User documentation

Nicolas Kielbasiewicz, Eric Lunéville

May 3, 2018

Contents

Preface

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.

1 Introduction

1.1 What XLiFE++ is

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);
- A parallel version using OpenMP.

1.2 How to download XLiFE++

XLiFE++ is downloadable at the following url . You can download releases and snapshots of either the source code or binaries. Snapshots are supposed to be generated automatically every day when necessary.

There are 2 kinds of archives (snapshots or releases):

1. a "source" archive that contains all XLiFE++ source files and tex/pdf documentation;
2. a "api" archive that contains only source documentation generated by Doxygen

1.2.1 How XLiFE++ sources are organized ?

XLiFE++ sources are organized with several directories, described as follows for the main ones:

bin contains the xlifepp_project_setup.exe for Windows and the user scripts xlifepp.sh and xlifepp.bat. This will be explained later.

doc contains the present user guide, the developer guide (also in pdf) and other specific documentations extracted from the present user guide, such as a tutorial, an install documentation, and explanations about examples.

etc contains a lot of stuff such as templates for installation, the multilingual files, ...

examples contains example files ready to compile and use.

ext contains source files for external dependencies, such as Arpack++, Eigen, Amos libraries

src contains all C++ sources of the XLiFE++ library

tests contains all unitary and system tests to check your installation

lib will contain the static libraries of XLiFE++, after the compilation step.

usr contains the user files to write and compile a C++ program using XLiFE++

You also have a very important file `CMakeLists.txt`, that is the CMake compilation script.

1.2.2 How XLiFE++ binaries are organized ?

XLiFE++ binaries are organized with several directories, described as follows for the main ones:

bin contains the xlifepp_project_setup.exe for Windows and the user scripts xlifepp.sh and xlifepp.bat. This will be explained later.

etc it contains a lot of stuff such as templates for installation, the multilingual files, ...

share/doc it contains the present user guide, the developer guide (also in pdf) and other specific documentations extracted from the present user guide, such as a tutorial, an install documentation, and explanations about examples.

share/examples it contains example files ready to compile and use.

ext it contains source files for external dependencies, such as Arpack++, Eigen, Amos libraries

tests it contains all unitary and system tests to check your installation

lib After the compilation, it will contain the static libraries of XLiFE++.

You also have a very important file `CMakeLists.txt`, that is the CMake compilation script.

1.3 Requirements

1.3.1 Extensions

To use XLiFE++ full capabilities, you may need some external libraries to activate extensions:

- The main mesh engine needs Gmsh (http://gmsh.info). It is not a strong dependency insofar as you just have to tell XLiFE++ where Gmsh binary is.
- To use them as solvers, you may install Arpack (http://www.caam.rice.edu/software/ARPACK/) and/or UmfPack (http://faculty.cse.tamu.edu/davis/suitesparse.html). On Unix systems, you may rely on your package manager to install them. On Windows, we highly recommend you to download files on the XLiFE++ website http://uma.ensta-paristech.fr/soft/XLiFE++/?module=main&action=dl
- To visualize solutions of your programs using XLiFE++, you may install Gmsh (http://geuz.org/gmsh), Paraview (http://www.paraview.org), Matlab or Octave (https://sourceforge.net/projects/octave/files/).

XLiFE++ includes 2 external libraries: Eigen (http://eigen.tuxfamily.org/, essentially for SVD, and Amos (http://www.netlib.org/amos/) for Bessel/Hankel functions on complex arguments.

1.3.2 Installation requirements

Basically, XLiFE++ compilation depends on the cross-platform builder CMake, available at http://cmake.org. To know how to install and use XLiFE++ this way, please read section 1.4.

For Unix systems, you can use an alternative installation procedure that does not require CMake. To know how to install and use XLiFE++ this way, please read section 1.5.

Another way to install XLiFE++ is to download a Docker container (like a virtual machine containing everything to build and run XLiFE++). It is cross-platform. To know how to install and use XLiFE++ this way, please read section 1.6.

1.4 Installation and usage with CMake

You download XLiFE++ from its website http://uma.ensta-paristech.fr/soft/XLiFE++/.

- Either you download XLiFE++ sources: you have to unzip the archive at any place you choose in the filesystem. Then, you follow configuration procedure by setting at least a C++ compiler, the path to Gmsh and eventually paths to external libraries Blas, Lapack, Arpack and UmfPack. When done, you will have to compile XLiFE++ source code.
- Or you download XLiFE++ binaries: you will have to run the installer if you are on Windows (see subsection 1.4.4), or follow the configuration procedure with cmake by setting a C++ compiler, paths to Gmsh, and to Blas, Lapack, Arpack, UmfPack libraries. In following sections, you will be guided on which options you may use or not as far as sources or binaries are concerned.

  If you are on Mac OS and want to use `clang++` as a compiler, please download dedicated binaries, as they are generated without OpenMP activated.

1.4.1 How to use CMake ?

CMake only needs a configuration file named `CMakeLists.txt`, at the root directory of the XLiFE++ 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 anywhere. It will contains compilation files (objects files, ...), a `Makefile` or an IDE project file named XLiFE++ (for Eclipse, CodeBlocks, Visual Studio, Xcode, ...). So we suggest you to set this directory as a subdirectory of XLiFE++ install directory, with the name *build* for instance.
CMake can be called and used different ways:

On the command line: On Linux and Mac OS, you can use the `cmake` command or its default GUI `ccmake`. On Windows, you can use the `cmake.exe` command.

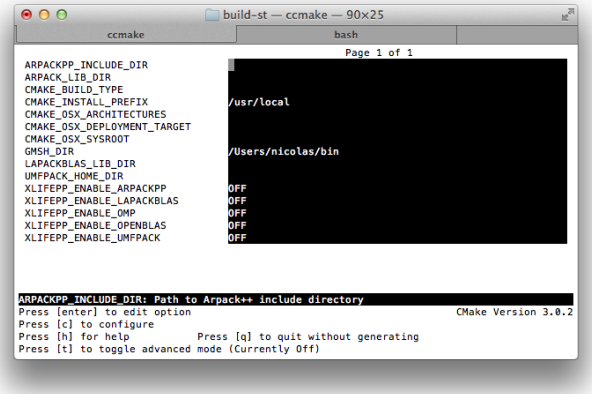


Figure 1.1: `ccmake` (MacOS, Linux)

When running CMake, 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 [options]
ccmake [options] path/to/CMakeLists.txt
```

Through GUI applications: 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 and the compiler you want. Then, you click the *Generate* button, to generate your IDE project file or your `Makefile`. It may also be useful to check *Grouped* and *Advanced* checkboxes.

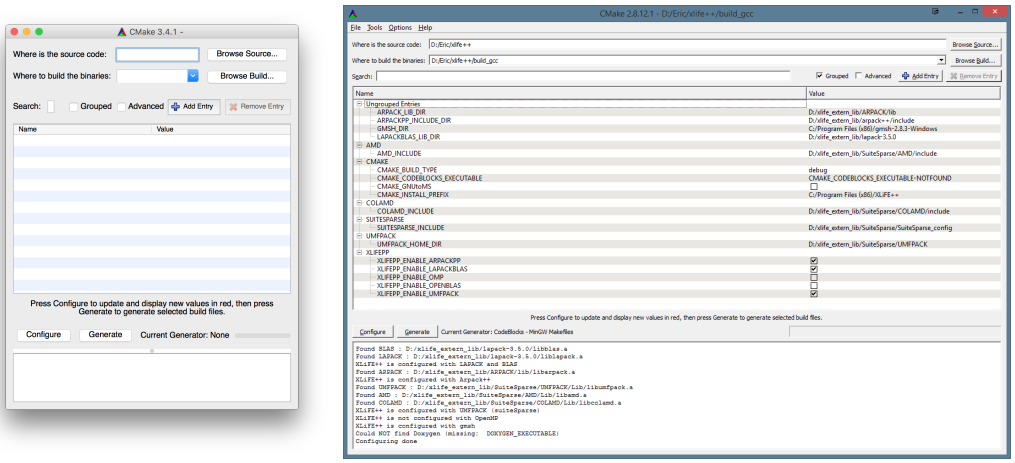


Figure 1.2: CMake application (MacOS on the left and WIndows on the right)

Let's now discuss about configuration options you may have to give to CMake. The first one is the generator. By default, the `cmake` command generates a `Makefile`. This is the "Unix Makefiles" generator on Linux and Mac OS or "MinGW Makefiles" generator on Windows. But you can use other generators to have IDE files for your favorite IDE, such as Eclipse, CodeBlocks, Xcode, Visual Studio, ...:

```
cmake path/to/CMakeLists.txt -G <generator_name> [options]
```

The following command chooses for instance to use codeblocks on unix platform:

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

Please read the `cmake` command help to known the potential list of available generators on your computer.

 Whatever the generator, to compile XLiFE++ sources, you will have to build the target *libs* to compile the libraries, and the target *tests* to compile tests.

In the following sections, we will discuss about the other options. Each of them will be of the form KEY=value and are used through the -D option with the following syntax:

```
cmake path/to/CMakeLists.txt [-G <generator_name>] -DKEY1=value1
    -DKEY2=value2 -DKEY3=value3 ...
```

 Please notice that the key is always sticked to the -D option, and that the equal sign is sticked to both key and value

1.4.2 Configuration step for XLiFE++ sources

General options

 In the following, we will consider CMake used in command line mode, from a build directory directly inside sources so that the path to `CMakeLists.txt` file is `..`

By default, external dependencies to Eigen, Amos, and OpenMP are activated. So without additional options, XLiFE++ will be configured with the first C++ compiler found in your PATH, in Release mode, and without Arpack and UmfPack.

```
cmake .. [-G <generator_name>]
```

If you want to change the compiler to use and/or the build type, you can use the following options: **CMAKE_CXX_COMPILER**, **CMAKE_Fortran_COMPILER** and **CMAKE_BUILD_TYPE**.

```
cmake .. -DCMAKE_CXX_COMPILER=g++-7 -DCMAKE_Fortran_COMPILER=gfortran-7
    -DCMAKE_BUILD_TYPE=Debug
```

When you look at the CMake log, you are supposed to read that the rightful compiler is used with the rightful build type, and that activated dependencies are found and used and that deactivated dependencies are not used.

 The fortran compiler is necessary to compile Amos library

 When CMake is run, it stores values of options and a lot of internal variables in a cache file `CMakeCache.txt` in the build directory. As a result, when you run CMake, you are not forced to give already given options. This is the reason why in the following examples, only options that are currently discussed will be used.

Basic options related to XLiFE++ dependencies

First, you can look at Gmsh and Paraview detection. If executables are reachable through the PATH, they are automatically found. If not, you can use **XLIFEPP_GMSH_EXECUTABLE** and **XLIFEPP_PARAVIEW_EXECUTABLE**.

 On Mac OS, you have to give the full path to Gmsh/Paraview executables and not applications. Executables are inside applications. If application names are Gmsh.app and paraview.app and are located in the *Applications* directory, Gmsh and Paraview will be correctly detected.

```
cmake .. -DXLIFEPP_PARAVIEW_EXECUTABLE=
    /Applications/Paraview-5.4.0.app/Contents/MacOS/paraview
```

To activate dependencies, you can use the following options:

XLIFEPP_ENABLE_ARPACK To enable/disable use of Arpack. Possible values are ON or OFF. Default is OFF.

XLIFEPP_ENABLE_UMFPACK To enable/disable use of UmfPack. Possible values are ON or OFF. Default is OFF.

XLIFEPP_ENABLE_AMOS To enable/disable use of Amos. Possible values are ON or OFF. Default is ON.

XLIFEPP_ENABLE_OMP To enable/disable use of OpenMP. Possible values are ON or OFF. Default is ON.

XLIFEPP_ENABLE_EIGEN To enable/disable use of Eigen. Possible values are ON or OFF. Default is the same as **XLIFEPP_ENABLE_OMP**, as Eigen needs OpenMP.

The default configuration being given, you may only use **XLIFEPP_ENABLE_ARPACK** and **XLIFEPP_ENABLE_UMFPACK**.


To activate/deactivate all external dependencies, you can use **XLIFEPP_DEPS** whose possible values are ENABLE_ALL, DISABLE_ALL or DEFAULT.

```
cmake .. -DXLIFEPP_ENABLE_ARPACK=ON -DXLIFEPP_ENABLE_UMFPACK=ON
cmake .. -DXLIFEPP_DEPS=ENABLE_ALL
```

If libraries are installed in standard directories, reachable from paths environment variables (it is often the case), they will be found. If not, additional options are available and explained in the following section.

Intermediate options related to XLiFE++ dependencies

You can use specific option to give to CMake additional search paths for external dependencies:

XLIFEPP_BLAS_LIB_DIR to specify an additional search directory CMake will use to find Blas library

XLIFEPP_LAPACK_LIB_DIR to specify an additional search directory CMake will use to find Lapack library

XLIFEPP_ARPACK_LIB_DIR to specify an additional search directory CMake will use to find Arpack library

XLIFEPP_UMFPACK_INCLUDE_DIR to specify an additional search directory CMake will use to find UmfPack header

XLIFEPP_UMFPACK_LIB_DIR to specify an additional search directory CMake will use to find UmfPack header

XLIFEPP_SUITESPARSE_HOME_DIR to specify an additional search directory CMake will use to find 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.

```
cmake .. -DXLIFEPP_DEPS=ENABLE_ALL -DXLIFEPP_BLAS_LIB_DIR=/usr/lib/
  -DXLIFEPP_LAPACK_LIB_DIR=/usr/lib/ ...
```

If external libraries have standard names, namely their filenames are like libarpack.a, libarpack.so, libarpack.dylib, libarpack.dll, arpack.lib, ..., they will be found. If their name contains a release number, you can ask your sysadmin to add symbolic links of each of the libraries (as it should be) and re-run CMake or you can use dedicated options to dodge the problem.

Advanced options related to XLiFE++ dependencies

You can use specific option to give directly external libraries:

XLIFEPP_BLAS_LIB to specify Blas library with full path

XLIFEPP_LAPACK_LIB to specify Lapack library with full path

XLIFEPP_ARPACK_LIB to specify ARPACK library with full path

XLIFEPP_XXX_INCLUDE_DIR to specify the XXX header, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, METIS, SUITESPARSECONFIG or UMFPACK.

XLIFEPP_XXX_LIB_DIR to specify the XXX library, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, METIS, SUITESPARSE (only on MAC OS), SUITESPARSECONFIG or UMFPACK.

XLIFEPP_FORTRAN_LIB_DIR to specify the directory where the gfortran library is. It is necessary for compilers that are not able to find it by itself, such as `clang++`

XLIFEPP_FORTRAN_LIB to specify the gfortran library with full path. It is necessary if the gfortran library has a non standard name.

1.4.3 Configuration step for XLIFE++ binaries

If you are under WINDOWS, you can go directly to subsection 1.4.4 to learn how to configure XLIFE++ with your installer.

General options

 In the following, we will consider CMAKE used in command line mode, from a build directory directly inside binary distribution so that the path to `CMakeLists.txt` file is `..`

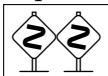
By default, all external dependencies to ARPACK, UMFPACK, EIGEN, AMOS, and OPENMP are activated. So without additional options, XLIFE++ will be configured with the first C++ compiler found in your PATH, in Release mode.

```
cmake .. [-G <generator_name>]
```

If you want to change the compiler to use, you can use the following option: **CMAKE_CXX_COMPILER**.

```
cmake .. -DCMAKE_CXX_COMPILER=g++-7
```

When you look at the CMAKE log, you are supposed to read that the rightful compiler is used with the rightful build type, and that activated dependencies are found and used and that deactivated dependencies are not used.

 When CMAKE is run, it stores values of options and a lot of internal variables in a cache file `CMakeCache.txt` in the build directory. As a result, when you run CMAKE, you are not forced to give already given options. This is the reason why in the following examples, only options that are currently discussed will be used.

Basic options related to location of XLIFE++ dependencies

Now, you can look at GMSH and PARAVIEW detection. If executables are reachable through the PATH, they are automatically found. If not, you can use **XLIFEPP_GMSH_EXECUTABLE** and **XLIFEPP_PARAVIEW_EXECUTABLE**.


On Mac OS, you have to give the full path to Gmsh/Paraview executables and not applications. Executables are inside applications. If application names are Gmsh.app and paraview.app and are located in the *Applications* directory, Gmsh and Paraview will be correctly detected.

```
cmake .. -DXLIFEPP_PARAVIEW_EXECUTABLE=
    /Applications/Paraview-5.4.0.app/Contents/MacOS/paraview
```

On Windows, external libraries are provided with the binary distribution. On Linux and Mac OS, if libraries are installed in standard directories, reachable from paths environment variables (it is often the case), they will be found. If not, additional options are available and explained in the following section.

Intermediate options related to XLiFE++ dependencies on Linux and Mac OS

You can use specific option to give to CMake additional search paths for external dependencies:

XLIFEPP_BLAS_LIB_DIR to specify an additional search directory CMake will use to find Blas library

XLIFEPP_LAPACK_LIB_DIR to specify an additional search directory CMake will use to find Lapack library

XLIFEPP_ARPACK_LIB_DIR to specify an additional search directory CMake will use to find Arpack library

XLIFEPP_UMFPACK_INCLUDE_DIR to specify an additional search directory CMake will use to find UmfPack header

XLIFEPP_UMFPACK_LIB_DIR to specify an additional search directory CMake will use to find UmfPack header

XLIFEPP_SUITESPARSE_HOME_DIR to specify an additional search directory CMake will use to find 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.

```
cmake .. -DXLIFEPP_DEPS=ENABLE_ALL -DXLIFEPP_BLAS_LIB_DIR=/usr/lib/
    -DXLIFEPP_LAPACK_LIB_DIR=/usr/lib/ ...
```

If external libraries have standard names, namely their filenames are like libarpack.a, libarpack.so, libarpack.dylib, libarpack.dll, arpack.lib, ..., they will be found. If their name contains a release number, you can ask your sysadmin to add symbolic links of each of the libraries (as it should be) and re-run CMake or you can use dedicated options to dodge the problem.

Advanced options related to XLiFE++ dependencies on Linux and Mac OS

You can use specific option to give directly external libraries:

XLIFEPP_BLAS_LIB to specify Blas library with full path

XLIFEPP_LAPACK_LIB to specify LAPACK library with full path

XLIFEPP_ARPACK_LIB to specify ARPACK library with full path

XLIFEPP_XXX_INCLUDE_DIR to specify the XXX header, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, METIS, SUITESPARSECONFIG or UMFPACK.

XLIFEPP_XXX_LIB_DIR to specify the XXX library, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, METIS, SUITESPARSE (only on MAC OS), SUITESPARSECONFIG or UMFPACK.

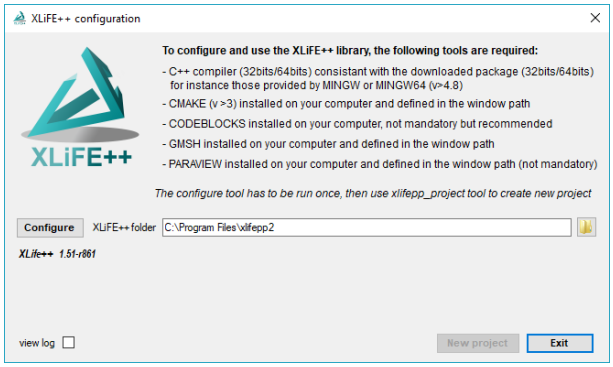
XLIFEPP_FORTRAN_LIB_DIR to specify the directory where the gfortran library is. It is necessary for compilers that are not able to find it by itself, such as `clang++`

XLIFEPP_FORTRAN_LIB to specify the gfortran library with full path. It is necessary if the gfortran library has a non standard name.

1.4.4 Installation of binaries under WINDOWS

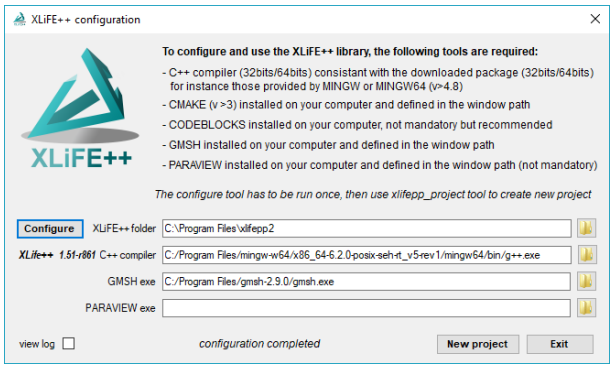
When downloading binaries under WINDOWS, you just have to run the installer. To do so, administrator elevation is required. If a previous distribution of XLIFE++ is installed in the folder you choose, the installer can remove it itself. Furthermore, it is highly recommended to install every component.
Now, in the bin subdirectory of the XLIFE++ install directory, you will find `xlifepp_configure.exe`. Tu run it, administrator elevation is required.

1. First, you have to set the folder containing XLIFE++



2. As mentioned in the banner, a C++ compiler, CMAKE and GMSH have to be installed on your computer and defined in the WINDOWS path[1]. An EDI such as CODEBLOCKS and PARAVIEW are not mandatory but highly recommended. Click on the `Configure` button

[1] A simple tool to edit the window path is PATHEDITOR2 that you can find easily on the web



3. When everything is OK, message "Configuration complete" is displayed. To compile your own program, you can click on the `New project` button or run `xlifepp_new_project.exe`. See section 1.4.5

1.4.5 Compilation of a program using XLiFE++

The manual way

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

1. You create your working directory,
2. You copy the main.cpp file into your working directory,
3. You copy the CMakeLists.txt file from the build directory (the directory in which you ran installation process) into your working directory,
4. You run CMake on the CMakelists.txt file to get your makefile or files for your IDE project (Eclipse, XCode, CodeBlocks, Visual C++, ...),
5. 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)]
                       [--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
    --generate, -g            generate the project. Used with --build option.
                              This is the default.
    --help, -help, -h         show the current help
    --interactive, -i         run xlifepp in interactive mode. Used with
                              --build option. This is the default
    --non-interactive, -noi   run 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>,  set the verbose level. Default value is 1
    -vl <value>

OPTIONS FOR BUILD IN NON INTERACTIVE MODE:
    --build-type <value>,     set cmake build type (Debug, Release, ...).
    -bt <value>
    --cxx-compiler <value>,   set the C++ compiler to use.
    -cxx <value>
    --directory <dir>,        set the directory where you want to build
    -d <dir>                  your project
    --generator-name <name>,  set the cmake generator.
    -gn <name>
    -f <filename>,            copy <filename> as a main file for the user
    --main-file <filename>    project.
    -nof,                     do not copy the sample main.cpp file. This is
    --no-main-file            the default.
    --info-dir, -id           set the directory where the info.txt file is
    --with-omp, -omp          activates OpenMP mode
    --without-omp, -nomp      deactivates OpenMP mode
```

The graphical way on Mac OS

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.

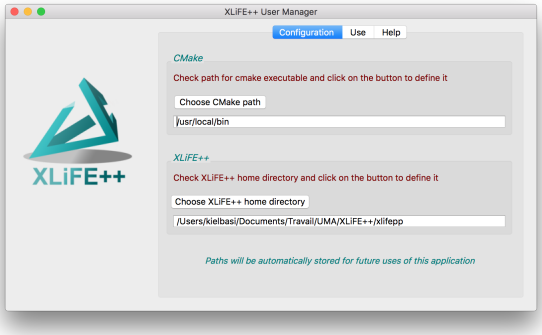


Figure 1.3: The "Configuration" tab of xlifepp-qt application

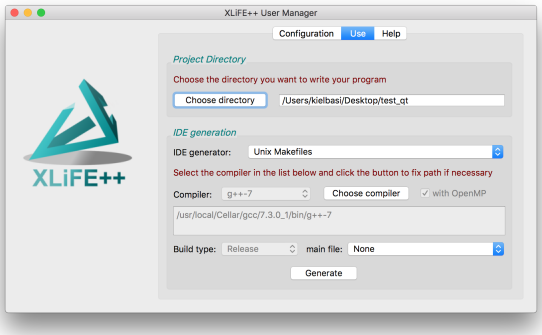
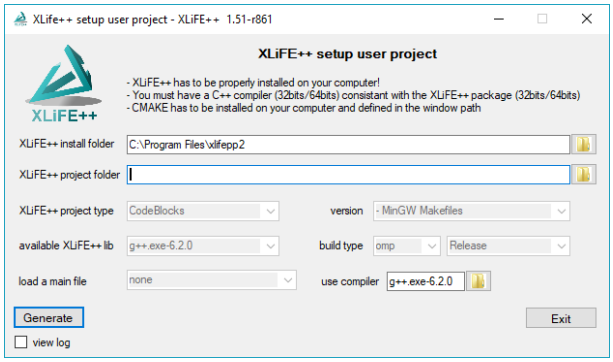


Figure 1.4: The "Use" tab of xlifepp-qt application

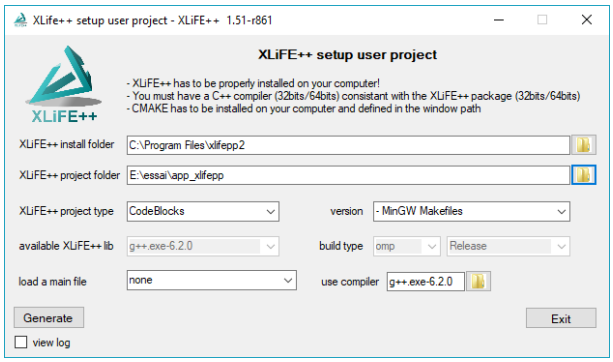
This application is a graphical user interface to the first 3 steps of the manual way.

The graphical way on Windows

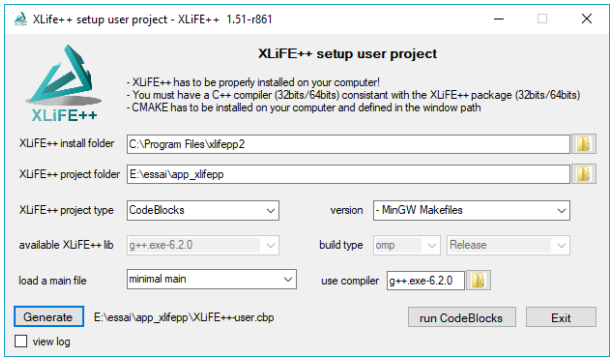
1. You run the generator `xlifepp_new_project.exe`, that is in the bin subdirectory of the XLiFE++ install directory. The XLiFE++ folder should be correct by you can fix it if necessary.



2. You select the folder in which you will write your rogram using XLiFE++. If it already exists, the generator asks you to clean it or not. This window gives some information about XLiFE++: the compiler used to generate it, if the library supports omp and the debug/release status. You should use a compatible compiler with this library. If the default C++ compiler found on your computer is not compatible, you can select another one by clicking on the `use compiler` folder button.



3. Select the type of your project. For the moment only `CodeBlocks-MinGW` and `Makefile` are working but CodeBlocks is highly recommended! Select a main file from the proposed list. This main fill will be copied in your application folder. Be care, if you choose "none", no main file will be copied and the generator will fail if there is no main file in your application folder. This option is only useful if you want to keep an existing main file in your application folder! Click on the Generate button and wait:



4. When everything is complete, you can either exit the tool or run the program that opens the generated project (CodeBlocks in the example) by clicking on the `run` button.

1.5 Installation and usage without cmake

1.5.1 Installation process

The procedure presented above requires CMake for both the installation and the usage of the libraries. Here is an alternative solution that do not use CMake, and is targeted for Unix-like systems, namely Linux and Mac OS, since it needs the execution of a shell script.
To install the libraries:

- Download the archive (release or snapshot containing the sources) from
  http://uma.ensta-paristech.fr/soft/XLiFE++/?module=main&action=dl
- Decompress the archive where the software is expected to be installed in the filesystem. This can be in the user's home or at system-wide level, in which case administrator rights will be necessary. Let's denote by `$XLDIR` the directory containing the files.

- Open a terminal and type in the command:

  `bash $XLDIR/etc/installLibs`

  This will create the libraries in the `$XLDIR/lib` directory.

XLiFE++ may use other libraries (UmfPack, Arpack, Lapack, Blas) or third party softwares (Gmsh, Paraview), depending on their presence on the computer. The script `installLibs` performs the installation in an automatic way, without any user action. This means that these libraries or softwares are really used only if they are detected or built.

The installation requires a C++ compiler. The C++ compiler to use can be imposed by the mean of the environment variable CPPCMP before calling the script `installLibs`. By default, its name is `g++`, which is the GNU compiler generally used under Linux ; under Mac OS, this will make use of the native compiler shipped with Xcode, but the GNU compiler may be used as well.
The Fortran library is needed if Arpack is used. Thus, the name of the Fortran compiler, from which is deduced the name of the Fortran library, can also be imposed by the mean of the environment variable FCMP. By default, its name is `gfortran`.

The installation process conforms to the following rules:

1. if they are not found in the filesystem, Lapack and Blas are not installed, neither any third party software,
2. UmfPack and Arpack libraries present on the system are used first and foremost,
3. if Arpack has not been found in the system and if a Fortran compiler is available, Arpack library is built locally,
4. if UmfPack has not been found in the system, SuiteSparse libraries are built locally.

Some options may be used to alter the default configuration:
`-noAmo` prevents XLiFE++ to use Amos library,
`-noArp` prevents XLiFE++ to use Arpack library,
`-noOmp` prevents XLiFE++ to use OpenMP capabilities,
`-noUmf` prevents XLiFE++ to use UmfPack (SuiteSparse) libraries.
Thus, in case of trouble, the installation script may be relaunched with one or more of these options. Using all the options leads to the standalone installation of XLiFE++, which is perfectly allowed. The complete calling sequence is then:

```
bash $XLDIR/etc/installLibs [-noAmo] [-noArp] [-noOmp] [-noUmf]
```

Finally, the details of the installation are recorded in the file `$XLDIR/installLibs.log`.

1.5.2 Compilation of a program using XLiFE++

To use XLiFE++:

1. Create a new directory to gather all the source files related to the problem to be solved.
2. In this directory, create the source files. This can be done with any text editor. One of them (only) should be a valid "XLiFE++ main file" (see section 1.7). For example, start by copying one of the files present in `$XLDIR/examples`.

3. In a terminal, change to this directory and type in the command:

   `$XLDIR/etc/xlmake`

   This will compile all the C++ source files contained in the current working directory (valid extension are standard ones `.c++`, `.cpp`, `.cc`, `.C`, `.cxx`) and create the corresponding executable file, named `xlifeppexec`.

4. Launch the execution of the program by typing in:

   `./xlifeppexec`

   The files produced during the execution are created in the current directory.

 To improve comfort, one can make a link to the script `xlmake` in the working directory, as suggested in the commentary inside the script:

```
ln -s $XLDIR/etc/xlmake .
```

or add `$XLDIR/etc` to the `PATH` environment variable. In both cases, the command typed in at step 3. above would then reduce to:

```
xlmake
```

 If OpenMP is used, it may be useful to adjust the number of threads to the problem size. Indeed, by default all threads available are used, which may be completely counter productive for example for a small problem size and a large number of threads. The number of threads to use can be modified at program level, generally in the main function, or at system level, by setting the environment variable `OMP_NUM_THREADS` before the execution is launched, e.g. with a Bourne shell:

```
export OMP_NUM_THREADS=2 ; ./xlifeppexec
```

or with a C shell:

```
setenv OMP_NUM_THREADS 2 ; ./xlifeppexec
```

1.6 Installation and usage with Docker

This procedure allows to get a pre-installed version of the libraries which are gathered in a so-called Docker container.

This first requires the installation of the Docker application, which can be downloaded from: https://www.docker.com/products/overview

Once this is done:

- Download the XLiFE++ image (use `sudo docker` on linux system):

  `docker pull pnavaro/xlifepp`

- Create a workspace directory, for example:

  `mkdir $HOME/my-xlifepp-project`

- Run the container with interactive mode and share the directory created above with the `/home/work` container directory:

  `docker run -it --rm -v $HOME/my-xlifepp-project:/home/work pnavaro/xlifepp`

This allows the files created in the internal `/home/work` directory of the container to be stored in the `$HOME/my-xlifepp-project` directory of the true filesystem, making them available after Docker is stopped.

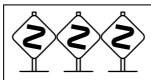
- Everything is now ready to use XLiFE++ as explained in section 1.4.5 above, for example:

  `xlifepp.sh`

  `make`

  `./exec-x86_64-linux-g++-5-Release`

  The files produced during the execution are in the directory `$HOME/my-xlifepp-project` shared with running Docker, and are then available for postprocessing.

 The Docker application requires Windows 10, or Mac OS 10.10 and higher. For older OSes, you have to download Docker ToolBox instead. See https://docs.docker.com/toolbox/toolbox_install_windows/ for Windows or https://docs.docker.com/toolbox/toolbox_install_mac/ for Mac OS.

1.7 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:

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

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

If the users have additional source files using XLiFE++ elements, they cannot include the "super" header file ***xlife++.h*** because of global variable definitions. Instead, they will include the "super" header file ***xlife++-libs.h*** that includes every XLiFE++ header except the one containing the definition of global variables.

1.8 License

directly for more information.

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

1.9 Credits

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

- Y. Lafranche (IRMAR lab), mesh tools using subdivision algorithms, wrapper to Arpack
- C. Chambeyron (POEMS lab), iterative solvers
- M.H N'Guyen (POEMS lab), eigen solvers and OpenMP implementation
- N. Salles (POEMS lab), boundary element methods
- L. Pesudo (POEMS lab), boundary element methods and HF coupling

2 Getting started

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

$$\begin{cases} -\Delta u(x,y) + u(x,y) = f(x,y) & \forall (x,y) \in \Omega \\ \dfrac{\partial u}{\partial n}(x,y) = 0 & \forall (x,y) \in \partial\Omega \end{cases} \tag{2.1}$$

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

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

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 Ω is the unit square.

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

Real cosxcosy(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); // mandatory initialization of xlifepp
  Square sq(_origin=Point(0.,0.), _length=1, _nnodes=11);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega = mesh2d.domain("Omega");
  Space Vk(omega, P1, "Vk", true);
  Unknown u(Vk, "u");
  TestFunction v(u, "v");
  BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v);
  LinearForm fv=intg(omega, cosxcosy * v);
  TermMatrix A(auv, "a(u,v)");
  TermVector B(fv, "f(v)");
  TermVector X0(u, omega, 1., "X0");
  TermVector U = cgSolve(A, B, X0, _name="U");
  saveToFile("U", U, vtu);
  return 0;
}
```

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 12 : 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* for English, *fr* for French, *de* for German, or *es* for Spanish. Default value is *en*.

_verbose 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); // mandatory initialization of xlifepp
```

lines 13-14 : 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 sq(_origin=Point(0.,0.), _length=1, _nnodes=11);
Mesh mesh2d(sq, triangle, 1, structured);
```

line 15 : The main domain, named "Omega" in the mesh, is defined.

```
Domain omega = mesh2d.domain("Omega");
```

line 16 : 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 Ω, 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).

```
Space Vk(omega, P1, "Vk", true);
```

lines 17-20 : 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 Ω, we obtain :

$$-\int_\Omega v\Delta u dxdy + \int_\Omega vudxdy = \int_\Omega vfdxdy$$

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

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v dxdy + \int_\Omega uvdxdy = \int_\Omega fvdxdy = l(v) \tag{2.3}$$

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

```
Unknown u(Vk, "u");
TestFunction v(u, "v");
BilinearForm auv = intg(omega, grad(u) | grad(v)) + intg(omega, u * v);
LinearForm fv=intg(omega, cosxcosy * v);
```

Please notice that:

- the test function is defined from the unknown. The reason is that the test function is dula 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. $cosx2$) 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`.

```
{
  Real x=P(1), y=P(2);
  return cos(pi_ * x) * cos(pi_ * y);
}
```

lines 21-22 : 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.

```
TermMatrix A(auv, "a(u,v)");
TermVector B(fv, "f(v)");
```

lines 23-24 : Matrix and vector being assembled, you can now choose the solver you want. Here, a conjugate gradient solver is used, with an initial guess constant equal to 1.

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.

```
TermVector X0(u, omega, 1., "X0");
TermVector U = cgSolve(A, B, X0, _name="U");
```

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

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

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

```
return 0;
```

3 Examples

3.1 A 1D problem

Solving 1D problems is sometimes regarded to be out of interest. Anyway, most of existing FE softwares do not handle this case. But in fact, 1D problems are of interest, often as a part of more complex problems. Thus, XLiFE++ deals with 1D problems.

3.1.1 Dirichlet condition

As a first example, we show how to solve the very simple problem, involving Dirichlet conditions:

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

Its variational formulation is

$$\left| \begin{array}{l} \text{Find } u \in V = \{v \in L^2(\Omega),\ v' \in L^2(\Omega),\ u(0) = u(1) = 0\} \text{ such that} \\ \qquad \displaystyle\int_0^1 u'(x)\, v'(x) dx = \int_0^1 f(x)\, v(x) dx \quad \forall v \in V. \end{array} \right.$$

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

```
#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=101, _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, unknows, 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");

  // solve linear system and save solution
  TermVector U=directSolve(A, F);
  saveToFile("U_1d", U, vtu);
  return 0;
}
```

The following figure shows a graphical representation of the solution using ParaView:

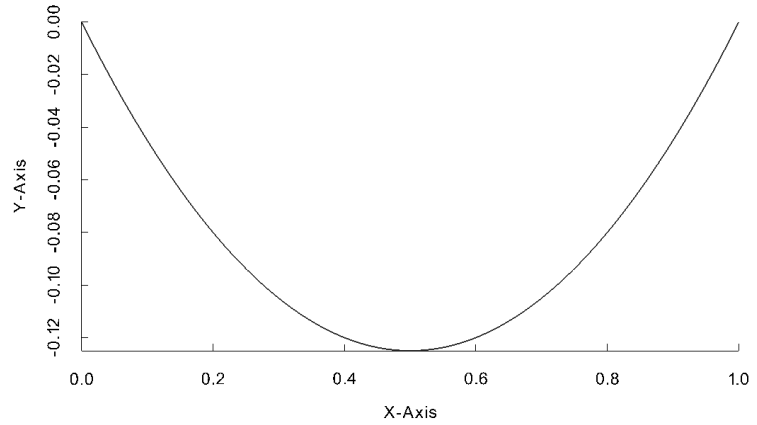


Figure 3.1: Solution of the Laplace 1D problem on the unit segment $[0, 1]$

3.1.2 Robin condition

The second example shows that XLiFE++ can also handle non homogeneous Neumann conditions or Robin-Fourier conditions. This problem also involve a Dirichlet condition. Given three real functions f_Ω, α and f_N, the problem is:

$$\begin{cases} -u'' + u = f_\Omega & \text{in } \Omega =]a, b[\\ u(a) = 0 \\ u'(b) + \alpha(b)\, u(b) = f_N(b) \end{cases}$$

Its variational formulation is:

$$\left| \begin{array}{c} \text{Find } u \in V = \{v \in L^2(\Omega),\ v' \in L^2(\Omega),\ u(a) = 0\} \text{ such that} \\ \displaystyle\int_a^b u'(x)\, v'(x)\, dx + \int_a^b u(x)\, v(x)\, dx + \alpha(b)\, u(b) = \int_a^b f(x)\, v(x)\, dx + f_N(b), \quad \forall v \in V. \end{array} \right.$$

$\alpha(b)u(b)$ can be interpreted as $\displaystyle\int_{\{b\}} \alpha(\gamma)u(\gamma)\, v(\gamma)d\gamma$ and $f_N(b)$ can be interpreted as $\displaystyle\int_{\{b\}} f_N(\gamma)\, v(\gamma)d\gamma$ where γ is the variable over the side domain here reduced to a point. This allows to handle these conditions in a uniform syntactic way by defining linear forms as shown in the previous examples.

The following main program corresponds to solving this problem with $\alpha(x) = \frac{7}{2}x^2 - 8x$ using the P10 Lagrange element over the interval $]a,b[= \left]0, \frac{13}{4}\pi\right[$ using 4 elements ; the functions $f_\Omega(x) = 2\,sin(x)$ and $f_N(x) = cos(x) + \alpha(x)\,sin(x)$ are chosen so that the solution is $sin(x)$:

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

/*
   Test problem:
     -u" + u = fOm    on the domain Om = [a,b]
        u(a) = 0
        u'(b) + alpha(b) u(b) = fN(b)
*/

Real fctEx (const Point& P, Parameters& pa = defaultParameters)
{ return sin(P[0]); }

Real fctOm (const Point& P, Parameters& pa = defaultParameters)
{ return 2 * sin(P[0]); }

Real alpha (const Point& P, Parameters& pa = defaultParameters)
{ return 3.5*P[0]*P[0] - 8*P[0]; }

Real fctfN (const Point& P, Parameters& pa = defaultParameters)
{ return cos(P[0]) + (3.5*P[0]*P[0] - 8*P[0]) * sin(P[0]); }

int main() {
  init(); // mandatory initialization of xlifepp

  // Mesh and domains
  Strings sidenames("x=a", "x=b");
  Segment seg(_xmin=0., _xmax=3.25*pi_, _nnodes=5, _domain_name="Omega",
      _side_names=sidenames);
  Mesh mesh1d(seg, 1, _structured);
  mesh1d.printInfo();
  Domain Omega = mesh1d.domain("Omega");
  Domain xA = mesh1d.domain("x=a");
  Domain xB = mesh1d.domain("x=b");

  // Space and unknowns
  Interpolation inter(_Lagrange, _standard, 10, _H1);
  Space Vh(Omega, inter, "Vh");
  Unknown u(Vh, "u");
  TestFunction v(u, "v");

  // Bilinear forms
  BilinearForm gugv = intg(Omega, grad(u)|grad(v)), uv = intg(Omega, u*v);
  BilinearForm aluv = intg(xB, alpha*u*v);
  LinearForm fOm = intg(Omega, fctOm*v), fN = intg(xB, fctfN*v);

  // Terms with essential conditions
  EssentialConditions ecs = (u|xA = 0);
  TermMatrix A(gugv + uv + aluv, ecs, "A");
  TermVector F(fOm + fN, "F");

  // Solve linear system and save solution
```

```
  TermVector U = directSolve (A, F);
  saveToFile("U", U, _matlab);

  // Compare with exact solution
  TermVector Uex(u, Omega, fctEx,"Uex");
  std::cout << "||U-Uex|| inf = " << norminfty(U-Uex) << std::endl;
  return 0;
}
```

The following figure shows a graphical representation of the solution using OCTAVE:

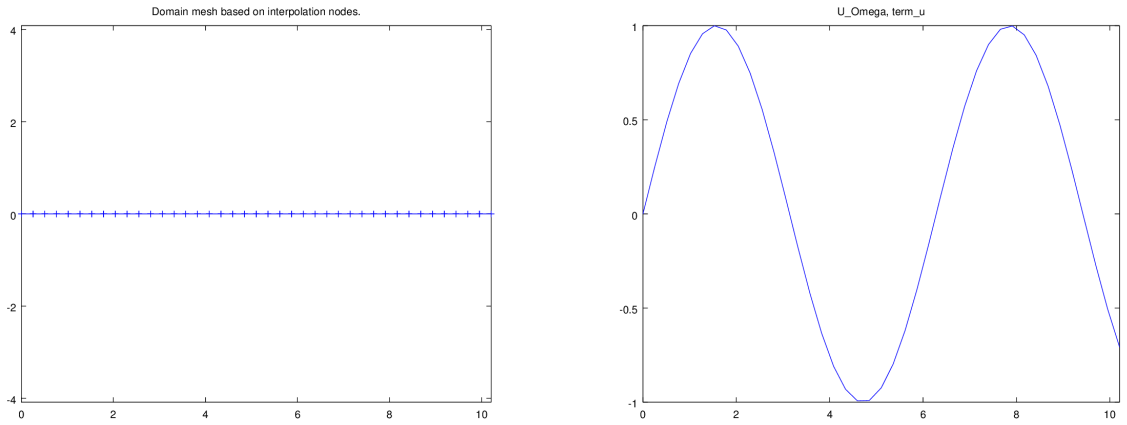


Figure 3.2: Solution of the Laplace 1D problem with Dirichelt and Robin conditions

The left figure shows the interpolation nodes which form a uniform distribution of points. This is the default behavior and the two lines

```
  Interpolation inter(_Lagrange, _standard, 10, _H1);
  Space Vh(Omega, inter, "Vh");
```

are equivalent to

```
  Space Vh(Omega, P10, "Vh");
```

Comparing the exact solution U_{ex} with the computed one U, at the interpolation abscissae, leads to $||U - U_{ex}||_\infty = 4.44278 \times 10^{-10}$, value which is currently printed by the program. By changing the keyword `_standard` for `_GaussLobatto`, one can toggle to the Gauss-Lobatto abscissae which are more suitable with higher interpolation degrees. With this example, choosing these abscissae leads to a better approximation: we then get $||U - U_{ex}||_\infty = 2.55367 \times 10^{-11}$.

