XLiFE++ tutorial

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++/`.

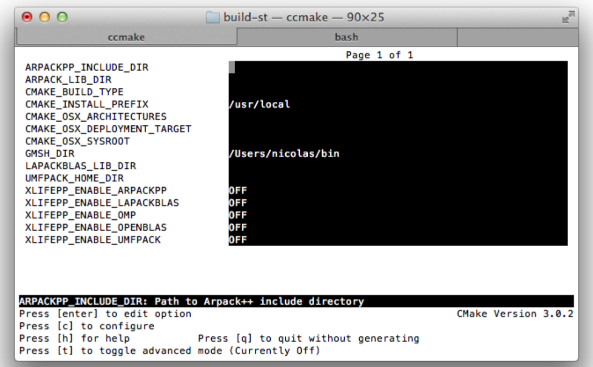
- Either you download XLiFE++ sources: you have to unzip the archive at any place you choose in the filesystem. Then, you follow configuration procedure by setting at least a C++ compiler, the path to Gmsh and eventually paths to external libraries Blas, Lapack, Arpack and UmfPack. When done, you will 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.



```
                                                     Page 1 of 1
 ARPACKPP_INCLUDE_DIR
 ARPACK_LIB_DIR
 CMAKE_BUILD_TYPE
 CMAKE_INSTALL_PREFIX            /usr/local
 CMAKE_OSX_ARCHITECTURES
 CMAKE_OSX_DEPLOYMENT_TARGET
 CMAKE_OSX_SYSROOT
 GMSH_DIR                        /Users/nicolas/bin
 LAPACKBLAS_LIB_DIR
 UMFPACK_HOME_DIR
 XLIFEPP_ENABLE_ARPACKPP         OFF
 XLIFEPP_ENABLE_LAPACKBLAS       OFF
 XLIFEPP_ENABLE_OMP              OFF
 XLIFEPP_ENABLE_OPENBLAS         OFF
 XLIFEPP_ENABLE_UMFPACK          OFF



ARPACKPP_INCLUDE_DIR: Path to Arpack++ include directory
Press [enter] to edit option                                          CMake Version 3.0.2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

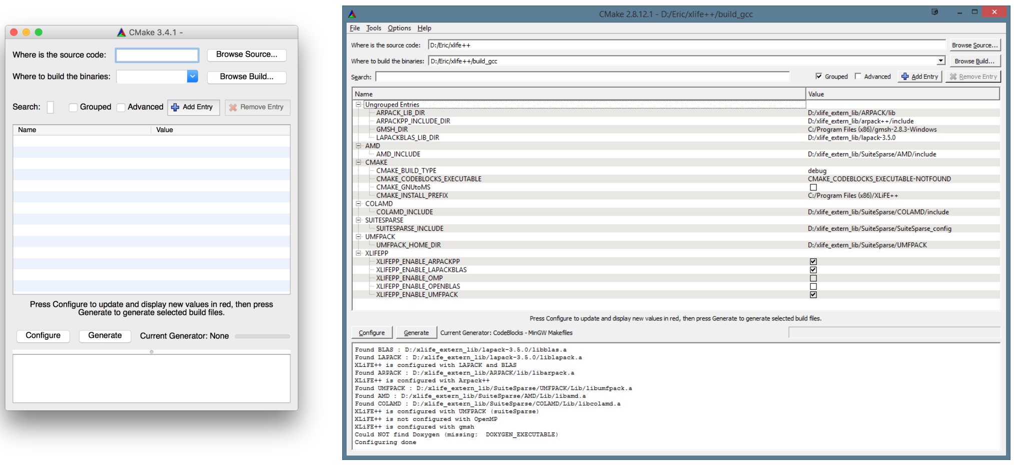
Figure 1.1: `ccmake` (MacOS, Linux)

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.


CMake 3.4.1 -
Where is the source code:
Browse Source...
Where to build the binaries:
Browse Build...
Search:
Grouped
Advanced
Add Entry
Remove Entry
Name
Value
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate
Current Generator: None
CMake 2.8.12.1 - D:/Eric/xlife++/build_gcc
File
Tools
Options
Help
Browse Source...
Browse Build...
Search:
Add Entry
Remove Entry
Press Configure to update and display new values in red, then press Generate to generate selected build files.
Configure
Generate

Where is the source code: D:/Eric/xlife++

Where to build the binaries: D:/Eric/xlife++/build_gcc

[✓] Grouped [ ] Advanced

Name	Value
Ungrouped Entries	
ARPACK_LIB_DIR	D:/xlife_extern_lib/ARPACK/lib
ARPACKPP_INCLUDE_DIR	D:/xlife_extern_lib/arpack++/include
GMSH_DIR	C:/Program Files (x86)/gmsh-2.8.3-Windows
LAPACKBLAS_LIB_DIR	D:/xlife_extern_lib/lapack-3.5.0
AMD	
AMD_INCLUDE	D:/xlife_extern_lib/SuiteSparse/AMD/include
CMAKE	
CMAKE_BUILD_TYPE	debug
CMAKE_CODEBLOCKS_EXECUTABLE	CMAKE_CODEBLOCKS_EXECUTABLE-NOTFOUND
CMAKE_GNUtoMS	☐
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/XLiFE++
COLAMD	
COLAMD_INCLUDE	D:/xlife_extern_lib/SuiteSparse/COLAMD/include
SUITESPARSE	
SUITESPARSE_INCLUDE	D:/xlife_extern_lib/SuiteSparse/SuiteSparse_config
UMFPACK	
UMFPACK_HOME_DIR	D:/xlife_extern_lib/SuiteSparse/UMFPACK
XLIFEPP	
XLIFEPP_ENABLE_ARPACKPP	✓
XLIFEPP_ENABLE_LAPACKBLAS	✓
XLIFEPP_ENABLE_OMP	☐
XLIFEPP_ENABLE_OPENBLAS	☐
XLIFEPP_ENABLE_UMFPACK	✓

Current Generator: CodeBlocks - MinGW Makefiles

