problem: the user class derives from \texttt{ARStdFrame<complex\_t>}, so it is a standard complex problem ; the eigenvalues of smallest magnitude are requested. All the other parameters are the default ones.

The output produced by this last program is the following:

Interpolation degree = 60
computing FE term intg\_Omega grad(u) \times grad(v), using 1 threads : done
computing FE term intg\_Omega u \times v, using 1 threads : done
Number of converged eigenvalues: 8
Computational mode: regular mode (standard problem)
Part of the spectrum requested: SM
Problem dimension = 61
Nb\_iter / Nb\_iter\_Max = 235 / 800
Tolerance = 1.11022e-16
Number of Arnoldi vectors = 17
Eigenvalues :
(1.6895427007126359e-10,-5.3921458890663075e-11)
User class example 3.

At last, we can create a user class defining a generalized problem, what we were starting from and thus redoing in fact what is already done internally when the first calling sequence of the function `arpackSolve` presented in the previous section is used. The corresponding user class `GenSym` is the following:

```cpp
class GenSym : public ARGenFrame<real_t> {
public:
    // ! constructor
    GenSym(TermMatrix& S, TermMatrix& M);

    // ! destructor
    ~GenSym(){ delete fact_p; }

    // ! matrix-vector products required : y <- inv(M)*S * x and x <- S * x
    void MultOPx (real_t *x, real_t *y);

    // ! matrix-vector product y <- M * x
    void MultBx (real_t *x, real_t *y);

private:
    // ! pointers to internal data objects
    const LargeMatrix<real_t> *matS_p, *matM_p;
    // ! pointer to temporary factorized matrix M
    LargeMatrix<real_t>* fact_p;
}; // end of Class GenSym
```

Take notice that this class derives from `ARGenFrame` and in accordance with ARPACK’s requirements for a generalized real symmetric problem (see the table above), this class provides the two matrix-vector products `MultOPx` and `MultBx`. The implementation is very similar to the `StdNonSym`’s one:

```cpp
/*
 * Assumptions (not checked) :
 * S real symmetric
 * M real symmetric positive definite
 */
GenSym::GenSym(TermMatrix& S, TermMatrix& M)
    : ARGenFrame(S), matS_p(S.matrixData()->rEntries_p),
      matM_p(M.matrixData()->rEntries_p) {

    fact_p = newSkyline(matM_p);
    ldltFactorize(*fact_p);
}

// ! Matrix-vector products y <- inv(M)*S * x and x <- S * x
void GenSym::MultOPx (real_t *x, real_t *y) {
    array2Vector(x, lx);
```
std::vector<real_t> Sx(GetN());
multMatrixVector(*matS_p, lx, Sx);
vector2Array(Sx, x);

// Solve linear system. Matlab equivalent: ly = matM_p \ Sx;
(fact_p->ldltSolve)(Sx, ly); // store the solution into ly
vector2Array(ly, y);
}

/// Matrix-vector product y <- M * x
void GenSym::MultBx(real_t *x, real_t *y) {
  array2Vector(x, lx);
multMatrixVector(*matM_p, lx, ly);
vector2Array(ly, y);
}

The constructor’s code is identical. The function MultOPx computes the same product $M^{-1}S$; it additionally stores $Sx$ in $x$ which is here both an input and an output argument as required by ARPACK. The function MultBx simply computes the product $Mx$.

Again, the initial program (A full example. above) can be modified by inserting the declaration and the implementation of the user class GenSym before the main function and replace the call to arpackSolve by:

```cpp
GenSym usrc1(S,M);
ArpackProb Arpb(usrc1,nev,"SM");
EigenElements areigs = arpackSolve(Arpb);
```

The first line creates an object called usrc1 by calling the constructor of the user class to which the stiffness and mass matrices are passed. Then, the intermediate object Arpb is created using the first of the constructors of the class ArpackProb given above. This completely defines the ARPACK problem: the user class derives from ARGenFrame<real_t>, so it is a generalized real problem; the eigenvalues of smallest magnitude are requested and the symmetric algorithm is chosen (since this is the default).

The output produced by this new program is the following:

```
Interpolation degree = 60
computing FE term intg_Omega grad(u) | grad(v), using 1 threads : done
computing FE term intg_Omega u * v, using 1 threads : done
Number of converged eigenvalues: 8
Computational mode: regular inverse mode (generalized problem)
Part of the spectrum requested: SM
Problem dimension = 61
Nb_iter / Nb_iter_Max = 443 / 800
Tolerance = 1.11022e-16
Number of Arnoldi vectors = 17
Eigenvalues :
1.6569854567517169e-10
0.9999999999869593
4.0000000000087512
9.0000000000560245
16.000000000016872
25.00000000006708
36.000000000077279
48.99999999728566
```
Once problem is solved, some particular tools may be applied to solution, for instance integral representation, export to files for graphic visualisation, ... This chapter is devoted to various post processing of solutions provided by XLiFE++.

### 8.1 Integral representation

In the context of integral equation, the solution of IE is a potential on the boundary \((\Gamma)\). This potential is not easy to interpret, so the final step of a BEM is often the reconstruction of the field outside \(\Gamma\). For instance, the Helmholtz diffraction Dirichlet problem may be solved using a single layer potential \(q = [\partial_n u]|\Gamma\) and the diffracted field outside the boundary \(\Gamma\) is given by

\[
    u(x) = \int_{\Gamma} G(x,y) q(y) \, dy.
\]

XLiFE++ adresses the general form of integral representation:

\[
    u(x) = \int_{\Gamma} opk(G(x,y)) \otimes opu(q(y)) \, dy.
\]

where \(opk\) is an operator on kernel, \(opu\) an operator on unknown and \(\otimes\) one of the operation \(*\), \(\mid\), \(\wedge\) or \(%\). The previous example corresponds to \(opk = id\), \(opu = id\) and \(\otimes = \ast\). To deal with such integral representation, the user has to define a linear form from \(\texttt{intg}\) constructor:

```plaintext
LinearForm ri=intg(Gamma, G*q); // default integration method
LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3); // specifying quadrature rule
IntgRepresentationIM irIM(Gauss_Legendre,3,Gauss_Legendre,2,0.1);
LinearForm ri=intg(Gamma, G*q, irIM); // specifying singular integration method
```

In these expressions, Gamma is a \texttt{Domain} object, G a \texttt{Kernel} object and q an \texttt{Unknown} object. Singular integration method is required if you intend to evaluate the integral representation at points close to the boundary \(\Gamma\).