3.2 Laplace Problems

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

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

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 Ω:

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

The variational formulation we deal with is

$$\left| \begin{array}{l} \text{find } u \in V = \{v \in L^2(\Omega),\ \nabla v \in (L^2(\Omega))^2\} \text{ such that} \\ \qquad \displaystyle\int_\Omega \nabla u.\nabla v + a \int_\Omega u\, v = \int_\Omega f\, v \quad \forall v \in V. \end{array} \right.$$

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):

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

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=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
  FEInterpolation 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;
  ldltFactorize(A, LD);
  TermVector U = factSolve(LD, B);

  saveToFile("U_LN", U, vtu);
  return 0;
}
```

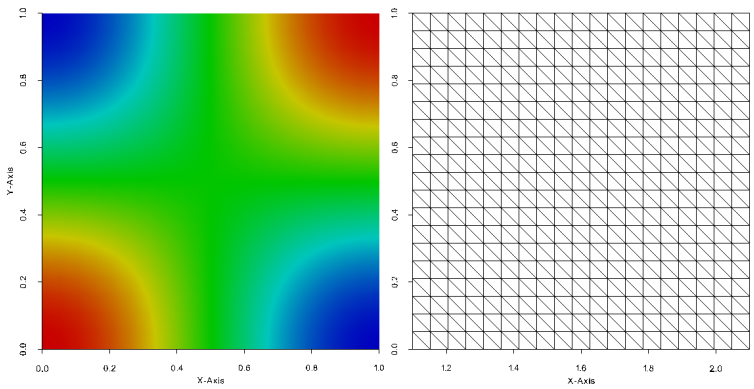


Figure 3.3: Solution of the Laplace 2D problem with Neumann condition on the square $[0,1]^2$

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:

```
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(_origin=Point(0.,0.,0.), _length=1, _nnodes=10), tetrahedron,
    1, _gmsh,"P1 mesh");
...
```

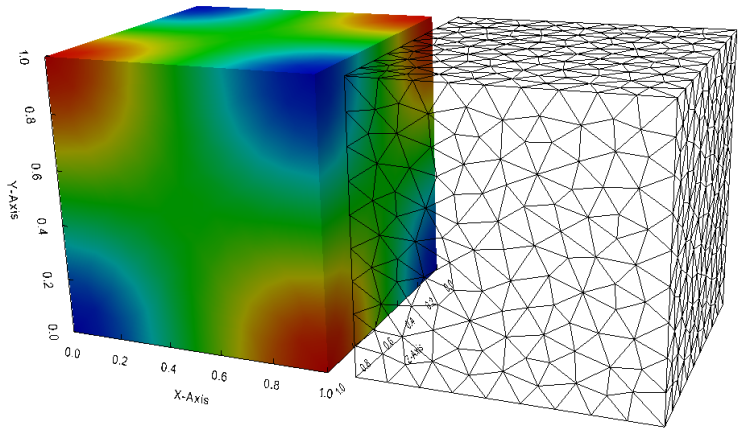


Figure 3.4: Solution of the Laplace 3D problem with Neumann condition on the unit cube $[0,1]^3$

3.2.2 Dirichlet condition

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

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

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

$$\left| \begin{array}{l} \text{find } u \in V = \{v \in L^2(\Omega), \nabla v \in (L^2(\Omega))^2\} \text{ such that} \\ \quad \displaystyle\int_\Omega \nabla u.\nabla v = \int_\Omega f\, v \quad \forall v \in V,\ v = 0 \text{ on } \Sigma^- \cup \Sigma^+ \\ \quad u = 1 \qquad\qquad\qquad\qquad \text{on } \Sigma^- \cup \Sigma^+ \end{array} \right.$$

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

```
#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");
  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;
}
```

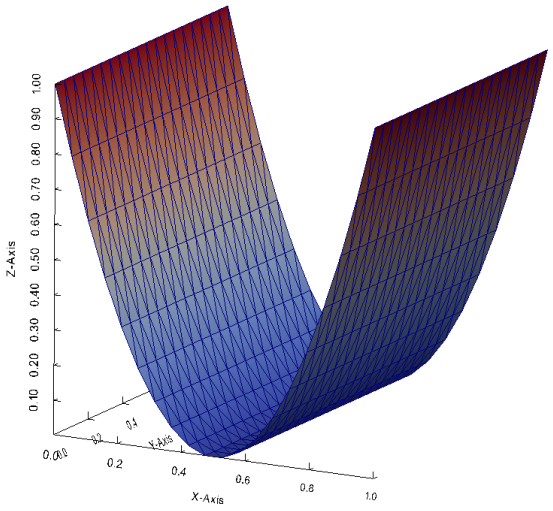


Figure 3.5: Solution of the Laplace 2D problem with Dirichlet condition on the unit square $[0,1]^2$

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:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u_{|\Gamma^-} = 0 \text{ and } u_{|\Gamma^+} = 0 \\ u_{|\Sigma^-} = u_{|\Sigma^+} \text{ and } \partial_x u_{|\Sigma^-} = \partial_x u_{|\Sigma^+} \end{cases}$$

and its variational formulation in $V = \{v \in H^1(\Omega),\ u_{|\Gamma} = 0 \text{ and } u_{|\Sigma^-} = u_{|\Sigma^+}\}$:

$$\left| \begin{array}{c} \text{find } u \in V \text{ such that} \\ \displaystyle\int_\Omega \nabla u.\nabla v = \int_\Omega f\, v \quad \forall v \in V. \end{array} \right.$$

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

```
#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"), sigmaP=mesh2d.domain("x=1");
  Domain gammaM=mesh2d.domain("y=0"), 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
```

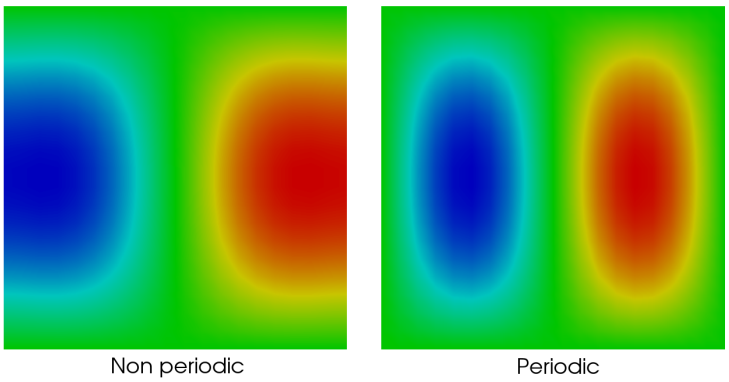
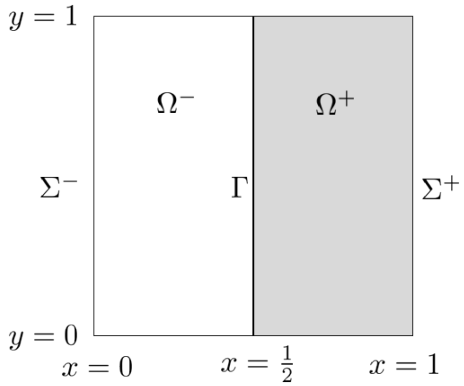


Figure 3.6: Solution of the Laplace 2D problem with periodic condition on the unit square $[0,1]^2$

3.2.4 Transmission condition



We turn to the Laplace problem with transmission condition:

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

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

$$\left| \begin{array}{l} \text{find } (u^-, u^+) \in H^1(\Omega^-) \times H^1(\Omega^+), u^-_{|\Sigma^-} = 1, u^+_{|\Sigma^-} = 1,\ u^-_{|\Gamma} = u^+_{|\Gamma} \text{ such that} \\ \displaystyle\int_{\Omega^-} \nabla u^-.\nabla v^- + \int_{\Omega^+} \nabla u^+.\nabla v^+ = \int_{\Omega^-} f\, v^- + \int_{\Omega^+} f\, v^+ \quad \forall v \in V. \end{array} \right.$$

Note that derivatives matching is taken into account in a weak sense. The implementation in XLiFE++, using P^2 Lagrange finite element, looks like:

```
#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 Γ.

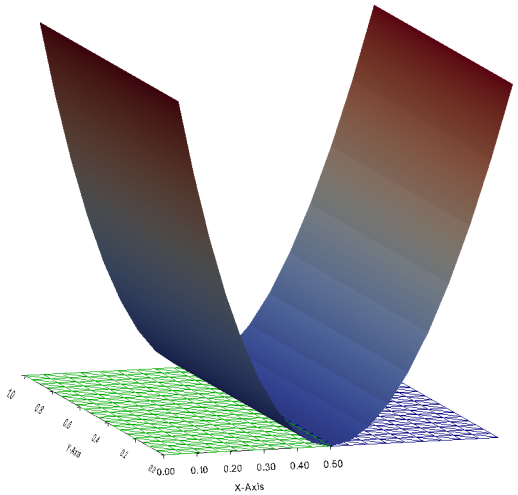


Figure 3.7: Solution of the Laplace 2D problem with transmission condition on the unit square $[0,1]^2$

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 adressed:

```
#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");

  //solve linear system AX=F using LU factorization
  TermVector U=directSolve(A, F);
  saveToFile("U_LA", U, vtu);
  return 0;
}
```

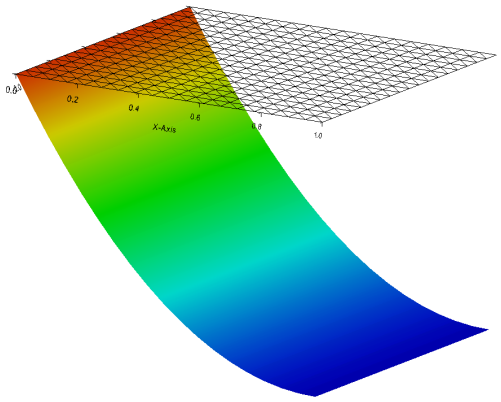


Figure 3.8: Solution of the Laplace 2D problem with average condition on the unit square $[0,1]^2$

 Be care with some average conditions. For instance, when adding the "full" average condition

$$\int_\Omega u = 0$$

the resulting reduced matrix is a full matrix. So, the problem is bigger and slower to solve!

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):

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

with the following variational formulation:

$$\left| \begin{array}{l} \text{find } (u,p) \in L^2(\Omega) \times H(div,\Omega) \text{ such that} \\ \quad -\displaystyle\int_\Omega \operatorname{div} p\, v = \int_\Omega f\, v \quad \forall v \in L^2(\Omega) \\ \quad \displaystyle\int_\Omega u \operatorname{div} q + \int_\Omega p.q = 0 \quad \forall q \in H(div,\Omega). \end{array} \right.$$

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, RaviartThomas, 1, "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), vtu);
  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:

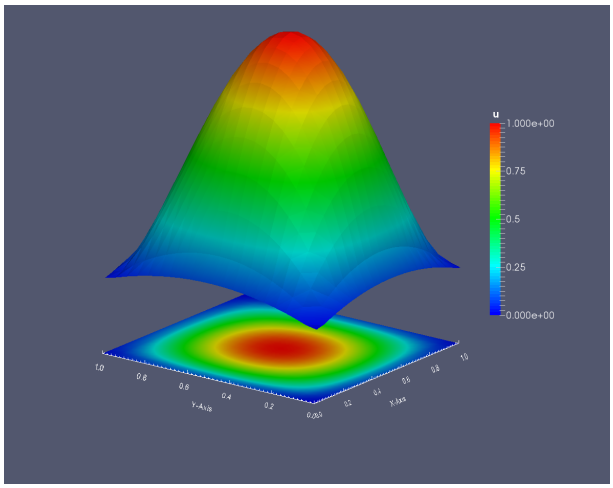


Figure 3.9: Solution of the Laplace 2D problem with mixed formulation P0-RT1 on the unit square $[0,1]^2$

3.4 2D Maxwell equations using Nedelec elements

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

$$\begin{cases} \operatorname{curl}\operatorname{curl}\mathbf{E} - \omega^2\mu\,\varepsilon\mathbf{E} = \mathbf{f} & \text{in } \Omega \\ \mathbf{E}\times n = 0 & \text{on } \partial\Omega \end{cases}$$

with the following weak form:

$$\left| \begin{array}{l} \text{find } \mathbf{E}\in V = \{\mathbf{v}\in H(curl,\Omega),\ \mathbf{v}\times n = 0 \text{ on } \partial\Omega\} \text{ such that} \\ \qquad \displaystyle\int_\Omega \operatorname{curl}\mathbf{E}\operatorname{curl}\mathbf{v} = \int_\Omega \mathbf{E}\,\mathbf{v} \quad \forall \mathbf{v}\in V. \end{array} \right.$$

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

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

Real omg=1, eps=1, mu=1, a=pi_, ome=omg* omg* 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
  Mesh mesh2d(Rectangle(_xmin=0, _xmax=1, _ymin=0, _ymax=1, _nnodes=50,
      _side_names="Gamma"), triangle, 1, gmsh);
  Domain omega=mesh2d.domain("Omega");
  Domain gamma=mesh2d.domain("Gamma");
  //define space and unknown
  Space V(omega, Nedelec, 1, "V");
  Unknown e(V, "E");
  TestFunction q(e, "q");
  //define forms, matrices and vectors
  BilinearForm aev=intg(omega, curl(e)|curl(q)) - ome*intg(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
  TermVector E=directSolve(A, b);
  // P1 interpolation, L2 projection on H1
  Space W(omega, P1, "W");
  TermVector EP1=projection(E, W, 2);
  EP1.name("E");
  saveToFile("E", EP1, vtu);
  return 0;
}
```

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:

$$\left| \begin{array}{l} \text{find } \mathbf{E}_1 \in L^2(\Omega) \text{ such that} \\ \displaystyle\int_\Omega \mathbf{E}_1 \, \mathbf{w} = \int_\Omega \mathbf{E} \, \mathbf{w} \quad \forall \mathbf{w} \in L^2(\Omega). \end{array} \right.$$

Using an H1 conforming approximation for $\mathbf{E}_1$ leads to a continuous representation of the projection. We show on the next figure the E_x component field provided by this example.

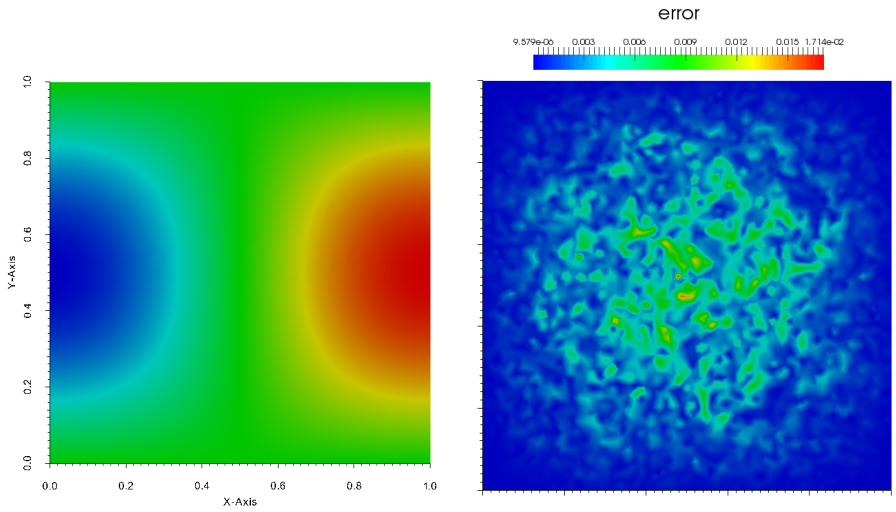


Figure 3.10: First component of the solution of the Maxwell 2D problem using Nedelec first family elements, and nodal error

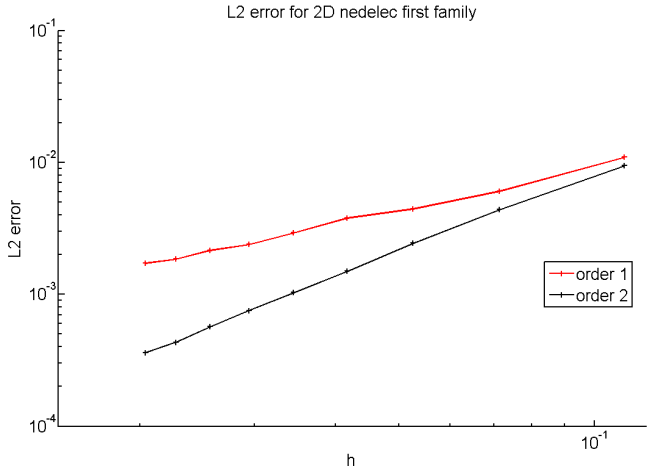


Figure 3.11: L^2 errors versus the step h for 1 and 2 order Nedelec first family approximation

3.5 Eigenvalues and eigenvectors of Laplace operator

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

$$\begin{cases} -\Delta u + u = \lambda u & \text{in } \Omega \\ \partial_n u = 0 & \text{on } \partial\Omega \end{cases}$$

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

$$\left| \begin{array}{c} \text{find } (u, \lambda) \in V \times \mathbb{R} \text{ such that} \\ \displaystyle\int_\Omega \nabla u.\nabla v + \int_\Omega u\, v = \lambda \int_\Omega u\, v \quad \forall v \in V. \end{array} \right.$$

```
#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 = eigenInternSolve(A, M, _nev=10, _mode=_krylovSchur,
      _which="SM");   //internal solver
  thePrintStream << eigs.values;
  saveToFile("eigs", eigs.vectors, vtu);
  return 0;
}
```

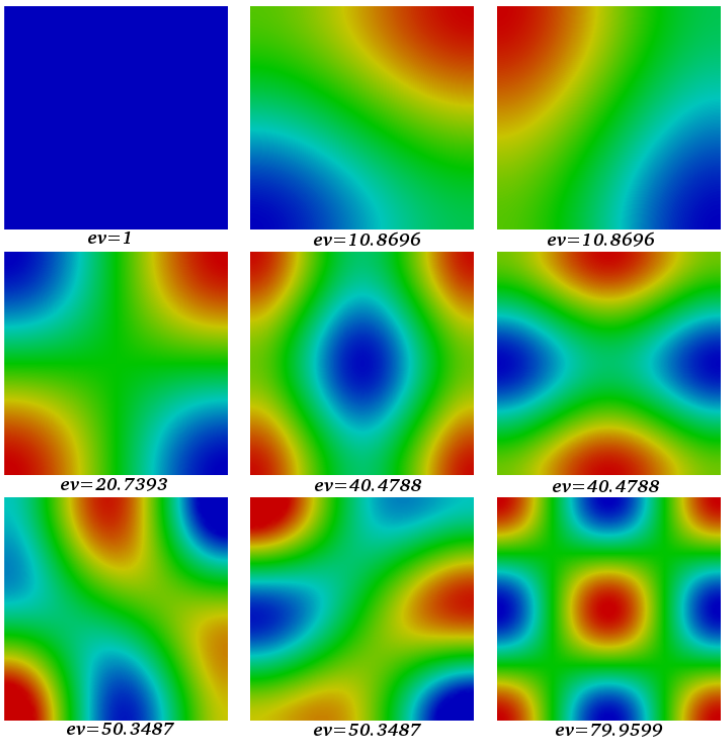


Figure 3.12: 9 first eigen vectors of the Laplace 2D problem with P2 elements

3.6 3D Helmholtz problem using single layer potential integral equation

XLiFE++ is also able to deal with integral equation. This example illustrates the computation of the acoustic scattering by a sphere:

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

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 Helmhotz equation:

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

We deal with the variational formulation in $V = H^{\frac{1}{2}}(S)$:

$$\left| \begin{array}{c} \text{find } p \in V \text{ such that} \\ \displaystyle\int_S \int_S p(x)\, G(x,y)\, \bar{q}(y)\, dx\, dy = -\int_S u_{inc}\bar{q} \quad \forall q \in V. \end{array} \right.$$

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

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

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

```
#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");

  // form definitions
  IntegrationMethods ims(Duffy,5, 0., Gauss_Legendre, 5, 1., Gauss_Legendre,
      4, 2.,Gauss_Legendre, 3 );
  BilinearForm blf0=intg(disk,disk, u1*G*v1, ims);
  LinearForm fv0 = -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;
  theCout << "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;
}
```

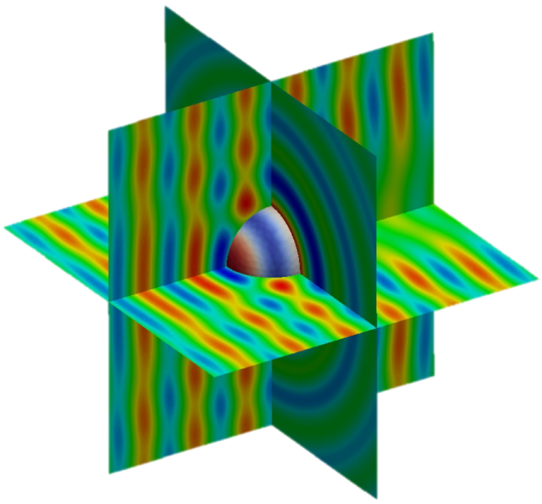


Figure 3.13: Solution of the 3D Helmholtz problem using single layer BEM on the unit sphere

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:

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

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

Let Ω be a domain that strictly surrounding the disk D and Σ its boundary. We have to point out that in this case, we use normals going outside the domain of computation Ω but then the normal on the obstacle (defined on Γ) is going inside the obstacle, that is opposite to usual case (see Figure 3.14). 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):

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

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 Ω not the obstacle) on Γ and n_x will denote the outward normal on Σ. Now matching values and normal derivative on Σ, 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 $\Gamma \times \Sigma$:

$$\begin{aligned}(\partial_{n_x} + \lambda)u(x) = \; & -\int_\Gamma \partial_{n_x}\partial_{n_y} G(x,y)\, u(y)\, dy - \lambda \int_\Gamma \partial_{n_y} G(x,y)\, u(y)\, dy \\ + \; & \int_\Gamma \partial_{n_x} G(x,y)\, g(y)\, dy + \lambda \int_\Gamma G(x,y)\, g(y)\, dy = \mathcal{R}_\lambda(u)(x)\end{aligned}$$

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

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

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

$$\left| \begin{array}{c} \text{find } u \in V \text{ such that } \forall v \in V \\ \displaystyle\int_\Omega \nabla u.\nabla \bar{v} - k^2 \int_\Omega u\, \bar{v} + \lambda \int_\Sigma u\, \bar{v} + \int_\Sigma \int_\Gamma u(y)\partial_{n_x}\partial_{n_y} G(x,y)\, \bar{v}(x) + \lambda \int_\Sigma \int_\Gamma u(y)\partial_{n_y} G(x,y)\, \bar{v}(x) \\ \displaystyle = \int_\Gamma g\, \bar{v} + \int_\Sigma \int_\Gamma g(y)\partial_{n_x} G(x,y)\, \bar{v}(x) + \lambda \int_\Sigma \int_\Gamma g(y) G(x,y)\, \bar{v}(x). \end{array} \right.$$

Considering the geometrical configuration:

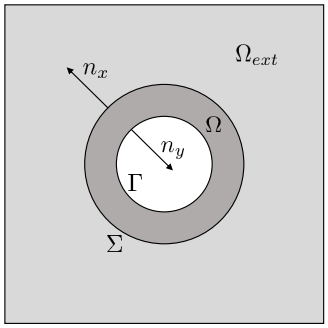


Figure 3.14: Geometrical configuration for the FEM-Integral Representation problem. The normal on Γ is going inside the obstacle (to point outside Ω).

the variational formulation is implemented as follows:

```
#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^{ikx}
}

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"));
  Square rect(_center=Point(0.,0.), _length=l, _nnodes=nr,
      _domain_name="Omega_ext");
  Mesh mesh(rect+(d2-d1), _triangle, 1, _gmsh);
  Domain omega=mesh.domain("Omega");
  Domain sigma=mesh.domain("Sigma");
  Domain gamma=mesh.domain("Gamma");
  Domain omega_ext=mesh.domain("Omega_ext");  //for integral representation
  sigma.setNormalOrientation(_outwardsDomain,omega); //outwards normals
  gamma.setNormalOrientation(_outwardsDomain,omega);

  //create P2 Lagrange interpolation
  Space V(omega,P2,"V",true);
  Unknown u(V,"u");  TestFunction v(u,"v");
  // create bilinear form and linear form
  Complex lambda=-i_*k;
  BilinearForm auv =
    intg(omega,grad(u)|grad(v))-k2*intg(omega,u*v)+lambda*intg(sigma,u*v)
    +intg(sigma,gamma,u*(grad_y(grad_x(H)|_nx)|_ny)*v)
    +lambda*intg(sigma,gamma,u*(grad_y(H)|_ny)*v);
```

```
  BilinearForm alv =
    intg(sigma,gamma,u*(grad_x(H)|_nx)*v)+lambda*intg(sigma,gamma,u*H*v);
  TermMatrix A(auv), ALV(alv);
  TermVector B(intg(gamma,g*v));
  TermVector G(u,gamma,g);
  B+=ALV*G;
  //solve linear system AU=F
  TermVector U=directSolve(A,B);
  saveToFile("U.vtk",U,vtk);
  //integral representation on omega_ext
  Space Vext(omega_ext,P2,"Vext",false);
  Unknown uext(Vext,"uext");
  TermVector Uext =
    -integralRepresentation(uext, omega_ext, intg(gamma,(grad_y(H)|_ny)*U))
    +integralRepresentation(uext, omega_ext, intg(gamma,H*G));
  saveToFile("Uext.vtk",Uext,vtk);
  //total field
  TermVector Ui(u, omega, ui), Utot=Ui+U;
  TermVector Uiext(uext, omega_ext, ui), Utotext=Uiext+Uext;
  saveToFile("Utot.vtk",Utot,vtk);
  saveToFile("Utotext.vtk",Utotext,vtk);
  return 0;
}
```

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 3.15.

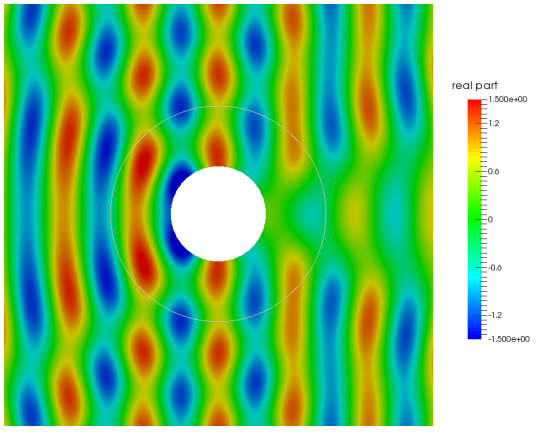


Figure 3.15: 2D Helmholtz diffraction problem using FE-IR method: real part of the total field

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 Ω that is heterogeneous, its boundary Γ and the exterior domain Ω_{ext} that is homogeneous (see Figure 3.16).

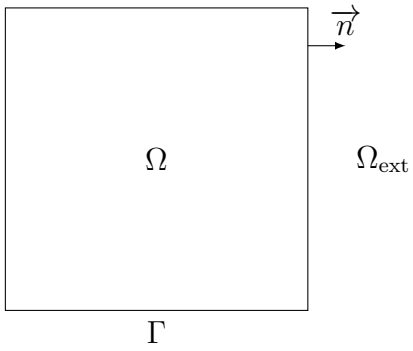


Figure 3.16: Domains for computation: Ω the heterogeneous medium, Ω_{ext} the homogeneous exterior domain and $\Gamma = \partial\Omega$.

We solve:

$$\begin{cases} \Delta u(x) + k^2\eta^2(x)u(x) = 0 & \text{in } \mathbb{R}^2 \\ u(x) = -u_i(x) & \text{on } \Gamma \end{cases}$$

with $\eta(x) = 1$ in Ω_{ext}, and $\eta(x)$ that can vary in Ω, and finally with $u_i = e^{ikx}$.
We will use: $\Omega = [-0.5, 0.5]^2$ and

$$\eta(x) = \begin{cases} \exp\left(-(x_1^2 - 0.25) * (x_2^2 - 0.25)/(2. * 0.05)\right), \text{ when } \max(x_1, x_2) < 0.5. \\ 1 \text{ otherwise.} \end{cases}$$

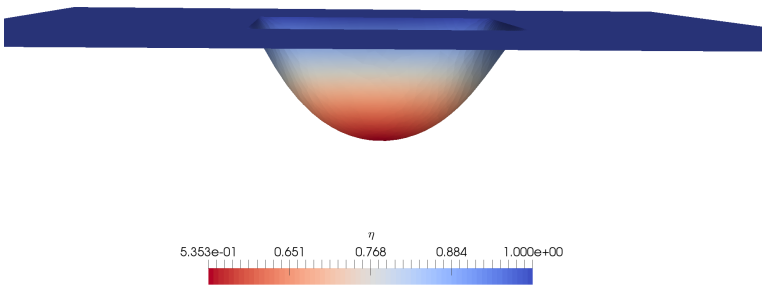


Figure 3.17: $\eta(x)$ in $\Omega \cup \Omega_{\text{ext}}$.

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:

$$\left|\begin{array}{c}\text{Find } u \in H^1(\Omega) \text{ such that :} \\ \int_\Omega \nabla u(x) \cdot \nabla \bar{v}(x) dx - k^2 \int_\Omega \eta^2(x) u(x) \bar{v}(x) dx - \int_\Gamma \lambda(x) \bar{v}(x) dx = 0, \quad \forall v \in H^1(\Omega)\end{array}\right. , \tag{3.2}$$

with $\lambda = \frac{\partial u}{\partial n}$ is the normal trace of u on Γ.

- in the BEM part, we solve:

$$\begin{cases} \Delta u + k^2 u = 0 & \text{in } \Omega_{\text{ext}} \\ u = -u_i & \text{on } \Gamma. \end{cases} \tag{3.3}$$

The scattered field verifies:

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

with u the total field solution of the equation and λ the normal trace of u on Γ, S_Γ and K_Γ 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 Γ, we obtain the integral equation:

$$\left(\frac{I}{2} - K_\Gamma\right) u(x) + S_\Gamma \lambda(x) = u_i(x), x \in \Gamma. \tag{3.5}$$

The resulting variational formulation for the BEM part is then:

$$\left|\begin{array}{c}\text{Find } u \in H^{1/2}(\Gamma) \text{ and } \lambda \in H^{-1/2}(\Gamma) \text{ such that :} \\ \frac{1}{2}\int_\Gamma u(x)\bar{\tau}(x)dx - \int_{\Gamma\times\Gamma} u(y)\frac{\partial G(x,y)}{\partial n_y}\bar{\tau}(x)dydx + \int_{\Gamma\times\Gamma}\lambda(y)G(x,y)\bar{\tau}(x)dydx \\ = \int_\Gamma u_i(x)\bar{\tau}(x)dx, \forall \tau \in H^{1/2}(\Gamma).\end{array}\right. \tag{3.6}$$

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 Ω and the BEM solution in Ω_{ext} to obtain a solution on the whole domain $\Omega \cup \Omega_{\text{ext}}$ to simplify the visualisation.

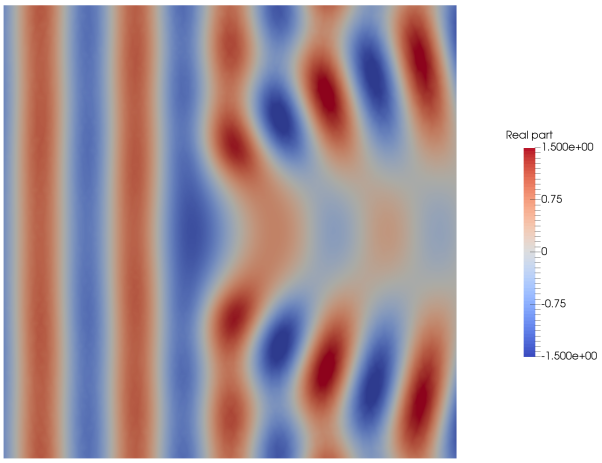


Figure 3.18: Solution of the FEM-BEM problem.

The code of this example follows:

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

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

int main(int argc, char** argv)
{
  init(_lang=en);    // mandatory initialization of xlifepp
  verboseLevel(10);
  Real k=10.;
  //meshing
  Real hsize=(2*pi_/k)/15.;
  Square sp(_center=Point(0.,0.), _length=1., _hsteps=hsize,
      _domain_name="Omega", _side_names="Gamma");
  Mesh m1=Mesh(sp, _triangle, 1, _gmsh);
  Domain omega = m1.domain("Omega");
  Domain gamma = m1.domain("Gamma");
  theCout << "Mesh size = " << hsize << eol;
  theCout << "Number of Triangles = " << m1.nbOfElements() << eol;
```

```
//defining parameter and kernel
Parameters pars;
pars << Parameter(k,"k");
Vector<Real> nv(2);
pars << Parameter(&nv,"_n");
Kernel G=Helmholtz2dKernel(pars);
Function finc(g1,pars);
//defining space, unknown and test function
Space V1(omega,P1,"V1",false);
Space V0(gamma,P1,"V0",false);
Unknown u1(V1,"u1"); TestFunction v1(u1,"v1");
Unknown l0(V0,"l0"); TestFunction lt0(l0,"lt0");
theCout<<"Nb dofs BEM= " << V0.nbDofs() << " Nb dofs FEM= " << V1.nbDofs()
   << eol;
//defining bilinear and linear form
IntegrationMethods ims(Duffy, 15, 0., _defaultRule, 12, 1., _defaultRule,
   10, 2., _defaultRule, 8);
BilinearForm blf=intg(omega,grad(u1)|grad(v1))-k*k*intg(omega,eta2*u1*v1)
                 - intg(gamma,l0*v1) + 0.5*intg(gamma,u1*lt0)
                 - intg(gamma,gamma,u1*ndotgrad_y(G)*lt0,ims)
                 + intg(gamma,gamma,l0*G*lt0,ims);
LinearForm lf=intg(gamma,finc*lt0);
//computing FEM/BEM matrix and right hand side vector
TermMatrix lhs(blf,"lhs");
TermVector rhs(lf);
//solving linear system using direct method
TermVector sol=directSolve(lhs,rhs);

// Representing 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");
Function Find(find,pars);
TermVector findex(ur,S,Find);
saveToFile("findex",findex,_vtu); // Representing eta
TermVector Uint=interpolate(ur,S_int,sol(u1)); // FEM solution (total
   field)
saveToFile("Uint",Uint,_vtu);

// Representing of the BEM part
IntegrationMethods imr(_GaussLegendreRule, 20, 1., _GaussLegendreRule, 10,
   2., _GaussLegendreRule,5);
TermVector Uext =
    - integralRepresentation(ur, S_ext, intg(gamma,G*sol(l0),imr))
    + integralRepresentation(ur, S_ext,
        intg(gamma,ndotgrad_y(G)*sol(u1),imr));

TermVector Uinc(ur,S_ext,finc);
saveToFile("Uinc",Uinc,_vtu); // Incident field
saveToFile("Uext",Uext,_vtu); // scattered field in exterior domain
TermVector Uext_t = Uext + Uinc;
saveToFile("Uext_t",Uext_t,_vtu); // Total field in exterior domain
```

```
  TermVector U=merge(Uint,Uext_t); // Merged FEM and BEM solutions
  saveToFile("U",U,_vtu);
  theCout << "Program finished" << eol;
  return 0;
}
```

3.9 3D Maxwell problem using EFIE

Solving diffraction of an electromagnetic plane wave on a obstacle using BEM is more intricate. Indeed, it is a vector problem and it involves Raviart-Thomas elements. We show how XLiFE++ can deal easily with.

Let Γ be the boundary of a bounded domain Ω of $\mathbb{R}^3$, we want to solve the Maxwell problem on the exterior domain Ω_e:

$$\begin{cases} \operatorname{curl}\mathbf{E} - ik\mathbf{H} = 0 & \text{in } \Omega_e \\ \operatorname{curl}\mathbf{H} + ik\mathbf{E} = 0 & \text{in } \Omega_e \\ \mathbf{E}\times\mathbf{n} = 0 & \text{on } \Gamma \\ \lim\limits_{|x|\to\infty}\left((\mathbf{H}-\mathbf{H}_{inc})\times\dfrac{x}{|x|} - (\mathbf{E}-\mathbf{E}_{inc})\right) = 0 & \text{(Silver-Muller condition)} \end{cases}$$

where $(\mathbf{E}_{inc}, \mathbf{H}_{inc})$ is an incident field (a solution of Maxwell equation in free space), for instance a plane wave.

The EFIE (Electric Field Integral Equation) consists in finding the potential $\mathbf{J}$ in the space

$$H_{\text{div}}(\Gamma) = \left\{\mathbf{V}\in L^2(\Gamma)^3, \mathbf{V}.\mathbf{n} = 0, \operatorname{div}_\Gamma\mathbf{V}\in L^2(\Gamma)\right\}$$

such that, $\forall\mathbf{V}\in H_{\text{div}}(\Gamma)$

$$k\int_\Gamma\int_\Gamma \mathbf{J}(y)\,G(x,y).\mathbf{V}(x) - \frac{1}{k}\int_\Gamma\int_\Gamma \operatorname{div}_\Gamma\mathbf{J}(y)\,G(x,y)\operatorname{div}_\Gamma\mathbf{V}(x) = -\int_\Gamma \mathbf{E}_{inc}.\mathbf{V}$$

where G is the Green function related to the Helmholtz 3D problem in free space.

This equation has a unique solution, except for a discrete set of wavenumbers corresponding to the resonance frequencies of the cavity Ω.

Using the Stratton-Chu representation formula, the scattered electric field may be reconstructed in Ω_e:

$$\mathbf{E}(x) = \mathbf{E}_{inc}(x) + \frac{1}{k}\int_\Gamma \nabla_x G(x,y)\operatorname{div}_\Gamma\mathbf{J}(y) + k\int_\Gamma G(x,y)\,\mathbf{J}(y).$$

This problem is implemented in XLiFE++ as follows:

```
#include "xlife++.h"
using namespace xlifepp;
Vector<complex_t> data_incField(const Point& P, Parameters& pars)
{
  Vector<real_t> incPol(3,0.); incPol(1)=1.; Point incDir(0.,0.,1.) ;
  Real k = pars("k");
  return  incPol * exp(i_*k * dot(P,incDir));
}
```

```
Vector<complex_t> uinc(const Point& P, Parameters& pars)
{
    Vector<real_t> incPol(3,0.); incPol(1)=1.; Point incDir(0.,0.,1.) ;
    Real k = pars("k");
    return  incPol*exp(i_*k * dot(P,incDir));
}

int main(int argc, char** argv)
{
  init(_lang=en);
  //define parameters and functions
  Real k= 1, R=1.; Parameters pars;
  pars << Parameter(k, "k") << Parameter(R, "radius");
  Kernel H = Helmholtz3dKernel(pars);           // load Helmholtz3D kernel
  Function Einc(data_incField, pars);           // define right hand side
  Function Uex(scatteredFieldMaxwellExn,pars);
  // meshing the unit sphere
  Number npa=15; Point O(0,0,0);
  Sphere sphere(_center=O,_radius=R,_nnodes=npa,_domain_name="Gamma");
  Mesh meshSh(sphere, triangle, 1, gmsh);
  Domain Gamma = meshSh.domain("Gamma");
  //define FE-RT1 space and unknown
  Space V_h(Gamma, RT_1, "Vh");
  Unknown U(V_h,"U"); TestFunction V(U,"V");
  //compute BEM system and solve it
  IntegrationMethods
      ims(_SauterSchwabIM,4,0.,_defaultRule,5,2.,_defaultRule,3);
  BilinearForm auv = k*intg(Gamma,Gamma, U*H|V,ims)
                   -(1./k)*intg(Gamma,Gamma, div(U)*H*div(V),ims);
  TermMatrix A(auv, "A");
  TermVector B(-intg(Gamma,Einc|V));
  TermVector J = directSolve(A,B);
  //get P1 representation of solution and export it to vtu file
  Space L_h(Gamma, P1, "Lh");
  Unknown U3(L_h,"U3",3);TestFunction V3(U3,"V3");
  TermVector JP1=projection(J, L_h, 3, _L2Projector);
  saveToFile("JP1", JP1(U3[1]), vtu);
  //integral representation on y=0 plane (excluding sphere),using P1 nodes
  Number npp=30, npc=5;
  Square sqx(_center=O,_length=20.,_nnodes=npp,_domain_name="Omega");
  Disk dx(_center=O, _radius=1.2*R, _nnodes=npc);
  Mesh mx0(sqx-dx, triangle, 1,gmsh);
  mx0.rotate3d(1.,0.,0.,pi_/2);
  Domain py0 = mx0.domain("Omega");
  Space Vy0(py0,P1,"Vy0",false);
  Unknown W(Vy0,"W",3);
  IntegrationMethods im(_defaultRule,10,1.,_defaultRule, 5);
  TermVector Uext=
   (1./k)*integralRepresentation(W, py0, intg(Gamma,grad_x(H)*div(U),im), J)
      + k*integralRepresentation(W, py0, intg(Gamma,H*U,im), J);
  saveToFile("Uext", Uext, vtu);
  //build exact solution, export to vtu file and compute error
  TermVector Uexa(W, py0, Uex);
  saveToFile("Uexa", Uexa, vtu);
  TermMatrix M(intg(py0,W|W));
  TermVector E=Uext-Uexa;
  theCout<<"L2 error = "<<sqrt(real(M*E|E))<<eol;
```

```
  return 0;
}
```

In order to build an approximated space of $H_{\text{div}}(\Gamma)$ we use the Raviart-Thomas element of order 1.
As the integrals involved in bilinear form are singular, we use here the Sauter-Schwab method to compute them when two triangles are adjacent, a quadrature method of order 5 if the two triangles are close ($0 < d(T1, T2) < 2h$) and a quadrature method of order 3 when the two triangles are far ($d(T1, T2) >= 2h$).
Note that the unknowns in RT approximation are the normal fluxes on the edge of the triangulation. In order to plot the potential $\mathbf{J}$, we have to move to a P1 representation, say $\widetilde{\mathbf{J}}$. This can be done using a L2 projection from $H_{\text{div}}(\Gamma)$ to $L^2(\Gamma)$:

$$\int_\Gamma \widetilde{\mathbf{J}} \,|\, \mathbf{V} = \int_\Gamma \mathbf{J} \,|\, \mathbf{V} \quad \forall V \in L^2(\Gamma)$$

This is what is done by the XLiFE++ function `projection`.
We obtain the following potential:

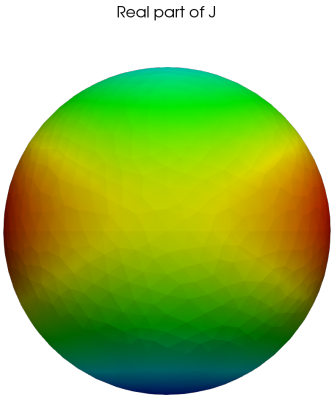


Figure 3.19: 3D Maxwell problem on the unit sphere, using EFIE, potential

On the following figures, we show the approximated electric field and the exact electric field. The component E_y is not shown because it is zero.