```
Found BLAS : D:/xlife_extern_lib/lapack-3.5.0/libblas.a
Found LAPACK : D:/xlife_extern_lib/lapack-3.5.0/liblapack.a
XLiFE++ is configured with LAPACK and BLAS
Found ARPACK : D:/xlife_extern_lib/ARPACK/lib/libarpack.a
XLiFE++ is configured with Arpack++
Found UMFPACK : D:/xlife_extern_lib/SuiteSparse/UMFPACK/Lib/libumfpack.a
Found AMD : D:/xlife_extern_lib/SuiteSparse/AMD/Lib/libamd.a
Found COLAMD : D:/xlife_extern_lib/SuiteSparse/COLAMD/Lib/libcolamd.a
XLiFE++ is configured with UMFPACK (suiteSparse)
XLiFE++ is not configured with OpenMP
XLiFE++ is configured with gmsh
Could NOT find Doxygen (missing:  DOXYGEN_EXECUTABLE)
Configuring done
```

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.

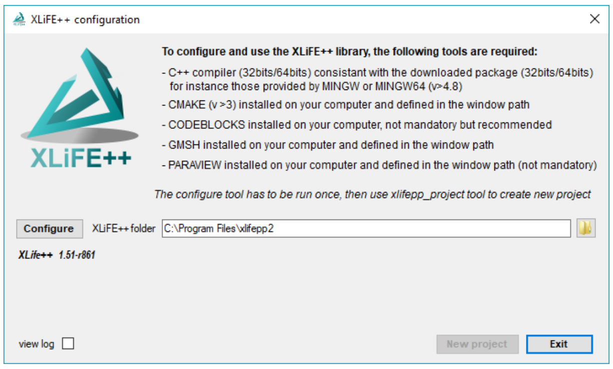
XLIFEPP_FORTRAN_LIB_DIR to specify the directory where the gfortran library is. It is necessary for compilers that are not able to find it by itself, such as `clang++`

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++



XLiFE++ configuration

To configure and use the XLiFE++ library, the following tools are required:

- C++ compiler (32bits/64bits) consistant with the downloaded package (32bits/64bits) for instance those provided by MINGW or MINGW64 (v>4.8)
- CMAKE (v >3) installed on your computer and defined in the window path
- CODEBLOCKS installed on your computer, not mandatory but recommended
- GMSH installed on your computer and defined in the window path
- PARAVIEW installed on your computer and defined in the window path (not mandatory)

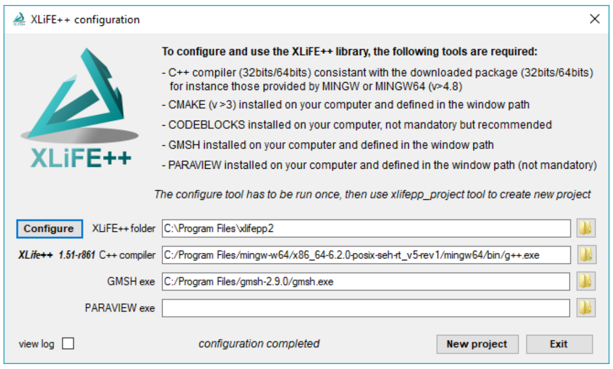
The configure tool has to be run once, then use xlifepp_project tool to create new project

XLiFE++ folder: C:\Program Files\xlifepp2

XLife++ 1.51-r861

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


XLiFE++ configuration

To configure and use the XLiFE++ library, the following tools are required:

- C++ compiler (32bits/64bits) consistant with the downloaded package (32bits/64bits) for instance those provided by MINGW or MINGW64 (v>4.8)
- CMAKE (v >3) installed on your computer and defined in the window path
- CODEBLOCKS installed on your computer, not mandatory but recommended
- GMSH installed on your computer and defined in the window path
- PARAVIEW installed on your computer and defined in the window path (not mandatory)

The configure tool has to be run once, then use xlifepp_project tool to create new project

XLiFE++ folder: C:\Program Files\xlifepp2

XLife++ 1.51-r861 C++ compiler: C:/Program Files/mingw-w64/x86_64-6.2.0-posix-seh-rt_v5-rev1/mingw64/bin/g++.exe

GMSH exe: C:/Program Files/gmsh-2.9.0/gmsh.exe

PARAVIEW exe:

view log ☐

configuration completed

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:

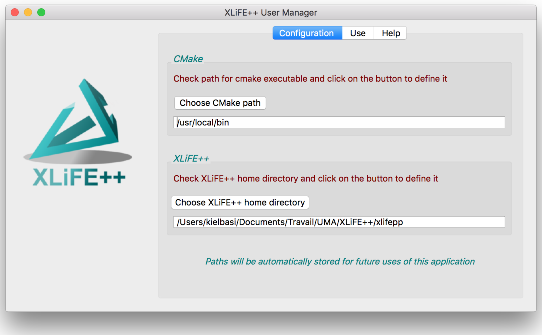
```
USAGE:
    xlifepp.sh --build [--interactive] [(--generate|--no-generate)]
    xlifepp.sh --build --non-interactive [(--generate|--no-generate)]
                       [--compiler <compiler>] [--directory <dir>]
                       [--generator-name <generator>]
                       [--build-type <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.