The linear form may be a linear combination of \(\texttt{intg}\):

```plaintext
IntgRepresentationIM irIM(Gauss_Legendre,3,Gauss_Legendre,2,0.1);
LinearForm ri=intg(Gamma, G*q, irIM) + intg(Gamma, (grad_x(G)|_nx)*q, irIM)
```

Now, to effectively compute integral representation you have to specify the vector representing the numerical potential, a \texttt{TermVector} object (say Q) and the points where to evaluate it. There are many way to give points:
• compute at one point $x$:

\[
\text{LinearForm } \mathbf{r}_i = \int_{\Gamma} \left( \Gamma, G \ast q, \text{Gauss-Legendre}, 3 \right) ;
\]
\[
\text{Complex } \mathbf{val} ;
\]
\[
\text{Point } x(0,0,2) ;
\]
\[
\text{integralRepresentation}(x, \mathbf{r}_i, \mathbf{Q}, \mathbf{val}) ;
\]

• compute at an explicit list of points:

\[
\text{LinearForm } \mathbf{r}_i = \int_{\Gamma} \left( \Gamma, G \ast q, \text{Gauss-Legendre}, 3 \right) ;
\]
\[
\text{Vector<Points> } \mathbf{xs}(10) ;
\]
\[
\mathbf{xs}(1) = \text{Point}(0,0,2) ; \ldots
\]
\[
\text{Vector<Complex> } \mathbf{val}(10) ;
\]
\[
\text{integralRepresentation}(\mathbf{xs}, \mathbf{r}_i, \mathbf{Q}, \mathbf{val}) ;
\]

• compute at an implicit list of points of a `Domain` object (say omega):

\[
\text{LinearForm } \mathbf{r}_i = \int_{\Gamma} \left( \Gamma, G \ast q, \text{Gauss-Legendre}, 3 \right) ;
\]
\[
\text{Vector<Points> } \mathbf{xs} ;
\]
\[
\text{Vector<Complex> } \mathbf{val} ;
\]
\[
\text{integralRepresentation}(\omega, \mathbf{r}_i, \mathbf{Q}, \mathbf{val}, \mathbf{xs}) ; \quad \text{// } \mathbf{val} \text{ and } \mathbf{xs} \text{ are filled by function}
\]

• compute at an implicit list of node points of an interpolation on a `Domain`:

\[
\text{LinearForm } \mathbf{r}_i = \int_{\Gamma} \left( \Gamma, G \ast q, \text{Gauss-Legendre}, 3 \right) ;
\]
\[
\text{TermVector } \mathbf{U} = \text{integralRepresentation}(\mathbf{u}, \omega, \mathbf{r}_i, \mathbf{Q}) ; \quad \text{// } \mathbf{u} \text{ unknown on a Lagrange space}
\]

\[\begin{array}{c}
\text{In the previous syntaxes the type of output } \mathbf{val} \text{ has to be consistent with data. For instance, } \mathbf{val} \text{ is of complex type if } G \text{ or } \mathbf{Q} \text{ is of complex type. The last syntax is more robust because the type is determined by the function. Besides, this syntax returns a TermVector that may be straight exported to a file for visualization.}
\end{array}\]

When points $x$ are far from boundary $\Gamma$, an alternate method consists in computing IR by interpolation method. Let $\Omega$ the domain where IR is evaluated and $V_{\Omega}$ a Lagrange finite element space of interpolation defined on $\Omega$. Denote $(w_i)_i$ the basis functions associated to $V_{\Omega}$ space. Let $W_\Gamma$ a Lagrange finite element space defined on $\Gamma$ and $(\tau_j)_j$ the basis functions associated to it. Interpolated the kernel at nodes $x_i \in \Omega$ and $y_j \in \Gamma$, IR is approximated by

\[
u(x_i) \approx \int_{\Gamma} \sum_i \sum_j G(x_i, y_j) w_i(x) \tau_j(y) q(y) \, dy \, dx.
\]

If $q$ has the following decomposition $q(y) = \sum_k q_k \sigma_k(y)$ we have:

\[
u(x_i) \approx \sum_i \sum_j \sum_k G(x_i, y_j) w_i(x) q_k \int_{\Gamma} \tau_j(y) \sigma_k(y) \, dy \, dx
\]

that reads in vector form $(U = (u(x_i)_i), Q = (q_k)_k)$:

\[
U = G \ast M \ast Q
\]

185
with $G$ the matrix $(G(x_i, y_j))_{ij}$ and $M$ the mass matrix:

$$M_{jk} = \int_{\Gamma} \tau_j \sigma_k.$$ 

This example shows how it is done with XLtFE++:

```plaintext
Space Vq(Omega, P0); Unknown q(Vq,"q");
computation of $Q$ ...
Space Vo(Omega, P1); Unknown u(Vo,"u");
Space Vg(Gamma, P1); Unknown v(Vg,"v");
TermMatrix Gi(u, Omega, v, Gamma, "Gi"); //G(xi, yj)
TermMatrix M(intg(Gamma, q*v,"M"); compute(M);
TermVector U=Gi*(M*Q);
```

Because kernel is interpolated, the mesh of $\Omega$ does not be too coarse. $Vg$ may be chosen equal to $Vq$. This method is generally faster than previous ones because computation of the mass matrix is a fast process but interpolation method fails at points $x_j$ too close to the boundary $Gamma$.

### 8.2 Output functions

#### 8.2.1 Print objects

Most of objects may be printed to screen, to file using the out streaming operator `<<`

```plaintext
BilinearForm a=intg(omega, u*v);
TermMatrix A(a,"'A");
compute(A);
TermVector Un(omega,u,1,"'U");
TermVector X=A*Un;
std::cout<<"A*un = "<<X<<eol;  //print to screen
thePrintStream<<"A*un="<<X<<eol;  //print to file print.txt
```

If you want to print to a specific file, create an ofstream object and print to:

```plaintext
std::ofstream out("myfile.dat");
out<<"A*un="<<X<<eol;  //print to file myfile.dat
out.close();
```

When using `<<`, objects are printing with verbose. The verbose level may be adjusted using the command `setVerboseLevel(n)` where $n$ is a positive number. The verbosity is proportional to $n$.

#### 8.2.2 Export TermMatrix and TermVector

Printing to file TermMatrix and TermVector objects using the `<<` operator produces files with a lot of verbosity. So, if you want raw outputs easier to read by other software, use the `saveToFile` commands.

For TermMatrix object:

```plaintext
BilinearForm a=intg(omega, u*v);
TermMatrix A(a,"'A");
A.saveToFile("A.dat","dense");  //dense format
A.saveToFile("As.coo","_coo",true);  //coordinate format
saveToFile(As,"As.coo","_coo",true);  //works also
```
Only two formats are available:

- dense format (_dense): all the matrix coefficients are written line by line
- coordinate format (_coo): non-zero matrix coefficients are written in form \(i\ j\ a_{ij}\)

When the last argument (default=false) is set to true, structure informations are added to the file name. In the previous example, the file name looks like **As(30_30_coo_real_scalar).coo**.

For **TermVector** object:

```
TermVector U1, U2, ...
U1.saveToFile("U1.dat", _dense);  // raw format
saveToFile("U1.dat", U1, true);   // name encoded
saveToFile("U.dat", U1, U2, _raw); // two TermVectors in same file
saveToFile("U.dat", U1, U2, _vtk); // two TermVectors export in vtk format
saveToFile("U.dat", U1, U2, _vtu); // two TermVectors export in vtu format
saveToFile("U.dat", U1, U2, _msh); // two TermVectors export in msh format
saveToFile("U.dat", U1, U2, _matlab); // two TermVectors export in matlab format
```

\(vtk/\)vtu and msh format are format embedding mesh informations. They may be read by visualization software (Paraview for instance). It is the reason why exporting TermVectors in the same \(vtk/\)vtu or msh file, TermVectors must be defined on the same space!

Note that mesh can also be exported in \(vtk\) and \(msh\) format using also the **saveToFile** command.
A.1 Installation and use of BLAS and LAPACK libraries

Using UmfPack or ARPACK means using BLAS and LAPACK libraries. XLiFE++ offers the ability to choose your BLAS/LAPACK installation:

- Using BLAS/LAPACK installed with UmfPack or ARPACK
- Using default BLAS/LAPACK installed on your computer
- Using standard BLAS/LAPACK libraries, such as OpenBLAS.