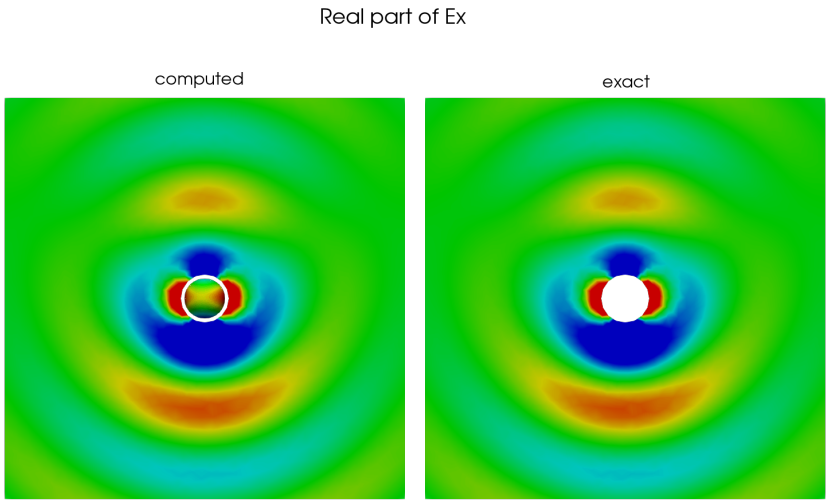


Figure 3.20: 3D Maxwell problem on the unit sphere, using EFIE, x component

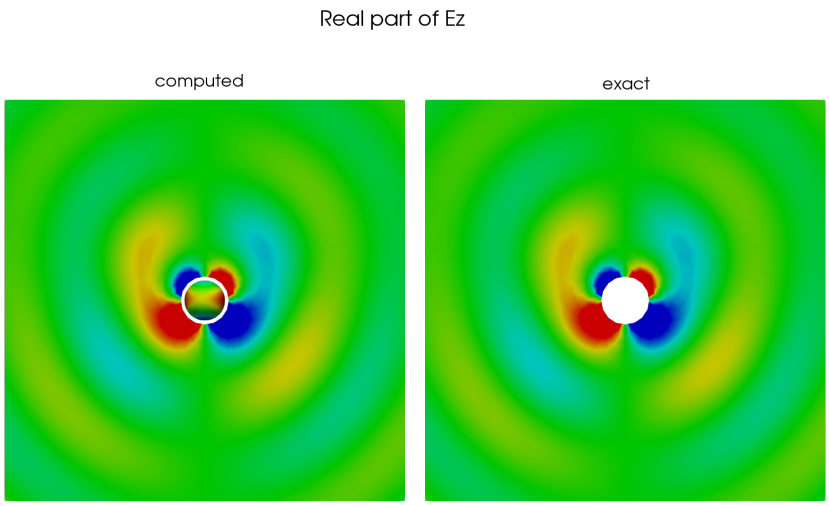


Figure 3.21: 3D Maxwell problem on the unit sphere, using EFIE, y component

3.10 Elasticity problem

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

$$\begin{cases} -\text{div}(\sigma(\mathbf{u}) - \omega^2 \mathbf{u} = \mathbf{f} & \text{in } \Omega \\ \sigma(\mathbf{u})\mathbf{n} = 0 & \text{on } \partial\Omega \end{cases}$$

For homogeneous isotropic material :

$$\sigma(\mathbf{u}) = \lambda \text{div}(\mathbf{u})\mathbb{I} + 2\mu\varepsilon(\mathbf{u}) \; \varepsilon_{ij}(\mathbf{u}) = \partial_i u_j.$$

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

$$\left| \begin{array}{c} \text{find } \mathbf{u} \in V \text{ such that} \\ \lambda \displaystyle\int_\Omega \varepsilon(\mathbf{u}) : \varepsilon(\bar{\mathbf{v}}) + 2\mu \int_\Omega \text{div}(\mathbf{u}) \, \text{div}(\bar{\mathbf{v}}) - \omega^2 \int_\Omega u \, \bar{v} = \int_\Omega \bar{\mathbf{f}}.\bar{\mathbf{v}} \quad \forall \mathbf{v} \in V. \end{array} \right.$$

This is implemented as follows:

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

//data function
Vector<Real> f(const Point& P, Parameters& pa = defaultParameters)
{  Vector<Real> F(2,0.); F(2)=-0.005; return F;}

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

 //mesh rectangle
  Rectangle rect(_center=Point(0.,0.), _xlength=20, _ylength=2,
      _nnodes=Numbers(50,5), _domain_name="Omega",
      _side_names=Strings("","","","Gamma"));
  Mesh mesh2d(rect, triangle, 1, gmsh);
  Domain omega=mesh2d.domain("Omega"), Gamma=mesh2d.domain("Gamma");
  // create P1 Lagrange interpolation
```

```
  Space V(omega, P1, "V");
  Unknown u(V, "u", 2);  TestFunction v(u, "v");
  // create bilinear form and linear form
  Real lambda=112.134, mu=83.53, omg2=0, rho=7.86;
  BilinearForm auv = lambda*intg(omega, epsilon(u) % epsilon(v))
                   + 2*mu*intg(omega, div(u)*div(v)) - omg2*intg(omega,u|v);
  LinearForm fv=intg(omega, f|v);
  EssentialConditions ecs= (u|Gamma=0);
  TermMatrix A(auv, ecs,"A");
  TermVector B(fv, "B");
  //solve linear system AX=B using direct method
  TermVector U=directSolve(A, B);
  thePrintStream<<U;
  saveToFile("U", U, vtu);

  //create the deformation of the mesh
  for(number_t i=0;i<mesh2d.nbOfNodes();i++)
      mesh2d.nodes[i] += U.evaluate(mesh2d.nodes[i]).value<std::vector<Real>
          >();
  mesh2d.saveToFile("Ud",msh);

  return 0;
}
```

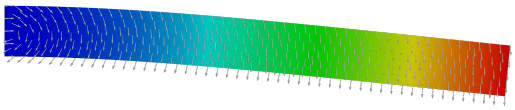


Figure 3.22: Displacement and modulus of the solution of the elasticity 2D problem

3.11 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:

$$\begin{cases} \dfrac{\partial^2 u}{\partial t^2} - c^2 \Delta u = f & \text{in } \Omega \times]0,T[\\ \dfrac{\partial u}{\partial n} = 0 & \text{in } \partial\Omega \times]0,T[\\ u(x,0) = \dfrac{\partial u}{\partial t}(x,0) = 0 & \text{in } \Omega \end{cases}$$

Using classical leap-frog scheme with time discretization $t^n = n\Delta t$, leads to (u^n approximates $u(x,t^n)$):

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

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

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

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:

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

where

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

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

```
#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");
  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 Figure 3.23 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.

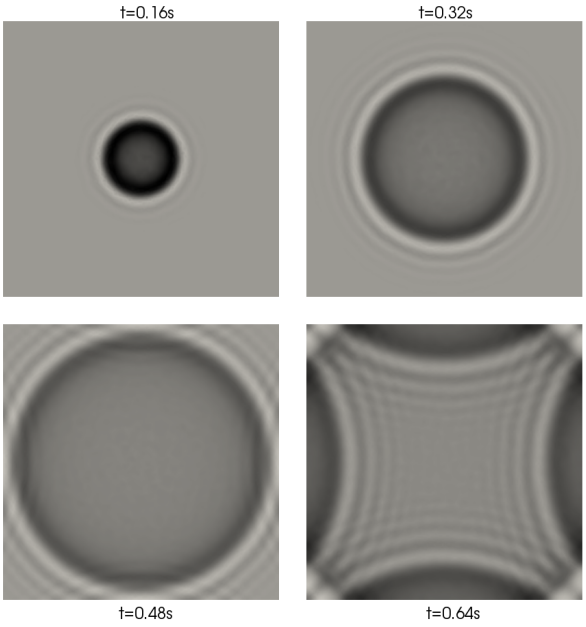


Figure 3.23: Solution of the wave equation at different instants for a constant source localized in disk with center $(0.5, 0.5)$

4 XLiFE++ written in C++

This chapter is devoted to the basics of C++ language required to use XLiFE++. It is adressed to people who does not know C++.

4.1 Instruction sequence

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

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

Instruction block may be nested in other one:

```
{
 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:

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

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

```
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 algebric 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:

```
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*):

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

For multiple choice, use the **switch** instruction:

```
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:

```
for( initialization; end_test; incrementing sequence)
{
 ...
}
```

The simplest loop is:

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

An other example with no initializer and two incrementing values:

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

A **for** loop may be replaced by a **while** loop:

```
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 <<:

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

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

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

```
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 :

```
ofstream out;
out.open("myfile");
Real x=2.25;
Number i=3;
String msg=" Xlife++ :";
out << msg << " x=" << x << " i=" << i << eol;
out.close();
```

To read from a file :

```
ifstream in;
in.open("myfile");
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**.

 All stream stuff id defined in the C++ standard template library (STL). To use it, write at the beginning of your c++ files :

```
#include <iostream>
#include <fstream>
...

using namespace std;
```

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 :

```
#include <cmath>
using namespace std;
...
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:

```
class Complex
{
  public:
  float x, y;
  Complex(float a=0, float b=0) : x(a), y(b){}
  float abs() {return sqrt(x*x+y*y);
  Complex& operator+=(const Complex& c)
```

```
        {x+=c.x;y+=c.y;return *this;}
  ...
};
```

Classes have special functions, called constructors, to create object. They have the name of the class and are invoked at the declaration of the object:

```
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, compiler are optimized to avoid copy.

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

```
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:

```
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 object (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 user documentation 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`, ...)
`Mesh` mesh structure containing nodes, geometric elements, ...
`Domain` alias of geometric domains describing part of the mesh, in particular boundaries, and `Domains` to deal with vectors of `Domain`'s
`Space` class handles discrete spaces (FE space or spectral space) and `Spaces` some vectors of `Space`'s
`Unknown`, `TestFunction` abstract elements of space and `Unknowns`, `TestFunctions` to handle vector of `Unknown`'s and `TestFunction`'s
`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)

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:

- 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, ...): section 5.2
3. How to extrude geometries (by translation or rotation): section 5.3
4. How to define a mesh from a geometry: section 5.4
5. How to transform a mesh: section 5.8
6. How to define a mesh from a file: section 5.7
7. How to use geometrical domains: section 5.9

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 **_nnodes** key. To set the local mesh step on each vertex, you will have to use the **_hsteps** key. These parameters are optional and only one of them is to be used.

```
Triangle tri(_v1 = Point(0.,0.), _v2 = Point(1.,0.), _v3 = Point(0.,1.),
  _nnodes = Numbers(11,15,11));
```

  What is the difference between **_nnodes** and **_hsteps** ? It is as in the GMSH documentation.

  _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.

  _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 :

  _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).

  _side_names is used to set the names of every side domain. You can give a vector of strings (`Strings` object) or a single `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 **_type**, for geometries fitted to the subdivision mesh generator (See subsection 5.4.2 for details).

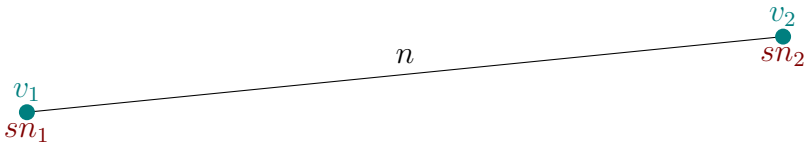
Let's summarize information about these keys:

key	authorized types	examples
_domain_name	`String` or const char*	**_domain_name**="Omega"
_hsteps	single real value, std::vector of real values or `Reals`	**_hsteps**=0.5, **_hsteps**=`Reals`(0.5, 0.2)
_nnodes	single (unsigned) integer value, std::vector of integer values, `Number` or `Numbers`	**_nnodes**=11, **_hsteps**=`Numbers`(11, 22)
_side_names	single string, std::vector of string, `String` or `Strings`	**_side_names**="Gamma", **_side_names**=`Strings`("Gam1", "Gam2", "Gam2")
_type	single (unsigned) integer value, or `Number`	**_type**=1

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.



The general case is to give points through parameters **_v1** and **_v2**, but when 1D, you can give directly the real coordinate.

```
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 $s3$ and $s4$ are identical. A better comprehensive way for s4 is to use parameters **_xmin** and **_xmax** instead of **_v1** and **_v2**.

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

In previous examples, you can notice that **_nnodes** take only a single integer value and **_hsteps** can take one real value or a vector of 2 real values (`Reals` object).
One of the combination **_xmin** and **_xmax** or **_v1** and **_v2** is needed.
After these arguments, you can give names of main domain and side domains as explained in preamble of this section.

Examples.

```
// 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:

```
Segment s1(_xmin=-2, _xmax=5, _nnodes=50);
s1.reverse(); // s1 is modified
Segment s2=~s1; // s1 is not modified
```

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

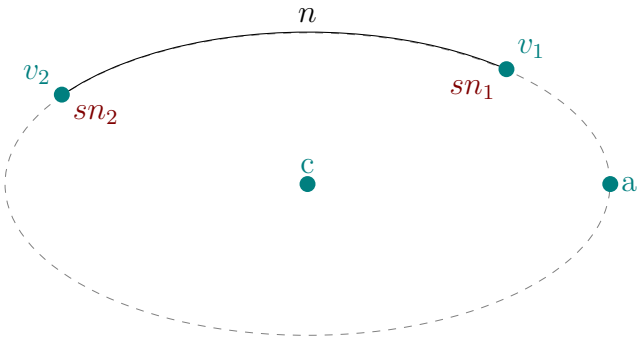
Let's summarize information about geometrical keys for segments:

key(s)	authorized types	examples
_v1, _v2	single integer or real value, or `Point`	**_v1**=`Point`(0.), **_v2**=`Point`(0.,0.), **_v1**=`Point`(0.,0.,0.), **_v2**=0.
_xmin, _xmax	single integer or real value	**_xmin**=1, **_xmax**=-2.5

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 : **_center**, **_apogee**, **_v1** and **_v2**. 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.
_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.

```
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 $\sim$ operator

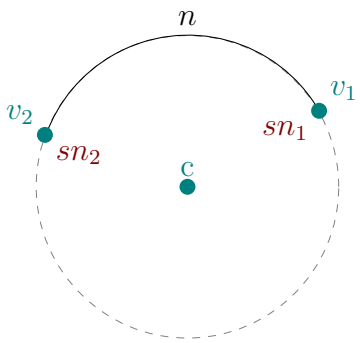
Let's summarize information about geometrical keys on elliptic arcs:

key(s)	authorized types	examples
_apogee, _center, _v1, _v2	`Point`	**_center**=`Point`(0.,0.), **_apogee**=`Point`(0.,0.,0.)

 Elliptic arcs cannot be defined if the angular sector is greater than π. This is a GMSH restriction !

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.

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

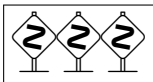
You can reverse the orientation of a circular arc by using one of the following:

```
CircArc c1(_center=Point(0.,0.), _v1=Point(1.,0.), _v2=Point(0.,1.),
    _nnodes=30, _domain_name="Omega");
c1.reverse(); // c1 is modified
CircArc c2=~c1; // c1 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 circular arcs:

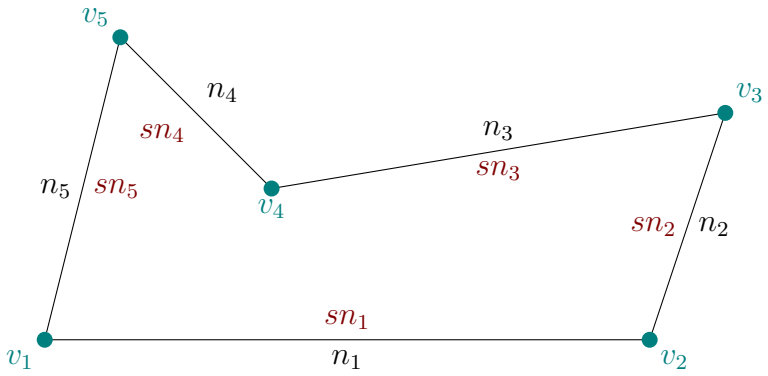
key(s)	authorized types	examples
_center, _v1, _v2	`Point`	**_v2**=`Point`(0.,0.), **_center**=`Point`(0.,0.,0.)

 Circular arcs cannot be defined if the angular sector is greater than π. This is a GMSH restriction !

5.1.3 Polygons and polygon-likes

Polygons

A polygon is defined by its ordered list of vertices.



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.

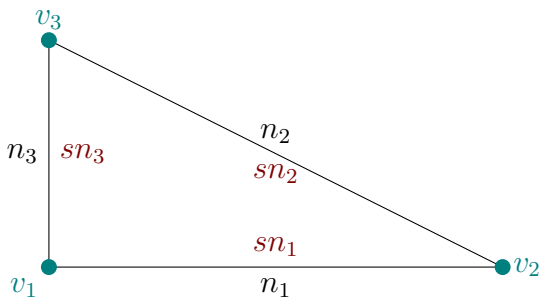
```
std::vector<Point> p(5);
p[0]=Point(0.,0.);
p[1]=Point(8.,0.);
p[2]=Point(9.,4.);
p[3]=Point(5.,2.);
p[4]=Point(1.,4.);
Polygon poly1(_vertices=p, _nnodes=Numbers(15, 10, 8, 8, 10),
    _domain_name="Omega", _side_names="Sigma");
```

Let's summarize information about geometrical keys on polygons:

key	authorized types	examples
_vertices	vector of `Point`	(std::vector<`Point`> vp; ...) **_vertices**=vp

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.

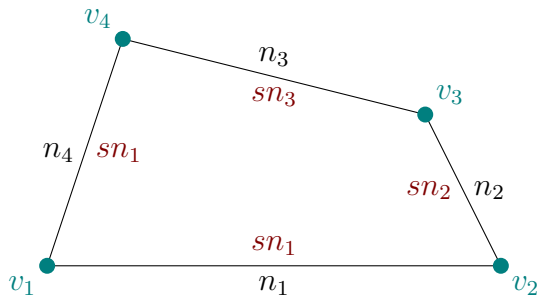
```
Point a(-1.,2.,0.);
Point b(1.,-4.,2.);
Point c(2.,3.,1.);
Triangle t1(_v1=a, _v2=b, _v3=c, _nnodes=Numbers(10,15,20),
    _domain_name="Omega", _side_names="Gamma");
```

Let's summarize information about geometrical keys on triangles:

key(s)	authorized types	examples
_v1, _v2, _v3	`Point`	**_v1**=`Point`(0.,0.), **_v2**=`Point`(0.,0.,0.)

Quadrangles

To define a quadrangle, you give the 4 vertices.



There is a parameter for each of them: **_v1**, **_v2**, **_v3** and **_v4**. These parameters take 2D or 3D points.
_nnodes can take one single value or a vector of 4 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.

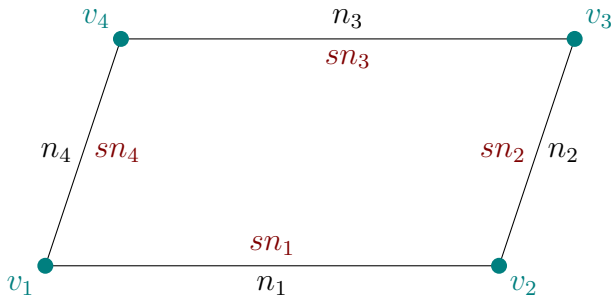
```
Quadrangle q1(_v1=Point(0.,0.), _v2=Point(2.,0.), _v3=Point(2.,1.),
    _v4=Point(0.,1.), _nnodes=Numbers(20, 10, 20, 10), _domain_name="Omega",
    _side_names="Gamma");
```

Let's summarize information about geometrical keys on quadrangles:

key(s)	authorized types	examples
_v1, _v2, _v3, _v4	`Point`	**_v1**=`Point`(0.,0.), **_v4**=`Point`(0.,0.,0.)

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 (`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.

Examples.

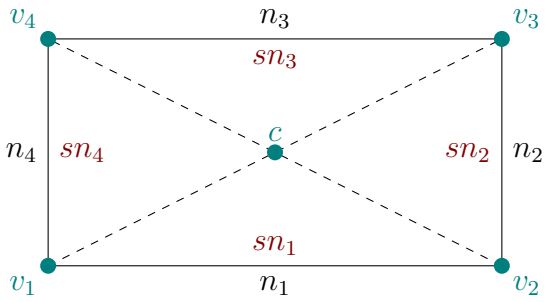
```
Parallelogram p1(_v1=Point(0.,0.), _v2=Point(2.,0.), _v4=Point(0.,1.),
    _nnodes=Numbers(20, 10, 20, 10), _domain_name="Omega",
    _side_names="Gamma");
Parallelogram p2(_v1=Point(0.,0.), _v2=Point(2.,0.), _v4=Point(0.,1.),
    _nnodes=Numbers(20, 10), _domain_name="Omega", _side_names="Gamma");
```

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 paralellograms:

key(s)	authorized types	examples
_v1, _v2, _v4	`Point`	**_v1**=`Point`(0.,0.), **_v2**=`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 (`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.

Examples.

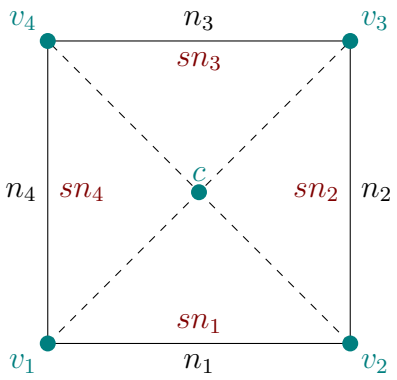
```
Rectangle r1(_v1=Point(0.,0.), _v2=Point(2.,0.), _v4=Point(0.,1.),
    _nnodes=Numbers(20, 10), _domain_name="Omega",
    _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));
Rectangle r2(_center=Point(1.,0.5), _xlength=2., _ylength=1.,
    _nnodes=Numbers(20, 10), _domain_name="Omega",
    _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:

key(s)	authorized types	examples
_center, _origin	`Point`	**_origin**=`Point`(0.,0.), **_center**=`Point`(0.,0.,0.)
_v1, _v2, _v4	`Point`	**_v1**=`Point`(0.,0.), **_v4**=`Point`(0.,0.,0.)
_xlength, _ylength	single unsigned integer or real positive value	**_xlength**=1, **_ylength**=2.5
_xmin, _xmax, _ymin, _ymax	single integer or real value	**_xmin**=1, **_ymax**=-2.5

Squares

To define a square, you give 3 vertices, as for rectangles and parallelograms.



There is a parameter for each of them: **_v1**, **_v2**, and **_v4**, 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 in the figure) and its length or p_1 (recalled origin in this case) and its length. You may use **_center** and **_length** or **_origin** and **_length** to do so. **_origin** and **_center** take 2D or 3D points. **_length** takes one single positive value.
_nnodes can take one single value or a vector of 2 or 4 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.

Examples.

```
Square s1(_v1=Point(0.,1.), _v2=Point(1.,1.), _v4=Point(0.,2.),
    _nnodes=Numbers(20, 10), _domain_name="Omega",
    _side_names=Strings("Gamma1", "Gamma2", "Gamma1", "Gamma2"));
Square s2(_center=Point(0.5,1.5), _length=1., _nnodes=Numbers(20, 10),
    _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1",
    "Gamma2"));
Square s3(_origin=Point(0.,1.), _length=1., _nnodes=Numbers(20, 10),
    _domain_name="Omega", _side_names=Strings("Gamma1", "Gamma2", "Gamma1",
    "Gamma2"));
```

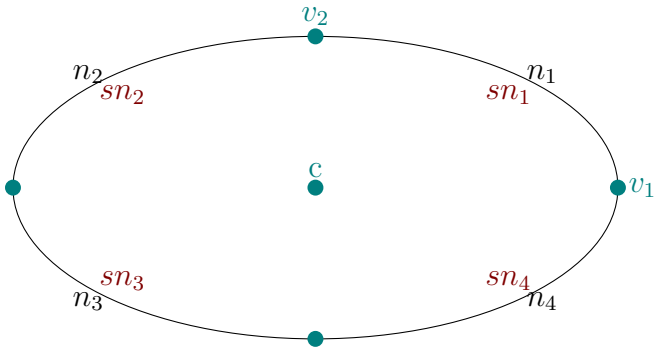
This is 3 definitions of the same `Square` object.
Let's summarize information about geometrical keys on squares:

key(s)	authorized types	examples
_center, _origin	`Point`	**_origin**=`Point`(0.,0.), **_center**=`Point`(0.,0.,0.)
_v1, _v2, _v4	`Point`	**_v2**=`Point`(0.,0.), **_v4**=`Point`(0.,0.,0.)
_length	single unsigned integer or real positive value	**_length**=1, **_length**=2.5

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 **_center**) and 2 other points, in order to have 3 unaligned points. These points are supposed to be both apogees of the ellipse (parameters **_v1** and **_v2**), namely c, p_1 and $p2$ 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.

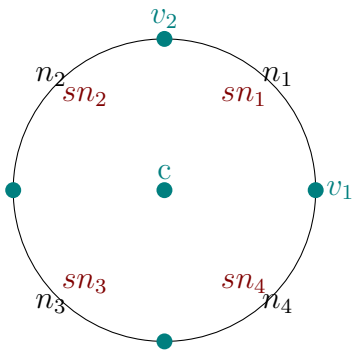
```
Ellipse e1(_center=Point(0.,0.), _v1=Point(2.,0.), _v2=Point(0.,1.),
    _nnodes=Numbers(5, 10, 5, 10), _domain_name="Omega",
    _side_names=Strings("Gamma5", "Gamma10", "Gamma5", "Gamma10"));
Ellipse e2(_center=Point(0.,0.,0.), _v1=Point(1.,0.,1.),
    _v2=Point(0.,1.,1.), _nnodes=40, _domain_name="Omega",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

Lets' summarize information about geometrical keys on ellipses:

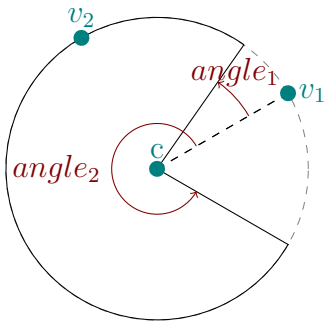
key(s)	authorized types	examples
_center, _v1, _v2	`Point`	**_center**=`Point`(0.,0.), **_v2**=`Point`(0.,0.,0.)
_xlength, _ylength	single unsigned integer or real positive value	**_xlength**=1, **_ylength**=2.5

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 **_center**, **_v1** and **_v2**, as for an ellipse. These parameters take 2D or 3D points.
Furthermore, you can define disk sectors with two additionnal parameters: **_angle1** and **_angle2**. Values of angles are given in degree and between 0 and 360.



_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.

```
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");
```


The `Disk` object has another name: `Circle`

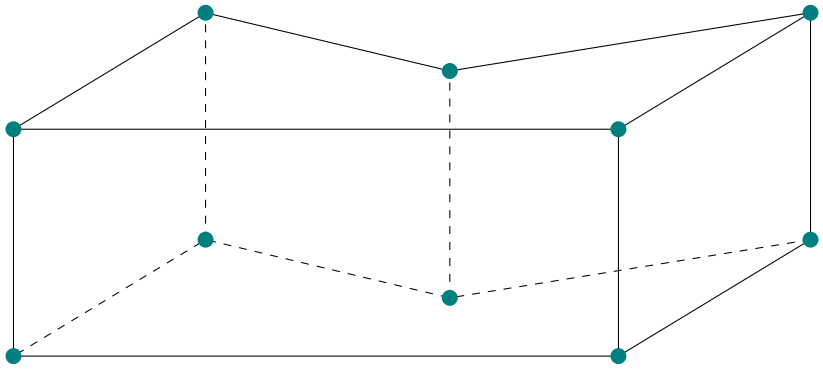
Let's summarize information about geometrical keys on disks:

key	authorized types	examples
_center, _v1, _v2	`Point`	**_center**=`Point`(0.,0.), **_v1**=`Point`(0.,0.,0.)
_radius, _angle1, _angle2	single unsigned integer or real positive value	**_radius**=1, **_angle1**=247.5

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.

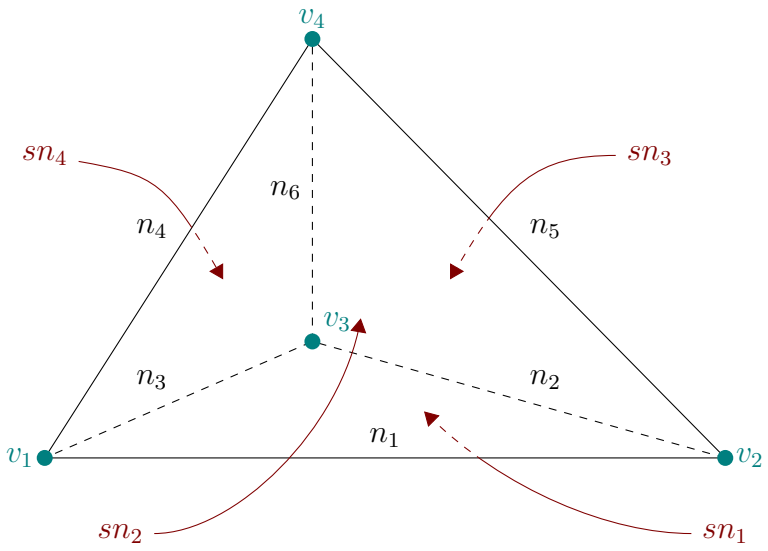
```
std::vector<Point> v(5), v2(4);
v[0]=Point(0.,0.,0.); v[1]=Point(2.,0.,0.); v[2]=Point(3.,1.,0.);
    v[3]=Point(1.,4.,0.); v[4]=Point(-1.,2.,0.);
Polygon pg1(_vertices=v, _nnodes="Gamma1");
v[0]=Point(0.,0.,1.); v[1]=Point(2.,0.,1.); v[2]=Point(3.,1.,1.);
    v[3]=Point(1.,4.,1.); v[4]=Point(-1.,2.,1.);
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.);
Polygon pg3(_vertices=v2, _nnodes="Sigma");
v2[0]=Point(2.,0.,0.); v2[1]=Point(3.,1.,0.); v2[2]=Point(3.,1.,1.);
    v2[3]=Point(2.,0., 1.);
Polygon pg4(_vertices=v2, _nnodes="Sigma");
v2[0]=Point(3.,1.,0.); v2[1]=Point(1.,4.,0.); v2[2]=Point(1.,4.,1.);
    v2[3]=Point(3.,1., 1.);
Polygon pg5(_vertices=v2, _nnodes="Sigma");
v2[0]=Point(1.,4.,0.); v2[1]=Point(-1.,2.,0.); v2[2]=Point(-1.,2.,1.);
    v2[3]=Point(1.,4., 1.);
Polygon pg6(_vertices=v2, _nnodes="Sigma");
v2[0]=Point(-1.,2.,0.); v2[1]=Point(0.,0.,0.); v2[2]=Point(0.,0.,1.);
    v2[3]=Point(-1.,2., 1.);
Polygon pg7(_vertices=v2, _nnodes="Sigma");
std::vector<Polygon> faces(7);
faces[0]=pg1; faces[1]=pg2; faces[2]=pg3; faces[3]=pg4; faces[4]=pg5;
    faces[5]=pg6; faces[6]=pg7;
Polyhedron poly1(_faces=faces, _domain_name="Omega");
```

Let's summarize information about geometrical keys on polyhedra:

key	authorized types	examples
_faces	vector of `Polygon`	(std::vector<`Polygon`> vp; ...) **_faces**=vp

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.

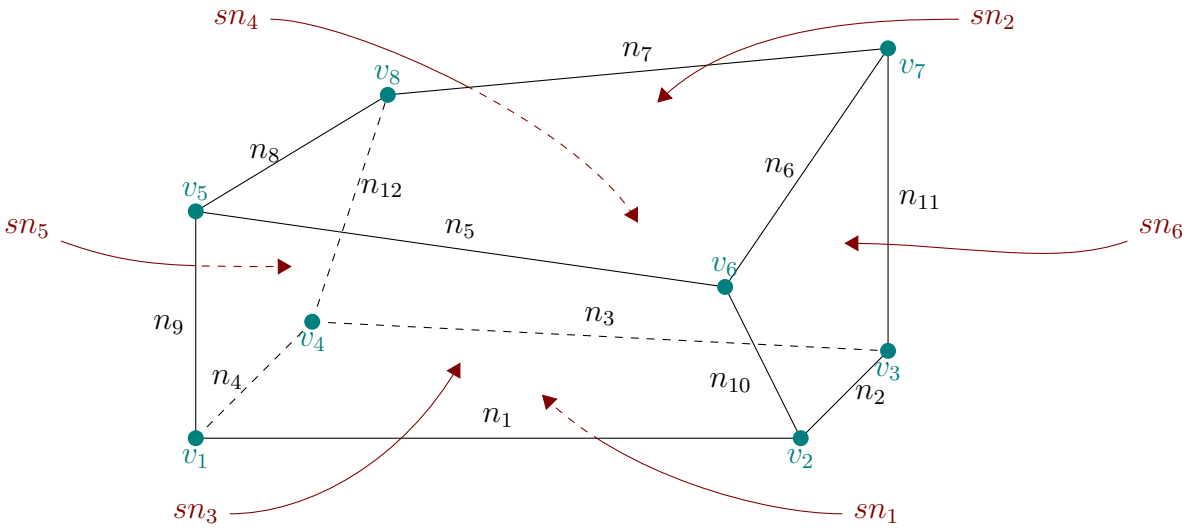
```
Point a(1.,0.,0.), b(0.,1.,0.), c(0.,0.,1.), d(0.,0.,0.);
Tetrahedron t1(_v1=a, _v2=b, _v3=c, _v4=d, _nnodes=10, _domain_name="Omega",
    _side_names="Gamma");
```

Let's summarize information about geometrical keys on tetrahedra:

key(s)	authorized types	examples
_v1, _v2, _v3, _v4	`Point`	**_v1**=`Point`(0.,0.,0.)

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 **_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.

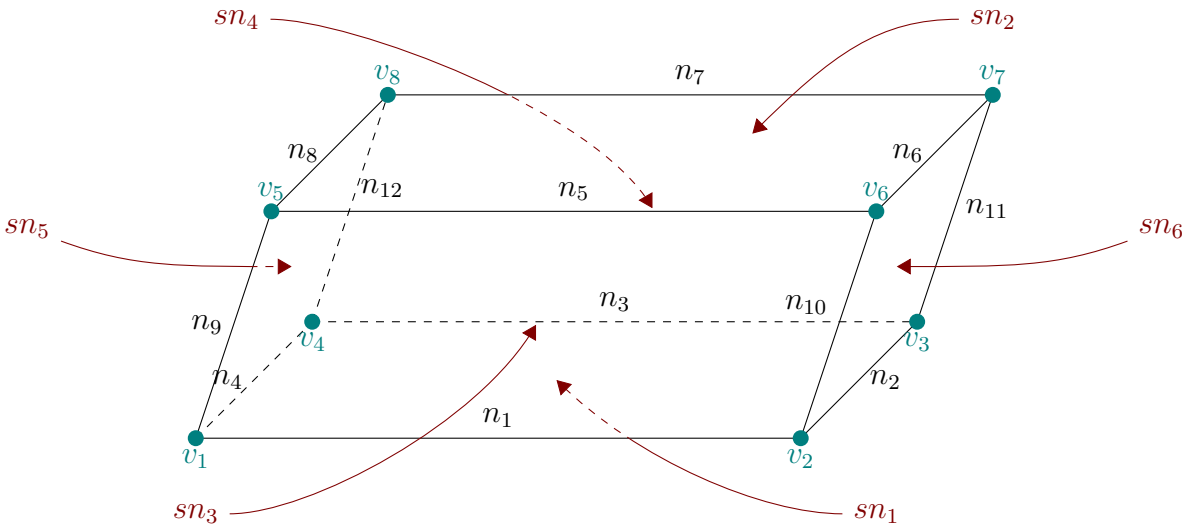
```
Point a(0.,0.,0.), b(4.,0.,0.), c(4.,2.,0.), d(0.,2.,0.);
Point aa(0.,0.,1.), bb(4.,0.,1.), cc(4.,2.,1.), dd(0.,2.,1.);
Hexahedron h1(_v1=a, _v2=b, _v3=c, _v4=d, _v5=aa, _v6=bb, _v7=cc, _v8=dd,
    _nnodes=Numbers(40, 20, 40, 20, 40, 20, 40, 20, 10, 10, 10, 10),
    _domain_name="Omega");
```

Let's summarize information about geometrical keys on hexahedra:

key(s)	authorized types	examples
_v1, _v2, _v3, _v4, _v5, _v6, _v7, _v8	`Point`	**_v4**=`Point`(0.,0.,0.)

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 :



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). **_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.

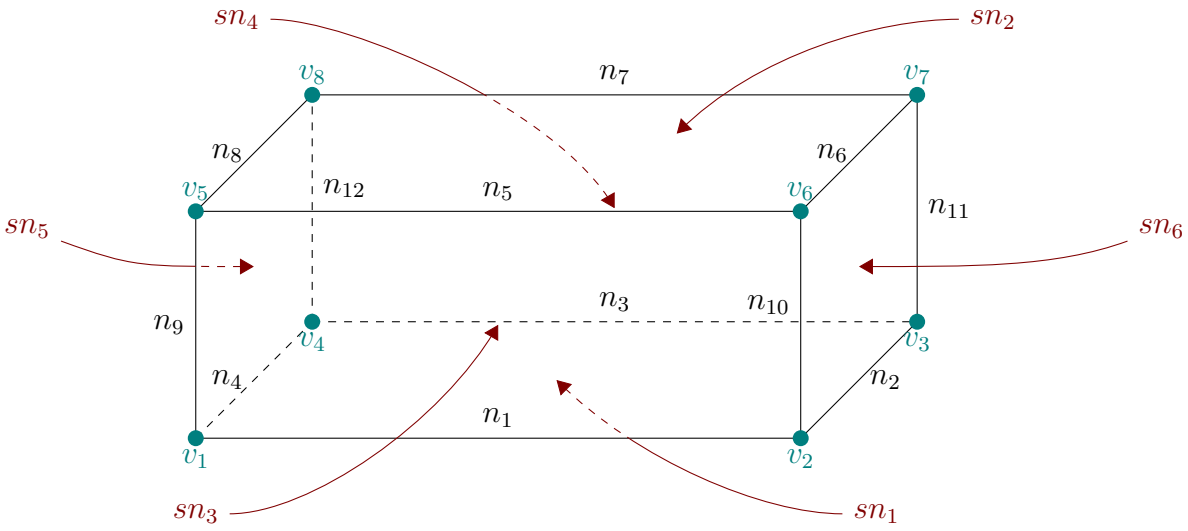
```
Point a(0.,0.,0.), b(4.,0.,0.), c(4.,2.,0.), d(0.,2.,0.);
Point aa(0.,0.,1.), bb(4.,0.,1.), cc(4.,2.,1.), dd(0.,2.,1.);
Parallelepiped p1(_v1=a, _v2=b, _v4=d, _v5=aa, _nnodes=Numbers(40, 20, 40,
    20, 40, 20, 40, 20, 10, 10, 10, 10), _domain_name="Omega");
Parallelepiped p2(_v1=a, _v2=b, _v4=d, _v5=aa, _nnodes=Numbers(40, 20, 10),
    _domain_name="Omega");
```

Both parallelepipeds of previous examples are identical. This explains the ability to give 3 values for **_nnodes**.
Let's summarize information about geometrical keys on parallelepipeds:

key(s)	authorized types	examples
_v1, _v2, _v4, _v5	`Point`	**_v5**=`Point`(0.,0.,0.)

Cuboids

To define a cuboid, you give 4 vertices, as for parallelepipeds.



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 p_1 (recalled origin in this case) and its lengths. You may use **_center**, **_xlength**, **_ylength** and **_zlength** or **_origin**, **_xlength**, **_ylength** and **_zlength** to do so. **_origin** and **_center** take points or a single value (in this case, it is like a 1D point). **_xlength**, **_ylength** and **_zlength** take one single positive value. There is another possibility : defining the rectangle by its bounds : parameters **_xmin**, **_xmax**, **_ymin**, **_ymax**, **_zmin** and **_zmax**. These parameters take one single value.
_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.

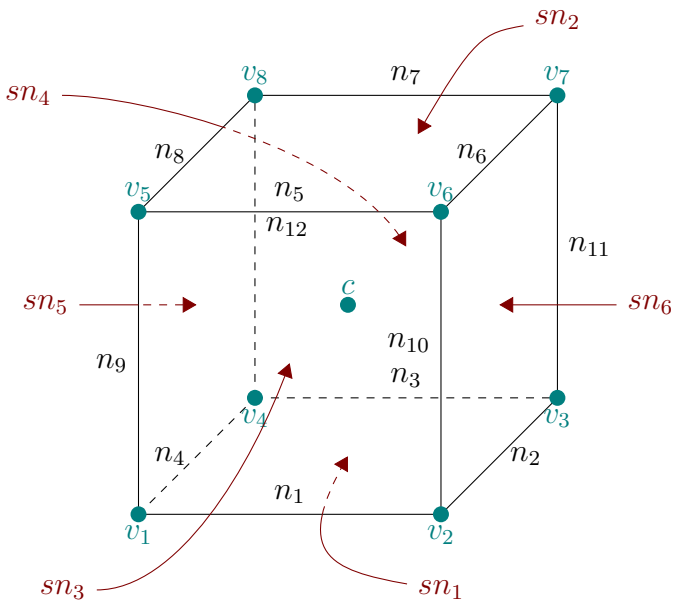
```
Cuboid c1(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v4=Point(0.,3.,0.),
    _v5=Point(0.,0.,4.), _nnodes=40, _domain_name="Omega");
Cuboid c2(_origin=Point(0.,0.,0.), _xlength=2., _ylength=3., _zlength=4,
    _nnodes=40, _domain_name="Omega");
Cuboid c3(_center=Point(1.,1.5.,2.),_xlength=2., _ylength=3., _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");
```

This is 4 definitions of the same `Cuboid` object.
Let's summarize information about geometrical keys on cuboids:

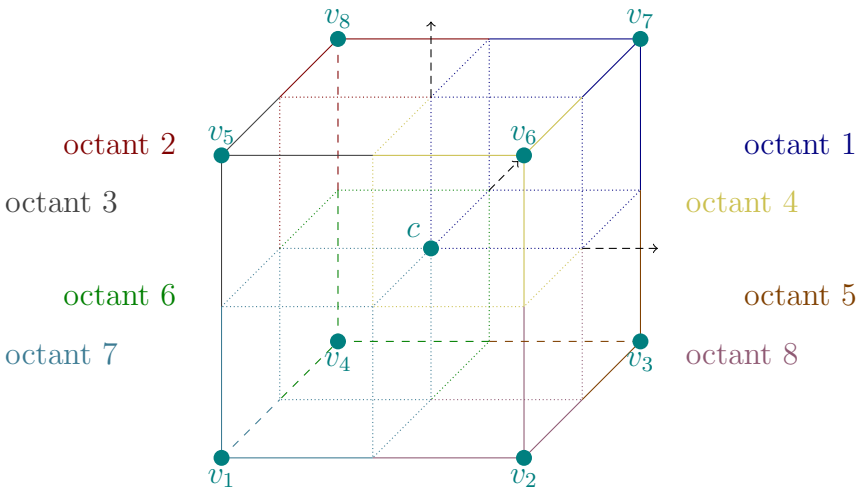
key(s)	authorized types	examples
_center, _origin	`Point`	**_center**=`Point`(0.,0.,0.)
_v1, _v2, _v4, _v5	`Point`	**_v1**=`Point`(0.,0.,0.)
_xlength, _ylength, _zlength	single unsigned integer or real positive value	**_xlength**=1, **_zlength**=2.5
_xmin, _xmax, _ymin, _ymax, _zmin, _zmax	single integer or real value	**_xmin**=1, **_zmin**=-2.5

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 p_1 (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:



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.