XLiFE++ User Manager

CMake

Check path for cmake executable and click on the button to define it

/usr/local/bin

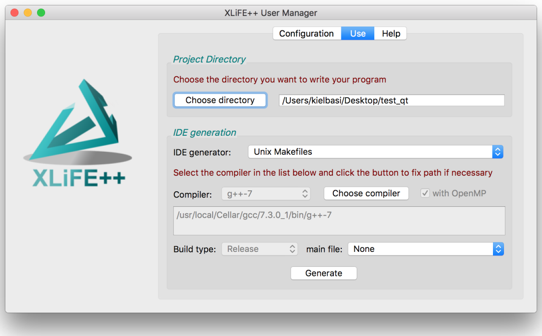
XLiFE++

Check XLiFE++ home directory and click on the button to define it

/Users/kielbasi/Documents/Travail/UMA/XLiFE++/xlifepp

Paths will be automatically stored for future uses of this application

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


XLiFE++ User Manager

Project Directory

Choose the directory you want to write your program

/Users/kielbasi/Desktop/test_qt

IDE generation

IDE generator: Unix Makefiles

Select the compiler in the list below and click the button to fix path if necessary

Compiler: g++-7

✓ with OpenMP

/usr/local/Cellar/gcc/7.3.0_1/bin/g++-7

Build type: Release

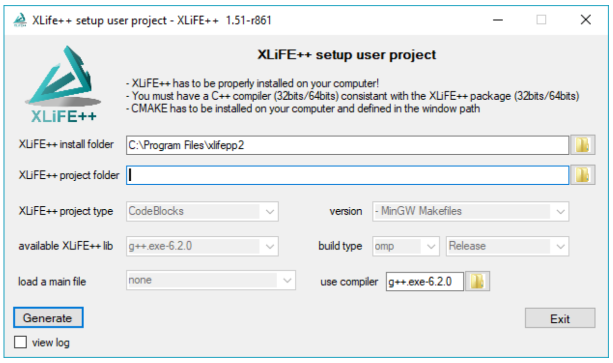
main file: None

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.


