XLiFE++ tutorial

Nicolas KIELBASIEWICZ, Eric LUNÉVILLE

April 8, 2022

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 activities.

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 modeling. A wide range of problems in Physics, Mechanics, Engineering, Mathematics, Health, Finance are modeled 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, scalar or vector) 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 formulation;
- 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 http://uma.ensta-paris.fr/soft/XLiFE++/. You can download releases and snapshots of either the source code or binaries. Snapshots are supposed to be generated automatically very often.

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

share/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.

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

ext contains source files for external dependencies, such as ARPACK++, EIGEN, AMOS libraries

tests contains all unitary and system tests to check your installation

lib contains 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 it as eigen solver, XLiFE++ provides its own ARPACK distribution (`http://www.caam.rice.edu/software/ARPACK/`). This internal distribution is patched to work with recent compilers, but you may prefer using another distribution. See section A.3 for details and recommendations.

- To use it as direct solver, you may install UMFPACK (http://faculty.cse.tamu.edu/davis/suitesparse.html). See section A.2 for details and recommendations.
- Tu use it as GPU solver, you may install MAGMA (http://icl.cs.utk.edu/magma/).
- To visualize solutions of your programs using XLIFE++, you may install GMSH (http://geuz.org/gmsh), PARAVIEW (http://www.paraview.org), MATLAB (https://fr.mathworks.com) or OCTAVE (https://sourceforge.net/projects/octave/files/).

XLIFE++ provides 2 external libraries: EIGEN (http://eigen.tuxfamily.org/, essentially for SVD, and AMOS (http://www.netlib.org/amos/) for Bessel/Hankel functions with 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.

Of course, you need a C++ compiler. To install a proper 64bits compiler on WINDOWS, see section A.4.

1.4 Main installation and usage process, with CMAKE

You download XLIFE++ from its website http://uma.ensta-paris.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, paths to GMSH and PARAVIEW 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 PARAVIEW, and eventually paths to external libraries BLAS, LAPACK, ARPACK and 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 want to use `clang++` as a compiler, please download dedicated `clang++` binaries, as they are necessarily generated without OPENMP activated.

If you need to learn how to use cmake, please go to Appendix B. Keep in mind that running CMAKE several times is not a problem. Indeed, as far as external dependencies are concerned, CMAKE is supposed to find a wide range of possible configurations, but not every time of course!

1.4.1 Configuration step

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 `..`
Cache entries that will be explained can be set directly through GUI applications.

The default build configuration is to use EIGEN and AMOS libraries, as they are provided directly by XLIFE++, and multi-threading with OPENMP, if the compiler is compatible with it (for instance, clang++ on MAC OS is not).

```
cmake ..
```

As explained in Appendix B, you can specify a generator with option *-G*, namely the native makefile or the IDE file you want. The default generator is "Unix Makefiles" on LINUX and MAC OS computers, and "MinGW Makefiles" on WINDOWS, but it can have a wide range of possible values. To know what you can do, please consult CMAKE help. For instance, if you want to generate a CodeBlocks project on LINUX computer, you may choose *Codeblocks - Unix Makefiles*

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

In the following, we will focus on cache entries you may have to set.

When CMAKE is running, it stores values of cache entries in a cache file `CMakeCache.txt` in the *build directory*. As a result, when you run again CMAKE, you are not forced to give already given options.

1.4.2 How to set compilers

By default, CMAKE consider each directory defined in your paths environment variables to find compilers, and select the first compiler found. Nevertheless, you may select another compiler. To do so, you need 2 CMAKE cache entries:

CMAKE_CXX_COMPILER to set the C++ compiler

CMAKE_Fortran_COMPILER to set the Fortran compiler (necessary to AMOS compilation, and eventually to ARPACK compilation if you choose provided source distribution). Please notice the spelling, as CMAKE is case-sensitive !!

```
cmake .. -DCMAKE_CXX_COMPILER=g++-7 -DCMAKE_Fortran_COMPILER=gfortran-7 [other_options]
```

If you want to configure XLIFE++ in Debug mode (or another one, you may use the following entry :

CMAKE_BUILD_TYPE Default value is "Release". Other possible values are "Debug", "RelWithDebInfo", ...