```
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");
```

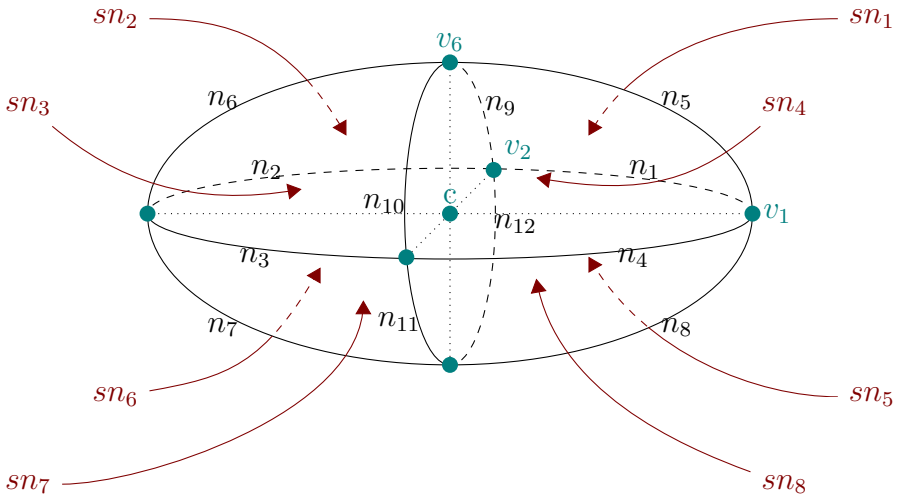
This is 3 definitions of the same `Cube` object.
Let's summarize information about geometrical keys on cubes:

key(s)	authorized types	examples
_center, _origin	`Point`	**_center**=`Point`(0.,0.,0.)
_v1, _v2, _v4, _v5	`Point`	**_v1**=`Point`(0.,0.,0.)
_length	single unsigned integer or real positive value	**_length**=1, **_length**=2.5
_nboctants	single unsigned integer value between 1 and 8	**_nboctants**=3

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.

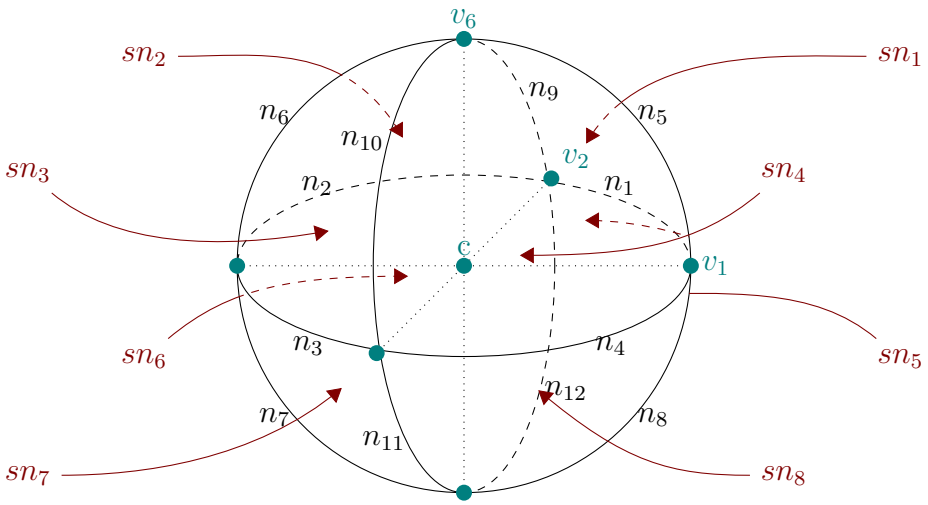
```
Ellipsoid e1(_center=Point(0.,0.,0.), _v1=Point(3.,0.,0.),
    _v2=Point(0.,2.,0.), _v6=Point(0.,0.,1.), _nnodes=Numbers(35, 30, 25),
    _domain_name="Omega1", _side_names="Gamma");
Ellipsoid e2(_center=Point(0.,0.,0.), _v1=Point(3.,0.,0.),
    _v2=Point(0.,2.,0.), _v6=Point(0.,0.,1.), _nnodes=Numbers(35, 35, 35, 35,
    30, 30, 30, 30, 25, 25, 25, 25), _domain_name="Omega1",
    _side_names="Gamma");
Ellipsoid e3(_center=Point(0.,0.,0.), _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:

key(s)	authorized types	examples
_center, _v1, _v2, _v6	`Point`	**_center**=`Point`(0.,0.,0.)
_xlength, _ylength, _zlength	single unsigned integer or real positive value	**_xlength**=1, **_zlength**=2.5

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:



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.

```
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");
Ball b3(_center=Point(0.,0.,0.), _radius=3, _nnodes=Numbers(35, 30, 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 **_nnodes**.


The `Ball` object has another name: `Sphere`

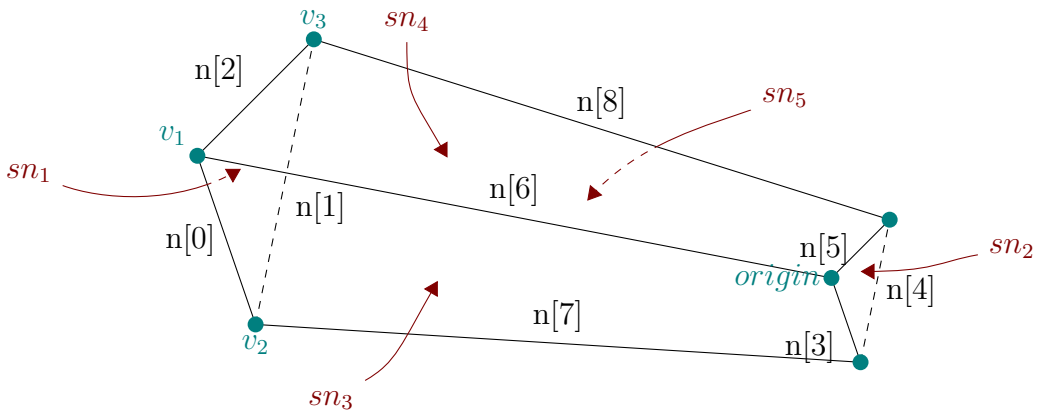
Let's summarize information about geometrical keys on balls:

key(s)	authorized types	examples
_center, _v1, _v2, _v6	`Point`	**_center**=`Point`(0.,0.,0.)
_radius	single unsigned integer or real positive value	**_radius**=1, **_radius**=2.5
_nboctants	single unsigned integer value between 0 and 8	**_nboctants**=3

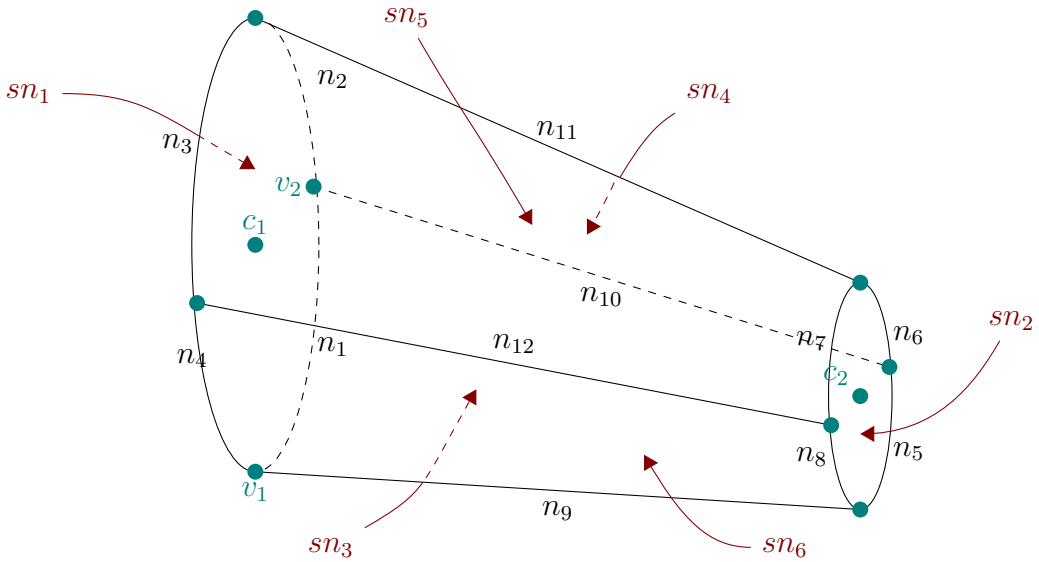
5.1.7 Trunks and trunk-likes

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.

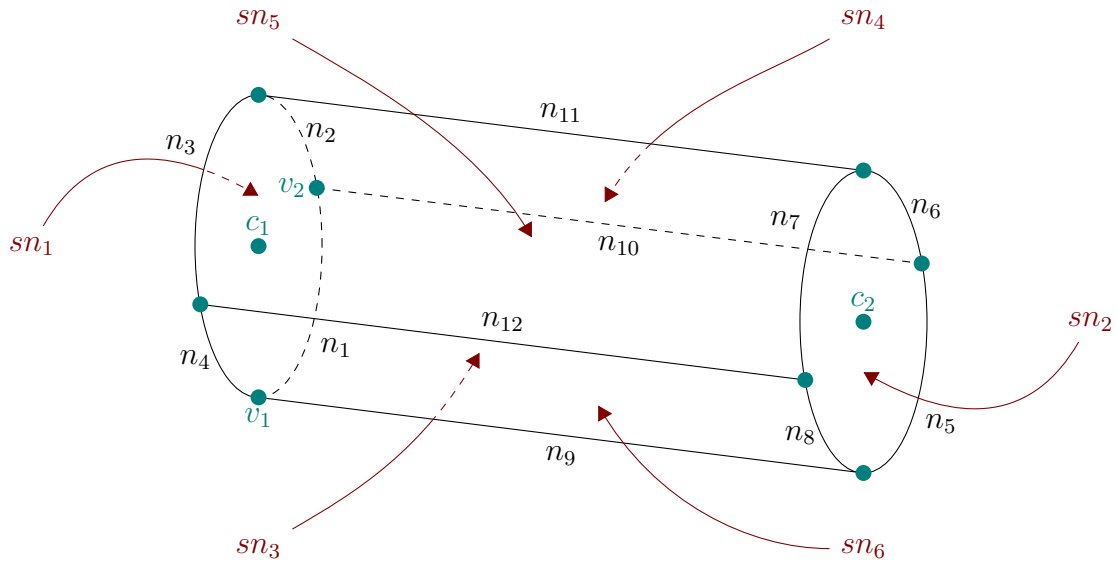
```
Trunk t1(_basis=Triangle(_v1=Point(0.,0.,0.), _v2=Point(3.,0.,0.),
    _v3=Point(0.,2.,0.)), _origin=Point(0.,2.,1.), _scale=0.5,
    _nnodes=Numbers(10, 10, 10, 5, 5, 5, 20, 20, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma"));
Trunk t2(_basis=Triangle(_v1=Point(0.,0.,0.),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

key(s)	authorized types	examples
_basis	`Polygon`, `Triangle`, `Quadrangle`, `Parallelogram`, `Rectangle`, `Square`, `Ellipse`, `Disk`	**_basis**=`Triangle`(...)
_origin	`Point`	**_origin**=`Point`(0.,0.,0.)
_scale	single unsigned integer or real positive value	**_scale**=2, **_scale**=0.5
_center1, **_center2**, **_v1**, **_v2**	`Point`	**_center1**=`Point`(0.,0.,0.)

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.
_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.

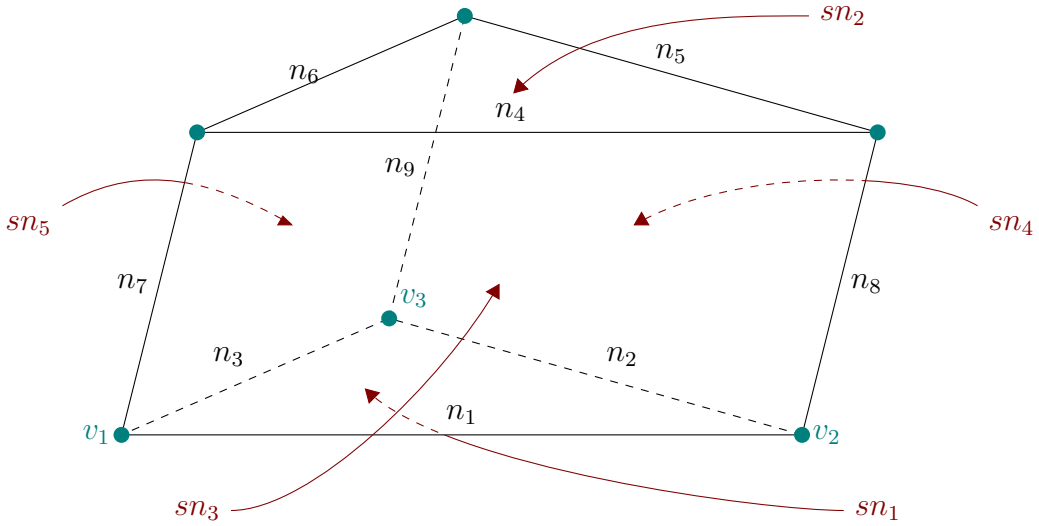
```
Cylinder c1(_basis=Disk(_center=Point(0.,0.,0.), _v1=Point(2.,0.,0.),
    _v2=Point(0.,2.,0.)), _direction=Point(0.,2.,1.), _nnodes=Numbers(10, 10,
    10, 10, 5, 5, 5, 5, 20, 20, 20, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma",
    "Sigma"));
Cylinder c2(_basis=Disk(_center=Point(0.,0.,0.), _v1=Point(2.,0.,0.),
    _v2=Point(0.,2.,0.)), _direction=Point(0.,2.,1.), _scale=0.5,
    _nnodes=Numbers(10, 5, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma",
    "Sigma"));
Cylinder c3(_center1=Point(0.,0.,0.), _v1=Point(2.,0.,0.),
    _v2=Point(0.,2.,0.)), _center2=Point(0.,2.,1.), _scale=0.5,
    _nnodes=Numbers(10, 5, 20), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma",
    "Sigma"));
```

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:

key(s)	authorized types	examples
_basis	Polygon, Triangle, Quadrangle, Parallelogram, Rectangle, Square, Ellipse, Disk	**_basis**=Triangle(...)
_direction	std::vector of real values, Reals or Point	**_direction**=Reals(0.,0.,1.), **_direction**=Point(0.,0.,1.)
_center1, _center2, _v1, _v2	Point	**_center1**=Point(0.,0.,0.)

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,
    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,
```

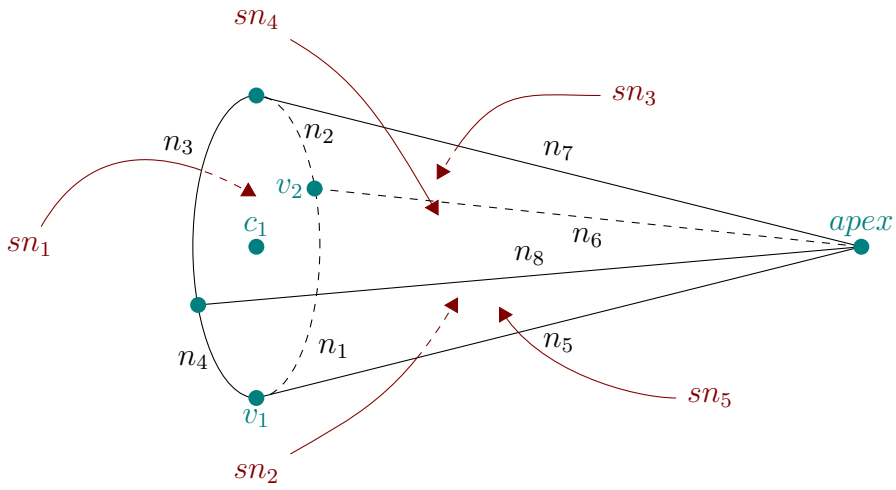
```
    20), _domain_name="Omega", _side_names=Strings("Gamma", "Gamma", "Sigma",
    "Sigma", "Sigma"));
Prism p3(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.), _v3=Point(0.,1.,0.),
    _direction=Reals(0.,2.,1.), _nnodes=Numbers(10, 5, 20),
    _domain_name="Omega", _side_names=Strings("Gamma", "Gamma",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

key(s)	authorized types	examples
_basis	`Polygon`, `Triangle`, `Quadrangle`, `Parallelogram`, `Rectangle`, `Square`	**_basis**=`Triangle`(...)
_direction	std::vector of real values, `Reals` or `Point`	**_direction**=`Reals`(0.,0.,1.), **_direction**=`Point`(0.,0.,1.)
_v1, _v2, _v3	`Point`	**_v2**=`Point`(0.,0.,0.)

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.

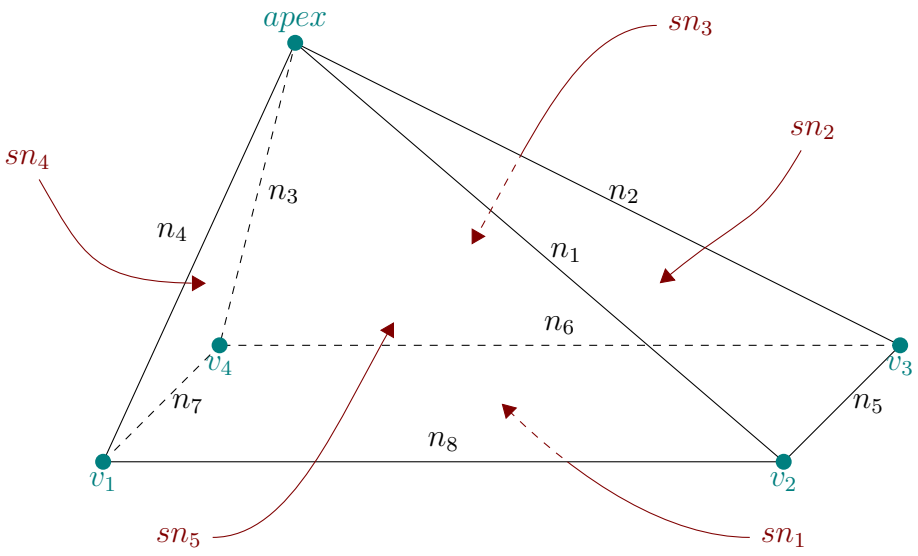
 Actually, this geometry cannot be meshed. Please use `Pyramid` for cones with polygonal basis, or `RevCone` for revolution cones.

Let's summarize information about geometrical keys on cones:

key(s)	authorized types	examples
_apex, _center1, _v1, _v2	`Point`	**_apex**=`Point`(0.,0.,0.)
_basis	`Polygon`, `Triangle`, `Quadrangle`, `Parallelogram`, `Rectangle`, `Square`, `Ellipse`, `Disk`	**_basis**=`Triangle`(...)

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. **_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.

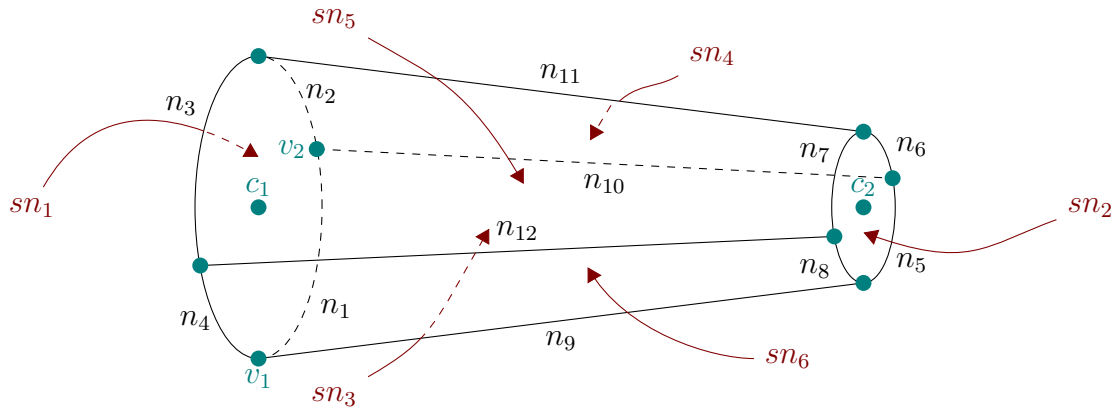
```
Pyramid p1(_basis=Quadrangle(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.),
    _v3=Point(1.,1.,0.), _v4=Point(-1.,2.,0.)), _apex=Point(0.,0.,1.),
    _nnodes=Numbers(20, 20, 20, 20, 10, 10, 10, 10), _domain_name="Omega",
    _side_names="Gamma");
Pyramid p2(_basis=Quadrangle(_v1=Point(0.,0.,0.), _v2=Point(2.,0.,0.),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

key(s)	authorized types	examples
_apex, _v1, _v2, _v3, _v4	`Point`	**_apex**=`Point`(0.,0.,0.)
_basis	`Polygon`, `Triangle`, `Quadrangle`, `Parallelogram`, `Rectangle`, `Square`	**_basis**=`Triangle`(...)

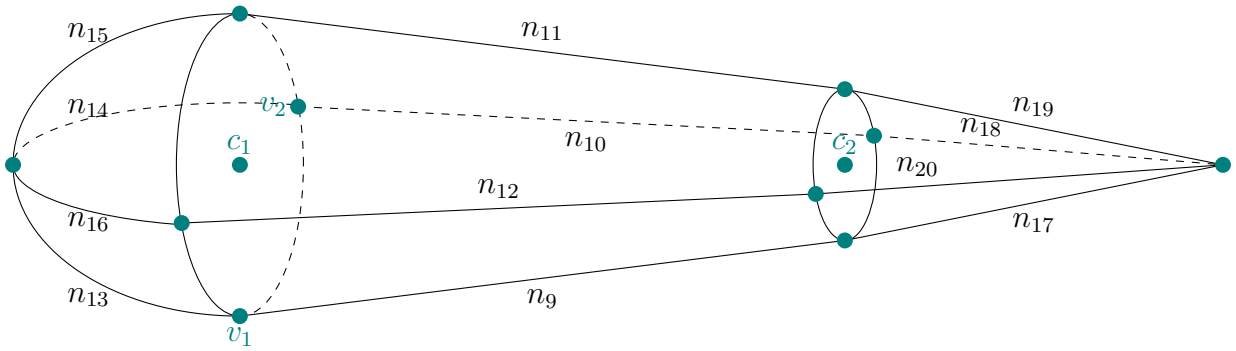
Revolution trunks

A revolution trunk is a right trunk with circular basis.



So, to define a revolution trunk, you just need to give centers and radiuses 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.

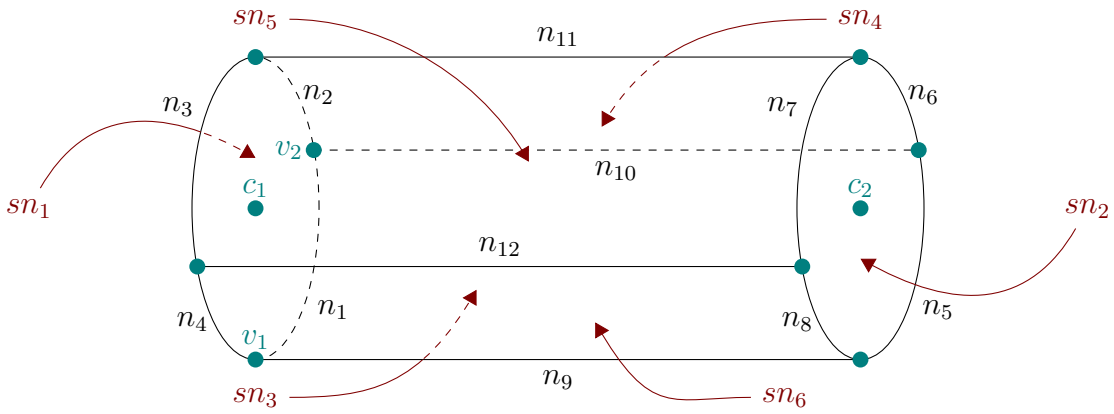


To do so, you will use parameters **_end1_shape** and **_end2_shape**, taking a `GeometricEndShape`, and **_end1_distance** and **_end2_distance**, taking one single positive value.
_nnodes can take one single value or a vector of 3 or n values (`Numbers` object), where n is 3 times the number of edges 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:

key(s)	authorized types	examples
_center1, _center2	`Point`	**_center2**=`Point`(0.,0.,0.)
_radius1, _radius2	single unsigned integer or real positive value	**_radius1**=1, **_radius2**=2.5
_end1_shape, _end2_shape	enum `GeometricEndShape`	**_end1_shape**=gesNone, **_end2_shape**=gesFlat, **_end2_shape**=gesCone, **_end1_shape**=gesEllipsoid, **_end2_shape**=gesSphere
_end1_distance, _end2_distance	single unsigned integer or real positive value	**_end1_distance**=1, **_end2_distance**=2.5
_nbsubdomains	single unsigned integer value	**_nbsubdomains**=2

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, **_nbsubdomains**, enabling you to slice the main cylinder (without its extensions) in as many domains as you want.

Examples.

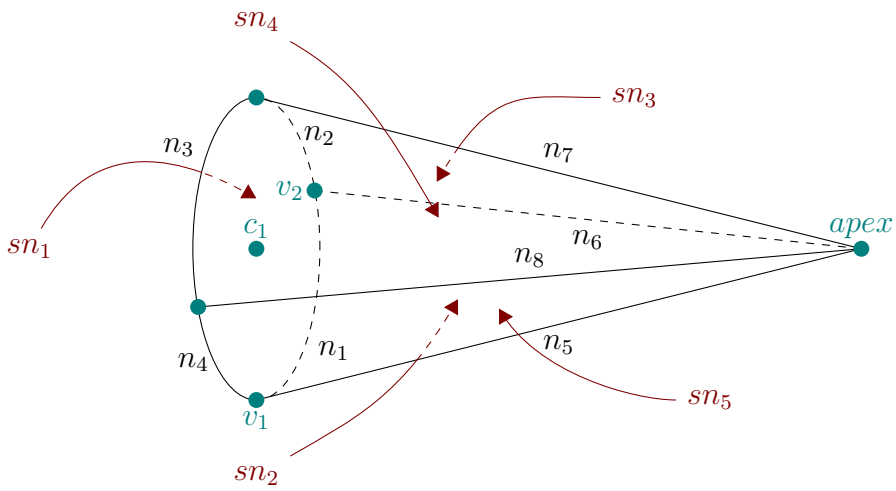
```
RevCylinder r1(\center1=Point(0.,0.,0.), _center2=Point(5.,0.,0.),
    _radius=1, _nnodes=Numbers(10, 10, 10, 10, 10, 10, 10, 10, 10, 50, 50,
    50, 50), _domain_name="Omega", _side_names=Strings("Gamma", "Gamma",
    "Sigma", "Sigma", "Sigma", "Sigma"));
RevCylinder r2(\center1=Point(0.,0.,0.), _center2=Point(5.,0.,0.),
    _radius=1, _nnodes=Numbers(10, 10, 50), _domain_name="Omega",
    _side_names=Strings("Gamma", "Gamma", "Sigma", "Sigma", "Sigma",
    "Sigma"));
```

Let's summarize information about geometrical keys on revolution cylinders:

key(s)	authorized types	examples
_center1, _center2	`Point`	**_center1**=`Point`(0.,0.,0.)
_radius	single unsigned integer or real positive value	**_radius**=1, **_radius**=2.5
_end1_shape, _end2_shape	enum `GeometricEndShape`	**_end1_shape**=gesNone, **_end2_shape**=gesFlat, **_end1_shape**=gesCone, **_end2_shape**=gesEllipsoid, **_end1_shape**=gesSphere
_end1_distance, _end2_distance	single unsigned integer or real positive value	**_end1_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 (`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, **_nbsubdomains**, enabling you to slice the main cone (without its extension) in as many domains as you want.
Let's summarize information about geometrical keys on revolution cones:

key(s)	authorized types	examples
_apex, _center	`Point`	**_apex**=`Point`(0.,0.,0.)
_radius	single unsigned integer or real positive value	**_radius**=1, **_radius**=2.5
_end_shape	enum `GeometricEndShape`	**_end_shape**=gesNone, **_end_shape**=gesFlat, **_end_shape**=gesCone, **_end_shape**=gesEllipsoid, **_end_shape**=gesSphere
_end_distance	single unsigned integer or real positive value	**_end_distance**=1, **_end_distance**=2.5
_nbsubdomains	single unsigned integer value	**_nbsubdomains**=2

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:

```
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 :

```
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");
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 s1, s2, s3 and s4 and circular arcs c1, c2, c3 and c4, in previous example, the following definitions are right :

```
Geometry g=planeSurfaceFrom(s2+c2+s3+c3+s4+c4+s1+c1, "Omega");
Geometry g=planeSurfaceFrom(s1+s2+s3+s4+c1+c2+c3+c4, "Omega");
```

The order of components here, and also the first component, has no meaning, but they all are oriented in the same way.

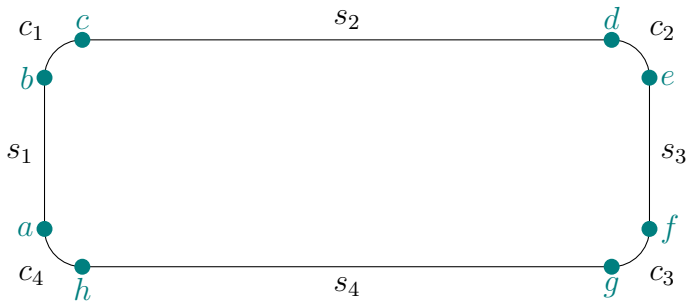


Figure 5.1: Rectangular geometry with rounded corners, defined with the `surfaceFrom` routine

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:

```
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);
```

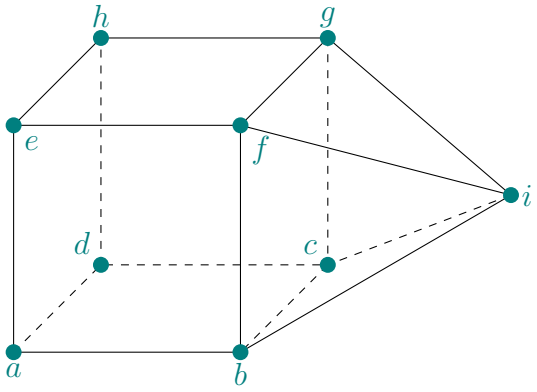
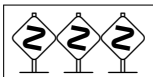


Figure 5.2: 3D geometry defined with the `volumeFrom` routine


3D loop geometries can be defined by a mix of 2D loop geometries and 2D canonical geometries.


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

```
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;
```

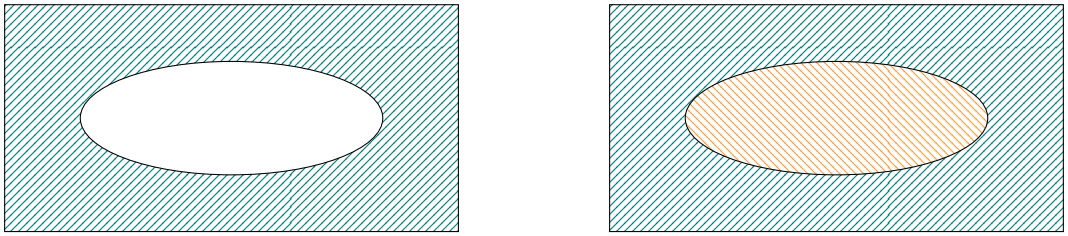


Figure 5.3: Composite geometry of an ellipse inside a rectangle

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 :

```
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); 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");
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=surfaceFrom(s5+c5+c6,"Omega6");
Geometry gmulti=(e1+sf1)-(e2+e3)+r2-sf2;
```

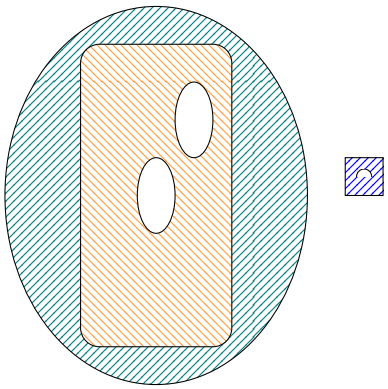


Figure 5.4: Composite geometry with multiple components and inclusions between components

 When at least 2 components share several vertices, several edges and/or several surfaces, everything works fine, shared geometrical entities are not duplicated.

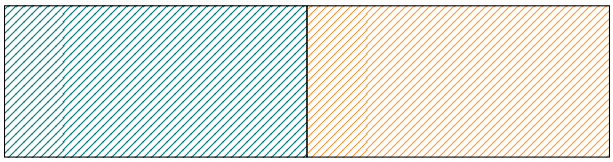


Figure 5.5: Composite geometry with edges shared by components.

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.

```
Geometry gmulti2=(e1+sf1)+(e2+e3)+r2+sf2;
```

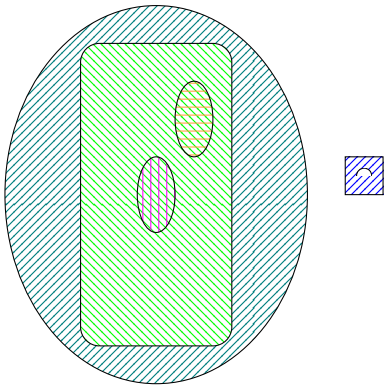


Figure 5.6: Composite geometry with multiple components and inclusions between components. Some inclusions are not detected correctly.

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 :

```
Geometry gmulti3=e1+(sf1+(+(e2+e3))+r2+sf2;
```

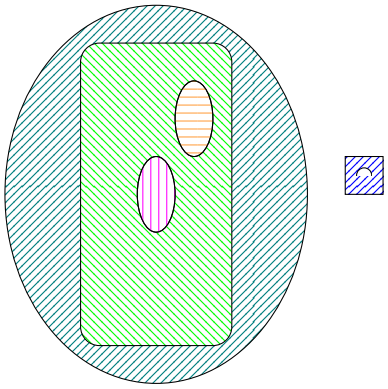
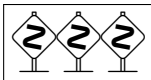


Figure 5.7: Composite geometry with multiple components and inclusions between components. Some inclusions are forced.

When you write (sf1+(+(e2+e3)), you tell explicitely 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.

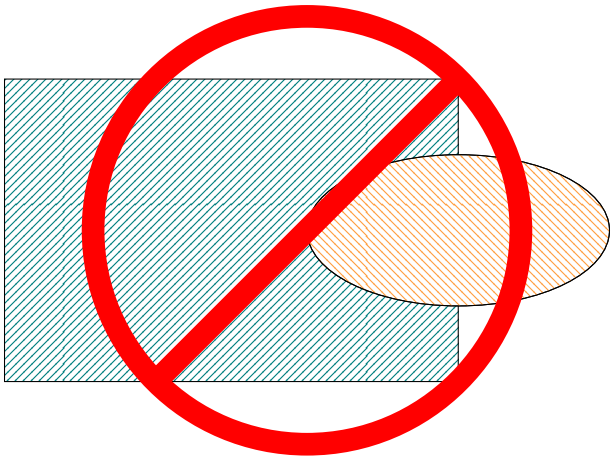


Figure 5.8: Partial inclusion is forbidden

5.2 Transformations on geometries

XLiFE++ 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 : $\left(\Omega, \vec{d}\right)$
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

$$\overrightarrow{AB} = \vec{u}$$

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

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

`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 Ω and of angle θ if and only if

$$\overrightarrow{\Omega B} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \overrightarrow{\Omega A}$$

A 2d rotation is defined by a point and an angle (in radians) :

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

`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 $\left(\Omega, \vec{d}\right)$ and of angle θ (in radians) if and only if

$$\overrightarrow{\Omega B} = \cos\theta\ \overrightarrow{\Omega A} + (1-\cos\theta)\ \overrightarrow{\Omega A} \cdot \vec{n} + \sin\theta\ \vec{n} \wedge \overrightarrow{\Omega A} \quad \text{(Rodrigues' rotation formulae)}$$

where $\vec{n} = \dfrac{\vec{u}}{||\vec{u}||}$ (the unitary direction).

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

```
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 Ω and of factor k if and only if

$$\overrightarrow{\Omega B} = k\ \overrightarrow{\Omega A}$$

```
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 Ω if and only if

$$\overrightarrow{\Omega B} = -\overrightarrow{\Omega A}$$

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

```
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 $\left(\Omega, \vec{d}\right)$ if and only if

$$\overrightarrow{AB} = 2\overrightarrow{AH} \quad \text{where } H \text{ is the orthogonal projection of } A \text{ on } \left(\Omega, \vec{d}\right)$$

```
Point c;
Vector<Real> d;
Real dx, dy;
Reflection2d r1(c, d), r2(c, dx, dy);
```


In the first syntax, `d` can be omitted. If so, its default value is the 2d zero vector and `c` can be omitted. If so, its default value is the 2d zero point.

3d reflections

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

$$\overrightarrow{AB} = 2\overrightarrow{AH} \quad \text{where } H \text{ is the orthogonal projection of } A \text{ on } [\Omega, \vec{n}]$$

```
Point c;
Vector<Real> n;
Real nx, ny, nz;
Reflection3d r1(c, n), r2(c, nx, ny, nz);
```


In the first syntax, `n` can be omitted. If so, its default value is the 3d zero vector and `c` can be omitted. If so, its default value is the 3d zero point.

5.2.2 Composition of transformations

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

```
Rotation2d r1(Point(0.,0.), 120.);
Reflection2d r2(Point(1.,-1.), 1.,2.5, -3.);
Translation t1(-1.,4.);
Homothety h (Point(-1.,0.), -3.2);
Transformation t = r1*h*r2*t1;
```

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

5.2.3 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 :

```
//! 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 angle);
Cube& Cube::rotate3d(Real ux, Real uy, Real uz, Real angle);
Cube& Cube::rotate3d(const Point& c, Real ux, Real uy, 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 reflection2d 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 reflection3d 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, Real ux, Real uy, Real angle);
Cube rotate3d(const Cube& m, Real ux, Real uy, Real uz, Real angle);
Cube rotate3d(const Cube& m, const Point& c, Real ux, Real uy, 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 reflection2d 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 reflection3d 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 Extrusion of geometries

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 :

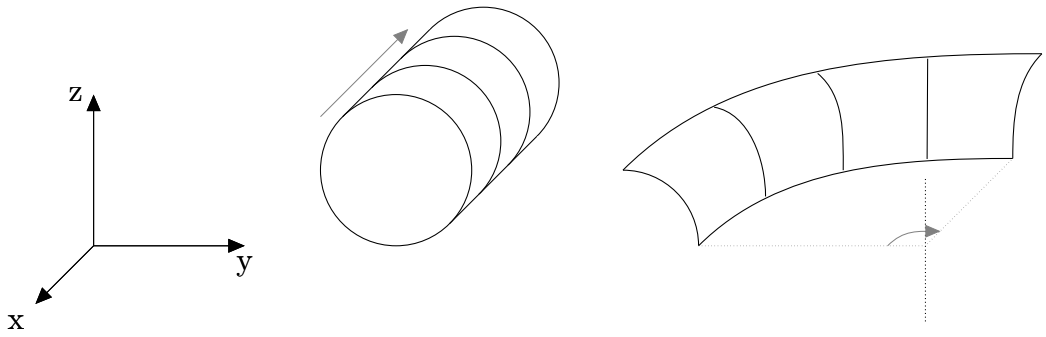


Figure 5.9: On the left, extrusion of a disk by a translation, with 3 layers. On the right, extrusion of a circular arc by rotation, with 4 layers

5.3.1 How to apply an extrusion ?

XLiFE++ 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`.

```
Point b(1.5,-4.,0.);
Point c(2.,-3.5,0.);
CircArc g(_center=Point(1.5,-3.5,0.), _v1=b, _v2=c, _nnodes=5,
    _domain_name="BC");
Geometry e2d=extrude(g, Translation(0.,0.,4.), 5, "Omega");
```

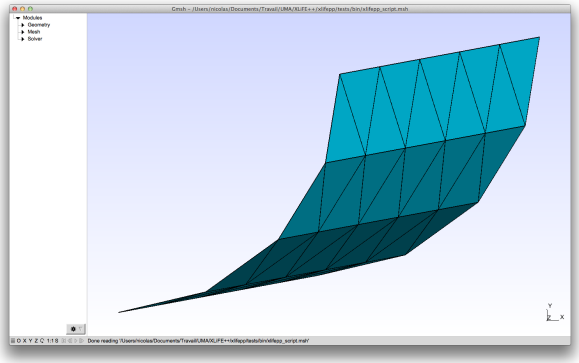


Figure 5.10: Extrusion of a circular arc by translation, with 5 layers

- A loop geometry (1D or 2D). Here, a rounded rectangle defines as in Figure 5.33

```
Point a(-1.5,-4.,0.); Point b(1.5,-4.,0.); Point c(2.,-3.5,0.); Point
    d(2.,3.5,0.);
Point e(1.5,4.,0.); Point f(-1.5,4.,0.); Point g(-2.,3.5,0.); Point
    h(-2.,-3.5,0.);
Segment s1(_v1=a, _v2=b, _nnodes=21, _domain_name="AB");
CircArc c1(_center=Point(3.5,0.5,0.), _v1=b, _v2=c, _nnodes=5,
    _domain_name="BC");
Segment s2(_v1=c, _v2=d, _nnodes=11, _domain_name="CD");
CircArc c2(_center=Point(3.5,1.5,0.), _v1=d, _v2=e, _nnodes=5,
    _domain_name="DE");
Segment s3(_v1=e, _v2=f, _nnodes=21, _domain_name="EF");
CircArc c3(_center=Point(0.5,1.5,0.), _v1=f, _v2=g, _nnodes=5,
    _domain_name="FG");
Segment s4(_v1=g, _v2=h, _nnodes=11, _domain_name="GH");
CircArc c4(_center=Point(0.5,0.5,0.), _v1=h, _v2=a, _nnodes=5,
    _domain_name="HA");
Geometry g=planeSurfaceFrom(s1+c1+s2+c2+s3+c3+s4+c4,"Omega");
Geometry e3d=extrude(g, Translation(0.,0.,4.), 10, "Omega");
```

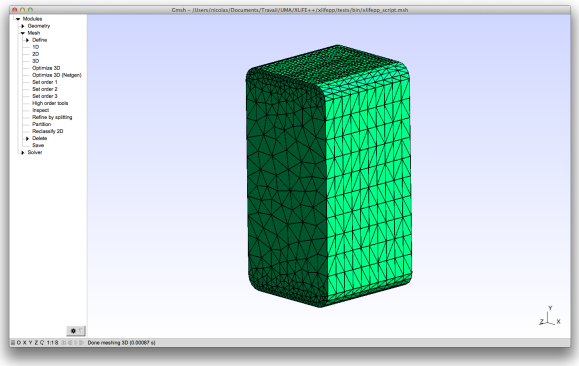


Figure 5.11: Extrusion of a rounded rectangle (loop geometry) by a rotation, with 10 layers

- 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

```
Ellipse e1(_center=Point(0.,0.,0.), _v1=Point(4,0.,0.), _v2=Point(0.,5.,0.),
    _nnodes=12, _domain_name="Omega1",
    _side_names=Strings("Gamma_1","Gamma_2","Gamma_3","Gamma_4"));
Ellipse e2(_center=Point(1.,2.,0.), _v1=Point(1.5,2.,0.),
    _v2=Point(1.,3.,0.), _nnodes=12, _domain_name="Omega3",
    _side_names=Strings("Gamma_9","Gamma_10","Gamma_11","Gamma_12"));
Geometry e3d2=extrude(e1-e2, Rotation3d(Point(5.,0.,0.), 0., 5., 0.,
    pi_/2.), 10, "Omega", "Gamma");
```

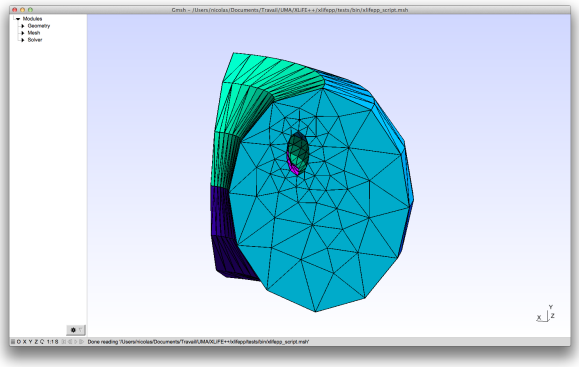


Figure 5.12: Extrusion of an ellipse with an elliptic hole by rotation, with 10 layers

5.3.2 How to define names of lateral domains of an extrusion ?

Instead of giving the same name to every lateral surface of an extrusion, you can give a name for each of them, but what about sides numbering ?
First example, let's take the extrusion of a `CircArc`:

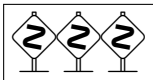
```
Point b(1.5,-4.,0.);
Point c(2.,-3.5,0.);
CircArc g(_center=Point(1.5,-3.5,0.), _v1=b1, _v2=c1, _nnodes=5,
    _domain_name="BC");
Geometry e2d=extrude(g, Translation(0.,0.,4.), 5, "Omega", Strings("Gamma1",
    "Gamma2");
```

In Figure 5.10, point b is the front below left corner and point c is the front top right corner. As g is defined from b to c, the first lateral side, corresponding to domain `Gamma1`, will be the edge below. If you had defined g from c to b, `Gamma1` would have correspond to the edge above.
Second example, let's take the extrusion of an ellipse with an elliptic hole:

```
Ellipse e1(_center=Point(0.,0.,0.), _v1=Point(4,0.,0.), _v2=Point(0.,5.,0.),
    _nnodes=12, _domain_name="Omega1");
Ellipse e2(_center=Point(1.,2.,0.), _v1=Point(1.5,2.,0.),
    _v2=Point(1.,3.,0.), _nnodes=12, _domain_name="Omega3");
Geometry e3d3=extrude(e1-e2, Rotation3d(Point(5.,0.,0.), 0., 5., 0.,
    pi_/2.), 10, "Omega", Strings("Gamma1", "Gamma2", "Gamma3", "Gamma4",
    "Gamma5", "Gamma6", "Gamma7", "Gamma8"));
```

This time lateral surfaces are ordered as follows:

- Lateral surfaces from the outer ellipse are ordered the same way as borders of the ellipse
- Lateral surfaces from the inner ellipse (and every hole in general) are ordered in the reverse order of borders of the ellipse

 Contrary to GMSH, you can extrude a geometry by rotation of angle greater than π, by splitting extrusion in 2 half extrusions when angle is not 2π or in 4 quarter extrusions when angle is 2π. As a result, the number of lateral surfaces is multiplicated by 2 or 4.

5.3.3 Example: definition of a conesphere

To define geometries based on cones, you always have to use extrusions. It is the case for the conesphere:

```
Real rb=1., hc=3.;
Real hs=rb*rb/hc;
Real rs=sqrt(rb*rb + hs*hs);

Point origin(0.,0.,0.), apex(0.,0.,hc), p1(rb,0.,0.), p2(0.,0.,-hs-rs);

Segment s1(_v1=p1, _v2=apex, _hsteps=0.05);
Segment s2(_v1=apex,_v2=origin, _hsteps=0.05);
Segment s3(_v1=origin, _v2=p2, _hsteps=0.05);
CircArc c1(_center=Point(0.,0.,-cssphereheight),_v1=p2,_v2=p1, _hsteps=0.05);
Disk d1(_center=0.5*p1, _v1=0.5*p1+Point(0.2*rb,0.,0.),
    _v2=0.5*p1+Point(0.,0.,0.2*rb), _domain_name="Sigma", _hsteps=0.05);

Geometry base=planeSurfaceFrom(s2+s3+c1+s1, "Gamma");
Geometry g=extrude(base,
    Rotation3d(Point(0.,0.,0.),0.,0.,1.,2.*pi_),"Omega1", Strings("Gamma1",
    "Gamma2", "Gamma3", "Gamma4", "Gamma5", "Gamma6", "Gamma7", "Gamma8"));
```

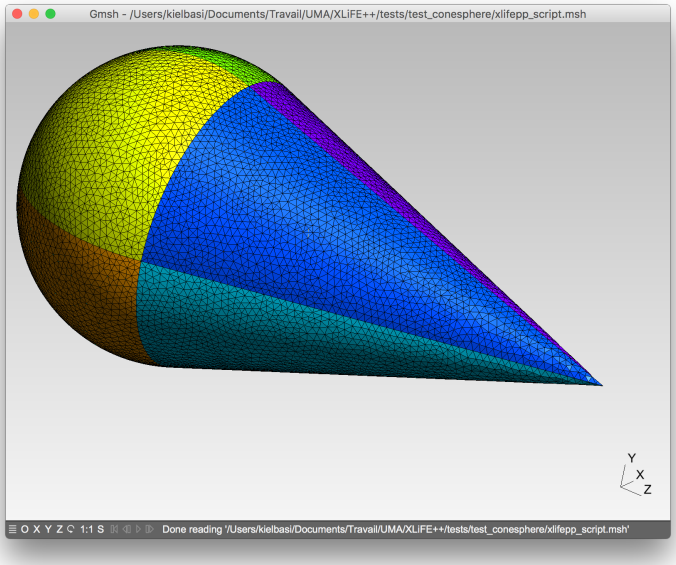


Figure 5.13: Mesh of a conesphere

5.4 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:

```
//! 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.

```
// 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 mentionned above. The corresponding constructor is defined as follows:

```
//! 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.

```
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.:

```
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.