To do so, you just have to use `XLIFEPP_LAPACK_LIB_DIR` and/or `XLIFEPP_BLAS_LIB_DIR` to set the directory containing LAPACK and BLAS libraries:

```
cmake [...] -DXLIFEPP_BLAS_LIB_DIR=path/to/Blas/library/directory
-DXLIFEPP_LAPACK_LIB_DIR=path/to/Lapack/library/directory
```

It is useless to use `XLIFEPP_LAPACK_LIB_DIR` and/or `XLIFEPP_BLAS_LIB_DIR` options if you do not activates configuration with UmfPack or ARPACK.

A.2 Installation and use of UmfPack library

The prerequisite to make use of UmfPack is to have it installed or at least its libraries are compiled. The `umfpackSupport` can be linked with or without UmfPack in case of none of its functions is invoked. Otherwise, any try to use its provided functions can lead to a linkage error.

Details to compile and install UmfPack can be found at `http://www.cise.ufl.edu/research/sparse/umfpack/`. All the steps will be described below supposing UmfPack already installed or compiled in the user’s system.

In order to make use of UmfPack routines, user must configure CMAKE with options: `XLIFEPP_ENABLE_UMFPACK`, `XLIFEPP_UMFPACK_INCLUDE_DIR` and `XLIFEPP_UMFPACK_LIB_DIR`.

```
cmake -DXLIFEPP_ENABLE_UMFPACK=ON
-DEXLIFEPP_UMFPACK_INCLUDE_DIR=path/to/UMFPACK/include/directory
-DEXLIFEPP_UMFPACK_LIB_DIR=path/to/UMFPACK/library/directory
```

path/to/CMakeLists.txt
Theoretically, UmfPack does not need to use BLAS/LAPACK, but as it is highly recommended by UmfPack (for accuracy reasons), XLiFE++ demand that you use BLAS/LAPACK.

UmfPack is provided by SuiteSparse. When looking how UmfPack is compiled, it seems that it can depend (maybe in the same way as BLAS/LAPACK) from other libraries provided by SuiteSparse.

In this case, you have a simpler way to define paths: by using XLIFEPP_SUITESPARSE_HOME_DIR. When given alone, it consider the given directory as the home directory containing every library provided by SuiteSparse with a specific tree structure. If it is not the case, you can use more specific options of the form XLIFEPP Xxx_INCLUDE_DIR and XLIFEPP Xxx_LIB_DIR, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, SUITESPARSECONFIG or UMFPACK.

A.3 Installation and use of ARPACK++ library

ARPACK++ distribution can be obtained from the following URL: http://www.ime.unicamp.br/~chico/arpack++. However, because of the deprecation of this version, a patch at http://reuter.mit.edu/index.php/software/arpackpatch/ needs applied to make sure a correct compilation. Users of Unix-like system can follow the instructions on https://help.ubuntu.com/community/Arpack%2B%2B to make the patch. And this patch is often not enough to compile correctly with recent compilers.

So XLiFE++ contains its own patched release of ARPACK++, used by default.

Because ARPACK++ is an interface to the original ARPACKFORTRAN library, this library must be available when installing the C++ code. Although FORTRAN files are not distributed along with ARPACK++, they can be easily downloaded from Netlib and are even available under some Unix-like systems.

The BLAS and LAPACK routines are required by the ARPACKFORTRAN, so make sure these two libraries are installed in the system. Like ARPACK, these two libraries are available under some Unix-like systems.

By default, the intern eigensolver of XLiFE++ is used for calculating eigenvalues and eigenvectors. To make use of ARPACK++, users must configure CMAKE with options: XLIFEPP_ENABLE_ARPACK and XLIFEPP_ARPACK_LIB_DIR.

The current directory is the root directory containing XLiFE++ source code. To enable ARPACK, we use the command:

```bash
cmake -DXLIFEPP_ENABLE_ARPACK=ON
-DXLIFEPP_ARPACK_LIB_DIR=path/to/arpack/libraries/directory
path/to/CMakeLists.txt
```

After configuring, we can make the library

```bash
make
```
The *utils* library collects all the general classes and functionalities required by the code: extended string capabilities, *Point*, *Vector* and *Matrix* (dense storage) objects, *Parameters* and *Function* objects to deal with user functions, *Timer* providing time computation tools and more internal useful classes intended mainly to developers (messages and traces management).

## B.1 String, Strings

*String* is a nice class allowing to deal with char of arrays without managing the memory. *String* is no more than an alias to the string class of the STL which is either standard string (utf8, by default) or wide string (utf16); this choice is made in the *config.hpp* header file by setting the macro *WIDE_STRING*. When this macro changes, all the library has to be rebuilt!

As a string or wstring class of the STL, *String* proposes all the functionalities of std::string. Mainly, you can create, concatenate string, access to char and find string in string:

```cpp
String s1("a string"); // create a String
String s2="an other string"; // create a String using =
String s12=s1+" and "+s2; // concatenate String, s3="a"+"b" does not work!
s1+=" and "+s2; // concatenate String, now s1 is the same as s12
int l=s1.size(); // number of char of s1, s1.length() gives the same
char a=s1[3]; // char a='t' (the fourth character)
s1[3]=p; // now s1="a spring and an other string";
int p=s1.find("string",0); // find first position of "string" from beginning
    (if p=−1 not found)
s1.replace(p,5,"spring"); // replace "string" by "spring"
s2=s1.substr(p,5); // extract string of length 5 from position p
s1.compare(s2); // alphanumeric comparison, return 0 if equal,
    // a negative (positive) value if s1<s2 (s1>s2)
char * c=s1.c_str(); // return pointer to the char array of the string
```

There a lot of variations of these string functions and other functions; see the STL documentation.

Some additional functions which may be useful have been introduced:

```cpp
template<typename T>
String tostring(const T& t); // 'anything' to String

template<typename T>
T stringto(const String& s); // String to 'anything'
    // returns String converted to lowercase
String lowercase(const String&);
    // returns String converted to uppercase
String uppercase(const String&);
    // returns String with initial converted to uppercase
String capitalize(const String&);
    // trims leading white space from String
```

190
String trimLeading(const String&);
// trims trailing white space from String
String trimTrailing(const String&);
// trims leading and trailing white space from String
String trimSpace(const String&);
// delete all white space from String
delSpace(const String& s);
// search capabilities
int findString(const String, const std::vector<String>&);

Be cautious with template conversion functions. The template T type has to be clarified when invoking stringto.

```cpp
// examples of conversion stringto
String s="1 2 3";
int i=stringto<int>(s); // i=1
Real r=stringto<Real>(s); // r=1.
Complex c=stringto<Complex>(s); // c=(1.,0)
void * p=stringto<void*>(s); // p=0x1
String ss=stringto<String*>(s); // ss="1"
s="(0,1)";
c=stringto<Complex>(s); // c=(0.,1.)
```

Besides, lists of strings are available using Strings:

```cpp
// list of strings
Strings ss("x=0","x=1","y=0","z=0"); //initialize list (up to 5 elements)
Strings ls(10); //10 empty strings
ls(1)="x=0"; //access to first string of list
String s=ss(3);
cout<<ss; //output list
```

Strings inherits from std::vector<String>.

### B.2 Int, Dimen, Number, Numbers

**Int** is a nice datatype allowing to deal with signed integers properly, whatever the OS (Windows, Unix/Linux, Mac OS) and the architecture (32/64 bits), so that an **Int** is 32 bits on 32 bits architectures and 64 bits on 64 bits architectures. **Number** is defined in the same way as **Int**, but for unsigned integers. **Dimen** is defined in the same way as **Int** and **Number**, but for short unsigned integers.