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

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.

1.4.3 How to set external dependencies

In the following, [general_entries] refers to options introduced in subsection 1.4.2

To activates all dependencies, you can type the following:

```
cmake .. [general_entries] -DXLIFEPP_DEPS=ENABLE_ALL
```

XLIFEPP_DEPS Default value is "DEFAULT". Other possible values are "ENABLE_ALL" or "DISABLE_ALL".

For XLiFE++ binaries, this option is used to generate them, so you don't have to use it.

When you use CMake GUI application, option **XLIFEPP_DEPS** is useless, as the list of specific options discussed in the following warning are displayed as checkboxes.

For every external dependency, CMake search files in standard environment paths and in specific paths defined in CMake configuration script. So if external dependencies are installed by package managers, CMake will find them.
If it is not the case, or if the library found is not the one you want to use, you can tell CMake where to find them, as we will see in the following.

If you don't want to activate every external dependency, but only some of them, you may use a subset of the following options (default behavior in bold):

XLIFEPP_ENABLE_ARPACK To enable/disable use of ARPACK. Possible values are ON or **OFF**.
XLIFEPP_ENABLE_UMFPACK To enable/disable use of UMFPACK. Possible values are ON or **OFF**.
XLIFEPP_ENABLE_AMOS To enable/disable use of AMOS. Possible values are **ON** or OFF.
XLIFEPP_ENABLE_OMP To enable/disable use of OPENMP. Possible values are **ON** or OFF.
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.
XLIFEPP_ENABLE_MAGMA To enable/disable use of MAGMA. Possible values are ON or **OFF**.

Configuring paths to GMSH and PARAVIEW executables

In the following, [general_entries] refers to options introduced in subsection 1.4.2 and [deps_entries] refers to others options introduced since subsection 1.4.3

To set the full path to GMSH executable, you can type the following:

```
cmake .. [general_entries] [deps_entries] -DXLIFEPP_GMSH_EXECUTABLE=path/to/gmsh/executable
```

To set the full path to PARAVIEW executable, you can type the following:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_PARAVIEW_EXECUTABLE=path/to/paraview/executable
```

You have to set 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.

Configuring dependency on BLAS and LAPACK libraries

In the following, [general_entries] refers to options introduced in subsection 1.4.2 and [deps_entries] refers to others options introduced since subsection 1.4.3

To tell to CMAKE where to find BLAS library, you just have to set the directory containing the BLAS library, with the option **XLIFEPP_BLAS_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries] -DXLIFEPP_BLAS_LIB_DIR=path/to/blas/library/directory
```

To tell to CMAKE where to find LAPACK library, you just have to set the directory containing the LAPACK library, with the option **XLIFEPP_LAPACK_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_LAPACK_LIB_DIR=path/to/lapack/library/directory
```

Setting the directory is sufficient only if the BLAS library name is standard, such as libblas.a, libblas.so, libblas.dylib, libblas.dll, blas.lib, ... (the same goes for LAPACK) If it is not the case, it means that BLAS/LAPACK is not installed properly (for instance with a version number as a suffix). Prefer using options **XLIFEPP_BLAS_LIB** and **XLIFEPP_LAPACK_LIB** instead, to set the full path to BLAS and LAPACK libraries

If you downloaded binary libraries provided by the XLIFE++ website, as recommended in subsection 1.3.1, it will be something like (if you downloaded 64bits binaries):

```
cmake .. [general_entries] [deps_entries] -DXLIFEPP_BLAS_LIB_DIR="C:/.../lapack-3.5.0_64"
    -DXLIFEPP_LAPACK_LIB_DIR="C:/.../lapack-3.5.0_64"
```

When using `clang++`, you have to set the path to FORTRAN library, as `clang++` is not able to find it by itself (contrary to `gcc`, as `gcc` and FORTRAN library are in the same distribution). To do so, you just have to use the option **XLIFEPP_FORTRAN_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_FORTRAN_LIB_DIR=path/to/gfortran/library/directory
```