5.4.1 Structured internal meshing tools: structured generator

When the **structured mesh generator** is chosen (mg = `_structured`), one can create a mesh of order 1:

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

One has to declare an object of type `Geometry`, more precisely of one of its derived type `Segment`, `Parallelogram`, `Rectangle`, `Square`, `Parallelepiped`, `Cuboid` or `Cube` using one of these constructors, that allow in particular to specifiy 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]x[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:
sideNames[0] is $[a,b] \times c$, sideNames[1] is $b \times [c,d]$,
sideNames[2] is $[a,b] \times d$, 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]x[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]x[1,3]x[1,5]");
```

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:
sideNames[0] is $[a,b] \times [c,d] \times e$, sideNames[1] is $[a,b] \times [c,d] \times f$,
sideNames[2] is $[a,b] \times c \times [e,f]$, sideNames[3] is $[a,b] \times d \times [e,f]$,
sideNames[4] is $a \times [c,d] \times [e,f]$, sideNames[5] is $b \times [c,d] \times [e,f]$.

5.4.2 Unstructured internal meshing tools: subdivision generator

When the **subdivision algorithm** is chosen (mg = `_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.:

```
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 5.14 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 *nboctants* varying from 1 to 8:

```
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);
```


The class `Sphere` could have been used instead of `Ball`, leading to the same result.

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 5.15.

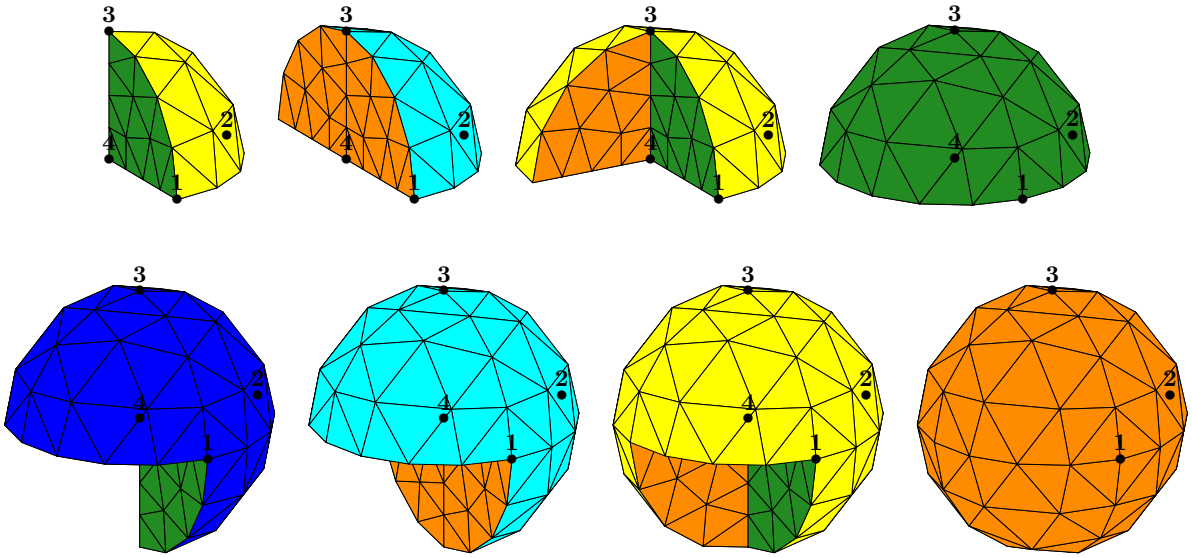


Figure 5.14: Volumic meshes of the different portions of the ball according to the number of octants.

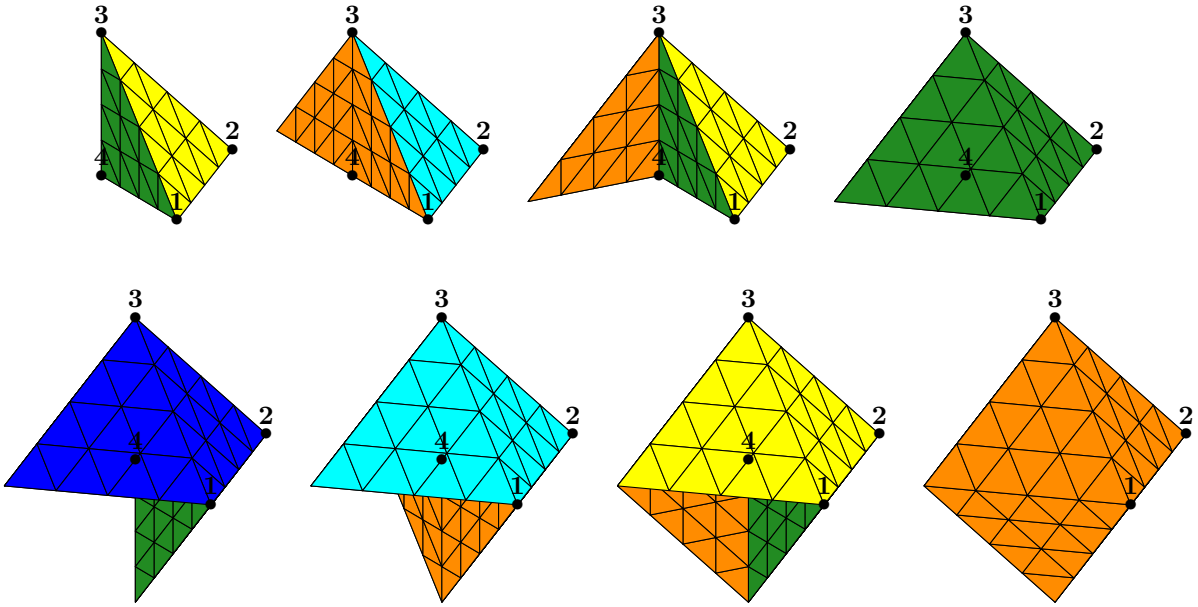


Figure 5.15: Volumic meshes of the different portions of the "flat ball" according to the number of octants.

If we specifically want the mesh inside the whole sphere, it can be usefull 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 5.16, which shows both the round and flat results of the algorithm:

```
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);
```

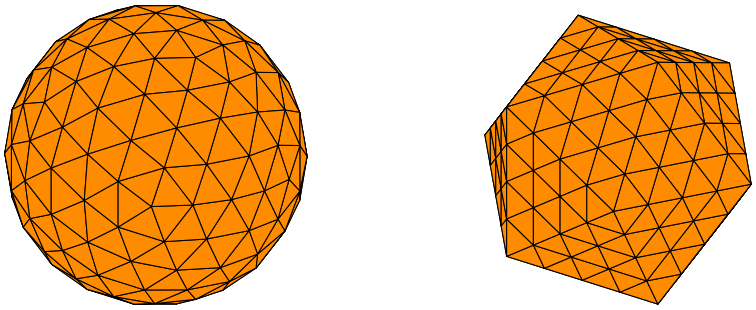


Figure 5.16: Volumic meshes starting from an icosahedron.

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 5.17, which is the result of the following code, with no subdivision and *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 5.17 in an unsual 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 5.17, 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.

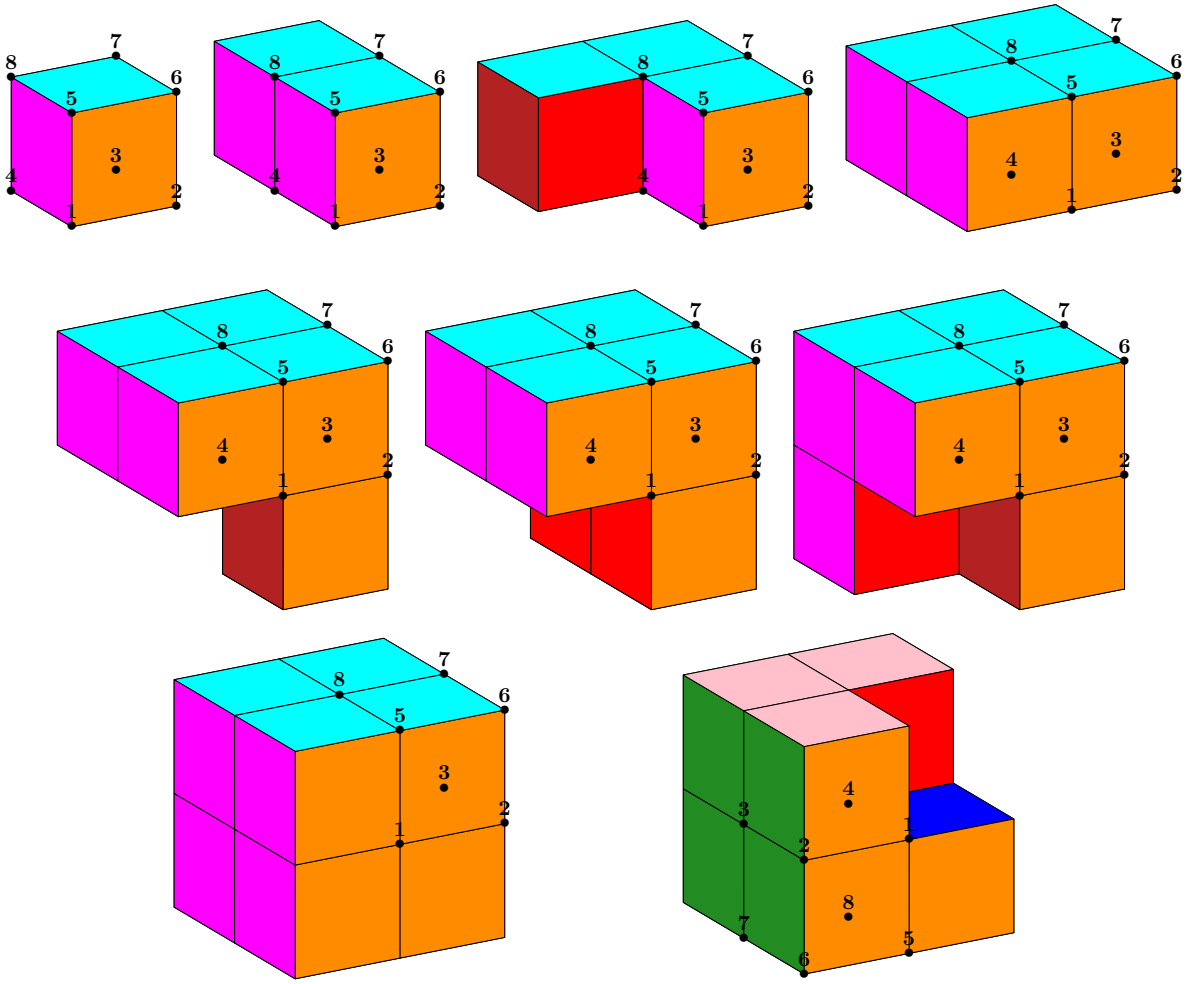


Figure 5.17: Volumic meshes of the different portions of the cube according to the number of octants.

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 5.18:

```
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);
```

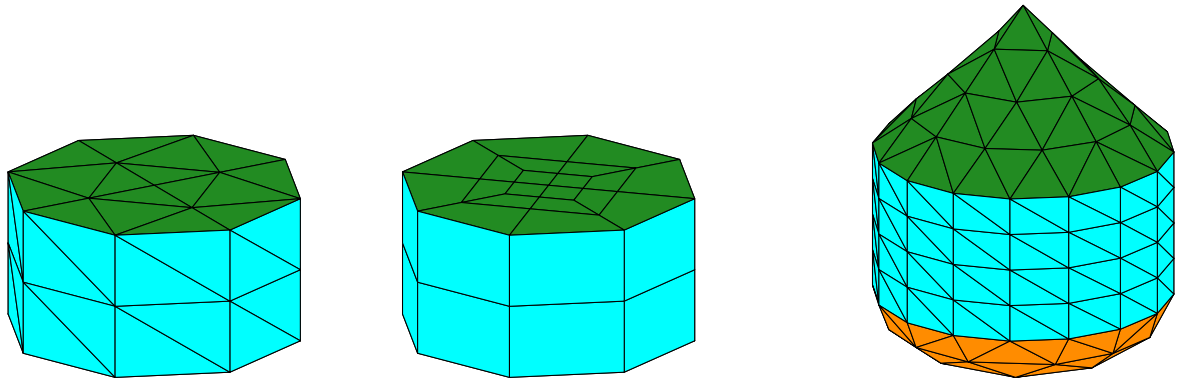


Figure 5.18: Volumic meshes of the “unit” cylinder with tetrahedra and hexahedra.

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 5.18 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 5.19:

```
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 trucated cone ; an example is shown on the middle drawing of the figure 5.19.

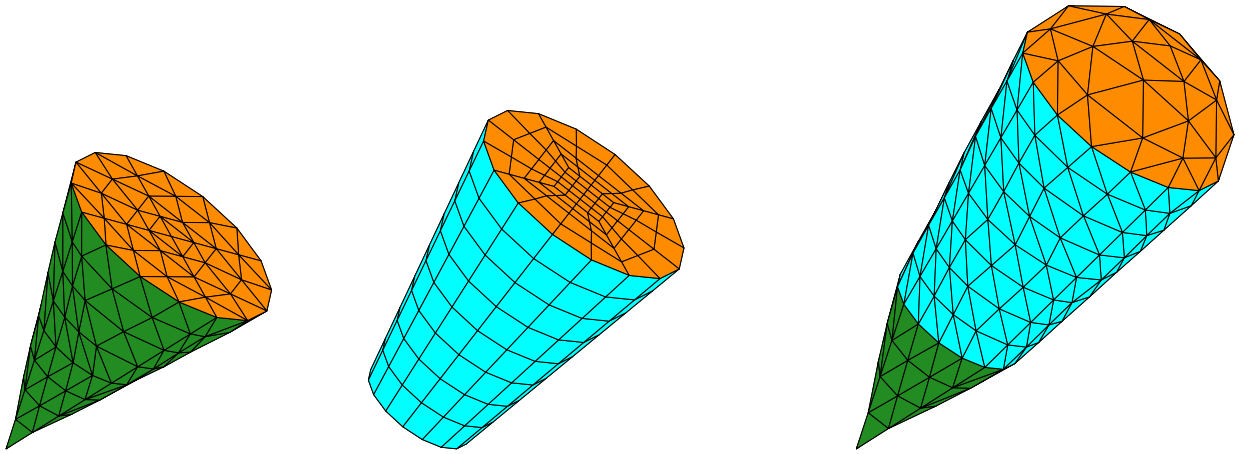


Figure 5.19: Volumic meshes of the cone and truncated cone with tetrahedra and hexahedra.

The following code produces the last mesh shown on figure 5.19:

```
nbpts=5;
radius1=0.6, radius2=1.;
RevTrunk cone1(_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(cone1, _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 5.20:

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

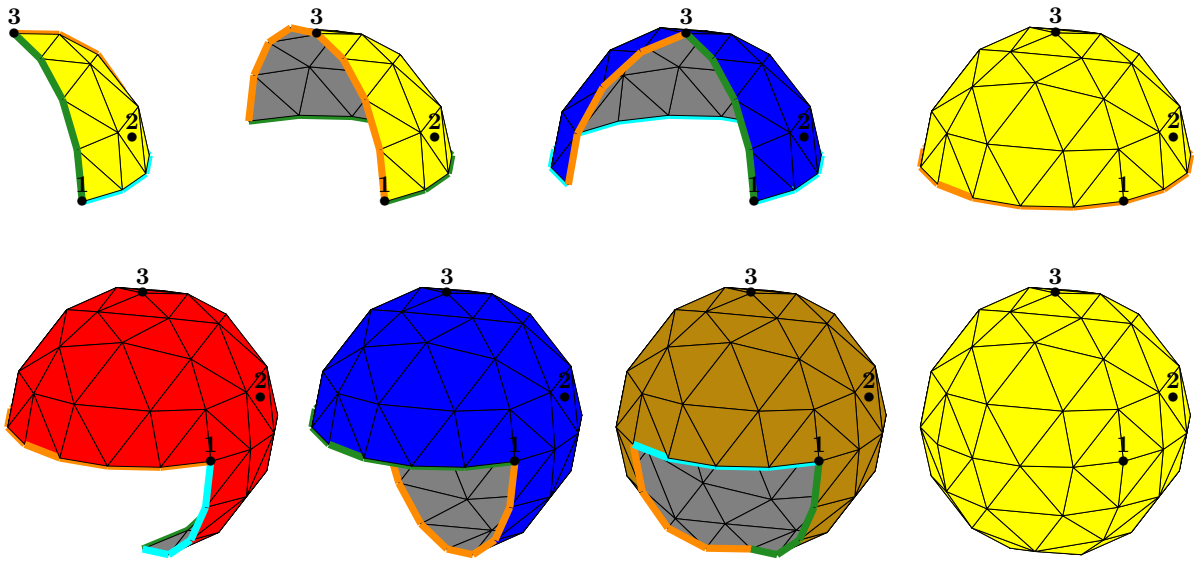


Figure 5.20: Surfacic meshes of the different portions of the boundary of the sphere according to the number of octants.

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 5.21).

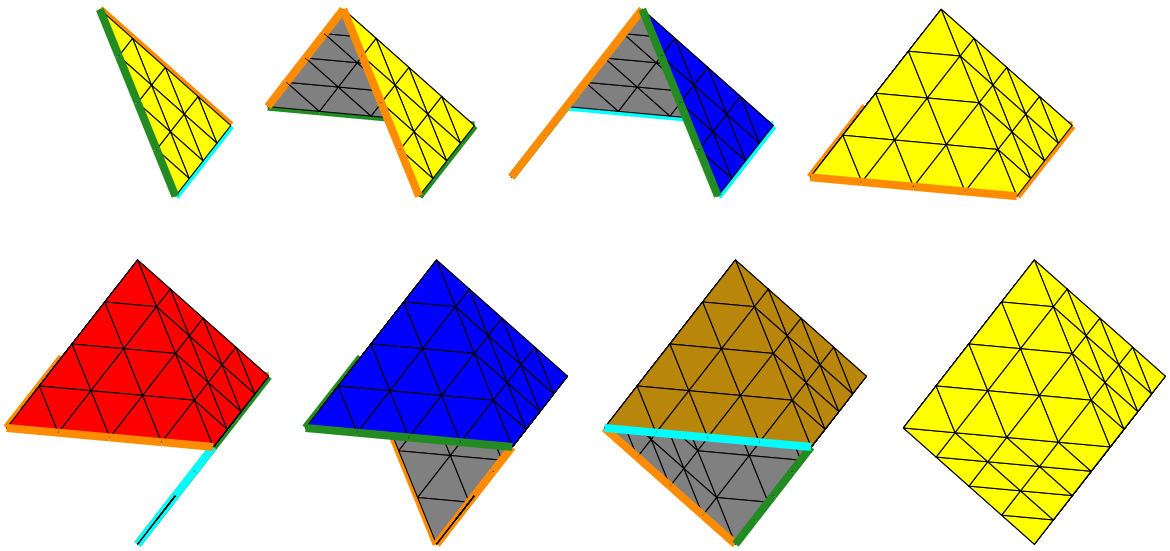


Figure 5.21: Surfacic meshes of the different portions of the "flat sphere" according to the number of octants.

If we specifically want the mesh of the whole sphere, it can be usefull 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 5.22, which shows both the round and flat results of the algorithm:

```
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);
```

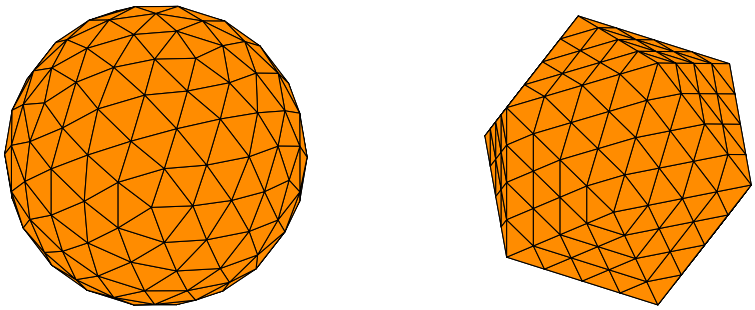


Figure 5.22: Surfacic meshes starting from an icosahedron.

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:

```
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 5.23 below.

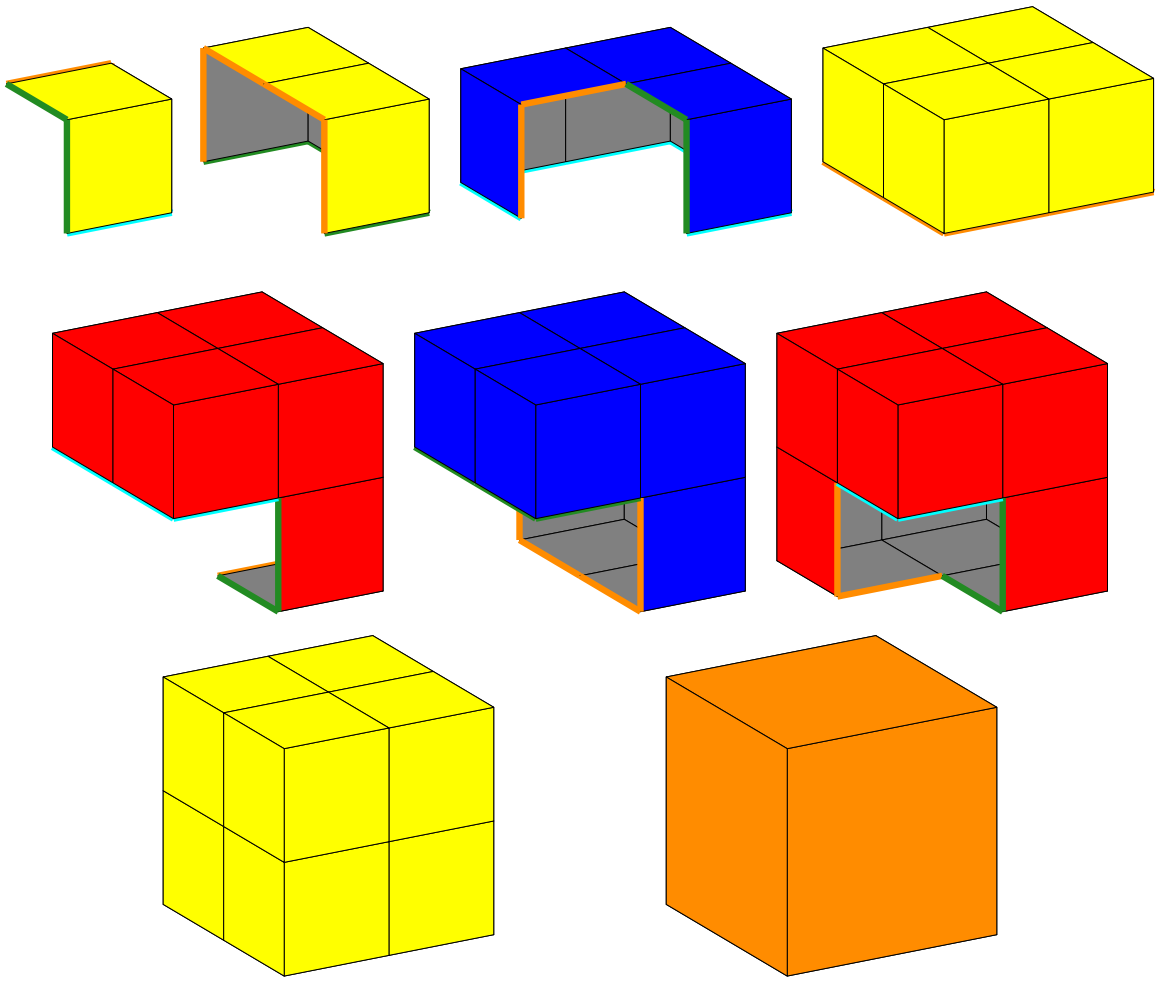


Figure 5.23: Surfacic meshes of the different portions of the cube according to the number of octants.

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 5.24:

```
radius=1.;
nbslices=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, _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);
```

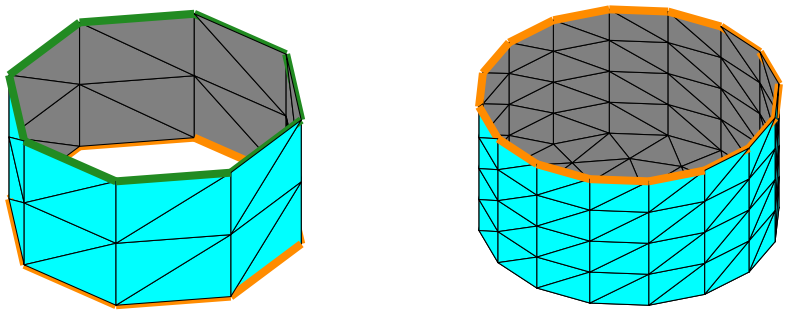


Figure 5.24: Surfacic meshes of a cylinder.

```
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);
```

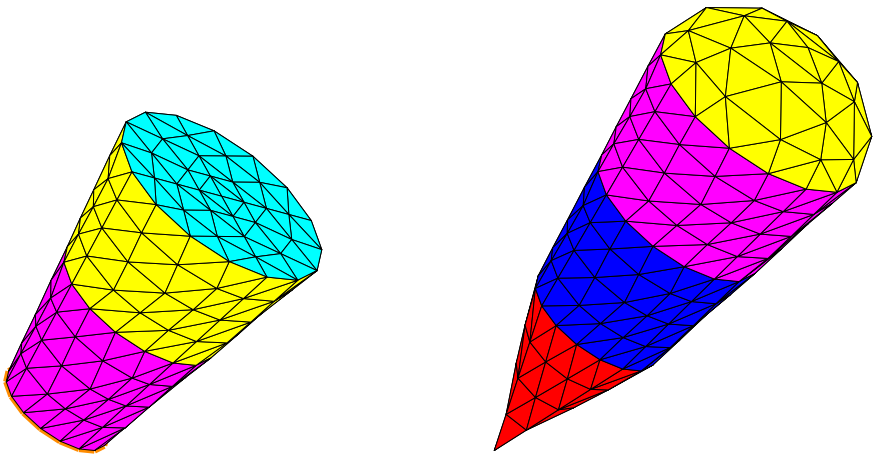


Figure 5.25: Surfacic meshes of a truncated cone.

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 5.26:

```
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 5.19).
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 objet 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)
```

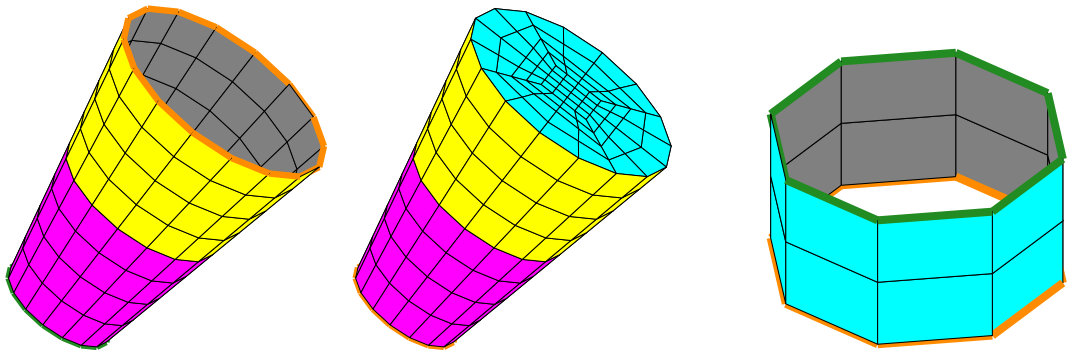


Figure 5.26: Surfacic meshes of a cone and a cylinder.

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 5.18. 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 5.27.

```
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);
```

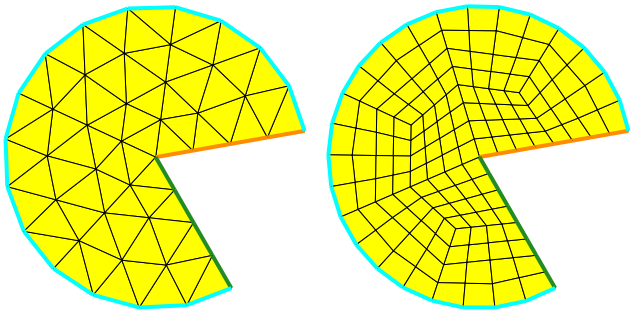


Figure 5.27: Meshes of a portion of disk with triangles and quadrangles.

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 5.28 shows two examples, in 3D with triangles, in 2D with quadrangles on a domain with a hole.

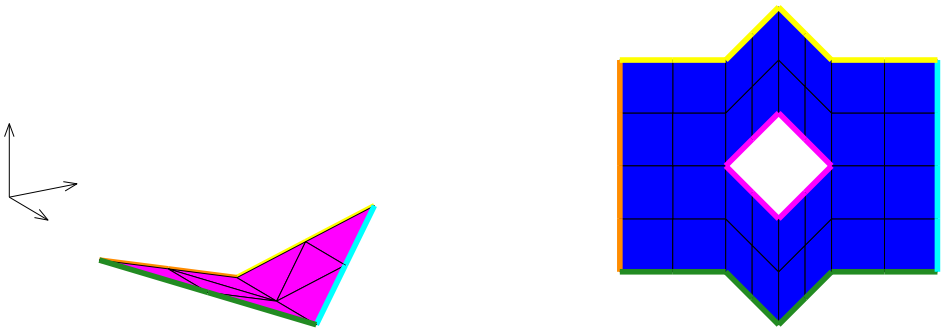


Figure 5.28: Meshes from initial set of triangles and quadrangles.

The program that produces it looks like the following:

```
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.

5.4.3 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++.

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 :

```
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);
```

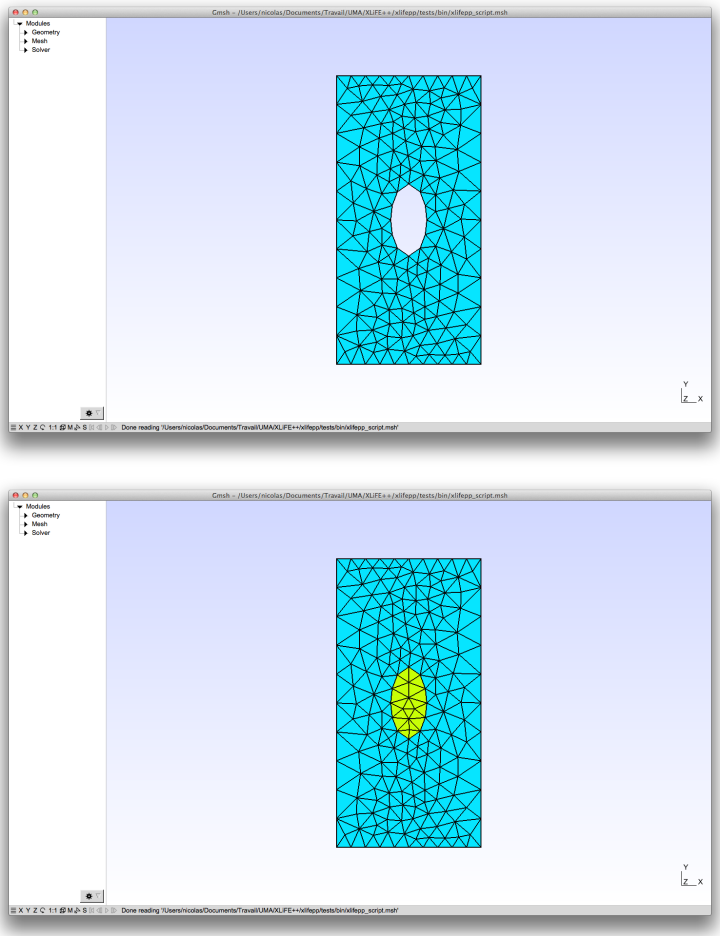


Figure 5.29: GMSH view of $m1$ and $m2$

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

```
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 pa1(_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-pa1,_tetrahedron,1,_gmsh);
```

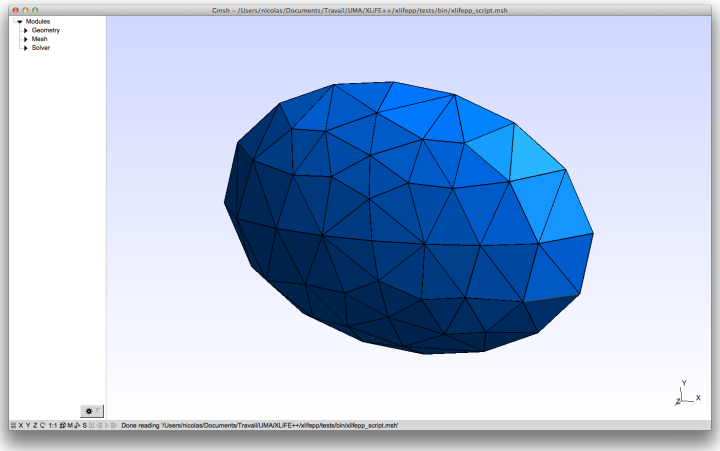


Figure 5.30: GMSH view of a 3d composite geometry (ellipsoid - parallelepiped)

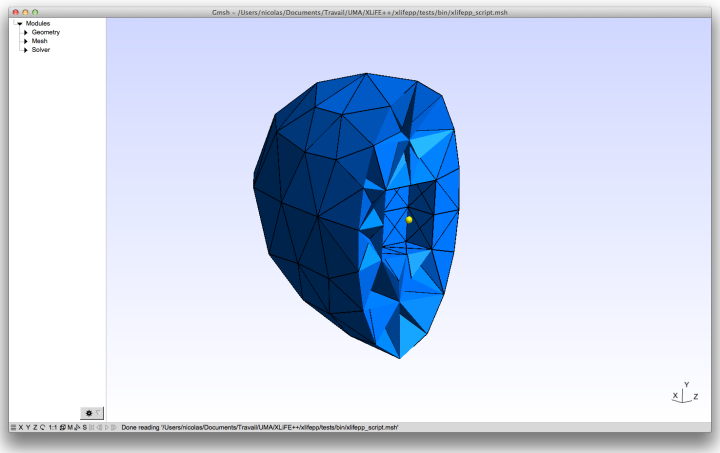


Figure 5.31: Gmsh view of the hole of a 3d composite geometry (ellipsoid - parallelepiped)

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";
Rectangle r1(_xmin=-2., _xmax=2., _ymin=-4., _ymax=4., _nnodes=11,
    _domain_name="Omega2", _side_names=sn);
sn[0]="Gamma_9"; sn[1]="Gamma_10"; sn[2]="Gamma_11"; sn[3]="Gamma_12";
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";
Ellipse e3(_center=Point(0.,0.,0.), _v1=Point(0.5,0.,0.),
    _v2=Point(0.,1.,0.), _nnodes=12, _side_names=sn);
sn[0]="Gamma_17"; sn[1]="Gamma_18"; sn[2]="Gamma_19"; sn[3]="Gamma_20";
Rectangle r2(_xmin=5., _xmax=6., _ymin=0., _ymax=1., _nnodes=6,
    _domain_name="Omega3", _side_names=sn);
sn[0]="Gamma_21"; sn[1]="Gamma_22"; sn[2]="Gamma_23"; sn[3]="Gamma_24";
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);
```

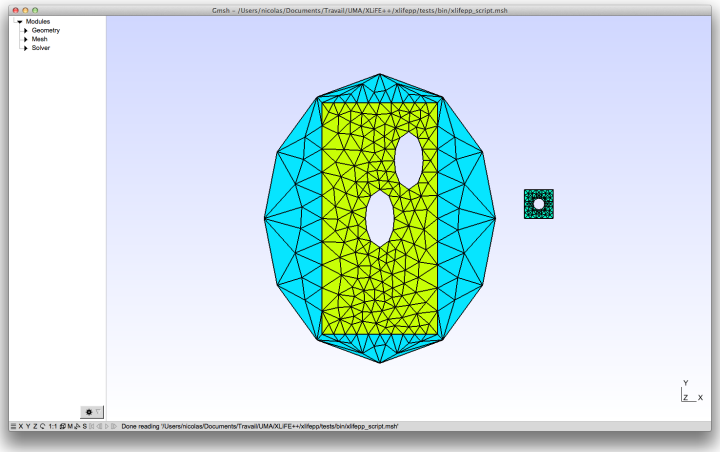


Figure 5.32: Gmsh view of complex 2d composite geometry

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);
```

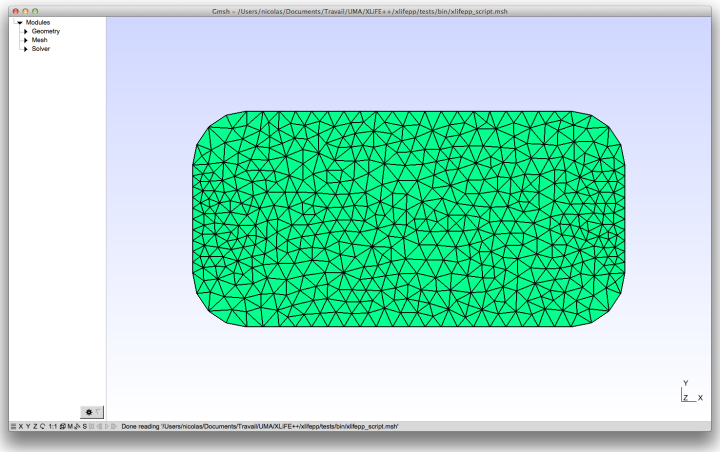


Figure 5.33: GMSH view of rectangular geometry with rounded corners, defined with the `surfaceFrom` routine

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, bith 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); 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");
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);
```

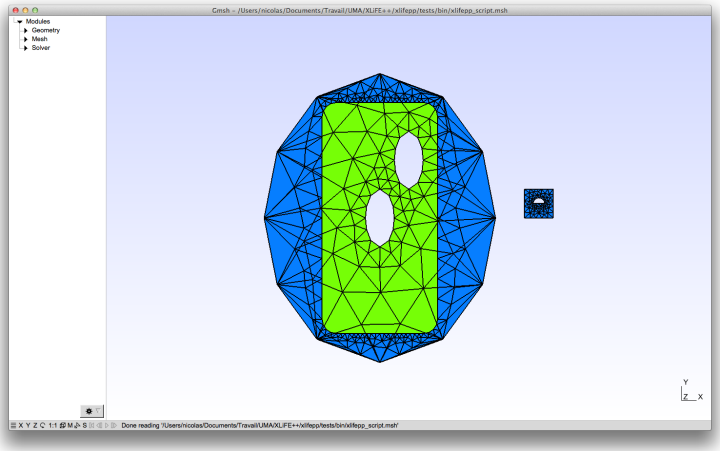


Figure 5.34: GMSH view of mesh2dP1Composite1

Finally, let's see now an example of complex composite geometry with use of forcing inclusion. Components are defines in example above:

```
Mesh mesh2dP1Composite3(e1+(sf1+(+(e2+e3))+r2+sf2,_triangle,1,_gmsh);
```

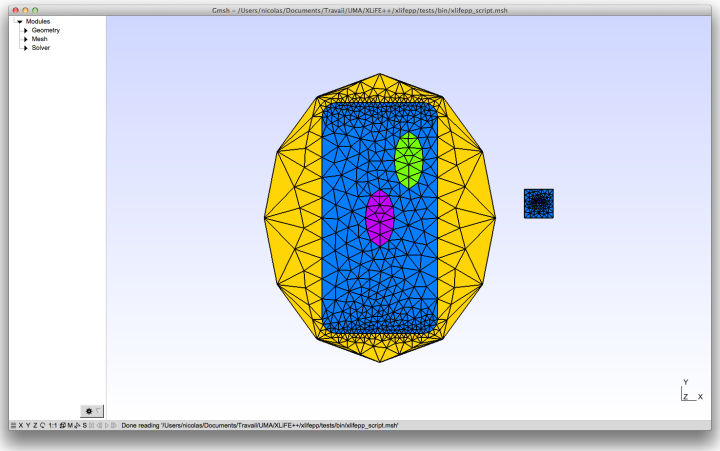


Figure 5.35: GMSH view of mesh2dP1Composite3

Structured mesh with GMSH generator

GMSH allows to generate structured and unstructured meshes. Even if XLIFE++ offers a

structured mesh generator, it may be useful to have it with the GMSH generator. To do so, you have an additional argument whose value is `structuredMesh`

```
Mesh m(Rectangle(_xmin=0., _xmax=2., _ymin=0., _ymax=4., _nnodes=11,
    _side_names=Strings("Gamma_1", "Gamma_2", "Gamma_3", "Gamma_4")),
    _triangle, 1, _gmsh, _structuredMesh);
```

5.5 Extrude a mesh

As an alternative to mesh an extruded geometry, it is possible to extrude a 1D or a 2D mesh using the following mesh constructor:

```
Mesh(const Mesh& ms, const Point& O, const Point& D , number_t nbl,
number_t namingDomain=0, number_t namingSection=0, number_t namingSide=0,
const string_t& meshName="");
```

where $\vec{OD}$ defines the direction of extrusion and *nbl* the number of layers of same width (regular extrusion). More precisely, any point M of the section mesh is extruded in *nbl* points :

$$M_k = M + O + k * \vec{OD}.$$

When a 1D section, extruded mesh is made with quadrangles. When a 2D triangular mesh section, extruded mesh is made with prisms and when a 2D quadrangular mesh section, extruded mesh is made with hexahedra.

The boundary domains created by the extrusion process come from the boundary domains of the original section. This process is controlled by the 3 parameters `namingDomain`, `namingSection`, `namingSide` taking one of of the values 0, 1 or 2, with the following rules:

- 0 : domains are not created
- 1 : one extruded domain is created for any domain of the original section
- 2 : for each layer, one extruded domain is created for any domain of the original section

Be cautious, the side domains of extruded domain are created from side domains of the given section. Thus, if the given section has no side domains, the extruded domains will have no side domains! The naming convention is the following:

- Domains and side domains keep their name with the extension "_e" or "_e1", "_e2", ..., "_en"
- Section domains have the name of original domains with the extension "_0",...,"_n"
- When `namingDomain=0`, the full domain is always created and named "Omega".

The following figure illustrates the naming rules of domains.

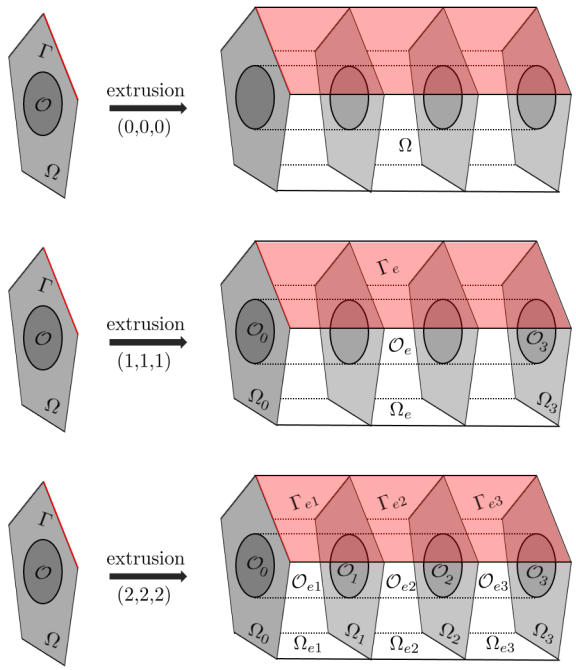


Figure 5.36: Mesh extrusion, domains naming

For instance, to mesh a tubular domain using the mesh extrusion of a crown:

```
Disk dext(_center=Point(0.,0.), _radius=1.,_nnodes=20, _domain_name="Omega",
         _side_names="Sigma");
Disk dint(_center=Point(0.,0.), _radius=0.5,_nnodes=10,_side_names="Gamma");
Mesh crown(dext-dint,_triangle,1,_gmsh);
Mesh tube(crown,Point(0,0,0),Point(0,0,1),10,1,1,1);
```

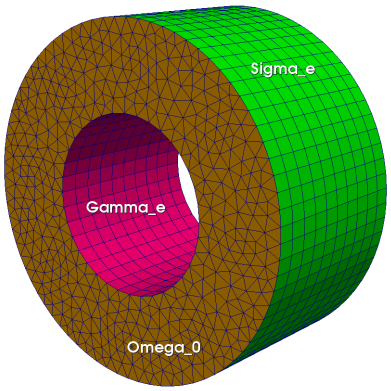


Figure 5.37: Tube prismatic mesh from extrusion of a crown mesh

5.6 Split mesh element

Sometimes it may be useful to split elements into elements of an other type, for instance to produce mesh of pyramids that are not provided by standard meshing softwares.To do this, a general constructor is offered :

```
Mesh(const Mesh& mesh, ShapeType sh, const string_t name="");
```

where `sh` is the desired shape type.
Up to now, only one splitting process is available: hexahedron of order 1 into six pyramids of order 1 :

```
Mesh meshQ1(Cuboid(_xmin=0, _xmax=1, _ymin=1, _ymax=3, _zmin=1, _zmax=5,
      _nnodes=Numbers(3, 5, 9), _side_names=sidenames), _hexahedron, 1,
      _structured, "Q1 mesh of [0,1]x[1,3]x[1,5]");
Mesh meshPyramid(meshQ1,_pyramid,"Py1 mesh of [0,1]x[1,3]x[1,5]");
```

The `Mesh` object passed to the splitting constructor may be any hexahedral mesh. Note that an hexahedron is split into the six pyramids based on the hexahedron faces and the centroid of the hexahedron as tip.

5.7 Loading a mesh from a file

XLiFE++ allows you to read various mesh file formats. The constructor to use is defined as follows:

```
//! 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.

```
// 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
- 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/

- To have more information on the PLY file format, please go to http://paulbourke.net/dataformats/ply.
- To have more information about GMSH, please go to 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.

5.8 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 :

```
//! apply a geometrical transformation on a Mesh
Mesh& Mesh::transform(const Transformation& t);
//! apply a translation on a Mesh
Mesh& Mesh::translate(std::vector<Real> u = std::vector<Real>(3,0.));
Mesh& Mesh::translate(Real ux, Real uy = 0., Real uz = 0.);
//! apply a rotation 2d on a Mesh
Mesh& Mesh::rotate2d(const Point& c = Point(0.,0.), Real angle = 0.);
//! apply a rotation 3d on a Mesh
Mesh& Mesh::rotate3d(const Point& c = Point(0.,0.,0.), std::vector<Real> u =
    std::vector<Real>(3,0.), Real angle = 0.);
Mesh& Mesh::rotate3d(Real ux, Real uy, Real angle);
Mesh& Mesh::rotate3d(Real ux, Real uy, Real uz, Real angle);
Mesh& Mesh::rotate3d(const Point& c, Real ux, Real uy, Real angle);
Mesh& Mesh::rotate3d(const Point& c, Real ux, Real uy, Real uz, Real angle);
//! apply a homothety on a Mesh
Mesh& Mesh::homothetize(const Point& c = Point(0.,0.,0.), Real factor = 1.);
Mesh& Mesh::homothetize(Real factor);
//! apply a point reflection on a Mesh
Mesh& Mesh::pointReflect(const Point& c = Point(0.,0.,0.));
//! apply a reflection2d on a Mesh
Mesh& Mesh::reflect2d(const Point& c = Point(0.,0.), std::vector<Real> u =
    std::vector<Real>(2,0.));
Mesh& Mesh::reflect2d(const Point& c, Real ux, Real uy = 0.);
//! apply a reflection3d on a Mesh
Mesh& Mesh::reflect3d(const Point& c = Point(0.,0.,0.), std::vector<Real> u
    = std::vector<Real>(3,0.));
Mesh& Mesh::reflect3d(const Point& c, Real ux, Real uy, Real uz = 0.);
```

For instance:

```
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 :

```
//! apply a geometrical transformation on a Mesh (external)
Mesh transform(const Mesh& m, const Transformation& t);
//! apply a translation on a Mesh (external)
Mesh translate(const Mesh& m, std::vector<Real> u = std::vector<Real>(3,0.));
Mesh translate(const Mesh& m, Real ux, Real uy = 0., Real uz = 0.);
//! apply a rotation 2d on a Mesh (external)
Mesh rotate2d(const Mesh& m, const Point& c = Point(0.,0.), Real angle = 0.);
//! apply a rotation 3d on a Mesh (external)
Mesh rotate3d(const Mesh& m, const Point& c = Point(0.,0.,0.),
    std::vector<Real> u = std::vector<Real>(3,0.), Real angle = 0.);
Mesh rotate3d(const Mesh& m, Real ux, Real uy, Real angle);
Mesh rotate3d(const Mesh& m, Real ux, Real uy, Real uz, Real angle);
Mesh rotate3d(const Mesh& m, const Point& c, Real ux, Real uy, Real angle);
Mesh rotate3d(const Mesh& m, const Point& c, Real ux, Real uy, Real uz, Real
    angle);
//! apply a homothety on a Mesh (external)
Mesh homothetize(const Mesh& m, const Point& c = Point(0.,0.,0.), Real
    factor = 1.);
Mesh homothetize(const Mesh& m, Real factor);
//! apply a point reflection on a Mesh (external)
Mesh pointReflect(const Mesh& m, const Point& c = Point(0.,0.,0.));
//! apply a reflection2d on a Mesh (external)
Mesh reflect2d(const Mesh& m, const Point& c = Point(0.,0.),
    std::vector<Real> u = std::vector<Real>(2,0.));
Mesh reflect2d(const Mesh& m, const Point& c, Real ux, Real uy = 0.);
//! apply a reflection3d on a Mesh (external)
Mesh reflect3d(const Mesh& m, const Point& c = Point(0.,0.,0.),
    std::vector<Real> u = std::vector<Real>(3,0.));
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.,1.);
```

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.