In fact, **Int** is no more than an alias to long int on 32 bits architectures and long long int on 64 bits architectures. **Dimen** is an alias of unsigned short int, and **Number** is an alias of size_t. As a result, **Int** and **Number** follows both the architecture.

This choice is made in the config.hpp header file automatically.

**Number** datatype is the most often used for the user, so that we defined **Numbers** to manage lists of **Number**, facilitating geometries definitions for instance:

```cpp
// list of numbers
Numbers ns(10,11,12,13); //initialize list (up to 20 elements)
ns(1)=11; //access to first number of list
Number n=ns(3);
cout<<ns; //output list
```

**Numbers** inherits from std::vector<Number>.
B.3 Real, Complex and Reals

**Real** is a nice datatype allowing to deal with floats, whatever the precision. **Real** is no more than an alias to `float`, `double` or `long double`; this choice is made in the `config.hpp` header file by setting the corresponding macro: `STD_TYPES`, `LONG_TYPES` or `LONG_LONG_TYPES`. When you change the macro, all the library has to be rebuilt!

**Complex** is no more than an alias to `std::complex<Real>`. To facilitate geometries definitions, **Reals** manages lists of **Real**:

```cpp
// list of strings
Reals rs(2.5,-3.,0.1); //initialize list (up to 10 elements)
Reals ls(10); //10 reals equal to zero
ls(1)=3.7; //access to first real of list
Real r=rs(3);
cout<<rs; //output list
```

**Reals** inherits from `std::vector<Real>.

B.4 Point

A finite element library deals obviously with points. The purpose of the **Point** class is to deal with points in any dimension and providing some algebraic operations on points and comparison capabilities. This class is used by the **Function** class encapsulating user functions.

There are mainly four ways to construct a point:

```cpp
Point p4(4.,0.); // dimension(4) and value(0) p1=(0,0,0,0)
Point p1(2.); // a 1D point p1=(2);
Point p2(1.,0.); // a 2D point p1=(1,0);
Point p3(1.,0.,1.); // a 3D point p1=(1,0,1);
Real v[]={1,2,3}; //array of real_t
Point p4(3,v); //dimension and real_t array
std::vector<Real> w(3,0); //the std::vector (0,0,0)
Point p5(w); //stl vector
```

To access to

- the dimension $n$ of a point $p$: $p$.size(),
- to the $i$-th coordinate ($1 \leq i \leq n$) of a point $p$: $p(i)$ or $p[i-1]$
- to the $x,y$ or $z$ coordinate (restricted to $n \leq 3$): $p.x()$, $p.y()$ or $p.z()$
- the vector storing the point: $p$.toVect();

You can use the coordinate accessors in reading or writing mode. A simple example:

```cpp
Point p(1.,0.,1.); //a 3D point p=(1,0,1);
p(1)=2; //modify the first coordinate
p.y()=3.; //modify the second coordinate
std::vector<Real> v=p.toVect(); //convert point to vector
```
Using standard operators (+, -, *, /), it is possible to perform algebraic computations on points up to linear combinations:

```
Point p(1.,0.,1.), q(0.,0.,1.), r(1.,2.,3.); // some 3D points
Point g=(p+q+r)/3; // compute the barycenter of p, q, r
(p+=q)/=2; // p contains the middle of p and q
```

Besides, there are some functions to compute the distance or the square of distance between two points:

```
Point p(1.,0.,1.), q(0.,0.,1.), r(1.,2.,3.); // some 3D points
real_t d=p.distance(q); // compute distance between p and q
d=pointDistance(p,q); // alternative syntax
d=squareDistance(p,q); // square of the distance between p and q
```

Finally, comparing points is possible using standard operators ==, !=, <, >, <= or >=. The comparison uses a tolerance factor \( \tau \) defined by the variable Point::tolerance (\( p, q \) being points of \( \mathbb{R}^n \)):

\[
p == q \text{ if } |p - q| \leq \tau \\
p < q \text{ if } \exists i \leq n, \forall j < i, \ |p_j - q_j| \leq \tau \text{ and } p_j < q_j - \tau.
\]

The other comparison operators !=, >, <= or >= are naturally defined from == and < operators. By default, the tolerance is set to 0. Below is an example:

```
Point p(1.,0.,1.), q(0.,0.,1.); // some 3D points
bool r=(p==q); // r=false
r=(p!=q); // r=true
r=(p<q); // r=false
Real eps=.00001;
Point::tolerance=eps; // change the tolerance factor to eps
r=(p==(p+eps/2)); // r=true
```

Geometrical transformations on points work as on geometries. Please see section 5.2 for definition and use of transformations routines.

Then, if you want to create a new Point by applying a transformation on a Point, you should use one of the following functions instead:

```
// ! apply a geometrical transformation on a Point (external)
Point transform(const Point& p, const Transformation& t);
// ! apply a translation on a Point (external)
Point translate(const Point& p, std::vector<Real> u = std::vector<Real>(3,0.));
Point translate(const Point& p, Real ux, Real uy = 0., Real uz = 0.);
// ! apply a rotation 2d on a Point (external)
Point rotate2d(const Point& p, const Point& c = Point(0.,0.), Real angle = 0.);
// ! apply a rotation 3d on a Point (external)
Point rotate3d(const Point& p, const Point& c = Point(0.,0.,0.),
               std::vector<Real> u = std::vector<Real>(3,0.),
               Real angle = 0.);
Point rotate3d(const Point& p, Real ux, Real uy, Real angle);
Point rotate3d(const Point& p, Real ux, Real uy, Real uz, Real angle);
Point rotate3d(const Point& p, const Point& c, Real ux, Real uy, Real angle);
Point rotate3d(const Point& p, const Point& c, Real ux, Real uy, Real uz, Real angle);
// ! apply a homothety on a Point (external)
Point homothetize(const Point& p, const Point& c = Point(0.,0.,0.), Real factor = 1.);
```
Point homothetize(const Point& p, Real factor);
// apply a point reflection on a Point (external)
Point pointReflect(const Point& p, const Point& c = Point(0.,0.,0.));
// apply a reflection2d on a Point (external)
Point reflect2d(const Point& p, const Point& c = Point(0.,0.),
    std::vector<Real> u = std::vector<Real>(2,0.));
// apply a reflection3d on a Point (external)
Point reflect3d(const Point& p, const Point& c = Point(0.,0.,0.),
    Real ux, Real uy = 0.);

For instance:
Point p1;
Point p2 = translate(p1, 0., 0., 1.);

B.5 Vector

The purpose of the Vector class is mainly to deal with complex or real vector. In particular, this class is used in the definition of the user functions (see the section Function). It is a templated class mainly used as a real or complex vector:

Vector<Real> u;   // u=[0.]
Vector<Real> v(3); // v=[0. 0. 0.]
Vector<Real> w(3,2.5); // w=[2.5 2.5 2.5]
Vector<Complex> cu; // cu=[(0.,0.)]
Vector<Complex> cv(3); // cv=[(0.,0.) (0.,0.) (0.,0.)]
Complex i(0,1); // the complex i
Vector<Complex> cw(3,i); // cw=[(0.,1.) (0.,1.) (0.,1.)]