XLife++ setup user project - XLiFE++ 1.51-r861

XLiFE++ setup user project

- XLiFE++ has to be properly installed on your computer!
- You must have a C++ compiler (32bits/64bits) consistant with the XLiFE++ package (32bits/64bits)
- CMAKE has to be installed on your computer and defined in the window path

XLiFE++ install folder: C:\Program Files\xlifepp2

XLiFE++ project folder:

XLiFE++ project type: CodeBlocks

version: - MinGW Makefiles

available XLiFE++ lib: g++.exe-6.2.0

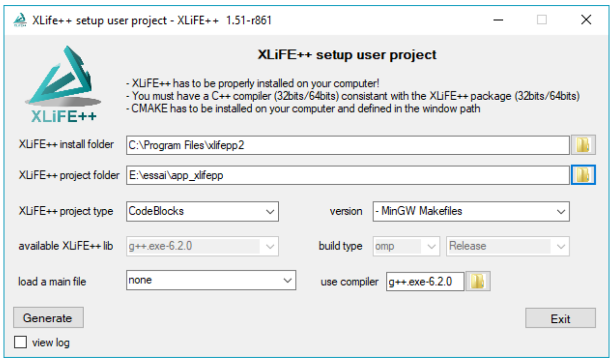
build type: omp Release

load a main file: none

use compiler: g++.exe-6.2.0

☐ view log

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.

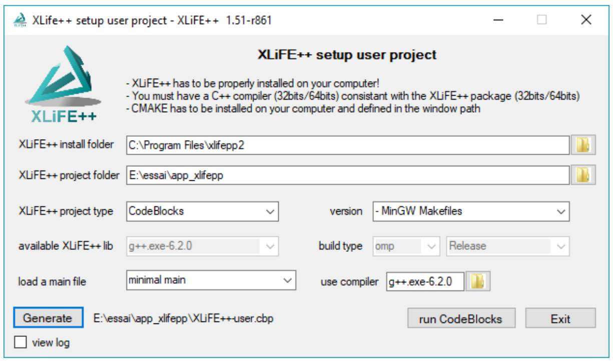

XLife++ setup user project - XLiFE++ 1.51-r861

XLiFE++ setup user project

- XLiFE++ has to be properly installed on your computer!
- You must have a C++ compiler (32bits/64bits) consistant with the XLiFE++ package (32bits/64bits)
- CMAKE has to be installed on your computer and defined in the window path

XLiFE++ install folder: C:\Program Files\xlifepp2
XLiFE++ project folder: E:\essai\app_xlifepp
XLiFE++ project type: CodeBlocks — version: - MinGW Makefiles
available XLiFE++ lib: g++.exe-6.2.0 — build type: omp, Release
load a main file: none — use compiler: g++.exe-6.2.0

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:


XLife++ setup user project - XLiFE++ 1.51-r861

XLiFE++ setup user project

- XLiFE++ has to be properly installed on your computer!
- You must have a C++ compiler (32bits/64bits) consistant with the XLiFE++ package (32bits/64bits)
- CMAKE has to be installed on your computer and defined in the window path

XLiFE++ install folder: C:\Program Files\xlifepp2
XLiFE++ project folder: E:\essai\app_xlifepp
XLiFE++ project type: CodeBlocks — version: - MinGW Makefiles
available XLiFE++ lib: g++.exe-6.2.0 — build type: omp, Release
load a main file: minimal main — use compiler: g++.exe-6.2.0

E:\essai\app_xlifepp\XLiFE++user.cbp

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

XLiFE++ is copyright (C) 2010-2018 by E. Lunéville and N. Kielbasiewicz and is distributed under the terms of the GNU General Public License (GPL) (Version 3 or later, see https://www.gnu.org/licenses/gpl-3.0.en.html). This means that everyone is free to use XLiFE++ and to redistribute it on a free basis. XLiFE++ is not in the public domain; it is copyrighted and there are restrictions on its distribution. You cannot integrate XLiFE++ (in full or in parts) in any closed-source software you plan to distribute (commercially or not). If you want to integrate parts of XLiFE++ into a closed-source software, or want to sell a modified closed-source version of XLiFE++, you will need to obtain a different license. Please contact us directly for more information.

The developers do not assume any responsibility in the numerical results obtained using the 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.

```cpp
#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 **??** 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 **??** 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 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++.

3.1 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
}
```

3.2 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.

3.3 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.

3.4 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++;
}
```

3.5 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;
```

3.6 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);
...
```

3.7 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.

3.8 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

3.9 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)