5.9 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.

5.9.1 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.), _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.), _xlength=0.5, _ylength=1,
    _nnodes=Numbers(20,40), _side_names=sn),_triangle,1,_structured);
mesh2d.renameDomain("Omega","Omega-");
sn[2] = "x=1/2+"; sn[3] = "x=1";
Mesh mesh2d_p(Rectangle(_origin=Point(0.5,0.), _xlength=0.5, _ylength=1,
    _nnodes=Numbers(20,40), _side_names=sn),_triangle,1,_structured);
mesh2d_p.renameDomain("Omega","Omega+");
mesh2d.merge(mesh2d_p);
Domain omegaM=mesh2d.domain("Omega-");
Domain omegaP=mesh2d.domain("Omega+");
Domain sigmaM=mesh2d.domain("x=0");
Domain sigmaP=mesh2d.domain("x=1");
Domain gamma=mesh2d.domain("x=1/2- or x=1/2+");
```

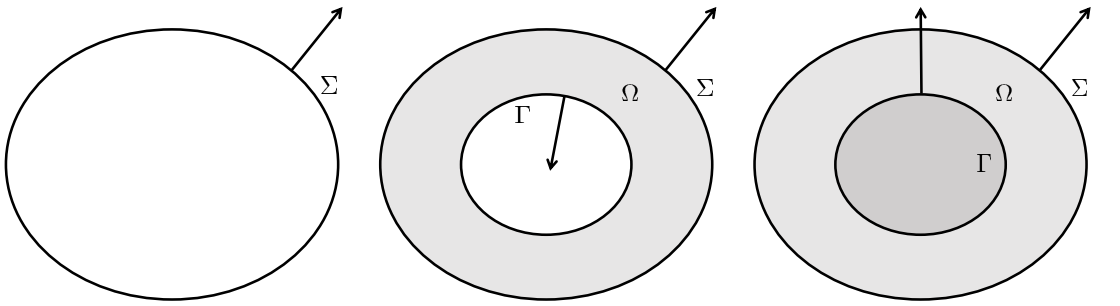
In this exemple, the unit square is split in two domains Ω^+ and Ω^- using the `merge` and `renameDomain` functions. Note that the merging process of meshes concatenates 'same' domain in a new one named "name1 or name2".

5.9.2 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 Γ, 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 Σ is a boundary (or a part of) of a unique domain Ω, the selected normals are the outwards vectors to Ω
- if Σ is a boundary between two domains (an interface), the selected normals are the towards infinite vectors
- if Σ is not a boundary, say an immersed manifold, the selected normals are the towards infinite vectors



Manifold : towards infinite normal Boundary : outwards Ω 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 member function of `Domain` class: `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
```

For visualization purpose, the normal vectors can be exported to a vtk/vtu file using:

```
Sigma.saveNormalsToFile("n_Sigma",_vtu);
```

They can be also collected in a `TermVector` object:

```
Space V1(Omega,P1,"V1");  Unknown u1(V1,"u1",3);
TermVector Ns = normalsOn(Sigma,u1);
```

The normals are computed by L2 projection on the space W related to the `Unknown` used. More precisely, the normals n in space W are get by solving the problem:

$$\int_\Sigma n|t = \int_\Sigma n_0|t \quad \forall t \in W$$

where n_0 is the normal to the element faces. If (w_j) denotes the basis of W and N the vector representing the normal in W basis ($n = \sum_j n_j\, w_j$), the following linear system is solved:

$$\mathbb{A}N = B, \quad \text{with } \mathbb{A} = \int_\Sigma w_i|w_j \text{ and } B_i = \int_\Sigma n_0|w_i$$

The unknown may be an explicit vector unknown or a scalar unknown if the space is a space of vectors. In the case of a P0 unknown, the normals are normals on element faces with no projection. In the case of Pk Lagrange unknown, the normals are some interpolated normals on Lagrange dofs.

5.9.3 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 $F\ \ \Sigma^+ \longrightarrow \Sigma^-$. The simple way to define such map is the following:

```
Reals mapPM(const Point& P, Parameters& pa = defaultParameters)
{
   Point Q(P);
   Q(1)-=1;
   return Q;
}
...
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.

5.9.4 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 ρ and deal with a unique integral:

$$\int_{\Omega} \rho(x)u(x)\,v(x)$$

and assign id to domains that are subdomains :

```
Real rho(const Point&P, Parameters& pars=defaultParameters)
{
 Number mat=materialIdFromParameters(pars);
 if(mat==1) return ...
 else return ...
}
...
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);
```

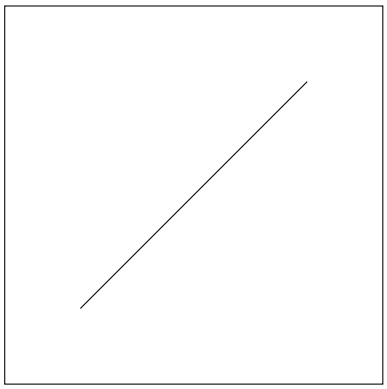
5.9.5 Cracking a domain

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. Inthis 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.

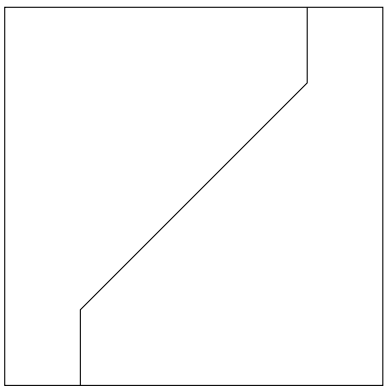


```
Point x1(0,0,0), x2(1,0,0), x3(1,1,0), x4(0,1,0),
    x5(0.2,0.2,0), x6(0.8,0.8,0), x7(0.2,0,0),
    x8(0.8,1,0);
Rectangle rrect8(_v1=x1, _v2=x2, _v4=x4,
    _domain_name="Omega", _side_names="Gamma");
Segment scrack(_v1=x5, _v2=x6, _nnodes=3,
    _domain_name="Crack", _side_names="Sigma");
openCrack(scrack,"Sigma");
Mesh m(rrect8+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 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);
Segment s1(_v1=x1, _v2=x7, _domain_name="Gamma");
Segment s2(_v1=x2, _v2=x7, _domain_name="Gamma");
Segment s3(_v1=x3, _v2=x2, _domain_name="Gamma");
Segment s4(_v1=x8, _v2=x3, _domain_name="Gamma");
Segment s5(_v1=x8, _v2=x4, _domain_name="Gamma");
Segment s6(_v1=x4, _v2=x1, _domain_name="Gamma");
Segment s7(_v1=x7, _v2=x5);
Segment s8(_v1=x5, _v2=x6, _nnodes=3,
    _domain_name="Crack");
Segment s9(_v1=x6, _v2=x8);
crack(s8);
Geometry
    sf1=surfaceFrom(s7+s8+s9+s5+s6+s1,"Omega1");
Geometry
    sf2=surfaceFrom(s7+s8+s9+s4+s3+s2,"Omega2");
Mesh m(sf1+sf2, _triangle, 1, _gmsh);
```

Here, it is a closed crack.

 In this example, surfaces have different domain names. You can also give the same domain name

Which way is better ?

	direct way	indirect way
1D crack in 2D mesh	100% safe	100% safe
2D crack in 3D mesh	not 100% safe	100% safe
1D crack in 3D mesh	to be tested	to be tested

 Gmsh team is currently working on improving their crack engine to be 100% whatever the case.

A look at the mesh file

Let's see the resulting mesh file for the indirect example above:

```
$MeshFormat
2.2 0 8
$EndMeshFormat
$PhysicalNames
4
1 1 "Crack"
1 2 "Gamma"
2 3 "Omega1"
2 4 "Omega2"
$EndPhysicalNames
$Nodes
18
1 0.2 0.2 0
2 0.8 0.8 0
3 0.4999999999991927 0.4999999999991927 0
4 0.8 1 0
5 0 1 0
6 0 0 0
7 0.2 0 0
8 1 1 0
9 1 0 0
10 0.4999999999991927 0.4999999999991927 0
11 0.4000000000000001 0.8999999999999999 0
12 0.09999999999935429 0.5999999999998384 0
13 0.6188775510203053 0.8489795918366316 0
14 0.1178571428570276 0.1714285714285426 0
15 0.5999999999991926 0.1 0
16 0.8999999999993543 0.3999999999998387 0
17 0.3811224489791758 0.1510204081631623 0
18 0.8821428571427419 0.8285714285713998 0
$EndNodes
```

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)

```
$Elements
36
1 1 2 1 2 1 3
2 1 2 1 2 3 2
3 1 2 2 4 4 5
4 1 2 2 5 5 6
5 1 2 2 6 6 7
6 1 2 2 8 4 8
7 1 2 2 9 8 9
8 1 2 2 10 9 7
9 1 2 1 11 1 10
10 1 2 1 11 10 2
11 2 2 3 7 2 13 3
12 2 2 3 7 1 3 12
13 2 2 3 7 3 11 12
14 2 2 3 7 3 13 11
15 2 2 3 7 2 4 13
16 2 2 3 7 7 1 14
17 2 2 3 7 7 14 6
18 2 2 3 7 5 12 11
19 2 2 3 7 1 12 14
20 2 2 3 7 4 5 11
21 2 2 3 7 4 11 13
22 2 2 3 7 5 6 12
23 2 2 3 7 6 14 12
24 2 2 4 11 1 10 17
25 2 2 4 11 2 16 10
26 2 2 4 11 10 16 15
27 2 2 4 11 10 15 17
28 2 2 4 11 7 1 17
29 2 2 4 11 2 4 18
30 2 2 4 11 4 8 18
31 2 2 4 11 9 15 16
32 2 2 4 11 7 15 9
33 2 2 4 11 7 17 15
34 2 2 4 11 2 18 16
35 2 2 4 11 8 9 16
36 2 2 4 11 8 16 18
$EndElements
```

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".

5.10 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:

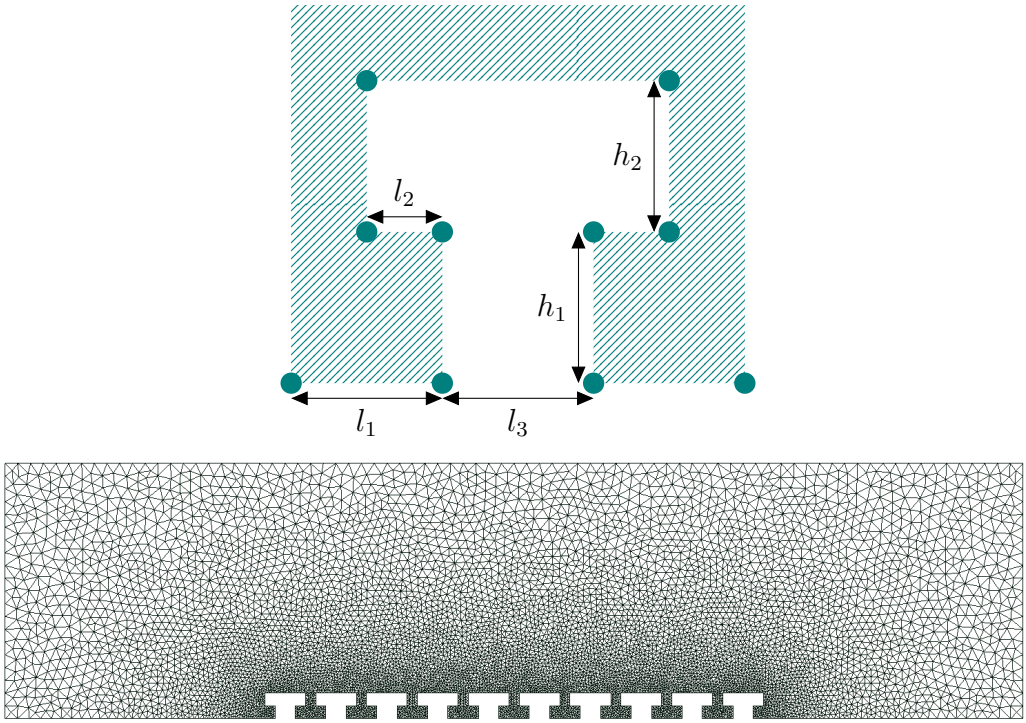


Figure 5.38: Definition of the cavity and the mesh you want

First, you have to define the first cavity, according to the previous figure:

```
// 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

```
// 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.); }
```

Finally, we define borders of the main domain, and mesh the resulting `Geometry` defined with `surfaceFrom`.

```
// full geometry
Real sb=0.05;
Point p1(0.,0.), p2(2.+nbcav*cL,0.), p3(2.+nbcav*cL,1.), p4(0.,1.);
Segment s0(_v1=p1, _v2=po, _hsteps=Reals(sb,s)),
        s10(_v1=Point(1.+nbcav*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 Defining the problem

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 φ_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:

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

int main() {

  init(); // 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:

```
  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&, PolynomType, const String&, bool opt = true);
Space(const GeomDomain,, QolynomType, const String&, bool opt = true);
Space(const GeomDomain&, FeFaceType , const String&, bool opt = true);
Space(const GeomDomain&, FeEdgeType , const String&, 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 NE1_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 optimization (reduction of the band width of the matrix) if its value is false.

Here are some examples of FE space construction :

```
//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");
FEInterpolation interp=interpolation(_Lagrange,_standard,20,H1);
Space V4(omega,interp,"P20");
//Hdiv Finite Element spaces
Space W1(omega,RT_1,"RT1");
Space W3(omega,interpolation(_RaviartThomas,_standard,3,Hdiv),"RT3");
//Hrot Finite Element spaces
Space R1(omega,N_1,"Ned1");
Space R2(omega,N_2,"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_1,"Hdiv_Ned1");
Space R2(omega,NE1_2,"Hrot_Ned2");
```

Note that it is possible to use some shortcuts in `Space` construction:

```
Space V0(omega,Lagrange,0,"V0");          // P0 Lagrange, L2 conforming
Space V3(omega,Lagrange,3,"V3");          // P3 Lagrange, H1 conforming
Space R1(omega,Raviart_Thomas,1,"RT1"); // Hdiv conforming
Space R2(omega,Nedelec_face,2,"NF1");    // Hdiv conforming
Space N1(omega,Nedelec_edge,1,"NE1");    // Hcurl conforming
Space CR(omega,CrouzeixRaviart,1,"CR"); // Other non conforming (degree 1 on
    triangle or tetrahedron)
```

When dealing with problem with vector unknown where each component is approximated in the same space (for instance $P1 \times P1 \times P1$ for the displacement field in elasticity problem), you have to build a 'vector' unknown on a 'scalar' space; see the `Unknown` section.

Some subspaces or trace spaces are automatically created by XLiFE++. For instance, when an integral on a boundary (Σ) is involved in a bilinear or linear form, the trace space $V_{|\Sigma} = \{v_{|\Sigma},\ v \in V\}$ is created. This subspace can be get using the following command:

```
Space& Vs=V|Sigma;
```

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:

```
Space(const GeomDomain& g, Function f, Number n, Dimen d, const String&
    name);
Space(const GeomDomain& g, Function f, Number n, Dimen d);
Space(const GeomDomain& g, Function f, Number n, const String& name);
```

Let us see an example :

```
Real sin_n(const Point& P, Parameters& pa = defaultParameters)
{
  Real h = pa("h");                    // get the parameter h (user definition)
  Real n = pa("basis index");          // get the index of function to compute
  return sqrt(2. / h) * sin(n * pi * P.x() / h); // computation
}

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");
  ...
}
```

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 an other space (`TermVectors`, see **Terms** chapter). The following example shows how it works:

```
int main (int argc, char** argv)
{
  Mesh m(...);
  Domain omega=m.domain("Omega");
  Space V(omega,P1,"V",true); //create FE space
  Unknown u("u",V);           //create FE unknown
  Real h=1;
  Parameters params(h,"h");
```

```
Function sinBasis(sin_n, params);

TermVectors sinInt(N);      //interpolated functions
for(Number n=0; n<N; n++)
  {
    sinBasis.parameter("basis index")=n;
    sinInt(n+1)=TermVector(u, omega, sinBasis, "c"+tostring(n));
  }
Space S(sinInt, "V interpolated sin(n*pi*y)");
```

The `Unknown` object is described in the next section.

Advance usage

It is possible to manipulate spectral basis by instanciate such objects :

```
SpectralBasisFun sbFun(omega, sinBasis, N, 1);
SpectralBasisInt sbInt(sinInt);
```

Thus it is possible to evaluate basis functions at a point :

```
Real r;
Point P(1.,0.);
sbFun.function(n,P,r);
sbInt.function(n,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.

```
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:

```
Unknown u(V, "u", 2);
Unknown p(V, "p");
TestFunction v("v",u);
TestFunction q("q",p);
cout<<u.rank()<<" "<<v.rank()<<" "<<p.rank()<<" "<<q.rank();
```

This exemple gives `1 3 2 4`.

It is possible to assign the rank of an unknown at the construction:

```
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()<<" "<<v.rank()<<" "<<p.rank()<<" "<<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:

```
...
setRanks(u,1,p,2,v,11,q,12);
```

6.1.4 Dealing with collections

When there is a lot of domains, spaces, unknowns it may be more friendly to work with indexed collection. This is the purpose of the classes `Domains`, `Interpolations`, `Spaces` , `Unknowns` and `TestFunctions`. As an example, suppose you want to deal with 4 domains:

```
Strings sn("Gamma1","Gamma2","Gamma3","Gamma4");
Mesh mesh2d(Rectangle(_xmin=0,_xmax=0.5,_ymin=0,_ymax=1,_nnodes=Numbers(3,6),
            _domain_name="Omega",_side_names=sn),_triangle,1,_structured);
//get the mesh domains in a Domains object
Domains doms(4);
for(Number i=1;i<=4;i++) doms(i)=mesh2d.domain(i-1);
//create one space by domain (say P1)
Spaces Vs(4);
for(Number i=1;i<=4;i++) Vs(i)=Space(doms(i),_P1,"V_"+tostring(i));
//create unknowns
Unknowns us(4);
for(Number i=1;i<=4;i++) us(i)=Unknowns(Vs(i),"u"+tostring(i));
//create TestFnctions
TestFunctions vs=dualOf(us);
```

Other syntaxes are available:

```
Unknown u1(V1,"u1"), u2(V2,"u2"), u3(V3,"u3");
Unknowns us1(u1,u2,u3);
Unknowns us2; us2<<u1<<u2<<u3;
Unknowns usi={u1,u2,u3}; //only in C++2011
```

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*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:

mathematical expression	XLiFE++ translation	comment
$\nabla(u)$	grad(u)	u unknown
$\nabla(u)\cdot\nabla(v)$	grad(u)|grad(v)	u unknown, v test function
$(A*\nabla(u))\cdot\nabla(v)$	(A*grad(u))|grad(v)	u unknown, v test function, A a matrix
$(F(x)*\nabla(u))\cdot\nabla(\overline{v})$	(F*grad(u))|grad(conj(v))	u unknown, v test function, F a function
u(x)*G(x,y)*v(y)	u*G*v	u unknown, v test function, G a kernel

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

```
Function F(f1, "name", params);
```

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

```
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

```
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):

mathematical	built in functions	unknown
identity	`id`(u) or u	scalar or vector
∂_t	`d0`(u) or `dt`(u)	scalar or vector
∂_x	`d1`(u) or `dx`(u)	scalar or vector
∂_y	`d2`(u) or `dy`(u)	scalar or vector
∂_z	`d3`(u) or `dz`(u)	scalar or vector
∇	`grad`(u) or `nabla`(u)	scalar or vector
div	`div`(u)	vector
curl	`curl`(u) or `rot`(u)	vector
∇_τ (surfacic)	`gradS`(u) or `nablaS`(u)	scalar or vector
div_τ (surfacic)	`divS`(u)	vector
curl_τ (surfacic)	`curlS`(u) or `rotS`(u)	vector
$\nabla_{abc} = (a\partial_x, b\partial_y, c\partial_z)$	`gradG`(u,a,b,c) or `nablaG`(u,a,b,c)	scalar or vector
$\nabla_{abc}.$	`divG`(u,a,b,c)	vector
$\nabla_{abc}\times$	`curlG`(u,a,b,c) or `rotS`(u,a,b,c)	vector

mathematical	built in functions	unknown
ε	`epsilon`(u)	vector
ε_{iabc}	`epsilonG`(u,i,a,b,c)	vector
ε_R	`epsilonR`(u) $= (\varepsilon_{11}, \varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, \varepsilon_{32}, \varepsilon_{31}, \varepsilon_{21})$	vector
voigtToM	`voigtToM`(u)= [u1 u6 u5; u6 u2 u4; u5 u4 u3]	vector
$n*$	`nx`(u) or _n*u	scalar
$n.$	`ndot`(u) or _n.u	vector
$n\times$	`ncross`(u) or _n^u	vector
$n \times n\times$	`ncrossncross`(u) or _n^_n^u	vector
$n.\nabla$	`ndotgrad`(u) or _n.`grad`(u)	scalar
$n \times \nabla$	`ncrossgrad`(u) or _n^`grad`(u)	scalar
n div	`ndiv`(u) or _n*`div`(u)	vector
$n \times$ curl	`ncrosscurl`(u) or _n^`curl`(u)	vector
[ ] (jump across)	`jump`(u)	scalar or vector
{ } (mean across)	`mean`(u)	scalar or vector

6.2.2 Operators on kernel

Similar to operator on unknowns, XLiFE++ allows to apply some operators on kernel (say $k(x, y)$). The complete list of operators is the following :

mathematical	built in functions	unknown
identity	`id`(k) or k	scalar or vector
∇_x	`grad_x`(k) or `nabla_x`(k)	scalar or vector
∇_y	`grad_y`(k) or `nabla_y`(k)	scalar or vector
div_x	`div_x`(k)	vector
div_y	`div_y`(k)	vector
curl_x	`curl_x`(k) or `rot_x`(k)	vector
curl_y	`curl_y`(k) or `rot_y`(k)	vector
n_x*	`ntimes_x`(k) or _nx*k	scalar
n_y*	`ntimes_y`(k) or _ny*k	scalar
$n_x.$	`ndot_x`(k) or _nx.k	vector
$n_y.$	`ndot_y`(k) or _ny.k	vector
$n_x\times$	`ncross_x`(k) or _nx^k	vector
$n_y\times$	`ncross_y`(k) or _ny^k	vector
$n_x \times (n_x\times)$	`ncrossncross_x`(k) or _nx^(_nx^k)	vector
$n_y \times (n_y\times)$	`ncrossncross_y`(k) or _ny^(_ny^k)	vector

mathematical	built in functions	unknown
$n_x.\nabla_x$	`ndotgrad_x`(k) or _nx.`grad_x`(k)	scalar
$n_y.\nabla_y$	`ndotgrad_y`(k) or _ny.`grad_y`(k)	scalar
$n_x\mathrm{div}_x$	`ndiv_x`(k) or _nx*`div_x`(k)	vector
$n_y\mathrm{div}_y$	`ndiv_y`(k) or _ny*`div_y`(k)	vector
$n_x \times \mathrm{curl}_x$	`ncrosscurl_x`(u) or _nx^`curl_x`(k)	vector
$n_x.n_y*$	`nxdotny_times`(k) or (_nx|_ny)*k	scalar or vector
$(n_x \times n_y).$	`nxcrossny_dot`(k) or (_nx^_ny)|k	vector
$(n_y \times n_x).$	`nycrossnx_dot`(k) or (_ny^_nx)|k	vector
$(n_x \times n_y)\times$	`nxcrossny_cross`(k) or (_nx^_ny)^ k	vector
$(n_y \times n_x)\times$	`nycrossnx_cross`(k) or (_ny^_nx)^ k	vector

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 5.9.2 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 some kernels available in XLiFE++ for 2D and 3D problems: Laplace and Helmholtz Green functions, and Mazxwell Green tensor in 3D:

- The Laplace kernel used for 2D problems is

$$L_{2d}(x,y) = -\frac{1}{2\pi}\log\left(\|x-y\|\right),$$

  and for 3D we use

$$L_{3d}(x,y) = \frac{1}{4\pi\|x-y\|}$$

- The Helmholtz Green function for 2D problems used is

$$H_{2d}(k;x,y) = \frac{i}{4}H_0^{(1)}\left(k\|x-y\|\right),$$

with $H_0^{(1)}$ then Hankel function. Finally, for 3D problems we use

$$H_{3d}(k;x,y) = \frac{e^{ik\|x-y\|}}{4\pi\|x-y\|}.$$

- The harmonic Maxwell Green tensor for 3D problems is defined as:

$$M_{3d}(k;x,y) = H_{3d}(k;x,y)\,\mathbb{I}_3 + \frac{1}{k^2}\mathrm{Hess}(H_{3d}(k;x,y)).$$

Examples of declaration of these kernels follow:

```
Kernel GLap2D=Laplace2dKernel();
Kernel GLap3D=Laplace3dKernel();
Parameters pars(k,"k"); // We provide the wavenumber k using a Parameters
Kernel GHelm2D=Helmholtz2dKernel(pars);
Kernel GHelm3D=Helmholtz3dKernel(k);
```

6.2.4 Interpolated function in operator

Interpolated functions are function defined from finite element approximation :

$$f(x) = \sum_{i=1,q} f_i\, w_i(x)$$

where w_i are some finite element shape functions and f_i are some real/complex scalar/vector coefficients. With Lagrange FE, $f_i = f(x_i)$ where x_i is the node related to the shape function w_i. In XLiFE++ lib, such function may be represented by a `TermVector` object that handles both a FE space, thus the shape functions, and the coefficients in an array of values. By specifying a `TermVector` object in an operator construction, an interpolated function will be handled.

```
Mesh
    m(Square(_origin=Point(0.,0.),_length=1.,_nnodes=5,_domain_name="Omega"),
        _triangle,1,_structured);
Domain omega=m.domain("Omega");
Space V(omega,P1,"V");  Unknown uh(V,"u");
TermVector x1(u,omega,_x1,"x1");  // interpolated function
LinearForm l1=intg(omega,x1*u);
LinearForm l2=intg(omega,(x1^3)*u);
```

Such approach may be very useful for non linear problem when non linear terms are taken into account at a previous step in a iterative scheme.

For the moment, the way that XLiFE++ processes, consists in transforming the `TermVector` object in a `Function` object that performs the computation of the sum outside of any context : first locate the finite element that contains the point x, then evaluate using local interpolation. As the localization algorithm has a log complexity, computation of the interpolated function at a point is not so time expansive but it is not optimal, in particular when computing integral.

6.2.5 Additional operation in operator

Transpose and conjugate

Some quantities (function or kernel) involved in a bilinear form may have to be transposed or conjugated. XLiFE++ provides 3 operators to do it :

- `tran` to transpose an expression
- `conj` to conjugate an expression
- `adj` to conjugate and transpose an expression

Obviously, transposition has only meaning for functions or kernels returning a matrix! In XLiFE++, the shape functions related to a FE space are real scalar/vector functions, so transpose or conjugate an unknown has no interest and, by the way, is not allowed!

Extension

Some problems require to deal with the extension of a function/kernel, say f or k, from a boundary (say Γ) to its parent domain (say Ω). XLiFE++ provides a particular extension process that extends function/kernel from the boundary Γ to its neighborhood, that is the set of elements that have at least one vertex located onto Γ, say Γ_{ext}. More precisely, the extension formula is:

$$E_\Gamma(f)(x) = \sum_{M_i \in \Gamma} f(M_i) w_i(x)$$

where M_i are some mesh vertices and w_i the one order Lagrange shape functions related to the vertex M_i. Such extension vanishes outside Γ_{ext}.

Define an extension is very easy. You have to specify the boundary domain to be extended, if not unique, the domain where you want to do the extension and the variable if you want to extend a kernel :

```
Extension Eg(Gamma);    //extend from Gamma to elements in its neighborhood
Extension Eg(Gamma,_y);        //extend from Gamma for variable y
Extension Eg(Gamma,Omega);     //extend from Gamma to Omega
Extension Eg(Gamma,Omega,_x); //extend from Gamma to Omega for variable x
```

The set of elements in the neighborhood of Γ can be explicitely constructed by using the `extendedDomain` member function of `Domain` :

```
Domain Gamma_ext=Gamma.extendDomain();     //elements having a side on Gamma
Domain Gamma_ext=Gamma.extendDomain(true);//elements having a vertex on Gamma
```

Finally, to use extension in bilinear form, write for instance

```
Extension Eg(Gamma, Omega);
BilinearForm a=intg(Omega,u*Eg(f)*v);          //extension of a function
Eg.var=_y;
BilinearForm a=intg(Gamma,Omega,u*Eg(k)*v);  //extension of a kernel along y
```

 The computation are automatically restricted to elements of the extended domain, so you do not have to restrict yourself.
Be cautious, when applying an extension to an object involving kernel derivatives; for instance

$$E_{\Gamma,x}[dy(k)](x,y) = \sum_{M_i \in \Gamma} dy(k)(M_i,y)\, w_i(x)$$

$$E_{\Gamma,x}[dx(k)](x,y) = \sum_{M_i \in \Gamma} dx(k)(M_i,y)\, w_i(x) + \sum_{M_i \in \Gamma} k(M_i,y)\, dx(w_i)(x).$$

Summary of main operator syntaxes

In the following

- `val` represents any constant value, that is a real or a complex value, a real or a complex vector or a a real or a complex matrix,
- `fun` represents a `Function` object or an explicit C++ function
- `opfun` represents an `OperatorOnFunction` object, say difop(Function); only few differential operators are available
- `ker` represents a `Kernel` object or an explicit C++ function
- `opker` represents an `OperatorOnKernel` object, say difop(Kernel); only few differential operators are available
- `tv` represents a `TermVector` object
- `aop` represents an algebraic operator, one of `* | ^ %`

`[val/fun/opfun/tv aop] [difop](Unknown) [aop val/fun/opfun/tv]`	`-> opu`
`opu [aop opu]`	`-> opus`
`opu aop ker/opker [aop opu]`	`-> kopus`

6.2.6 Integration method

When defining a linear or a bilinear form, the user may specify the integration method or a list of integration methods to use. Currently the following objects are available:

- `QuadratureIM` : quadrature methods based on quadrature points and weights, see quadrature rule in the next section
- `IntegrationMethods`: specific methods to integrate singular kernel in bilinear form, see details in the next section

To use it in a computation, specify an integration object in the definition of the form:

```
QuadratureIM quadIM(Gauss_Legendre,2);          //standard quadrature method
BilinearForm blf=intg(omega,uv,quadIM);
BilinearForm blf=intg(omega,uv,Gauss_Legendre,2);    //shortcut syntax
//for singular integrals
IntegrationMethods ims(Sauter_Schwab,3,0.,Gauss_Legendre,3);
BilinearForm blf=intg(sigma,sigma,u*G*v,ims);
IntegrationMethods imr(Gauss_Legendre,3);
LinearForm lf=intg(sigma,G*u,imr),U);
```

Note that integration method is attached to the integral definition. So you can mix different integration methods in a bilinear form :

```
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 wich integrates exactly the polynoms of order

$$k = (deg(u) - order(dif(u))) * [deg(u)] * [(deg(v) - order(dif(v))) * [deg(v)]]$$

where $deg(u)$ (resp. $deg(v)$) is the degree of polynoms used by u-interpolation (resp. v-interpolation), $order(dif(u))$ (resp. $order(dif(v))$) is the order of differential operator applied to u (resp. v). [] 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 :

$$\int_{\widehat{E}} f(\widehat{x}) d\widehat{x} \approx \sum_{i=1,q} \omega_i \, f(\widehat{x}_i)$$

where $(\widehat{x}_i)_{i=1,q}$ are quadrature points belonging to reference element $\widehat{E}$ and $(w_i)_{i=1,q}$ 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 :

General rules

	Gauss-Legendre	Gauss-Lobatto	Grundmann-Muller	symmetrical Gauss
segment	any odd degree	any odd degree		
quadrangle	any odd degree	any odd degree		odd degree up to 21
triangle	any odd degree		any odd degree	degree up to 10
hexahedron	any odd degree	any odd degree		odd degree up to 11
tetrahedron	any odd degree		any odd degree	degree up to 10
prism				degree up to 10
pyramid	any odd degree	any odd degree		degree up to 10

Particular rules

	nodal	miscellaneous
segment	P1 to P4	
quadrangle	Q1 to Q4	
triangle	P1 to P3	Hammer-Stroud 1 to 6
hexahedron	Q1 to Q4	
tetrahedron	P1, P3	Stroud 1 to 5
prism	P1	centroid 1, tensor product 1,3,5
pyramid	P1	centroid 1, Stroud 7

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 `intg` with a specific pair of arguments:

- `QuadRule` qr, to give the quadrature rule formulae (possible values are Gauss_Legendre, Gauss_Lobatto, nodalQuadrature, miscQuadrature, Grundmann_Moller or symmetrical_Gauss),
- `Number` qro, to give the quadrature rule degree:

```
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)linearform 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

$$\int_\Omega f\, u\, v$$

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.


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 or integral representation

Integral equation involves singular kernels. To deal with the singularity in integrals, some particular methods are proposed to users. `LenoirSallesxx` classes compute in an analytic way integrals involving 2D or 3D Laplace kernel but with low order finite elements (P0 or P1) whereas there are methods for any finite element order but specific problem dimension such as `SauterSchwabIM` for 3D problems and `DuffyIM` for 2D problems. Currently, only $log(r)$ in 2D and r^{-1} in 3D singularities are adressed.

- `SauterSchwabIM`[1] class adresses computation of integral

$$\int_\Gamma \int_\Sigma K(x,y)\, dx\, dy$$

[1] *Integral over a product of geometric elements with singularity using Sauter-Schwab technique*, Stefan A. Sauter, Christoph Schwab, "Boundary Element Methods", Springer, 2010

where Γ and Σ 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. When creating such method, you may specify the quadrature order on segment:

```
SauterSchwabIM ssIM(5);
```

The default order is 3. Sauter-Schwab method works for any finite element on triangle.


The Sauter-Schwab method consist in transforming the integral over a triangle pair to some integrals over the unity cube of $\mathbb{R}^4$ and then computing each integral using a 4 tensor product of standard quadrature formula on segment. As a consequence, the number of points where the kernel is evaluated grows as a power 4 of the number of quadrature points used on segments. So increasing the order of quadrature on segment may be very time expansive.

- `DuffyIM`[2] class adresses computation of integral

$$\int_\Gamma \int_\Sigma K(x,y)\,dx\,dy$$

where Γ and Σ 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 specify the quadrature order on segment:

```
DuffyIM dufIM(5);
```

The default order is 6. Duffy method works for any finite element on segment.

- `LenoirSalles2dIM` and `LenoirSalles3dIM` [3] classes adress computation of integral

$$\int_\Gamma \int_\Sigma p(x)\,K(x,y)\,q(y)\,dx\,dy$$

or

$$\int_\Gamma \int_\Sigma p(x)\,\partial_{n_y} K(x,y)\,q(y)\,dx\,dy$$

where Γ and Σ 2D domains in 3D (partition of triangles) or 1D domains in 2D (partition of segments), $K(x,y)$ the Laplace kernel and p, q are either piecewise constant functions or piecewise linear functions. It deals only the case of elements sharing at least one vertex. So for elements that are not close you a standard quadrature has to be used. These classes do not manage any parameter:

```
LenoirSalles3dIM ls2();
LenoirSalles2dIM ls3();
```

- `LenoirSalles2dIR` and `LenoirSalles3dIR` classes address computation of integral

$$i(x) = \int_\Gamma K(x,y)\,q(y)\,dy$$

[2] *Integral over a product of segments with singularity using the Duffy transformation*

[3] *Integral over a product of geometric elements with singularity using Lenoir-Salles analytic technique*, Marc Lenoir, Nicolas Salles, Evaluation of 3-D Singular and Nearly Singular Integrals in Galerkin BEM for Thin Layers, SIAM Journal on Scientific Computing, vol. 36, pp. 3057-3078, 2012

or

$$i(x) = \int_\Gamma \partial_{n_y} K(x,y)\, q(y)\, dy$$

where Γ is either a 2D domain in 3D (partition of triangles) or a 1D domain in 2D (partition of segments), $K(x,y)$ the Laplace kernel and q is a either piecewise constant function or piecewise linear function. These classes do not manage any parameter:

```
LenoirSalles3dIR ls2();
LenoirSalles2dIR ls3();
```

Define several integration methods

Computing integral with kernel is a costly business because of the kernel singularity. But in fact, this singularity is effective only in particular situations : when two elements share at least one vertex in BEM computation or when the point is to close to an element in IR computation. This is the reason why XLiFE++ provides a way to choose different integration methods regarding a geometric criteria : the relative distance between the centroids of elements :

$$dr(E_i, E_j) = \frac{||C_i - C_j||}{\max(diam(E_i), diam(E_j))}$$

or the relative distance between a point and the centroid of element:

$$dr(x, E_i) = \frac{||x - C_i||}{diam(E_i)}.$$

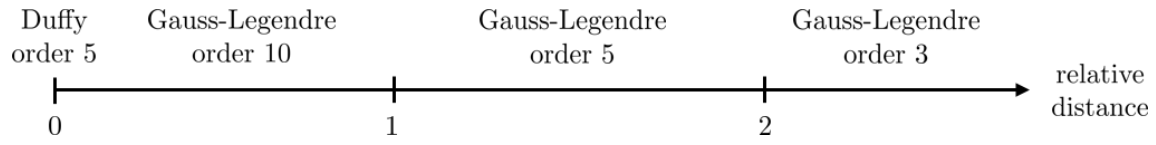
The `IntegrationMethods` class collect integration methods with two additional informations:

- the bound value b telling the integration method is applied when $dr \leq b$
- the part of the function concerned by the integration method, one of `_allFunction, _regularPart, _singularPart`, the default value is `_allFunction`.

There is a lot of way to define an `IntegrationMethods` object. Here are given some classical forms:

```
IntegrationMethods ims1(Sauter_Schwab,3,0.,defaultQuadrature,5);
IntegrationMethods ims2(Duffy,5,0.,Gauss_Legendre,10,1.,
                                   Gauss_Legendre,5,2.,
                                   Gauss_Legendre,3) ;
IntegrationMethods ims3(Lenoir_Salles_3d, Gauss_Legendre,5);
IntegrationMethods imsh(LenoirSalles2dIR(),_singularPart,theRealMax,
                   QuadratureIM(_GaussLegendreRule,4),_regularPart,theRealMax);
```

For instance, the definition of `ims2` corresponds to the following choice:




Be care when defining an `IntegrationMethods` object. In particular, check that all the cases are handled.

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 Γ (homogeneous) or $u = g$ on Γ (non homogeneous). But there are others : transmission condition on a boundary, periodic condition between two boundaries, null average on a domain, ... XLiFE++ provides a symbolic description of such conditions based on operator's stuff already described.

The general syntax of an essential condition is the following

$$(a1 \otimes op1(u1))|D1 +/- (a2 \otimes op2(u2))|D2 = f$$

where

- a1, a2 are some constants
- $\otimes$ is any algebraic operator (`*, |, % , ^`)
- 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 :

u|D = 0	homogeneous Dirichlet condition
u|D = f	non homogeneous Dirichlet condition
u1|D - u2|D = 0	homogeneous transmission condition
u|D1 - u|D2 = 0	homogeneous periodic condition
u|D1 - g * u|D2 = 0	quasi periodic condition (g function)

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 instanciate `EssentialConditions` object, which handles a set of conditions:

```
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:

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

EssentialConditions ecs = (u|sigmaM = un);
```

To concatenate conditions, use the operator & :

```
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:

```
...
Domain sigmaM=mesh2d.domain("x=0");
Domain sigmaP=mesh2d.domain("x=1");
Domain gamma=mesh2d.domain("x=1/2- or x=1/2+");
Space VM(omegaM,P2,"VM",true);
Unknown uM(VM, "u-");
Space VP(omegaP,P2,"VP",true);
Unknown uP(VP, "u+");
EssentialConditions ecs = (uM|sigmaM = 1) & (uP|sigmaP = 1)
                        & ((uM|gamma) - (uP|gamma) = 0);
```

To deal with periodic condition, the map related to the two domains involved is required:

```
Vector<Real> mapPM(const Point& P, Parameters& pa = defaultParameters)
{
   Point Q(P);
   Q(1)-=1;
   return Q;
}
...
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");
Space V(omega,P,"V",true);
Unknown u(V,"u");

defineMap(sigmaP, sigmaM, mapPM);
EssentialConditions ecs = (u|gammaM = 0) & (u|gammaP = 0)
                        & ((u|sigmaP) - (u|sigmaM) = 0);
```

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 responsability 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.

7 Solving the problem

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 :

$$L_i = l(\tau_i) \text{ and } \mathbb{A}_{ij} = a(w_j, \tau_i)$$

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:

$$L = \begin{bmatrix} L_{v_1} \\ L_{v_2} \\ \cdots \end{bmatrix} \text{ and } A = \begin{bmatrix} A_{v_1 u_1} & A_{v_1 u_2} \\ A_{v_2 u_1} & A_{v_1 u_2} \\ \cdots & \end{bmatrix}$$

where u_1, u_2 stands for unknowns and v_1, v_2 stands for test functions.