It is also possible to deal with vector of vectors, for instance:

Vector<Real> ones(3,1); // ones=[1. 1. 1.]
Vector<Vector<Real>> U(4,ones);
    // U=[[1. 1. 1.] [1. 1. 1.] [1. 1. 1.] [1. 1. 1.]]

To access to a vector component (both read and write access) use the operator () with index from 1 to the vector length:

Vector<Real> v(3);    // v=[0. 0. 0.]
v(1)=1.;v(2)=2.;v(3)=3.;   // v=[1. 2. 3.]
Vector<Complex> cv(3);  // cv=[(0.,0.) (0.,0.) (0.,0.)]
cv(2)=Complex(1,1);    // cv=[(0.,1.) (1.,1.) (0.,0.)]

Note that access using operator [] with index from 0 to the vector length -1, is also possible. Advanced users can use member functions begin and end returning respectively iterators (or const iterators) to the beginning and the end of the vector.

It is also possible to extract some vector components in a new vector or to set some vector components by specifying a set of indices either given by lower and upper indices or given by a vector of indices:
Vector<Real> v(5); // v = [0. 0. 0. 0. 0.]
for (Number i=1; i<=5; i++)
v(i)=i; // v = [1. 5. 9. 16. 25.]

Vector<Real> w = v(3, 5); // w = [9. 16. 25.]

Vector<Number> is(3);
is(1)=i; is(2)=3; is(3)=5; // is = [1 3 5]
w = v(is); // w = [1. 9. 25.]
v.set(is, w); // v = [1. 9. 25. 16. 25.]

Standard algebraic operations (+, -, *, /, +, -, *, /) are supported by the Vector. Some shortcuts are also possible, for instance a vector plus a scalar, a scalar plus a vector, ... Here are a few examples:

Vector<Real> u(3, 1); // u = [1. 1. 1.]
Vector<Real> v(3); // v = [0. 0. 0.]
v = 2.; // v = [2. 2. 2.]

For algebraic operations involving two vectors, the compatibility of the size of vectors is checked. All the algebraic operations involving a real vector (resp. a complex vector) and a real scalar (resp. a complex scalar) are supported. Be cautious, as an integer value is not always certainly cast to a real value, some operations may be failed during the compiling process. For instance, the addition between a real vector and an integer does not work, cast explicitly to a real!

Vector<Real> u(3, 1); // u = [1. 1. 1.]
Vector<Real> v(3); // v = [0. 0. 0.]
v = u + 2; // DOES NOT WORK
v = u + 2.; // v = [3. 3. 3.]

Automatic cast from real vector to complex vector is supported. For instance, the following instructions are legal:

Complex i(0, 1); // complex number i
Vector<Complex> cv(3, i); // cv = [(0., 1.) (0., 1.) (0., 1.)]
cv = cv * i; // cv = [(-1., 0.) (-1., 0.) (-1., 0.)]
cv /= 2.; // cv = [(-0.5, 0.) (0.5, 0.) (0.5, 0.)]

Be cautious, automatic cast is not supported for vector of vectors.

The class also provides some various functionalities:

Vector<Real> u(3, 1); // u = [1. 1. 1.]
u.normInfinity(); // the sup norm of u
u.norm2squared(); // squared quadratic norm of u
u.norm2(); // quadratic norm of u
cout<<u;           // output the vector u: [1 1 1]
Vector<Complex> cv(3,i); // cv=[(0.,1.) (0.,1.) (0.,1.)]
conj(u);           // conjugate of cv
cv=cmplx(u);       // transform a real vector in a complex one
u=real(cv);        // take the real parts
u=imag(cv);        // take the imaginary parts

Contrary to the Point class, the Vector class offers no comparison function. Note also that there is no link between these two classes except that a Point may be automatically constructed from a Vector:

Vector<Real> u(3,1);
Point P=u;

To avoid explicit templates in user program, the following aliases are provided:

- Reals or RealVector stands for Vector<Real>,
- Complexes or ComplexVector stands for Vector<Complex>,
- RealVectors stands for Vector<Vector<Real>>,  
- ComplexVectors stands for Vector<Vector<Complex>>.

## B.6 Matrix

The purpose of the Matrix class is mainly to deal with complex or real dense matrices. In particular, this class is used in the definition of the user functions (see the section Function). This class is compliant with the Vector class. Although, it can deal with matrices of anything, it is only fully functional for real or complex matrices:

```
Matrix<Real> rA;           // an empty matrix
Matrix<Real> rB(3,2);      // a 3×2 zeros matrix
Matrix<Real> rC(3,2,1);    // a 3×2 ones matrix
Vector<Real> w(3,2.5);     // w=[2.5 2.5 2.5]
Complex i(0,1);           // the complex i
Matrix<Real> cA(3,2,i);    // a 3×2 i matrix
```

It is possible to construct diagonal matrix from a Vector or a matrix from a Vector of Vector, to load (and save) a matrix from a file and to construct particular matrices (_zeroMatrix, _onesMatrix, _idMatrix, _hilbertMatrix):

```
Vector<Real> u(3,2.);      // vector [2. 2. 2.]
Matrix<Real> rA(u);        // 3×3 matrix with u as diagonal
Matrix<Real> rB("mat.dat");   // matrix loaded from "mat.dat" file
Matrix<Real> rO(_zeroMatrix); // a 3×3 zeros matrix
Matrix<Real> rI(_onesMatrix);  // a 3×3 ones matrix
Matrix<Real> rI(_idMatrix);   // a 3×3 identity matrix
Matrix<Real> rH(_hilbertMatrix); // the 3×3 Hilbert matrix
```

Construction of complex matrix from real data are allowed (automatic cast). But the contrary is not.

There are some functions to access to the matrix properties:

- numberOfRows()
- numberOfColumns()
- isSymmetric()
- isSkewSymmetric()
- isSelfAdjoint()
- isSkewAdjoint()
ans some utilities to access to a coefficient, a row or a column or the diagonal of the matrix:

```cpp
Matrix<Real> A(2,2,1);  // a 2x2 ones matrix
A(1,1) = 2.;  // change the coefficient A11
Vector<Real> r = A.row(1);  // first row of A
r = A.column(2);  // second column of A
r = A.diag();  // diagonal of A
A.column(1,r);  // assign a vector to the first column
A.row(2,r);  // assign a vector to the second row
A.diag(r);  // assign a vector to the diagonal
```

All these functions support automatic cast from real to complex but not the contrary. Advanced users can use member functions `begin()` and `end()` returning respectively iterators (or const iterators) to the beginning and the end of the Matrix. The data values of the matrix are stored according to the C convention, i.e. row-wise.

There are also generalized access tools either to extract submatrix (`get()` or `operator()`) or to set submatrix of matrix (`set()`):

```cpp
Matrix<Real> M(3,3);
for (Number i =1; i<=3; i++)
   for (Number j =1; j<=3; j++)
      M(i,j)=i+j;  // M = [2 3 4; 3 4 5; 4 5 6]
Matrix<Real> N = M.get(2,3,2,3);  // N = [4 5 6]
// N = M(2,3,2,3) gives the same
Vector<Number> is (2);
is(1) =1; is(2) =3;
N = M.get(is,is);  // N = [2 4; 4 6]
// N = M(is,is) gives the same
Matrix<Real> Z(2,2,0);
// Z = [0 0; 0 0]
M.set(1,2,1,2,Z);  // M = [0 0; 0 0; 0 5; 4 5 6]
Matrix<Real> U(2,2,1);
// U = [1 1; 1 1]
M.set(is,is,Z);  // M = [1 0 1; 0 0 5; 1 5 1]
```

Other syntaxes are proposed, see the developer's documentation.

Besides, the `Matrix` class proposes some transformations either as internal functions or external functions:

```cpp
Matrix<Real> A(2,2,1),B;
Matrix<Complex> C(2,2,i),D;
A.transpose();  // self transposition of A
B=transpose(A);  // transposition of A, A not changed
C.adjoint();  // self transposition and conjugate C
D=adjoint(A);  // transpose and conjugate, C not changed
B=diag(A);  // from diagonal of A to a diagonal matrix
A=real(C);  // real part of C
B=imag(C);  // imaginary part of C
D=conj(C);  // conjugate of C
D=cmplx(A);  // forced casting from real to complex
```

Standard algebraic operations (`+=,=-,*=,=/,+,+,*,/`) are supported by the `Matrix` class. Some shortcuts are also possible, for instance a matrix plus a scalar, a scalar plus a matrix, ... Automatic cast from real to complex is supported. There is no comparison operator.
It is also possible to deal with matrix of matrices, for instance:

```
Matrix<Real> ones(2,2,1); // a 2x2 ones matrix
Matrix<Matrix<Real>> A(2,2,ones); // a 2x2 matrix of 2x2 ones matrix
```

but all operations are not supported for such matrices!

To avoid explicit templates in user program, the following aliases are provided:

- `RealMatrix` stands for `Matrix<Real>`,
- `ComplexMatrix` stands for `Matrix<Complex>`,
- `RealMatrices` stands for `Matrix<Matrix<Real>>`,
- `ComplexMatrices` stands for `Matrix<Matrix<Complex>>`.

## B.7 Parameters

In order to attached some user’s data to anything (in particular functions), two classes (`Parameter` and `Parameters`) are proposed. The `Parameter` class handles a single data of type `integer, real, complex, string, real/complex vector/matrix` or `void *` with the possibility to name the parameter. The `Parameters` class handles a list of `Parameter` objects.

### B.7.1 The `Parameter` object

It is easy to define a parameter by its constructor or the assignment operation:

```
Parameter p(value,[name]);
Parameter p=value;
```

where `value` is of type `integer, real, complex, string (or char*)`, `RealVector`, `ComplexVector`, `RealMatrix`, `ComplexMatrix` or `void *` and `name` is an optional string defining the parameter name.

Once a parameter is set, it is possible to get its name (if defined), its type, its value and print it:

```
Parameter k(1.,"frequency");
cout<"parameter "<k.name()<<" type "<k.type()<<" value="<real(k);
k.print(); // print name and value
cout<"k; // print only its value
RealMatrix H(5,_hilbertMatrix); // hilbert matrix 5x5
Parameter mat(H,"Hilbert matrix"); // H as parameter
```

The use of type `void *` allows the user to deal with data of any kind. This nice possibility is for advanced users because a `void *` variable is unsafe in C++:

```
list<String> lst; // a list of string
lst.push_back("Helmholtz"); lst.push_back("Laplace");
Parameter par(&lst,"problem list"); // void * parameter
cout<par; // print the pointer not the list
list<String> >& rlst=static_cast<list<String>&>(*pointer(par)); // be sure !!!
```
The functions to get the value are integer(), real(), cmplx(), string() and pointer(). Be cautious, the user must invoke the "get" function compatible with the parameter type. In case of misfit call, an error may occur or not if a logical cast is possible (only integer to real and real to complex).

A Parameter object can be automatically cast to its right value:

For numerical type parameters (integer, real or complex), it is possible to apply algebraic operations (+=, -=, *=, /=,+, -, *, /), comparison operations (==, !==, >, >=, <, <= ). The result is a Parameter. These operations do not yet work on vectors and matrices.

B.7.2 The Parameters: list of Parameter

The Parameter object is a brick of the more interesting class Parameters which handles a list of Parameter. With this class, the user is able to deal with lists of anything of the type of numerics (integer, real, vector, matrix) or string type or pointer type. In particular, these parameters lists can be attached to functions as Parameters object of the function (see the class Function documentation).

A Parameters object is simply defined by constructors taking one explicit data of type supported by the Parameter class or one Parameter object:

Parameters ps(value,[name]) ;
The main operations on the list are the insertion and the extraction of parameter values. To insert a parameter in the list, you can use the `push()` function or the stream operator `<<`:

```c++
Parameters ps;
ps.push(param);
ps<<value;
```

where `value` is of type integer, real, complex, string (or char*), void * or is a `Parameter` object. For instance:

```c++
Parameters params(2.,"k"); // initialize from one data
params<<Parameter(1.,"rho")<<Parameter(3.,"eps"); // insert 2 real
params<<3.1415926; // insert a real with no name get it by its index (4)
params<<Parameter(RealVector(5,1.),"v"); // insert a vector
params<<Parameter(RealxMatrix(5,_hilbertMatrix),"H"); // insert a matrix
```

To extract a parameter from the list, you have to use the direct access operator () specifying its rank (from 1) in the parameters list or its parameter name or the parameter itself:

```c++
Parameter p=ps(i); // i is an integer index
Parameter p=ps(name); // name is a string
Parameter p=ps(q); // q is a parameter
```

If a parameter has no name (case of a value insertion with no name) a default name is given (`parameterx` with x its rank in the list)! To get the value of the parameter, capabilities of the `Parameter` class may be used. It also possible to use the assignment operator =:

```c++
Real k=params.get_r("k"); // use get with name
Real rho=params("rho"); // work also
Real pi=params(4); // no name available
RealVector v=params("v"); // get vector
RealMatrix H=params("H"); // get matrix
```

where `value_type` is the type of the parameter (be cautious with type compatibility).

This class provides print facilities of a list of parameters:

```c++
Parameters params;
params.print(); // print on a default printf file
params.print(out); // out is an output stream
out<<params;
```

Finally, the class provides a void list of parameters: `Parameters::default_Parameters`.

An example:

```c++
Parameter height=3;
Parameters data;
data<<height<<Parameter(4,"width")<<"case 1"<<1.5;
// String "case 1" has default name "parameter3"
// Real "1.5" has default name "parameter4"
Parameter a=data("height"); // acces by name, contains height
Parameter c=data(4); // acces by rank, contains 1.5
data(1)= 2; // replace the value 3 by 2
data("height")=2; // same effect, height is thereafter modified
data(height)=2; // same effect
double x=data(4); // x contains 1.5
x=data("width"); // x contains 4
```
Note that there is no possibility to delete a parameter of the list and, contrary to the Parameter class, no algebraic operations may be performed on list of parameters.

B.8 Function

In order to deal with functions with parameters of any kind it is necessary to use an object function which is related to a Parameters object (a list of parameters, see Parameters documentation). This approach allows to pass friendly, at low level of the code, some user’s functions, say functions defined in the main program.

B.8.1 User function and object function

When you want to deal with the integral term:

\[
\int_{\Omega} e^{ikx} u(x, y, z) v(x, y, z) \, d\Omega,
\]

where the loop of finite element computation requires the computation of the function \( f(x) = e^{ikx} \) on quadrature points, it is necessary to pass the function \( f \) to the low-level code where the finite element loop is implemented. Most functions are function of a point or a list of points. So, only four kinds of function are concerned:

- function of a n-dimensional point (see ? for the class Point);
- function of two n-dimensional points, usually named kernel;
- function of a vector of n-dimensional point;
- function of two vectors of n-dimensional points.