Unknowns 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`:

```
Mesh mesh2d(Square(_origin=Point(0.,0.), _length=1,
    _nnodes=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 omega1=mesh2d.domain("Omega1"); Domain omega2=mesh2d.domain("Omega2");
Space V1(omega1,P1,"V",true);          Space V2(omega2,P1,"V",true);
Unknown u1(V1,"u1");                   Unknown u2(V2,"u2");
TestFunction v1(u1,"v1");              TestFunction v2(u2,"v2");

LinearForm fv = intg(omega1,f*v1) + intg(omega2,f*v2);
TermVector F(fv,"F");

BilinearForm
    auv=intg(omega1,grad(u1)|grad(v1))+intg(omega2,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

- **_compute**, _notCompute : to manage the automatic computation of the TermMatrix
- **_assembled**, _unassembled : to manage the automatic assembling of the matrix; not assembled implies not computed

- _nonSymmetricMatrix, _symmetricMatrix, _selfAdjointMatrix, _skewSymmetricMatrix, _skewAdjointMatrix : to enforce symetry property when bilinear form has such symetry and XLiFE++ has not detected it.
- _csRowStorage, _csColStorage, _csDualStorage, _csSymStorage, _denseRowStorage, _denseColStorage, _denseDualStorage, _skylineSymStorage, _skylineDualStorage : to enforce the storage if the default one chosen by XLiFE++ is not well suited
- **_pseudoReductionMethod**, _realReductionMethod, _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 `ReductionMethod` object:

```
BilinearForm auv=intg(omega,grad(u)|grad(v));
EssentialConditions ecs= (u|sigmaM = 0);
TermMatrix A(auv, ecs,ReductionMethod(_pseudoReduction,10.), "A");
```

If you choose to declare the TermMatrix with the *_notCompute* option, its computation may be done later using the `compute` command:

```
...
BilinearForm auv=intg(omega,grad(u)|grad(v)));
TermMatrix A(auv,_notCompute,"A");
...
compute(A);
```

`TermMatrix` and `TermVector` manages some additional parameters and a lot of facilities are provided. Let us go to details.

7.1.1 TermVector in details

`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:

```
TermVector(name);
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.

The constructors of `TermVector` from linear forms 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 :

```
B.computed()=false;
```

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(Unknown, GeomDomain, T, String name)
```

The `T` argument may be a function (C++ function,`Function` object or `SymbolicFunction` 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);                          // vector unknown
TermVector B(u, omega, f, "B");              // from C++ function
Unknown v(V,"v");                            // scalar unknown
TermVector F(v, omega, x_1*x_2,"F");         // from SymbolicFunction
TermVector G(v, omega, 1.,"G");              // from constant
```

There are also a copy constructor and a constructor assigning a constant value from an other `TermVector`:

```
TermVector(TermVector, name);
template <typename T> TermVector(TermVector, T, name);
```

TermVector can be constructed from one or two TermVector by applying a C++ function or a symbolic function. C++ function has to be of the following forms:

```
Real fun(const Real& x1);
Real fun(const Real& x1, const Real& x2);
Complex fun(const Complex& x1);
Complex fun(const Complex& x1, const Complex& x2);
```

For instance, to build a new TermVector that is the squared of an other one:

```
Real fsq(const Real& x1){return x1*x1;}
...
TermVector F(intg(omega,u*v));
TermVector F2(F,fsq);      // from C++ function
TermVector F2(F,x_1^2);    // from symbolic function
```

Finally, by using algebric operators `+ - * / ^` and standard mathematical function `abs, real, imag, complex, sqrt, squared, sin, cos, tan, sinh, cosh, tanh, exp, log, log10, ...` on TermVector's, new TermVector may also be constructed:

```
TermVector F(intg(omega,u*v));
TermVector G = sqrt(F)+abs(F);
```

 It is possible to mix single unknown scalar TermVector and single unknown vector TermVector in some operations but only one single unknown vector TermVector is allowed. In that case, the result is a single unknown vector TermVector where the operation has been performed on each component of the input single unknown vector TermVector. For instance if X is a single unknown scalar TermVector and Y a single unknown vector TermVector, you can do

```
TermVector Z= X*X*Y;
```

but not

```
TermVector Z= X*Y*Y;
```

When many TermVector's are involved, they have to be of the same size!

Concatenate some scalar TermVector's into a one vector TermVector is also possible by using the following construction process:

```
Space V(omega,_P1,"V");      //P1 Lagrange space
Unknown u(V,"V");            //scalar unknown
TermVector V1(u,omega,1.); //a scalar TermVector (1,1,...)
TermVector V2(u,omega,2.); //an other scalar TermVector (2,2,...)
Unknown u2(V,"V",2);         //vector unknown
TermVector W(u2,V1,V2);      //vector TermVector ([1,2],[1,2],...)
```

This process works only for single unknown TermVector.

In case of a multiple unknowns vector, a unknown block may be extracted as follows:

```
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`:

```
...
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(fvs,"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.

```
...
TermVector W=U+V;     // Ok
cout<<U+V;            // NOT EVALUATED
```

Besides, it is possible to convert `TermVector`:

```
...
Space V(omega,P1,"V",true);
Unknown u(V,"u");
LinearForm fv=intg(omega,f*u);
TermVector B(fv,"B");

B.toAbs();
B.toReal();
B.toImag();
B.toComplex();
```

Be cautious, once it is converted it is not possible to go back.

In some circumstances, it may be useful to restrict a `TermVector` to a smaller domain. Use the member function onDomain or the operator `|`:

```
...
Space V(omega,P1,"V"); Unknown u(V,"u");
TermVector B(u,omega,x_1);              // TermVector on omega
TermVector Bg = B.onDomain(gamma);      // TermVector on gamma, boundary of omega
TermVector Bg = B|gamma;                // same
```

The merging of two `TermVector` living on different domains of a same mesh is also available:

```
...
 TermVector B1(u,omega1,x_1);       // TermVector on omega1
```

```
TermVector B2(u,omega2,x_1);      // TermVector on omega2
TermVector B=merge(B1,B2);        // merging
```


On dofs shared by the two domains, the merged value is those of the first domain. This behaviour is different from the addition of two `TermVector` living on different domains where the value of shared dofs is the addition of the values.

Inner/hermitian product and standard norms are provided:

```
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:

```
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:

```
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
TV.setValue(Sigma,0.);              // set to 0 the values on Sigma
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:

```
...
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);

Summary of main TermVector operations

A function marked ◇ means that its usage is restricted to single unknown TermVector.

• Constructors

`TermVector(LinearForm,[EssentialCondition],[option],[option],...[name])`	-> TermVector
`TermVector(Unknown, Domain, value, [name])◇`	-> TermVector
`TermVector(Unknown, Domain, Function, [name])`	-> TermVector
`TermVector(Unknown, Domain, SymbolicFunction, [name])◇`	-> TermVector
`TermVector(Unknown, Domain, VariableName, [name])◇`	-> TermVector
`TermVector(TermVector, value, [name])◇`	-> TermVector
`TermVector(TermVector, [TermVector], Function, [name])◇`	-> TermVector
`TermVector(TermVector, [TermVector], SymbolicFunction, [name])◇`	-> TermVector
`TermVector(Unknown,TermVector, [TermVector], [name])◇`	-> TermVector

• Accessors

`TermVector.name()`	-> String
`TermVector.valueType()`	-> ValueType
`TermVector.size()` counted in scalar	-> Number
`TermVector.nbdofs()` counted in dofs	-> Number
`TermVector.numberOfUnknowns()`	-> Number
`TermVector.asRealVector()` ◇	-> RealVector
`TermVector.asComplexVector()` ◇U	-> ComplexVector
`TermVector.Dof(Unknown, number)`	-> Dof
`TermVector(Unknown)`	-> TermVector

• Modifiers

`TermVector.compute()`	
`TermVector.toAbs()/ toReal()/ toImag()/ toComplex()`	-> TermVector
`TermVector.setUnknown(Unknown)` ◇	

• Operations

`[scalar][*]TermVector[+-][scalar][*]TermVector...`	-> TermVector
`TermVector*TermVector` component product ◇	-> TermVector
`TermVector/TermVector` component division ◇	-> TermVector
`TermVector^p` component exponent ◇	-> TermVector
`abs/real/imag (TermVector)`	-> TermVector
`sqrt/squared (TermVector)` ◇	-> TermVector
`sin/cos/tan/sinh/cosh/tanh (TermVector)` ◇	-> TermVector
`asin/acos/atan/asinh/acosh/atanh (TermVector)` ◇	-> TermVector
`exp/log/log10 (TermVector)` ◇	-> TermVector
`norm/norm1/norm2/norminfty (TermVector)`	-> Real
`innerProduct/hermitianProduct(TermVector,TermVector)`	-> Complex
`TermVector|TermVector`	-> Complex

• Value management

`TermVector.getValue(Unknown, number)`	-> Value
`TermVector.setValue(Unknown, number, val)`	
`TermVector.setValue(number, val)` ◇	
`TermVector.setValue(Unknown, Domain, T)`	
`TermVector.setValue(Domain, T)` ◇	
`TermVector.setValue(Unknown, Domain, Function)`	
`TermVector.setValue(Domain, Function)` ◇	
`TermVector.setValue(Unknown, Domain, TermVector)`	
`TermVector.setValue(Domain, TermVector)` ◇	
`TermVector.evaluate(Unknown, Point)`	-> Value
`TermVector.evaluate(Point)` ◇	-> Value
`TermVector(Unknown, Point, T)`	-> T
`TermVector(Point, T)` ◇	-> T

• Domain operations

`TermVector.mapTo(GeomDomain, Unknown,[errOutDom]`	-> TermVector
`merge(TermVector, TermVector)`	-> TermVector
`TermVector|Domain`	-> TermVector
`normalsOn(GeomDomain, Unknown)`	-> TermVector
`setColor(GeomDomain, TermVector, ColoringRule)`	

• Output

`ostream << TermVector`
`TermVector.print([ostream])`
`saveToFile(FileName, TermVector, [termVector], [IOFormat], [withDomains])`
`plot(TermVector, [IOFormat])`

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:

```
//no essential condition
TermMatrix(BilinearForm, opt1, opt2, opt3, name);
//same essential condition on unknown and test function
TermMatrix(BilinearForm, EssentialConditions, opt1, opt2, opt3, name);
//diffrenet essential conditions on unknown and test function
TermMatrix(BilinearForm, EssentialConditions, EssentialConditions, opt1,
    opt2, opt3, name);
```

`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, _nnodes=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 symetry 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 (_coo). 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",_coo);
```

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");
...
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 !

TermMatrixs may be combined using standard algebraic operators (+=, -=, *=, /=, +, -, *, /)

```
BilinearForm kuv=intg(omega,grad(u)|grad(v)),
             muv=intg(omega,u*v),
             mguv=intg(sigma,u*v);
TermMatrix K(kuv,"A"),
           M(muv,"M"),
           Mg(mguv,"Mg");
M*=3;
K+=M;
TermMatrix A=K-3*M+Mg;
```

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 :

```
BilinearForm auv=kuv-3*muv+mguv;
TermMatrix A(auv,"A");
```

Regarding memory consuming and time performance, this method is better.

Product of TermMatrix and TermVector are provided using the * operator:

```
TermMatrix A(auv,"A");
```

```
TermVector L(luv,"L");
TermVector AL=A*L;
```

Matrix and vector must have compatible unknowns but some may be omitted (void blocks are ignored).

 When test function is used to construct matrix and vector, this compatibility rule in product `TermMatrix` × `TermVector` implies that the matrix column unknowns should be the same as vector unknowns. This rule may be too boring. It is the reason why a permissive behaviour is allowed : `TermVector` unknowns may be the dual unknowns of test functions, in other words the matrix row unknowns. It requires that test functions are declared as dual of unknowns !

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` .

```
...
//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");
//contruct DtN TermMatrix using matrix product
BilinearForm euv=intg(sigmaP,phiP*v),
             fuv=intg(sigmaP,u*phiP);
TermMatrix E(euv,"E"), F(fuv,"F");
TermMatrix L(phiP, sigmaP, lambda,"L"); //diagonal matrix
TermMatrix ELF=E*L*F;
TermMatrix A2=A-i*ELF;
```

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` whith 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 ot 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 adressing up to four nodes :

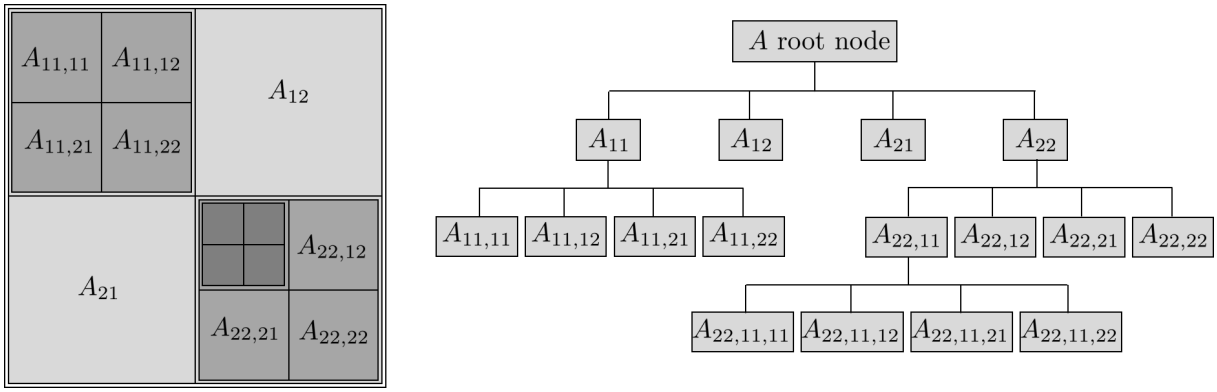
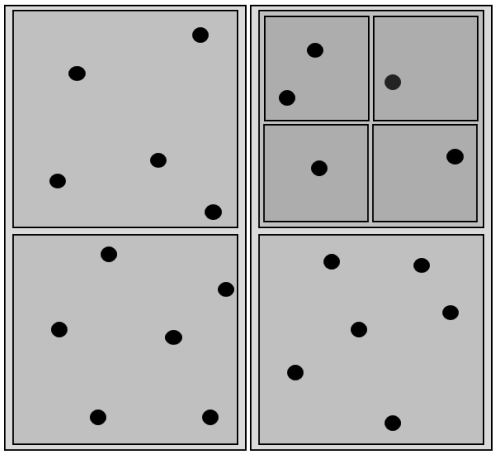


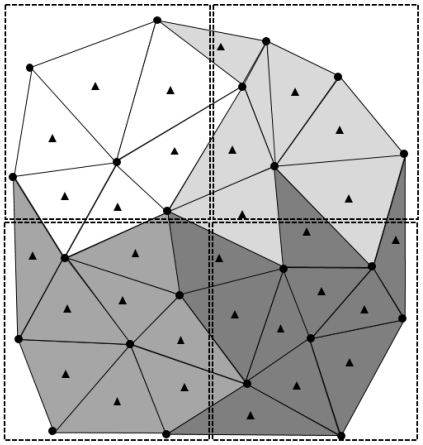
Figure 7.1: Hierarchical matrix

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



Clustering of triangles using centroids

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:

$$\operatorname{diam}(B_r) \leq 2\,\eta\,\operatorname{dist}(B_r, B_c).$$

Default value of η 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:

$$U\,D\,V^*$$

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++:

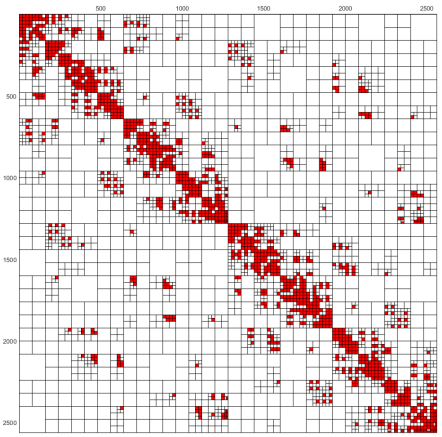


Figure 7.2: HMatrix with a sphere cluster, non admissible blocks are in red

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 bilinearform describing the BEM term : the `HMatrixIM` object that can be handled as follows

```
HMatrixIM him(clmeth, minrow, mincol, hmapp, rank, im);
```

or

```
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.

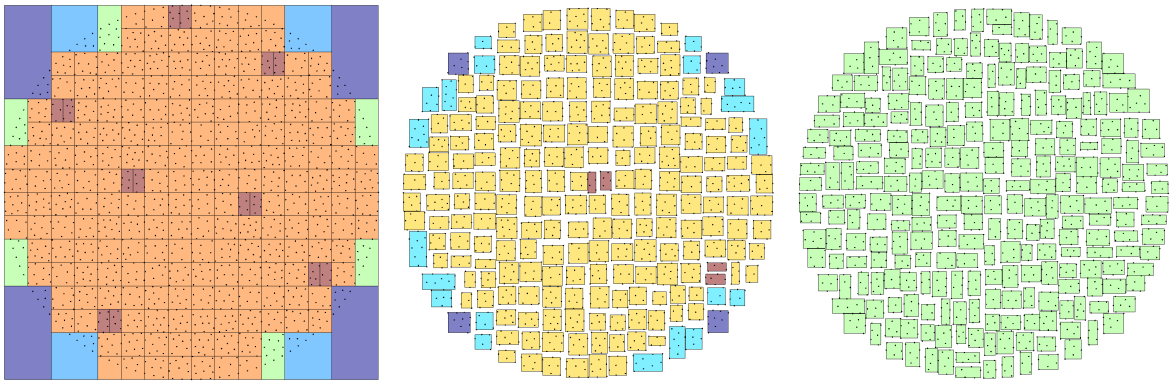


Figure 7.3: clustering of a disk mesh using regular (left), bounding box (middle), cardinality (right) bisection methods.

and the figure 7.4 shows the cluster get when using the kdtree (quadtree in 2D) methods with at most 5 nodes by box.

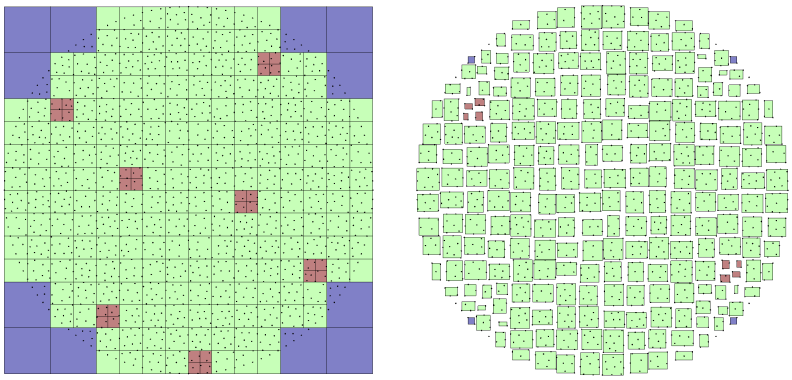


Figure 7.4: clustering of a disk mesh using kdtree methods - `_uniformKdtree` (left) and `_nonuniformKdtree`(right).

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:

```
//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,_acaplus,0.00001,ssim);
//compute single layer matrix
BilinearForm alf=intg(omega,omega,u*G1*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 with 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:

```
...
HMatrixIM him(_cardinalityBisection, 20, 20,_acaplus,0.00001,ssim);
BilinearForm alf=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.

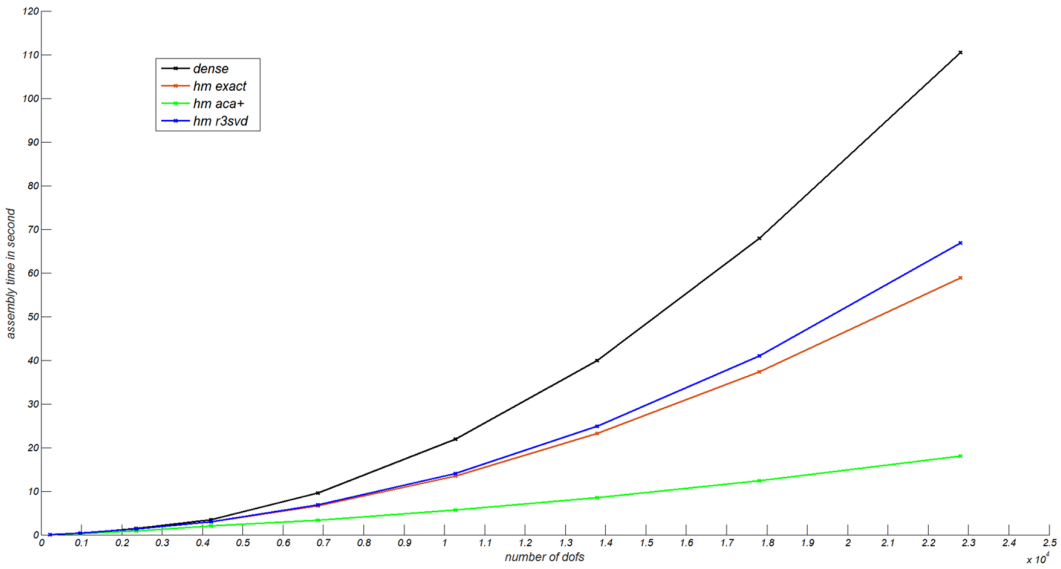


Figure 7.5: assembly time for different HMatrix approximate methods

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!

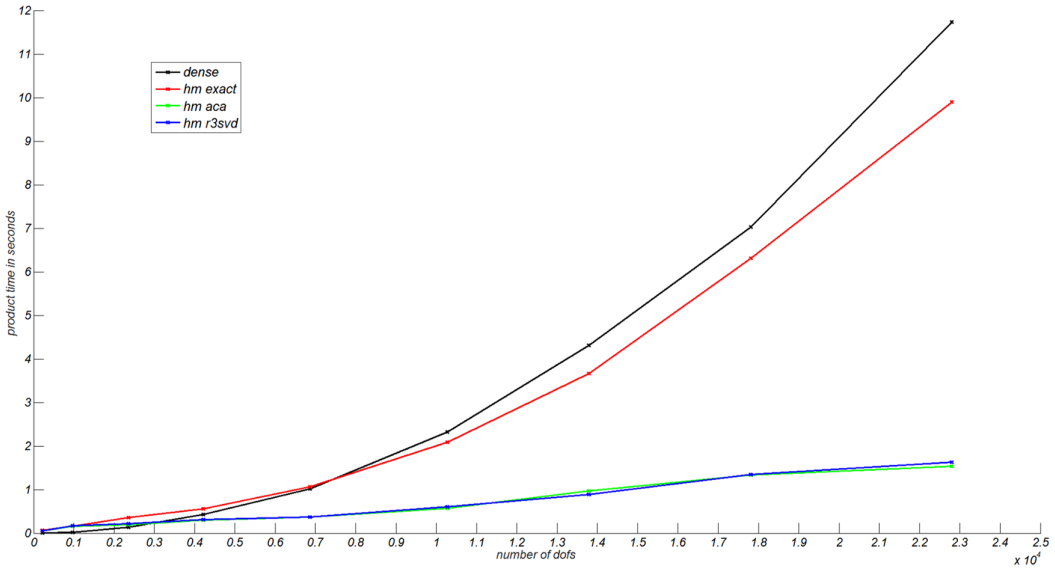


Figure 7.6: matrix/vector product time for different HMatrix approximate methods

HMatrix supports the matrix/vector product so it may be used with iterative methods but it is not supported by direct linear solvers !

7.1.4 Projector

Sometimes, it may be useful to project an element of one FE space, say V to an other FE space, say W. XLiFE++ deals with projection based on a bilinear form, say $a(.,.)$, defined on both projection spaces. The projection $w \in W$ of $v \in V$ is the solution of

$$a(w, \tilde{w}) = a(v, \tilde{w}) \;\; \forall \tilde{w} \in W.$$

Let $(w_i)_{i=1,n}$ a basis of W and $(v_i)_{i=1,m}$ a basis of V, the above problem is equivalent to the matrix problem:

$$\mathbb{A}\, W = \mathbb{B}\, V$$

where $\mathbb{A}_{ij} = a(w_j, w_i)$, $\mathbb{B}_{ij} = a(v_j, w_i)$, $v = \sum_{i=1,m} V_i\, v_i$ and $w = \sum_{i=1,n} W_i\, w_i$. The bilinear form should be symmetric and positive on the space W in order to the matrix $\mathbb{A}$ be invertible. The most simple example is the L^2 projection related to the bilinear form:

$$a(w, \tilde{w}) = \int_\Omega w\, \tilde{w}\, d\Omega.$$

Projection operations are related to the `Projector` class. Let us see how to construct such `Projector` object :

```
Mesh mesh2d(Rectangle(_xmin=0,_xmax=1,_ymin=0,_ymax=1,_nnodes=6,
                      _side_names="Gamma"), _triangle,1,_structured);
Domain omega=mesh2d.domain("Omega"), gamma = mesh2d.domain("Gamma");
// create some spaces
Space V1(omega,_P1,"V1",false);    Unknown u1(V1,"u1"); TestFunction
    v1(u1,"v1");
Space V2(omega,_P2,"V2",false);    Unknown u2(V2,"u2");
Space N1(omega,_N1_1,"N1",false); Unknown n1(N1,"n1");   //Nedelec FE
```

```
// create some L2 projectors
Projector P2toP1(V2,V1,_L2Projector,"P2toP1");
Projector N1toP1(N1,1,V1,2,_L2Projector,"N1toP1");
```

Note the particular construction of the N1 to V1 $\times$ V1 projector. Because the space V1 $\times$ V1 does not exist, you have to give the size of the vector unknowns (1 for N1, 2 for P1) related to spaces.
The projector types available are `_L2Projector, _H1Projector, _H10Projector` associated respectively to the L^2, H^1 and H_0^1 inner product. Be cautious, H^1 and H_0^1 projectors are not consistent with some FE spaces (e.g. P0)! It is also possible to use your own bilinear form :

```
BilinearForm myblf=intg(gamma,u1*v1)+intg(omega,u1*v1);
Projector P1toP2(V1,V2,myblf,"P1toP2");
```

It may happen that you want restrict your projection to a subdomain:

```
Projector P2toP1_Gamma(V2,V2,gamma,_L2Projector,"P2toP1_Gamma");
```

When a `Projector` is constructed, the matrix $\mathbb{A}$ and $\mathbb{B}$ are computed and the matrix $\mathbb{A}$ is factorized. So you can compute the projection of some `TermVector`'s:

```
TermVector B2(u2,omega,fx2,"B2"); // fx2 user function
TermVector B1=P2toP1(B2,u1);
```

By specifying the unknown `u1` as second argument, the result `B1` will have `u1` as unknown. It is also possible to omit the unknown argument or to pass the TermVector result as argument:

```
TermVector B1=P2toP1(B2);
TermVector B1b;
P2toP1(B2,B1b);
```

When no unknown is given, XLiFE++ will choose the first unknown that has been defined on result space.

You can also compute some standard projections without managing in a explicit way a `Projector` object:

```
TermVector B1=projection(B2,V1);
```

In that case the `Projector` object is constructed in the back and kept in memory.


You can only project a single unknown `TermVector`. If your `TermVector` has multiple unknowns, extract the part related to the unknown of interest before projection.

Sometimes, it may be interesting to build the matrix $\mathbb{A}^{-1}\mathbb{B}$:

```
TermMatrix P21 = P2toP1.asTermMatrix(u2,u1,"P21");
```

To save memory, the original matrices $\mathbb{A}$ and $\mathbb{B}$ are destroyed. Be careful, the matrix $\mathbb{A}^{-1}\mathbb{B}$ is a dense matrix (column dense storage), so it may waste a lot of memory.

Finally, as almost objects of XLiFE++, you can print a Projector

```
P2toP1.print(cout);
cout<<P2toP1;
```

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:

```
TermMatrix LD; // Create a new TermMatrix to store factorized result
ldltFactorize(A, LD); // LDLt-Factorize the TermMatrix
```

Then the linear system is solved with a very simple code

```
TermVector X = factSolve(LD, B);
```

The `TermVector` U is returned as a solution of the solver. Or a `TermMatrix` can be factorized into LU before being solved

```
TermMatrix LD; // Create a new TermMatrix to store factorized result
luFactorize(A, LD); // LU-Factorize the TermMatrix
TermVector X = factSolve(LD, B);
```

Factorisation and solving may be called in one time:

```
TermVector X = luSolve(LD, B);
```

The available factorisations and direct solvers are

- LDLt factorisation and solver (for symmetric matrix): functions `ldltFactorize`, `ldltSolve`
- LDLstar factorisation and solver (for self-adjoint matrix): functions `ldtstarFactorize`, `ldlstarSolve`
- LU factorisation and solver (for any matrix): functions `luFactorize`, `luSolve`
- umfpack factorisation and solver (for any matrix) if umfpack is installed and configured: `umfpackFactorize`, `umfpackSolve`
- gauss elimination with pivoting for matrix stored in dense format: `gaussSolve`
- Schur method on 2x2 `TermMatrix` : `schurSolve`
- lapack solver for matrix stored in dense format if lapack is installed and configured: `lapackSolve`
- magma solver for matrix stored in dense format if magma lib is available: `magmaSolve`

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 `directSolve` which performs tests before calling the well adapted method:

```
TermMatrix Af;     // create a new TermMatrix to store factorized result
factorize(A, Af); // factorize the TermMatrix
TermVector X = factSolve(Af, B); //solve factorized linear system

TermVector X = directSolve(A, B); //same in one call
```

The behaviour of`directSolve` is the following:

- if matrix is dense : use lapack solver if available else use XLiFE++ gauss solver
- if matrix is sparse (compressed or skyline) use umfpack if available else use XLiFE++ factorization method

Note that direct solver may induce storage conversion so the `TermMatrix` storage may be modified. if you want not, specify `true` as last argument of solver functions:

```
TermVector X = directSolve(A, B, true);  //keep original matrix
```

The right hand side is never modified.

 umfpack solver is always faster than XLiFE++ solvers but lapack solver may be slower than the XLiFE++ gauss solver if non optimized lapack-blas libraries are used.

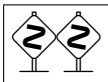
Most of solvers support multiple right hand sides given as a `TermVectors` or a `std::vector<TermVector>`:

```
TermVectors Bs;
...
TermVectors Xs = directSolve(A, Bs);
```

The `factSolve` and `directSolve` functions can also be used with a `TermMatrix` right hand side, say $\mathbb{B}$. Thus they produce a `TermMatrix` which is $\mathbb{A}^{-1}\mathbb{B}$. Even both the matrices $\mathbb{A}$ and $\mathbb{B}$ are sparse matrix, the matrix $\mathbb{A}^{-1}\mathbb{B}$ is not sparse. It is stored using a column dense storage.

```
TermMatrix invM1M2=directSolve(M1,M2,_keep);
TermMatrix Id(M1,_idMatrix,"Id");
TermMatrix invM1=directSolve(M1,Id,_keep);
TermMatrix invM1=inverse(M1);
```

Note that `inverse(M1)` is strictly equivalent to `directSolve(M1,Id,_keep);`.

 Up to now, the usage of `factSolve`, `directSolve` functions with a `TermMatrix` as right hand side and `inverse()` is restricted to single unknown `TermMatrix`.

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`:

```
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:

```
TermVector U = iterativeSolve(A, B, _solver=_cg); // Solve with default
    initial guest X0=0
```

The available solvers (through their keys to the _solver parameter) are: _bicg, _bicgstab, _cg, _cgs, _gmres and _qmr.
There are shortcuts specific to each solver:

```
TermVector U = cgSolve(A, B);
```

The available functions are `bicgSolve`, `bicgStabSolve`, `cgSolve`, `cgsSolve`, `gmresSolve`, `qmrSolve`.They all call the general function `iterativeSolve`. All these functions take parameters in the following orders:

1. the matrix A (`TermMatrix`)
2. the right hand side B (`TermVector`)
3. optionally the initial guess X0 (`TermVector`)
4. optionally the preconditioner P (`PreconditionerTerm`)
5. optionally one or more keyvalue parameters, among the following:

   _solver Only for routine `iterativeSolve`. 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.

   _name the name of the `TermVector` solution computes by the routines. Default value is "U".

   _krylovDim Only for routine `gmresSolve`, the krylov dimension.

An advanced use of solvers would be to instantiate an iterative solver object and call it using operator ():

```
CgSolver mySolver;                    // Define an iterative solver object
TermVector U = mySolver(A,B,X0);      // Solve the system with initial guess X0
```

For objects `BicgSolver`, `BicgStabSolver`, `CgSolver`, `CgsSolver`, `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 `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).

```
CgSolver mySolver(1.e-04, 20);
TermVector U = mySolver(A,B,X0); // 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 **??**) ; as far as possible, the latter should be preferred.

In the following, we will denote the problems using the generic form:

- $A\,x = \lambda\,x$, for a *standard* eigenvalue problem,
- $A\,x = \lambda\,B\,x$, for a *generalized* eigenvalue problem.

The couple (λ, x) is called an eigen pair, that consists of an eigenvalue λ 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:

```
EigenElements ees = eigenSolve(A);    // standard eigenvalue problem
EigenElements eeg = eigenSolve(A, B); // generalized eigenvalue problem
```

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 `eigenSolve` automatically selects ARPACK if it is available, or the internal solver otherwise.

Remark.
The user may choose himself by adding the argument `_solver=_intern` *or* `_solver=_arpack`.

If they are present, the other parameters, given in the form "**_key = value**", are checked and passed to the specific solver. Moreover, when ARPACK is used, some of them may be *modified* to benefit from experience feedback. Also, ARPACK requires the RHS matrix B to be hermitian to ensure the convergence of the computation. When this does not seem to be the case, the original generalized problem is automatically *transformed* into a standard problem. Whenever such a decision is made or a parameter is modified, an information message is printed on the terminal and in the main print file of XLiFE++.

The user has always direct access and full control over the parameters (which are *never modified in this case*) by using the specific functions: the function `eigenInternSolve` provides a direct access to the built-in solver, while `arpackSolve` uses ARPACK. With these functions, the statements in the previous example would become:

```
        // specific call to internal engine
EigenElements eesi = eigenInternSolve(A);     // standard eigenvalue problem
EigenElements eegi = eigenInternSolve(A, B); // generalized eigenvalue
    problem

        // specific call to Arpack engine
EigenElements eesa = arpackSolve(A);          // standard eigenvalue problem
EigenElements eega = arpackSolve(A, B);       // generalized eigenvalue
    problem
```

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:

```
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:

```
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:

```
saveToFile("EV", eeg, _vtk);
```

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 σ used in the spectral transformation in order to scan a portion of the spectrum around σ.

_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:

```
Number nev = 20;
EigenElements ee = 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:

```
Complex sig = (2.5, 1.);
EigenElements ee = 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:

Value	Description
BE	eigenvalues from both ends of the spectrum
LA	eigenvalues with largest algebraic value
SA	eigenvalues with smallest algebraic value
LM	eigenvalues with largest magnitude
SM	eigenvalues with smallest magnitude
LR	eigenvalues with largest real part
SR	eigenvalues with smallest real part
LI	eigenvalues with largest imaginary part
SI	eigenvalues with smallest imaginary part

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 σ used in the spectral transformation in order to scan a portion of the spectrum around σ.

_mode (enumeration) 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.

_convToStd parameter specified to force the conversion of a generalized problem into a standard one. This key does not take any value: it is present or absent (any assigned value is ignored).

_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 (any assigned value is ignored).

_verbose (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:

Kind of problem	Computational mode	Relevant parameters
Standard, real symmetric, nonsymmetric or complex	Regular	`_which`
	Shift and invert	`_sigma`
Generalized real symmetric	Regular	`_which`
	Shift and invert	`_sigma`
	Buckling	`_sigma`, `_mode=_buckling`
	Cayley	`_sigma`, `_mode=_cayley`
Generalized real nonsymmetric	Regular	`_which`
	Real shift and invert	`_sigma`
	Complex shift and invert (Re)	`_sigma`, `_mode=_cshiftRe`
	Complex shift and invert (Im)	`_sigma`, `_mode=_cshiftIm`
Generalized complex	Regular	`_which`
	Shift and invert	`_sigma`

Hypotheses. 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 bukling 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 section **Advanced usage of** ARPACK below). When A is real nonsymmetric and σ 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 $\Re\big((A - \sigma B)^{-1}\big)$; it should be set to **_cshiftIm** to select $\Im\big((A - \sigma B)^{-1}\big)$, and in this case obviously, the imaginary part of σ should not be nul.

The parameter **_convToStd** can be used to convert the generalized problem $A\,x = \lambda\,B\,x$ into the standard one $B^{-1}\,A\,x = \lambda\,x$. This feature increases the number of computational modes available. This is done internally with the help of a so-called user-class in a way described in the section **Advanced usage of** ARPACK below.

When the function `eigenSolve` is called to solve a generalized problem, tests are performed to check the hypotheses given above. Thus, if it happens that they do not seem to be fulfilled, the conversion into a standard problem is automatically done and an information message is printed.

In the same spirit, **_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 of** ARPACK 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:

```
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 Armoldi 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 computation with ARPACK, one can inquire about informations related to this *last* computation. The simplest way to get these informations is to call the function

```
string_t arEigInfos();
```

which gathers the main informations into a string and returns it. This string can then be printed out.
To do that, `arEigInfos` calls external 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:

```
bool arParametersDefined();
```

which returns `true` if all internal variables were correctly defined, `false` otherwise.

```
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.

```
int arGetMaxit();
```

which returns the maximum number of Arnoldi update iterations allowed.

```
int arGetMode();
```

which returns the computational mode used as described in the following table:

Value	Mode
1	regular mode (standard problems)
2	regular inverse mode (generalized problems)
3	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 $\Re\big((A-\sigma B)^{-1}\big)$
4	buckling mode (real symmetric generalized problems) or shift and invert mode with complex shift and the operator is $\Im\big((A-\sigma B)^{-1}\big)$ (real nonsymmetric generalized problems)
5	Cayley mode (real symmetric generalized problems)

```
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 σ 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 $[0, \pi]$, with 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 and printed, as long as the converged eigenvalues and the corresponding eigenvectors.

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

int main() {
  using std::cout;
  using std::endl;
  init(); // mandatory initialization of XLiFE++

  cout << " Eigenvalues of the 1D Laplace operator with Neumann
     conditions\n";
  cout << "
     ==========================================================\n";

  int nbint=1; // number of intervalls
  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 nev first eigenvalues with smallest magnitude with Arpack
  int nev = 8;
  EigenElements areigs = arpackSolve(S,M, _nev=nev, _which="SM");

  cout << arEigInfos();

  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:

```
 Eigenvalues of the 1D Laplace operator with Neumann conditions
 ==============================================================
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 eigenvalues (requested / converged): 8 / 8
 Computational mode: regular inverse mode (generalized problem)
 Part of the spectrum requested: SM
 Problem size = 61,  Tolerance = 1.11022e-16
 Nb_iter / Nb_iter_Max = 443 / 800,  Number of Arnoldi vectors = 17
Eigenvalues :
1.6569854567517169e-10
0.99999999999869593
4.0000000000087512
9.0000000000560245
16.000000000016872
25.000000000006708
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:

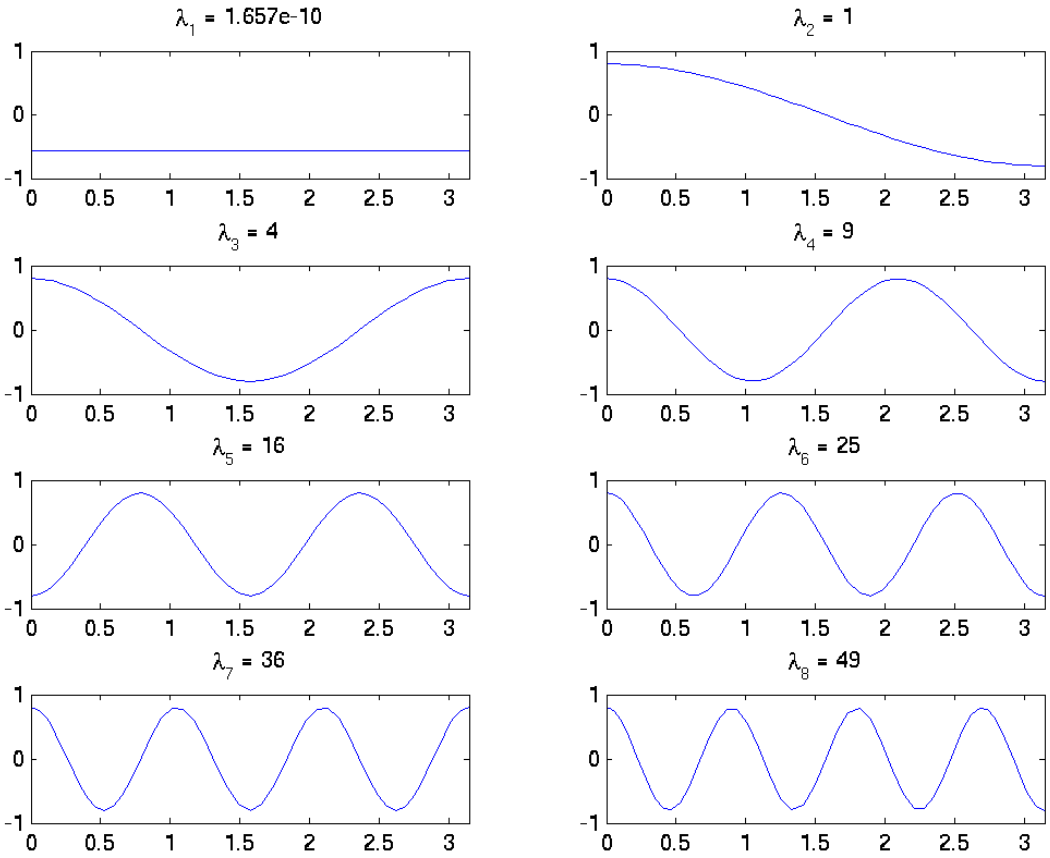


Figure 7.7: First eigenvectors of Laplace operator on a segment with Neumann conditions.

7.3.4 Advanced usage of ARPACK

The way the function `arpackSolve` is to be used, as presented so far, is sufficient if the operator A can be expressed as a linear combination of `TermMatrix` objects, or more frequently as a unique `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.

Remark.
This strategy is used internally in XLiFE++ *when the function* `eigenSolve` *is called to transform a generalized problem into a standard one when the matrix B is not hermitian.*

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. In order to benefit from the following paragraphs, the user should know some fundamentals of the C++ programming language.

♠ *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 `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:

Kind of problem		**Computational mode**	**Matrix-vector products**	**Member fcts to use**
Standard	all	Regular	$y \leftarrow A\,x$	MultOPx
		Shift and invert	$y \leftarrow (A - \sigma\,Id)^{-1}\,x$	MultOPx
Generalized	real symmetric	Regular	$y \leftarrow B^{-1}\,A\,x$; $x \leftarrow A\,x$ $y \leftarrow B\,x$	MultOPx MultBx
		Shift and invert	$y \leftarrow (A - \sigma B)^{-1}\,x$ $y \leftarrow B\,x$	MultOPx MultBx
		Buckling	$y \leftarrow (A - \sigma B)^{-1}\,x$ $y \leftarrow A\,x$	MultOPx MultBx
		Cayley	$y \leftarrow (A - \sigma B)^{-1}\,x$ $y \leftarrow A\,x$ $y \leftarrow B\,x$	MultOPx MultAx MultBx
	real nonsymmetric	Regular	$y \leftarrow B^{-1}\,A\,x$ $y \leftarrow B\,x$	MultOPx MultBx
		Real shift and invert	$y \leftarrow (A - \sigma B)^{-1}\,x$ $y \leftarrow B\,x$	MultOPx MultBx
		Complex shift and invert (real part)	$y \leftarrow \Re((A - \sigma B)^{-1})\,x$ $y \leftarrow A\,x$ $y \leftarrow B\,x$	MultOPx MultAx MultBx
		Complex shift and invert (imag part)	$y \leftarrow \Im((A - \sigma B)^{-1})\,x$ $y \leftarrow A\,x$ $y \leftarrow B\,x$	MultOPx MultAx MultBx
	complex	Regular	$y \leftarrow B^{-1}\,A\,x$ $y \leftarrow B\,x$	MultOPx MultBx
		Shift and invert	$y \leftarrow (A - \sigma B)^{-1}\,x$ $y \leftarrow B\,x$	MultOPx MultBx

In order to help to 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 facilitate 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:

```
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

```
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 $S\,x = \lambda M\,x$. Now, assume we want to write this problem as a standard eigenvalue problem $M^{-1}\,S\,x = \lambda\,x$, 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:

```
class StdNonSym: public ARStdFrame<real_t> {
public:
   //! constructor
   StdNonSym(TermMatrix& S, TermMatrix& M);

   //! destructor
   ~StdNonSym(){ delete fact_p; }

  //! matrix-vector product required : y <- inv(M)*S * x
   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;

}; // end of Class StdNonSym ========================================
```

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 mentionned in the previous table.

Since we plan to use the regular computational mode, the function MultOPx should compute the result y of $M^{-1}\,S\,x$ for a given x. This is done in two steps: first compute $z = S\,x$, second solve $M\,y = z$ 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 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:

```
/*!
  Assumptions (not checked) :
    S real
    M real symmetric positive definite
*/
StdNonSym::StdNonSym(TermMatrix& S, TermMatrix& M)
: ARStdFrame(S), matS_p(&S.matrixData()->getLargeMatrix<real_t>()),
                 matM_p(&M.matrixData()->getLargeMatrix<real_t>()) {

  fact_p = newSkyline(matM_p);
  ldltFactorize(*fact_p);
}
//!    Matrix-vector product y <- inv(M)*S * x
void StdNonSym::MultOPx (real_t *x, real_t *y) {
   array2Vector(x, lx);
   std::vector<real_t> Sx(GetN());
   multMatrixVector(*matS_p, lx, Sx);
// Solve linear system. Matlab equivalent: ly = matM_p \ Sx;
   (fact_p->ldltSolve)(Sx, ly); // store the solution into ly
   vector2Array(ly, y);
}
```

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 $My = Sx$ 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 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

```
ArpackProb(const ARInterfaceFrame<Real>& 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

```
ArpackProb(const ARInterfaceFrame<Complex>& usrcl, int nev, const char*
    which);
```

- Constructors for *shifted* computational mode (for standard or generalized problems)

1. • real symmetric case:
   – for standard problems: shift and invert mode (default)
   – for generalized problems: shift and invert mode (default), buckling and Cayley mode
   • real nonsymmetric case:
   – for standard or generalized problems: (real) shift and invert mode

```
ArpackProb(const ARInterfaceFrame<Real>& usrcl, int nev, bool sym, double
    sigma, char cMode = 'S');
```

As above, the argument **sym** tells if the algorithm designed for the symmetric case (`true`) or nonsymmetric case (`false`) should be used. The argument **sigma** is the value of the shift (real number here). The buckling and Cayley modes can be selected by giving the argument **cMode** the character value 'B' or 'C' respectively.

Note: for standard eigenvalue problems, this last argument (computational mode **cMode***) is irrelevant and thus has not to be specified.*

2. real nonsymmetric case, for generalized problems only: complex shift and invert mode

```
ArpackProb(const ARInterfaceFrame<Real>& usrcl, int nev, double sigmaR,
    double sigmaI, char cMode = 'R');
```

The shift is given by both its real part, **sigmaR**, and its imaginary part, **sigmaI**. The argument **cMode** should be set to 'R' to select $\Re\big((A-\sigma B)^{-1}\big)$; it should be set to 'I' to select $\Im\big((A-\sigma B)^{-1}\big)$ (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

```
ArpackProb(const ARInterfaceFrame<Complex>& usrcl, int nev, double sigmaR,
    double sigmaI = 0.0);
```

The shift is given by both its real part, **sigmaR**, and its imaginary part, **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 *A full example.* and do the following:

1. copy the declaration of the user class `StdNonSym`, followed by its implementation as given above, just before the `main` function,

2. replace the call to `arpackSolve`:

```
EigenElements areigs = arpackSolve(S,M, _nev=nev, _which="SM");
```

   by the three lines:

```
StdNonSym usrcl(S,M);
ArpackProb Arpb(usrcl,nev,"SM",false); // false means "use the
    nonsymmetric algorithm"
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 real standard 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:

```
for (int i = 0; i < nconv; i++) { cout << areigs.value(i+1) << endl; }
```

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 eigenvalues (requested / converged): 8 / 8
 Computational mode: regular mode (standard problem)
 Part of the spectrum requested: SM
 Problem size = 61,  Tolerance = 1.11022e-16
 Nb_iter / Nb_iter_Max = 266 / 800,  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

```
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:

```
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 eigenvalues (requested / converged): 8 / 8
 Computational mode: regular mode (standard problem)
 Part of the spectrum requested: SM
 Problem size = 61,  Tolerance = 1e-15
 Nb_iter / Nb_iter_Max = 252 / 800,  Number of Arnoldi vectors = 17
Eigenvalues :
(-3.9815928332131989e-12,0)
(0.99999999998896794,0)
(3.9999999999885101,0)
(8.9999999999974225,0)
(15.999999999996653,0)
(25.000000000000909,0)
(35.999999999996525,0)
(48.999999999971514,0)
```

♠ *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`:

```
class StdComp: public ARStdFrame<complex_t> {
public:
  //! constructor
  StdComp(TermMatrix& S, TermMatrix& M);

  //! destructor
  ~StdComp(){ delete fact_p; }

 //! matrix-vector product required : y <- inv(M)*S * x
  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:

```
/*!
  Assumptions (not checked) :
    S complex
    M real symmetric positive definite
*/
StdComp::StdComp(TermMatrix& S, TermMatrix& M)
: ARStdFrame(S), matS_p(&S.matrixData()->getLargeMatrix<complex_t>()),
                 matM_p(&M.matrixData()->getLargeMatrix<real_t>()) {

  fact_p = newSkyline(matM_p);
  ldltFactorize(*fact_p);
}
//!    Matrix-vector product y <- inv(M)*S * x
void StdComp::MultOPx (complex_t *x, complex_t *y) {
   array2Vector(x, lx);
   std::vector<complex_t> Sx(GetN());
   multMatrixVector(*matS_p, lx, Sx);
// Solve linear system. Matlab equivalent: ly = matM_p \ Sx;
   (fact_p->ldltSolve)(Sx, ly); // store the solution into ly
   vector2Array(ly, y);
}
```

One can just mention the initialization of the pointer `matS_p` to the complex data values. The factorization of M is unchanged and it should be noticed that here the Cholesky solver handles complex data.

The final step consists in modifying the initial program (*A full example.* above) by inserting the declaration and the implementation of the user class `StdComp` as given above before the `main` function, and replace the call to `arpackSolve` by the three lines:

```
  TermMatrix Sc = toComplex(S);
  StdComp usrcl(Sc,M);
  ArpackProb Arpb(usrcl,nev,"SM");
```

The first statement converts the matrix S to the complex one Sc. Then the object corresponding to the user class `usrcl` is created from the complex stiffness matrix and the real mass matrix. The intermediate object `Arpb` is created using the second constructor in the list given above. This completely defines the ARPACK problem: the user class derives from `ARStdFrame<complex_t>`, so it is a complex standard 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) | grad(v), using 1 threads :  done
computing FE term intg_Omega u * v, using 1 threads :  done

 Number of eigenvalues (requested / converged): 8 / 8
 Computational mode: regular mode (standard problem)
 Part of the spectrum requested: SM
 Problem size = 61,  Tolerance = 1.11022e-16
```

```
 Nb_iter / Nb_iter_Max = 235 / 800,  Number of Arnoldi vectors = 17
Eigenvalues :
(1.6895427007126359e-10,-5.3921458890663075e-11)
(1.0000000000208538,-1.1338949016552641e-13)
(4.0000000000304183,7.7808203477527148e-12)
(9.0000000000377973,1.1273274779629001e-11)
(16.000000000012292,2.7943890983902115e-11)
(24.999999999985345,1.3070479010254708e-11)
(36.00000000008869,-3.2309836334379703e-11)
(49.000000000591157,-2.8194647537903334e-10)
```

♠ *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:

```
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 real symmetric generalized 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:

```
/*!
  Assumptions (not checked) :
    S real symmetric
    M real symmetric positive definite
*/
GenSym::GenSym(TermMatrix& S, TermMatrix& M)
: ARGenFrame(S), matS_p(&S.matrixData()->getLargeMatrix<real_t>()),
                 matM_p(&M.matrixData()->getLargeMatrix<real_t>()) {

  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 nearly identical to the one of `StdNonSym`. The function `MultOPx` computes the same product $M^{-1}S$; it additionally stores Sx in $\mathbf{x}$ 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:

```
  GenSym usrcl(S,M);
  ArpackProb Arpb(usrcl,nev,"SM");
  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 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 real generalized 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 eigenvalues (requested / converged): 8 / 8
 Computational mode: regular inverse mode (generalized problem)
 Part of the spectrum requested: SM
 Problem size = 61,  Tolerance = 1.11022e-16
 Nb_iter / Nb_iter_Max = 443 / 800,  Number of Arnoldi vectors = 17
Eigenvalues :
1.6569854567517169e-10
0.99999999999869593
4.0000000000087512
9.0000000000560245
16.000000000016872
25.000000000006708
36.000000000077279
```

```
48.999999999728566
```

8 Post processing and outputs

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 (Γ). This potential is not easy to interpret, so the final step of a BEM is often the reconstruction of the field outside Γ. 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 Γ 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 $*$, $|$, $\wedge$ or $\%$. The previous exemple corresponds to $opk = id$, $opu = id$ and $\otimes = *$. To deal with such integral representation, the user has to define a linear form from `intg` constructor:

```
LinearForm ri=intg(Gamma, G*q);  //default integration method

LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3);//specifying quadrature rule

IntegrationMethods ims(Gauss_Legendre,10, 1., Gauss_Legendre, 3);
LinearForm ri=intg(Gamma, G*q, ims); //specifying 2 quadrature rules
```

In these expressions, Gamma is a `Domain` object, G a `Kernel` object and q an `Unknown` object. Singular integration method is required if you intend to evaluate the integral representation at points close to the boundary Γ.

The linear form may be a linear combination of `intg` :

```
IntegrationMethods ims(Gauss_Legendre,10, 1., Gauss_Legendre, 3);
LinearForm ri=intg(Gamma, G*q, ims) + intg(Gamma, (grad_x(G)|_nx)*q, ims)
```

There are several methods to compute the integral representation.

8.1.1 Direct method

To effectively compute integral representation you have to specify the vector representing the numerical potential, a `TermVector` object (say Q) and the points where to evaluate it. There are many way to give points :

- compute at one point x :

```
LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3);
Complex val;
Point x(0,0,2);
integralRepresentation(x,ri,Q,val);
```

- compute at an explicit list of points :

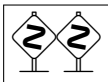
```
LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3);
Vector<Point> xs(10);
xs(1)= Point(0,0,2); ...
Vector<Complex> val(10);
integralRepresentation(xs,ri,Q,val);
```

- compute at an implicit list of points of a `Domain` object (say omega) :

```
LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3);
Vector<Point> xs;
Vector<Complex> val;
integralRepresentation(omega, ri, Q, val, xs); //val and xs are filled by
    function
```

- compute at an implicit list of node points of an interpolation on a `Domain` :

```
LinearForm ri=intg(Gamma, G*q, Gauss_Legendre,3);
TermVector U=integralRepresentation(u, omega, ri, Q); //u unknown on a
    Lagrange space
```

 In the previous syntaxes the type of output `val` has to be consistent with data. For instance, `val` is of complex type if G or Q 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.

Note that, integral representations may return a vector of vectors but not a vector of matrices. For instance, if yo want to compute the gradient of the integral representation (scalar), write :

```
Vector<Vector<Complex> > gsl;
integralRepresentation(xs, intg(Gamma,grad_x(K)*u,ims), dnU, gsl);
```

or

```
Unknown us(V,"us",2); //vector unknown !
TermVector gSL=integralRepresentation(us,omega,intg(Gamma,grad_x(K)*u,ims),
    dnU);
```

In this last form, attach to your `TermVector` a vector unknown with the dimension of the result.

8.1.2 Matrix method

There exists an other way to deal with integral representations. It consists in computing the matrix

$$R_{ij} = \int_\Gamma opk(G(x_i, y)) \otimes opu(\tau_j(y))\, dy.$$

Special functions will produce such matrices embedded in a `TermMatrix` object:

```
TermMatrix integralRepresentation(const Unknown&, const GeomDomain&,
                                  const LinearForm&);
TermMatrix integralRepresentation(const GeomDomain&, const LinearForm&);
TermMatrix integralRepresentation(const std::vector<Point>&,
                                  const LinearForm&);
```

When no domain is explicitly passed, one shadow `PointsDomain` object will be created from the list of points given. When no unknown is explicitly passed, one (minimal) space and related shadow unknown will be created to represent the row unknown.
The following example (2D diffraction problem) shows how to use this method of integral representation:

```
Number nmesh=25;
Disk disk_int(_center=Point(0.,0.,0.),_radius=1.,
              _nnodes=nmesh,_side_names="Gamma");
Disk disk_ext(_center=Point(0.,0.,0.),_radius=2.,_nnodes=2*nmesh,
              _domain_name="Omega",_side_names="Sigma");
Mesh mesh(disk_ext-disk_int,_triangle, 1, _gmsh);
Domain Gamma = mesh.domain("Gamma"), Sigma = mesh.domain("Sigma"),
    Omega=mesh.domain("Omega");
// define spaces, unknown and testfunction
Space V0(Gamma,P0,"V0",false); Unknown u0(V0,"u0"); TestFunction v0(u0,"v0");
Space V1(Omega,P1,"V1",false); Unknown u(V1,"u");
//define Kernel and integration method
Kernel H=Helmholtz3dKernel(k);
IntegrationMethods ims(Duffy,8,0.,Gauss_Legendre,6,2.,Gauss_Legendre,3);
//define forms
LinearForm lf = intg(Gamma, g*v0); //rhs linear form
BilinearForm aSL=intg(Gamma,Gamma, u0*H*v0,ims); //single layer bininear form
LinearForm lSL(intg(Gamma,H*u0,Gauss_Legendre,6)); //intg. rep. linear form
//build system and solve it
TermVector rhs(lf);
TermMatrix ASL(aSL);
TermVector uSL = gmresSolve(ASL, rhs, _tolerance=1.0e-6, _maxIt=500);
//intg rep on Omega producing a TermMatrix
TermMatrix R=integralRepresentation(u,Omega,lSL);
TermVector U=R*uSL;
```

This method is a little more time expansive than computing directly the integral representation. Thus, if there are a lot of integral representations to do with different data, it may be of interest. Obviously, it is memory consuming.

8.1.3 Kernel interpolation method

When points x are far from boundary Γ, an alternate method consists in computing IR by interpolation method. Let Ω the domain where IR is evaluated and V_Ω a Lagrange finite element space of interpolation defined on Ω. Denote $(w_i)_i$ the basis functions associated to V_Ω space. Let W_Γ a Lagrange finite element space defined on Γ 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 ($U = (u(x_i)_i)$, $Q = (q_k)_k$) :

$$U = \mathbb{G} * \mathbb{M} * Q$$

with $\mathbb{G}$ the matrix $(G(x_i, y_j))_{ij}$ and $\mathbb{M}$ the mass matrix :

$$\mathbb{M}_{jk} = \int_\Gamma \tau_j\, \sigma_k.$$

This exemple shows how it is done with XLiFE++:

```
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, G, "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 Ω 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 the objects appearing in the user main program may be printed in a simple way to the screen or into a file, using the output operator <<.

```
BilinearForm a=intg(omega, u*v);
TermMatrix A(a,"'A");
compute(A);
TermVector Un(omega,u,1,"'U");
TermVector X=A*Un;
theCout << "A*un = " << X << eol;        //print to screen and to the file
    print.txt
thePrintStream << "A*un=" << X << eol; //print to the file print.txt
```

`thePrintStream` is a XLiFE++ predefined object allowing to print into the file `print.txt` created in the current directory. `theCout` is a XLiFE++ object allowing to print both to the screen and file `print.txt`.

The verbosity of printing is only controled by the global parameter `theVerboseLevel` that can take value from 0 to the largest integer. To modify its value, use the `verboseLevel(...)` function:

```
verbosLevel(10);
...
verbosLevel(0);
...
```

When the verbose level is set to 0, nothing is printed except the errors and warnings.

 In multithreading environment, it may appear other print files (`printxx.txt`) coprresponding to outputs of each thread.

In order to print into a specific file, first create an *ofstream* object, then print to it:

```
std::ofstream out("myfile.dat"); // out is an ofstream object associated
                                 // with the file myfile.dat
out << "A*un=" << X << eol;      // print into this file
out.close();
```

8.2.2 Export `TermMatrix` and `TermVector`

We want to exploit easily the data contained in the objects produced during the computation. The objects concerned are mainly `TermVector` objects since `TermVector` is the type of nearly all the computation results of interest to the user. This may also concern `TermMatrix` objects if further postprocessing is envisaged.
As mentionned just above, printing objects using the `<<` operator may produce big files containing a lot of informations, generally used to control the different steps of the computation. But we often need to handle the data values outside of XLiFE++, either in a raw format or in a specific one corresponding to some particular software.
The `saveToFile` commands have been designed for that purpose. They exist in two main forms:

- member function : `Object.saveToFile("FileName", options);`
- external function : `saveToFile("FileName", Object, options);`

They behave slightly differently according to the kind of object they act on. Details follow:
1. `TermMatrix` object:

```
BilinearForm a=intg(omega, u*v);
TermMatrix A(a,"A"); compute(A);
A.saveToFile("A.dat",_dense);        // dense format
A.saveToFile("A.coo",_coo, true);    // coordinate format
saveToFile(A,"As.coo",_coo, true);   // works also (external function form)
```

Only two formats are available :

- dense format (*_dense* option) : all the matrix coefficients are written, line by line,
- coordinate format (*_coo* option), which corresponds to the MATLAB/OCTAVE sparse format: only non zero matrix coefficients are written in the form $i \quad j \quad a_{ij}$.

When the last argument is set to true (false by default), structure informations are added to the file name. In the previous exemple, the file name looks like `As(30_30_coo_real_scalar).coo`.

2. `TermVector` object:

```
TermVector U1, U2;
... computation of U1 and U2 ...
U1.saveToFile("U1.dat");             // raw format only
U1.saveToFile("U1.dat", true);       // raw format, structure added to name
```

With the member function form, the only available output format is raw. To select another format, the external function form should be used:

```
saveToFile("U1.dat",U1, _raw, true);// idem previous via external function

saveToFile("U.dat", U1, U2, _raw);  // two TermVectors in the same file
saveToFile("U.vtk", U1, U2, _vtk);  // idem, export in vtk format
saveToFile("U.vtu", U1, U2, _vtu);  // idem, export in vtu format
saveToFile("U.msh", U1, U2, _msh);  // idem, export in msh format
saveToFile("U.m", U1, U2, _matlab); // idem, export in Matlab/Octave format
saveToFile("U.dat", U1, U2, _xyzv); // idem, export nodes and values
```

Except the *raw* format which outputs only data values, all formats embed mesh informations. The corresponding files are intended to be read by visualization softwares:

- *vtk* and *vtu* formats are compatible with PARAVIEW,
- *msh* format is compatible with GMSH,
- *matlab* format is compatible with MATLAB and OCTAVE,
- *xyzv* format produces ascii files with *x y z v1 v2 ...* on each line.

Remark 1: Because of the geometrical informations involved, different TermVectors which are exported using one of those formats must be defined on the same space in order to be compared.

Remark 2: A mesh can also be exported in *vtk*, *msh* or *mel* format using a command bearing the same name, `saveToFile`. This can be useful for conversion from one format to another, or for visualization purpose.

```
Mesh demo = ...;
demo.printInfo();  // prints general information about the mesh to the screen

demo.saveToFile("demo.vtk", _vtk, true); // export the mesh in vtk format
demo.saveToFile("demo.msh", _msh);       // export the mesh in msh format
demo.saveToFile("demo.mel", _mel);       // export the mesh in melina format

// Idem with the external function form:
saveToFile("demo.vtk", demo, _vtk, true);
saveToFile("demo.msh", demo, _msh);
saveToFile("demo.mel", demo, _mel);
```

The last argument is optional ; ist default value is false. For the *vtk* and the *msh* formats, if this argument is true, a individual file is created for each subdomain or boundary subdomain. In this case, each filename contains the name of the corresponding subdomain.

8.3 Graphical exploitation

By itself, XLIFE++ is a finite element library and as such does not own any graphical possibilities. At present, three main softwares are targeted through the ouput formats mentionned in the previous section.

From a practical point of view, in order to obtain a graphical representation of the computation result, one has to process the output file produced by the `saveToFile` command by the corresponding software. A minimal knowledge is required in order to use GMSH and PARAVIEW. We invite the user to refer to the documentation of these softwares. Most of the computation

results shown is this documentation are produced by PARAVIEW and most of the meshes are displayed using GMSH.

The MATLAB/OCTAVE format is intended to be (easily) used as follows. We assume we have a `.m` script file, say `eigs_1_Omega.m`, containing the first vector computed and related to the domain `Omega` (maybe an eigenvector or the solution to a linear system, stored as a `TermVector` object in both cases). One has to launch MATLAB or OCTAVE, eventually change to the directory containing the `.m` file and type in `eigs_1_Omega` at the prompt (the execution of the script can also be achieved using the menus if the GUI is available). We thus automatically get several figures, one showing the mesh based on the interpolation nodes, and one for each component of each unknown. The user is then free to make further computations using the data present in memory (see below) or modify the attributes of the figures.

The curves gathered on the figure 7.7 are obtained via this procedure. The corresponding mesh is shown on figure 8.1.

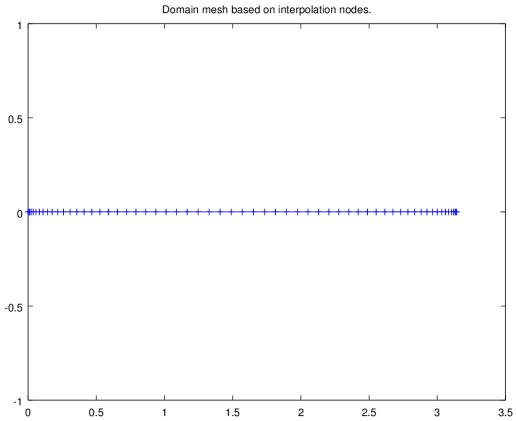


Figure 8.1: Computational domain: $[0, \pi]$.

Another example with a 2D geometry is shown on figure 8.2.

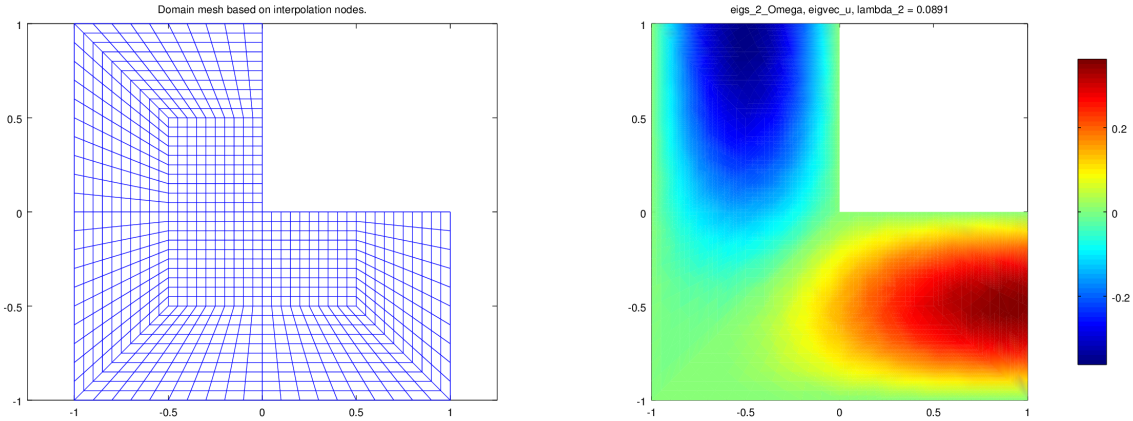


Figure 8.2: A 2D example.

The script is not interactive except for volumic data: the visualization is made by slicing the 3D domain with a plane and the user is invited to choose the cutting plane defined by a point and a vector normal to the plane, shown by a red thick line on the mesh figure. The position of the

cutting plane is updated on this figure as its definition changes and the corresponding slices are displayed in other figures.
Let's show what the command window looks like in such a case, here with OCTAVE:

```
GNU Octave, version 4.0.3
Copyright (C) 2016 John W. Eaton and others.
This is free software; see the source code for copying conditions.
 ...

>> what
M-files in directory /tmp/demo/xlifepp:

   eigs_1_Omega.m
>>
>> eigs_1_Omega
The intersection plane is defined by the point P and the orthogonal vector V.

Current value of P = 12.566      12.566      12.566
Current value of V = 1  -1   0
  1 : change P
  2 : change V
  0 : quit
Your choice (other value = no change) : 3

Current value of P = 12.566      12.566      12.566
Current value of V = 1  -1   0
  1 : change P
  2 : change V
  0 : quit
Your choice (other value = no change) : 0
Tuning suggestions:
 figure(N)
 subplot(2,2,k)
 rotate3d,   grid,   box
 view(2),  view(3),  view([Nx Ny Nz])
 shading faceted
 axis equal,  axis normal
 set(gca,'xtick',[])       or  set(gca,'xtick',[...])
 caxis([...])
 xlabel(...), ylabel(...), title(...)
 print('-dpng',eigs_1_OmegaFig2.png)
>>
```

Before the user types in 3 to answer the first question, the figure 8.3 has been displayed, showing the domain mesh and an initial cutting plane. By choosing the value 3, the user accepts the current settings and the figure 8.4 is drawn. The user then terminates by answering 0, and some hints to modify the aspect of the figures are displayed.

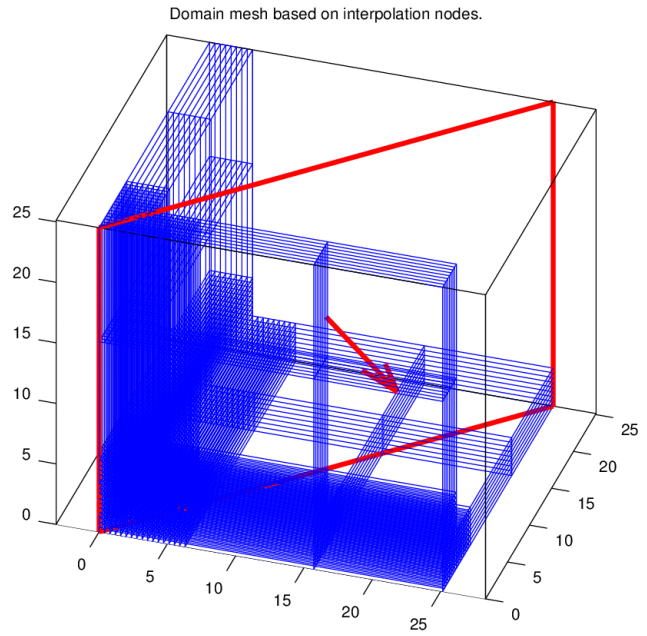


Figure 8.3: Computational domain Omega.

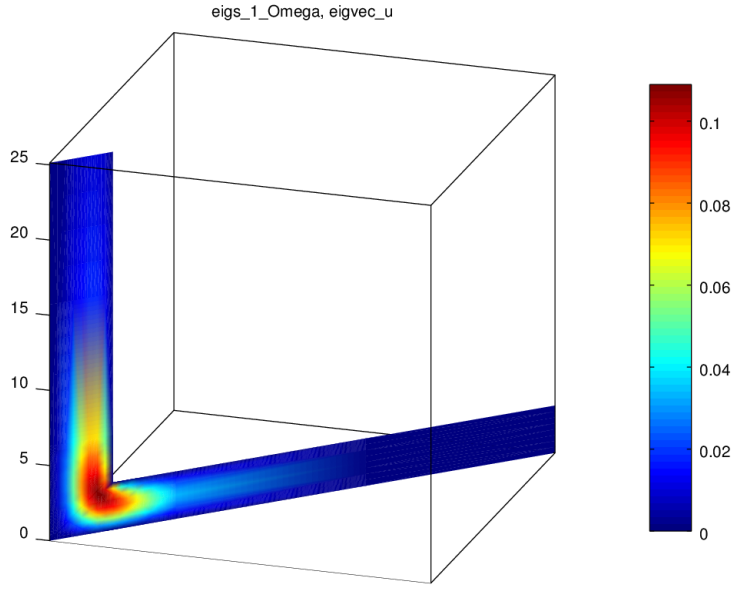


Figure 8.4: Data representation corresponding to the selected slice.

When the script has terminated, we can observe the variables present in the workspace:

```
>> whos
Variables in the current scope:

   Attr Name            Size                     Bytes  Class
   ==== ====            ====                     =====  =====
        coord        5859x3                    140616  double
        domaindim       1x1                         8  double
```

```
  domainname      1x5                          5  char
  elem         4832x8                     309248  double
  elemtype     4832x1                      38656  double
  interpDeg       1x1                          8  double
  spacedim        1x1                          8  double
  unknown         1x8                          8  char
  val          5859x1                      46872  double

Total is 66940 elements using 535429 bytes

>> quit
```

This allows the user to make any processing of his own with these data. The definitions of the variables are the following:

- `coord` (real)
  Coordinates of the interpolation nodes, one node per row. The nodes are implicitly numbered from 1 to `size(coord,1)`.

- `domaindim` (integer)
  Dimension of the domain (1, 2 or 3).

- `domainname` (string)
  Name of the domain.

- `elem` (integer)
  Array containing the lists of elements: `elem(i,:)` is an element of type `elemtype(i)`. Format of the array: one element per row, column `i` holds the i-th interpolation node, given by its number in the array `coord` above.

- `elemtype` (integer)
  Vector containing the type of each element present in the mesh, in the same order as the array `elem` above. Each type is a code number in XLiFE++'s internal codification:
  2 = point, 3 = segment, 4 = triangle, 5 = quadrangle,
  6 = tetrahedron, 7 = hexahedron, 8 = prism, 9 = pyramid.

- `interpDeg` (integer)
  Interpolation degree used during the computation.

- `spacedim` (integer)
  Dimension of the space (1, 2 or 3).

- `unknown` (string)
  1-column array containing the name of the unknowns, or their components in the vector case, one name per row.

- `val` (real or complex)
  Values corresponding to the unknowns, stored column-wise, one column per component unknown. Each row contains the value at a node, or in an element if the interpolation degree is 0, in the same order as the array `coord` or `elem` respectively. Indeed, the graphical function 'patch' used, makes the correspondence according to the number of rows of this array, which should be `size(coord,1)` or `size(elem,1)` respectively.

- `rootfn` (string)
  Filename of the calling script.

The variables `domaindim`, `domainname`, `interpDeg` and `spacedim` are defined for information or consistency check purpose. Indeed, `spacedim` should equal `size(coord,2)` and `domaindim` should be less or equal `spacedim` and be consistent with the element types.

A External libraries

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 juste 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
    path/to/CMakeLists.txt
```

 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
    -DXLIFEPP_UMFPACK_INCLUDE_DIR=path/to/UMFPACK/include/directory
    -DXLIFEPP_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 library can be obtained from the following URL: http://www.caam.rice.edu/software/ARPACK/. It requires Blas and Lapack routines, 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 instead, 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:

```
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

```
make
```


XLiFE++ uses the wrapper Arpack++. Because of its deprecation, a patch at http://reuter.mit.edu/index.php/software/arpackpatch/ needs to be applied to ensure a correct compilation. With the evolution of compilers, this patch is often not enough now. This is the reason why XLiFE++ contains its own patched release of Arpack++, used by default.

A.4 Installation of MinGW 64 bits

When you download `CodeBlocks`, the default compiler is MinGW 32bits. To use the full capabilities of XLiFE++, you may want to use a 64 bits compiler. 2 ways to download MinGW-W64:

- Download the installer from the url https://downloads.sourceforge.net/project/mingw-w64/Toolchains%20targetting%20Win32/Personal%20Builds/mingw-builds/installer/mingw-w64-install.exe?r=&ts=1522254087&use_mirror=netcologne. The installer
- Download the binaries directly from the url from the https://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win64/Personal%20Builds/mingw-builds/. You select the version you need, for instance 7.3.0. You click on

"threads-posix" directory, on one of the directories (sjlj, seh, ...). Personnally, I would choose the directory having the best download rate per week. FInally, you click on the archive to donwload it.

B Utility types in details

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:

```
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:

```
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
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
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*.

```
// 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:

```
// 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:

```
// 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:

```
// 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:

```
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:

```
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 (+=, -=, +, -, * and /), 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=p.squareDistance(q);  // square of the distance between p and q
d=squareDistance(p,q);  // alternative syntax
```

Finally, comparing points is possible using standard operators ==, !=, <, >, <= or >=. The comparison uses a tolerance factor τ 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.));
Point reflect2d(const Point& p, const Point& c, Real ux, Real uy = 0.);
//! apply a reflection3d on a Point (external)
Point reflect3d(const Point& p, const Point& c = Point(0.,0.,0.),
    std::vector<Real> u = std::vector<Real>(3,0.));
Point reflect3d(const Point& p, const Point& c, Real ux, Real uy, Real uz =
    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); // cv=[(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.,0.) [(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*i;                   // v =[1. 5. 9. 16. 25.]
Vector<Real> w=v(3,5);          // w =[9. 16. 25.]
Vector<Number> is(3);
is(1)=1;is(2)=3;is(3)=5;        // is=[1 3 5]
w=v(is);                        // w =[1. 9. 25.]
v.set(1,3,w);                   // v =[1. 9. 25. 16. 25.]
Vector<Real> z(3,0.);           // w =[0. 0. 0.]
v.set(is,z);                    // v =[0. 9. 0. 16. 0.]
```

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.]
Vector<Real> w=u+v;           // v=[3. 3. 3.]
w=2.*v;                       // v=[4. 4. 4.]
w-=2.;                        // v=[2. 2. 2.]
Complex i(0,1);               // complex number
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.)]
```

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<Real> u(3,1);        // u=[1. 1. 1.]
Vector<Complex> cv(3);      // cv=[(0.,0.) [(0.,0.) (0.,0.)]
cv=u+2.*i;                  // cv=[(1.,2.) [(1.,2.) (1.,2.)]
cv*=3.;                     // cv=[(3.,6.) [(3.,6.) (3.,6.)]
```

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.norminfty();              // 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(3,_zeroMatrix);     // a 3×3 zeros matrix
Matrix<Real> r1(3,_onesMatrix);     // a 3×3 ones matrix
Matrix<Real> rI(3,_idMatrix);       // a 3×3 identity matrix
Matrix<Real> rH(3,_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 :

```
Matrix<Real> A(2,2,1);    //a 2×2 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()`):

```
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; 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 4; 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:

```
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
Real n2=norm2(A);      //Frobenius norm
Real ninf=norminfty(A);//infinite norm
```

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.

The `Matrix` proposes some solvers:

```
Matrix<Real> A
RealVector B;
...
// solve AX=B or AXs=Bxs using Gauss reduction
gaussSolver(A, B, piv, row);
gaussMultipleSolver(A, B, nbrhs, piv, row);
//inverse of a square matrix
RealMatrix invA=inverse(A);
// QR factorization
RealMatrix Q,R;
qr(A,Q,R);
// SVD factorization A= U S V*, if A a (m,n)-matrix, U is a (m,r)-matrix, V
    a (n,r)-matrix and S a r-vector where r=min(m,n)
RealMatrix U, V;
RealVector S;   //singular values
svdMat(A, U, S, V);
```


QR and SVD are available only if Eigen library is set on.


It is also possible to deal with matrix of matrices, for instance:

```
Matrix<Real> ones(2,2,1);            //a 2×2 ones matrix
Matrix<Matrix<Real> > A(2,2,ones); //a 2×2 matrix of 2×2 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
    !!!
//or
list<String> >& rlst=static_cast<list<String>&>(*par.get_p());  // be sure
    !!!
cout<<rlst;                                // print the list not the pointer
```

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).

```
Parameter k(1.,"frequency"); // a real parameter
Real r=real(k);              // compatible get
Complex c=cmplx(k);          // no compatible get, but cast real to complex
String s=string(k);          // no compatible get, error
void * q=pointer(k);         // no compatible get, error
```

A `Parameter` object can be automatically cast to its right value:

```
Parameter k(1.,"frequency");      // a real parameter
Parameter i(complex_t(0,1),"i"); // a complex parameter
Parameter mat(RealMatrix(5,_hilbertMatrix),"Hilbert matrix");// a matrix
    parameter
Real r=k;     // auto cast to real
r=k;          // auto cast to real
Complex c=k; // auto cast to complex
c=i;          // does not work !!! cannot resolve ambiguity of complex class
c=k;          //          complex-> complex, double -> complex
String s=string(k);  // error, incompatible types
void * q=pointer(k); // error, incompatible types
RealMatrix H=mat;    // ok
```

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.

```
Parameter k(1.,"frequency");
Parameter k2=k*k;
```

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, complex, 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]);
```

where *value* is of type integer, real, complex, real/complex vector/matrix, string (or char*) or void * and *name* is an optional string defining the parameter name. When *value* is a Parameter object, *name* is not required.

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 << :

```
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 :

```
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:

```
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 = :

```
Parameters params;
Real k=params("k").get_r();  //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:

```
Parameters params;
params.print();             // print on a default print 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:

```
Parameter height=3;
Parameters data;
data<<height<<Parameter(4,"width")<<"case 1"<<1.5;
// String "case 1" has default name "parameter3"
// Real_t "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 B.4 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:

```
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 k=pa("k"); is available if your have a parameter with name "k"
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
}
```

Then create in your main program a `Parameters` object, a `Function` object from C++ function and you `Parameters` object

```
{
 Parameters pars(1,"k");
 Function F(f,pars);
}
```

If you have to deall with the integral term involving a real value matrix (with no parameter involved in this example):

$$\int_\Omega A\nabla u\,.\nabla v\,d\Omega,$$

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
Function funcf(f,par);         //define a scalar function object using
    parameters
Function funcA(A);             //define a matrix function object
Function funcG(Green_Hemholtz_3D, par); //define a scalar kernel
Function funcvf(vf,par);       //define a scalar function in its vector form
```

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 requiring an object function
```

Dealing with normal vectors

Sometimes, user functions has to deal with some normal vectors and obviously the normal vectors will depend on the point where the function is evaluated. For instance, a function computing the normal derivative of a given incident field (e.g. a plane wave e^{ikx}), will look like

```
Complex dnuinc(const Point& P,  Parameters& pa = default_Parameters)
{
 Real x=P(1), k=pa("k");      // get k from parameters
 Reals n=getN();             // get the normal vector at P
 return i_*k*exp(i_*k*x)*n(1);
}
```

When this function is passed to FE computation routines, the normal vector will be evaluated and transmitted to the function only if the `Function` object associated to the function, has declared to use the normal vector. It is done by the following instructions in the main program:

```
Parameters pars(1,"k");        // declare k in the parameters
Function f(dnuinc,pars);       // associate parameters to Function object
f.require(_n);                 // tell the function will use the normal
TermVector B(intg(Sigma,f*v)); // use function f
```

The normal vector refers to the domain on which the linear or bilinear form (involving the function) acts. The orientation of normal vector is described in section 5.9.2.

When using a function returning a vector or a matrix, because XLiFE++ checks the dimension(s) of the returned object by using a fake point or a normal vector, it may occur some consistency problems with the dimension of points or normals that is set to 3 by default. You can change it by specifying explicitly the dimension of the function when building it:

```
Function f(dnuinc,pars);
f.dimPoint=2;                  //change the point dimension
//or
Function f(dnuinc,2,pars); // specifying the point dimension when building
```

It works in the same way for kernels.

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 T and its associated object Function:

```
Vector<T> namefunction(const Vector<Point> &,const Vector<Point> &,
    Parameters& pa=default_Parameters)
Function nameofobjectfunction(namefunction,[param]);
```

- to avoid the explicit construction of the object function (useful when you have to pass the function as an argument)

```
Function(namefunction,[param]);
```

- to declare that the object function uses the normal vector

```
Function nameofobjectfunction(namefunction,[param]);
nameofobjectfunction.require(_n);
```

 For the people who used MELINA Fortran, this approach replaces the famous *fctrm.f* and the *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)

```
Point P=Point(0,0), Q=Point(1,1);
Complex c=funcf.funSC(P);       //a function returning a complex scalar (funSC)
Complex g=funcG.kerSC(P,Q);     //a kernel returning a complex scalar (kerSC)
Matrix<Real> m=funcA.funMR(P);//a function returning a real matrix
    scalar(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

```
Point P=Point(0,0), Q=Point(1,1);
Complex c;                  // complex to store the result
funcf.checkTypeOn();        // activate checking mode
funcf(P,c);                 // compute a complex scalar function at point P
                            // checking mode is disabled after the computation
Vector<Point> pts;          // a vector of points
pts(1)=P; pts(2)=Q;
vector<Complex> vc;         // vector to store the result
funcf.checkTypeOn();        // activate the checking mode
funcf(Pts,vc);              // compute function at a vector of points
```

  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
Function ndotgradx;         // nx.gradx if available
Function ndotgrady;         // ny.grady if available
Function curlx;             // curl_x if available
Function curly;             // curl_y if available
```

```
Function curlxy;                // curl_x curl_y if available
Function divx;                  // div_x if available
Function divy;                  // div_y if available
Function divxy;                 // div_x div_y if available
Function dx1;                   // d_x1 if available
Function dx2;                   // d_x2 if available
Function dx3;                   // d_x3 if available
Kernel* singPart;               // singular part of kernel
Kernel* regPart;                // regular part of kernel

Dimen dimPoint;                 // dimension of points
SingularityType singularType; // singularity (_notsingular, _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
```

When dealing with a matrix kernel it may be useful to give `d_x1, d_x2, d_x3` because it is not possible to define the gradient of a matrix kernel as a `Function`.

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 :

```
// 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;
}

Vector<Complex> Helmholtz3dGradx(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;
}

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;
}
```

Define regular part functions :

```
// 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.e-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;
}

Vector<Complex> Helmholtz3dGradxReg(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;
}

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;
}
```

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;
H3Dreg.userData = pars;
H3Dreg.dimPoint = 3;
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";
H3Dsing.singularType =_r;
H3Dsing.singularOrder = -1;
H3Dsing.singularCoefficient = over4pi;
H3Dsing.symmetry=_symmetric;
H3Dsing.userData = pars;
H3Dsing.dimPoint = 3;
H3Dsing.kernel = Function(Helmholtz3dSing, pars);
H3Dsing.gradx  = Function(Helmholtz3dGradxSing, pars);
H3Dsing.grady  = Function(Helmholtz3dGradySing, pars);

Kernel H3D;    //kernel
H3D.name="Helmholtz 3D kernel";
H3D.singularType =_r;
H3D.singularOrder = -1;
H3D.singularCoefficient = over4pi;
H3D.symmetry=_symmetric;
H3D.userData = pars;
H3D.dimPoint = 3;
```

```
H3D.kernel = Function(Helmholtz3d, pars);
H3D.gradx  = Function(Helmholtz3dGradx, pars);
H3D.grady  = Function(Helmholtz3dGrady, pars);
H3D.regPart  = &H3Dreg;
H3D.singPart = &H3Dsing;
```


If you do not define singular and regular part kernels, some computations will not be available.

In fact the Helmholtz kernels are defined in *mathsResources* library of XLiFE++. To load it, use the following code:

```
Parameters pars;
pars<<Parameter(1.,"k");
Kernel H3D = Helmholtz3dKernel(pars);
```

For `Kernel` objects, one can pass to some kernel functions either _nx vector or _ny vector or both using the same method as one used for `Function` :

```
Complex G(const Point& P,const Point& Q, Parameters& pa = default_Parameters)
{
Reals nx= getNx(); // get the normal vector at P
Reals ny= getNy(); // get the normal vector at Q
...
}
...
Parameters pars(1,"k"); // declare k in the parameters
Kernel K(g,pars);       // associate parameters to Kernel object
K.require(_nx);         // declare that kernel uses x-normal
K.require(_ny);         // declare that kernel uses y-normal
TermMatrix B(intg(Sigma,Sigma,u*K*v));  // use kernel K
```

Kernel functions defined in XLiFE++ managed the normal vectors, so the user has not to deal with.


By default, the dimension of points of a Kernel is 3. When you define a 2D kernel, it may happen some troubles with point dimensions, in particular if the kernel function involves some operations sensitive to the point dimension. You can cure this problem by testing point dimensions in the kernel function or by specifying the point dimension when building Kernel:

```
Real ker(const Point& x, const Point& x, Parameters&
    pa=defaultParameters)
{...}
...
Kernel K(ker); K.dimPoint=2;
//or
Kernel K(ker,2);
```


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 SymbolicFunction

Some finite element constructions require to pass as argument a `SymbolicFunction` object, that is a symbolic expression of a function. Such object is built in by writting any expression involving

- some constants (real or complex)
- some variables : `x_1`, `x_2`, `x_3`
- some algebraic operators : `+ - * / ^` (power)
- some boolean operators : `&& || < <= > >= == != !` (not)
- some mathematical functions : `abs, realPart, imagPart, sqrt, squared, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, exp, log, log10`

The result of a boolean expression is either 0 (false) or 1 (true).


The functions `asinh, acosh, atanh` are only available when compiling with C++ 2011.

To deal with a symbolic function is very easy. For instance consider the function

$$f(x_1, x_2) = [1 - x_1 + \cos(x_2)]^3.$$

To define it as a symbolic function, write:

```
SymbolicFunction fs = (1-x_1+cos(x_2))^3;
```

To evaluate it, write:

```
Real r = fs(-1,0);
Complex c = fs(i_,0);
```

You can mix algebraic and boolean expressions. Consider the following function:

$$f(x_1) = \begin{cases} e^{x_1} & x_1 \leq 0 \\ 1 & x_1 > 0 \end{cases}$$

To define it, write:

```
SymbolicFunction fs = (x_1<=0)*exp(x_1)+(x_1>0);
```

B.11 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 :

```
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-mm-dd)
String theIsoTime();    // returns ISO8601 format of current time (hh-mi-ss)
```

and others to get cpu or elapsed time :

```
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&) ;        // same and prints it with comment
double totalElapsedTime();                 // 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 :

```
#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");
}
```

B.12 Memory

Using the `Memory`, the memory usage can be inspected using the following functions:

```
Real phyMem  = Memory::physicalMem();          //physical memory
Real freeMem = Memory::physicalFreeMem();      //free physical memory
Real procMem = Memory::processPhysicalMem();//physical memory used by the
    process
```

The default unit is the MegaBytes (Mo), but you can change the unit by specifying one of the following units in the argument of memory functions:

```
enum MemoryUnit {_byte,_kilobyte,_megabyte,_gigabyte,_terabyte};
```