The functions may also have a Parameters as input argument if necessary (default value); for example, the real \( k \) in the previous example. The output argument may be of any type (real, complex, vector, matrix, ...), but has to be compatible which the type required in computation. The way to define such a function is the following. First, define a standard C++ function, for instance:

```cpp
Complex f(const Point & P, Parameters pa = default_Parameters)
{
    Real k=pa(1);  // k is the first parameter of the parameter's list pa
    Real x=P(1);   // x is the first coordinate of the point P
    return exp(i*k*x);  // return a complex value result,
                         // return exp(i*pa(1)*P(1)) is also possible
}
```

If you have to deal with the integral term involving a real value matrix (with no parameter involved in this example):

\[
\int_{\Omega} A \nabla u \cdot \nabla v \, d\Omega,
\]

you may define:
Matrix<Real> A(const Point & P, Parameters& pa = default_Parameters)
{
    Matrix<Real> vA(3,3);
    ... return vA;
}

Note that even if your function does not involve some parameters, the second argument of type Parameters is mandatory in the function definition.

For a kernel type function, it is quite similar. You have to specify two points as input arguments:

Complex Green_Helmholtz_3D(const Point& M, const Point& P, Parameters& pa = default_Parameters)
{
    Real r=distance(M,P); // we assume a distance function exists
    Real k=pa(1);
    Real eps=pa(2);
    if(r>eps) return exp(i*k*r)/r;
    else ...
}

The vector form of a Function is a function working with a vector of points and returning a vector of results (values on each point). It may be useful when computing n values together is really faster than computing n times a single value. Such a function should be declared, for instance, as follows:

Vector<Complex> vf(const Vector<Point>& vP, Parameters& pa = default_Parameters)
{
    Real k=pa(1); // k is the first parameter of the parameter’s list pa
    uint n=vP.size();
    Vector<Complex> res(n);
    for(int j=1;j<=n;j++) res(j)=exp(i*k*vP(j)(1));
    return res;
}

Note that the function has to return a Vector object. For a function involving a couple of vectors of points, the syntax of the declaration of the function is:

Vector<Complex> vf(const Vector<Point>& vP, const Vector<Point>& vQ, Parameters& pa = default_Parameters)

The functions defined by the user may be used directly as argument of some functions of the library. But in most of cases it is necessary to define explicitly the object Function associated to the user function. This is the way to do this:

Parameters par;
par <<2.<<.000001; //k and eps values, inserted in a parameter list
Do not confuse the vector form of a `Function` and a function which returns a vector! The vector form means a function which computes a quantity (scalar, vector, matrix) on a set of a points or bipoints, the result being a vector of scalars, vectors or matrices. For most applications, scalar form of `Function` are sufficient. Vector form is an extension allowing the user to compute the function more efficiently in the case of multiple evaluations.

When the user wants to associate some parameters to his function, it is mandatory to define the object `Function` because it stores the list of parameters. To understand the role of the object `Function`, note that if P is a `Point`, the two instructions:

```
r=f(P, par);  // call directly the user function f
funcf(P, r);  // call the user function f using the object function funcf
```

are allowed and give the same result. In other words, the object `Function` shadows the `Parameters` object. In this example, using an object function seems to be artificial. The object function has a real interest when internal computational routines requires user’s functions, because it is easier to send one type of object encapsulating various type of function rather than many different objects.

When you have to pass an object `Function` to a function which requires such an object, it is possible to use the constructor syntax `Function(f,param)`, avoiding the explicit creation of the object `Function`:

```
Parameters par;
par <<=2.<<<.000001;   // value of k and eps
compute(Function(f,par));  // compute is a function requiring an object function
```

To sum up,

- to define a function of one point returning a value of type T and its associated object `Function`:

  ```
  T namefunction(const Point& P, Parameters& pa=default_Parameters)
  Function nameofobjectfunction(namefunction,[param]);
  ```

- to define a function of two points (a kernel) returning a value of type T and its associated object `Function`:

  ```
  T namefunction(const Point& P, const Point& Q, Parameters& pa=default_Parameters)
  Function nameofobjectfunction(namefunction,[param]);
  ```

- to define a vector function of one point returning a vector of value of type T and its associated object `Function`:

  ```
  Vector<T> namefunction(const Vector<Point>& P, Parameters& pa=default_Parameters)
  Function nameofobjectfunction(namefunction,[param]);
  ```
• to define a vector function of two points (a vector kernel) returning a vector of value of type 
\text{T} and its associated object Function:

\[
\text{Vector<} \text{T} \text{>} \text{namefunction(const Vector<Point> } & \text{, const Vector<Point}> & \text{, Parameters& pa=default_Parameters}) \text{;}
\]

\[
\text{Function nameofobjectfunction(namefunction ,[param]);}
\]

• to avoid the explicit construction of the object function (useful when you have to pass the 
function as an argument)

\[
\text{Function(namefunction ,[param]);}
\]

For the people who used MELINA Fortran, this approach replaces the famous \text{fctrm.f} and 
the \text{tbasso} vector machinery.

B.8.2 Advanced user

Delaying computations

It may occur that the function you plan to use is a very complex one, involving some heavy 
computations that you want compute only once by storing some intermediate results somewhere. 
In order to allow flexibility to the user, it is advised to use the capabilities of Parameters object 
to store void pointers. For instance, the first time the function is called, you can compute some 
reusable quantities and store them in any structure with dynamic memory allocation and store 
the pointer of this structure in the Parameters object. The next time the function is called, as 
you have an access to this void pointer (do not forget to recast it), you can recover your data.

Calling a Function object

If you have to compute the values of the function via the object Function, there are mainly two 
ways to do it:

• using an alias to the pointer function (requires that you know the type of function and 
arguments)

\[
\text{Point } \text{P=Point(0,0) , Q=Point(1,1);}
\]
\[
\text{Complex } c=\text{func}.\text{funSC}(P); \quad // \text{a function returning a complex scalar} \text{(funSC)}
\]
\[
\text{Complex } g=\text{funcG.kerSC}(P,Q); \quad // \text{a kernel returning a complex scalar} \text{(kerSC)}
\]
\[
\text{Matrix<} \text{Real}> m=\text{funcA.funMR}(P); \quad // \text{a function returning a real matrix scalar} \text{(funMR)}
\]

As this method uses a recasting of a void pointer with no checking, it can cause segmentation 
errors if there is a misfit between the type of function required and the real function stored in 
the void pointer! It is possible to check the type of arguments by using the utility functions 
typeReturned(), structReturned(), typeFunction() and typeArg(). This direct method is 
offered to developers in order to have the best performance.

• using the safe overloaded operator (), allowing to deal with point or vector of points

\[
\text{Point } \text{P=Point(0,0) , Q=Point(1,1);}
\]
\[
\text{Complex } c; \quad // \text{complex to store the result}
\]
\[
\text{funcf.checkTypeOn();} \quad // \text{activate checking mode}
\]
This method does not require the knowledge of the exact type of the function (the output argument must be compatible!). It allows scalar or vector form independently of the form of the user function. Note that, contrary to the first method, this method uses a reference to return the values, so you have to manage its memory allocation. When the function is called with a vector of points as input, the vector result is resized if it is too small. Using the checkTypeOn function, it is possible to activate the checking of the type of argument. After computation, the checkType variable is reset to false in order to avoid unnecessary rechecking. As the checking process invokes RTTI functions (expensive in time), activate wisely this option. So, if you have to evaluate many times the function, activate the checking only for the first evaluation. Note that when the checking process is deactivated, this method is still slightly more expensive than the first one.

### B.9 Kernel

The Function class allows to define kernel type function, say function of two points. But, to deal with integral equation, more informations are required. It is the role of the Kernel class. A Kernel object manages mainly:

- `Function kernel;` // kernel function
- `Function gradx;` // x derivative
- `Function grady;` // y derivative
- `Function gradxy;` // x,y derivative
- `Kernel* singPart;` // singular part of kernel
- `Kernel* regPart;` // regular part of kernel
- `SingularityType singularType;` // singularity (nonsingular, r, logr, loglogr)
- `Real singularOrder;` // order of singularity
- `Complex singularCoefficient;` // coefficient of singularity
- `SymType symmetry;` // kernel symmetry (noSymmetry, symmetric ...)
- `String name;` // kernel name
- `Parameters userData;` // to store some additional informations

So when defining a new one, you have to provide such informations. To understand how it works, this is the example of Helmholtz3d kernel.

First define all the functions as ordinary c++ functions:

```c++
// kernel G(k; x, y)=exp(i*k*r)/(4*pi*r)
Complex Helmholtz3d(const Point& x, const Point& y, Parameters& pa)
{
    Real k = real(pa("k"));
    Real r = x.distance(y);
    Complex ikr = Complex(0., 1.) * k * r;
    return over4pi * std::exp(ikr) / r;
}
```

```c++
Vector<Complex> Helmholtz3dGradx(const Point& x, const Point& y, Parameters& pa)
```
Define regular part functions:

```cpp
// regular part: G_reg(k; x, y)=(exp(i*k*r)-1)/(4*pi*r)
Complex Helmholtz3dReg(const Point& x, const Point& y, Parameters& pa)
{
    Complex g;
    Real k = real(pa("k"));
    Real kr = k * x.distance(y);
    Complex ikr = Complex(0., kr);
    if (std::abs(kr) < 1.e-04)
    { int n=4; // for abs(kr)<1.0e-4 this is a good choice for n (checked)
        g = 1 + ikr / n--;
        while (n > 1) {g = 1 + g * ikr / n--;}
        return g *= Complex(0., over4pi * k);
    }
    else return over4pi * k * (std::exp(ikr) - 1.) / kr;
}
```

```cpp
Vector<Complex> Helmholtz3dGradyReg(const Point& x, const Point& y, Parameters& pa)
{
    Real k = real(pa("k"));
    Real r = x.distance(y);
    Complex ikr = Complex(0., k * r);
    Complex t = over4pi * (1. + std::exp(ikr) * (ikr - 1.)) / r;
    Vector<Complex> g(3);
    scaledVectorTpl(t / r, x.begin(), x.end(), y.begin(), g.begin());
    return g;
}
```

```cpp
Vector<Complex> Helmholtz3dGrady(const Point& x, const Point& y, Parameters& pa)
{
    Real k = real(pa("k"));
    Real r2 = x.squareDistance(y);
    Real r = std::sqrt(r2);
    Complex ikr = Complex(0., 1.) * k * r;
    Complex dr = (ikr - 1.) / r2;
    Vector<Complex> g1(3);
    scaledVectorTpl(over4pi * exp(ikr) * dr / r, x.begin(), x.end(), y.begin(),
                    g1.begin());
    return g1;
}
```
Real k = real(pa("k"));
Real r = x.distance(y);
Complex ikr = Complex(0., k*r);
Complex t = - over4pi * (1. + std::exp(ikr)*(ikr - 1.))/ r;
Vector<Complex> g(3);
scaledVectorTpl(t/ r, x.begin(), x.begin(), y.begin(), g.begin());
return g;
}

Define singular part functions :

// construct Helmholtz3d Kernel singular part: G_sing(k; x, y) = 1/(4*pi*r)
Complex Helmholtz3dSing(const Point& x, const Point& y, Parameters& pa) {
    Real r = x.distance(y);
    return over4pi / r;
}
Vector<Complex> Helmholtz3dGradxSing(const Point& x, const Point& y, Parameters& pa) {
    Real r = x.distance(y);
    return - over4pi / (r*r);
    Complex t = - over4pi / (r*r);
    Vector<Complex> g(3);
    scaledVectorTpl(t, x.begin(), x.end(), y.begin(), g.begin());
    return g;
}
Vector<Complex> Helmholtz3dGradySing(const Point& x, const Point& y, Parameters& pa) {
    Real r = x.distance(y);
    Complex t = over4pi / (r*r);
    Vector<Complex> g(3);
    scaledVectorTpl(t, x.begin(), x.end(), y.begin(), g.begin());
    return g;
}

Now construct Kernel objects :

Parameters pars;
pars<<Parameter(1., "k");

Kernel H3Dreg; // regular part
H3Dreg.name="Helmholtz 3D kernel regular part";
H3Dreg.singularType = _notsingular;
H3Dreg.singularOrder = 0;
H3Dreg.singularCoefficient = over4pi;
H3Dreg.symmetry = _symmetric;
H3DreguserData = pars;
H3Dreg.kernel = Function(Helmholtz3dReg, pars);
H3Dreg.gradx = Function(Helmholtz3dGradxReg, pars);
H3Dreg.grady = Function(Helmholtz3dGradyReg, pars);

Kernel H3Dsing; // singular part
H3Dsing.name="Helmholtz 3D kernel, singular part";
If you do not define singular and regular part kernels, some computations will not be available.

In fact the Helmholtz kernel is defined in `mathsResources` library of XLiFE++. To load it, use the following code:

```cpp
Parameters pars;
pars << Parameter(1., "k");
Kernel H3D = Helmholtz3dKernel(pars);
```

If you develop a new kernel for your own use, contact the administrators. May be they will be happy to integrate your work in XLiFE++.

**B.10 Timer**

The `Timer` class is a utility class to perform computational time analysis (cpu time and elapsed time) and manage dates. For a user, only a few functions are useful. They do not involve explicitly some `Timer` objects. There are some functions to get date in various forms:

```cpp
String theTime();     // returns current time
String theDate();     // returns current date as dd.mmm.yyyy
String theShortDate(); // returns current date as mm/dd/yyyy or dd//mm/yyyy
String theLongDate();  // returns current date as Month Day, Year or Day Month Year
String theIsoDate();  // returns ISO8601 format of current date (yyyy-mn-dd)
String theIsoTime();  // returns ISO8601 format of current time (hh–mi–ss)
```
and others to get cpu or elapsed time:

```cpp
double cpuTime(); // user time ("cputime") in sec. since last call
double cpuTime(const String&); // same and prints it with comment
double totalCpuTime(); // elapsed time in sec. since first call
double totalCpuTime(const String&); // same and prints it with comment
double elapsedTime(); // elapsed time in sec. since last call
double elapsedTime(const String&); // elapsed time in sec. since last call
double totalElapsedTime(const String&); // elapsed time in sec. since first runtime call
double totalElapsedTime(const String&); // same with comment
```

Using these functions, it is easy to perform time computation analysis. For instance:

```cpp
#include "xlife++.h"
using namespace xlifepp;
int main()
{
    init(fr); // initializes timers
    // task 1
    ...
    cpuTime("cpu time for task 1");
    elapsedTime("elapsed time for task 1");
    // task 2
    ...
    cpuTime("cpu time for task 2");
    elapsedTime("elapsed time for task 2");
    // end of tasks
    totalCpuTime("total cpu time");
    totalElapsedTime("total elapsed time");
}
```