Setting the directory is sufficient only if the FORTRAN library name is standard, such as libgfortran.a, libgfortran.so, libgfortran.dylib, libgfortran.dll, gfortran.lib, ... If it is not the case, it means that FORTRAN is not installed properly (for instance with a version number as a suffix). Prefer using option **XLIFEPP_FORTRAN_LIB** instead, to set the full path to FORTRAN library

Configuring dependency on ARPACK library

In the following, [general_entries] refers to options introduced in subsection 1.4.2 and [deps_entries] refers to others options introduced since subsection 1.4.3

First of all, you have to choose if you want to use the internal ARPACK distribution of XLIFE++, or an external one. Of course, the one provided by XLIFE++ is detected automatically. To select the rightful mode, you have to set the option:

XLIFEPP_SYSTEM_ARPACK To choose an external distribution of ARPACK or the one provided by XLIFE++. Possible values are ON or OFF. Default is OFF, to use the distribution provided by XLIFE++

If you choose to use an external distribution of ARPACK, you now have to tell CMAKE where to find ARPACK library. To do so, you just have to set the directory containing the ARPACK library, with the option **XLIFEPP_ARPACK_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries] -DXLIFEPP_BLAS_LIB_DIR=path/to/blas/library/directory
```

Setting the directory is sufficient only if the ARPACK library name is standard, such as libarpack.a, libarpack.so, libarpack.dylib, libarpack.dll, arpack.lib, ...If it is not the case, it means that ARPACK is not installed properly (for instance with a version number as a suffix). Prefer using option **XLIFEPP_ARPACK_LIB** instead, to set the full path to ARPACK library

If you downloaded binary libraries provided by the XLIFE++ website, as recommended in subsection 1.3.1, it will be something like (if you downloaded 64bits binaries):

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_ARPACK_LIB_DIR="C:/.../ARPACK_64/lib_mingw64"
```

When using `clang++`, you have to set the path to FORTRAN library, as `clang++` is not able to find it byt itself (contrary to `gcc`, as `gcc` and FORTRAN library are in ther same distribution). To do so, you just have to use the option **XLIFEPP_FORTRAN_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_FORTRAN_LIB_DIR=path/to/gfortran/library/directory
```

Setting the directory is sufficient only if the FORTRAN library name is standard, such as libgfortran.a, libgfortran.so, libgfortran.dylib, libgfortran.dll, gfortran.lib, ...If it is not the case, it means that FORTRAN is not installed properly (for instance with a version number as a suffix). Prefer using option **XLIFEPP_FORTRAN_LIB** instead, to set the full path to FORTRAN library

Configuring dependency on SUITESPARSE/ UMFPACK libraries

In the following, [general_entries] refers to options introduced in subsection 1.4.2 and [deps_entries] refers to others options introduced since subsection 1.4.3

UMFPACK is a part of SUITESPARSE distribution (that also contains amd, colamd, camd, ccolamd, cholmod, metis, and suitesparseconfig, ...).

To tell to CMAKE where to find UMFPACK library/SUITESPARSE distribution, you just have to set the home directory containing the SUITESPARSE distribution, with the option **XLIFEPP_SUITESPARSE_HOME_DIR**:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_SUITESPARSE_HOME_DIR=path/to/suitesparse/home/directory
```

If you downloaded binary libraries provided by the XLIFE++ website, as recommended in subsection 1.3.1, it will be something like (if you downloaded 64bits binaries):

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_SUITESPARSE_HOME_DIR="C:/.../SuiteSparse_64"
```

When using `clang++`, you have to set the path to FORTRAN library, as `clang++` is not able to find it byt itself (contrary to `gcc`, as `gcc` and FORTRAN library are in ther same distribution). To do so, you just have to use the option **XLIFEPP_FORTRAN_LIB_DIR**:

```
cmake .. [general_entries] [deps_entries]
    -DXLIFEPP_FORTRAN_LIB_DIR=path/to/gfortran/library/directory
```

Setting the directory is sufficient only if the FORTRAN library name is standard, such as libgfortran.a, libgfortran.so, libgfortran.dylib, libgfortran.dll, gfortran.lib, ... If it is not the case, it means that FORTRAN is not installed properly (for instance with a version number as a suffix). Prefer using option **XLIFEPP_FORTRAN_LIB** instead, to set the full path to FORTRAN library

If setting **XLIFEPP_SUITESPARSE_HOME_DIR** is not enough to find every library of SUITESPARSE distribution, you can use specific options of the form:

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.

Setting the directories is sufficient only if the SUITESPARSE library names are standard, such as libamd.a, libamd.so, libamd.dylib, libamd.dll, amd.lib, ... (the same goes for the other libraries) If it is not the case, it means that SUITESPARSE is not installed properly (for instance with a version number as a suffix). Prefer using option **XLIFEPP_XXX_LIB** instead, to set the full path to SUITESPARSE libraries, where XXX can be AMD, COLAMD, CAMD, CCOLAMD, CHOLMOD, METIS, SUITESPARSE (only on MAC OS), SUITESPARSECONFIG or UMFPACK.

Configuring dependency on MAGMA library

In the following, [general_entries] refers to options introduced in subsection 1.4.2 and [deps_entries] refers to others options introduced since subsection 1.4.3

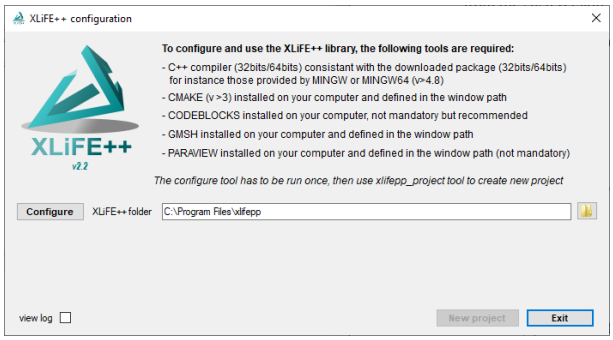
To tell to CMAKE where to find MAGMA library, you just have to set the directory containing the MAGMA library, with the option **XLIFEPP_MAGMA_LIB_DIR** and the directory containing the MAGMA header files, with the option **XLIFEPP_MAGMA_INCLUDE_DIR**:

```
cmake .. [general_entries] [deps_entries] -DXLIFEPP_MAGMA_LIB_DIR=path/to/magma/library/directory
    -DXLIFEPP_MAGMA_INCLUDE_DIR=path/to/magma/include/directory
```

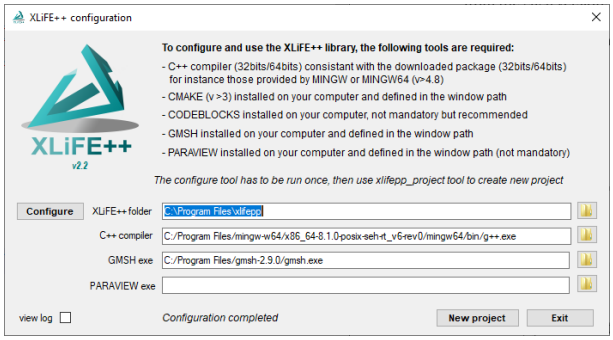
1.4.4 Installation of binaries under WINDOWS

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`. To 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.



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:

[1] A simple tool to edit the window path is WINDOWS PATH EDITOR (https://rix0rrr.github.io/WindowsPathEditor/)

```
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.

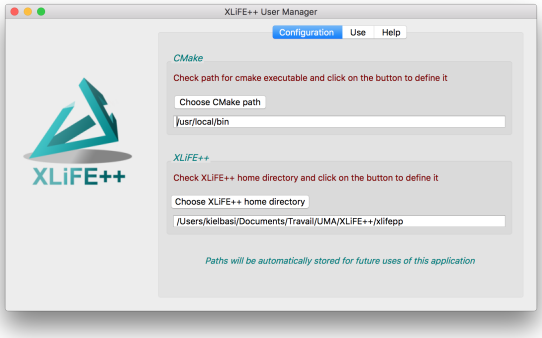


Figure 1.1: The "Configuration" tab of xlifepp-qt application

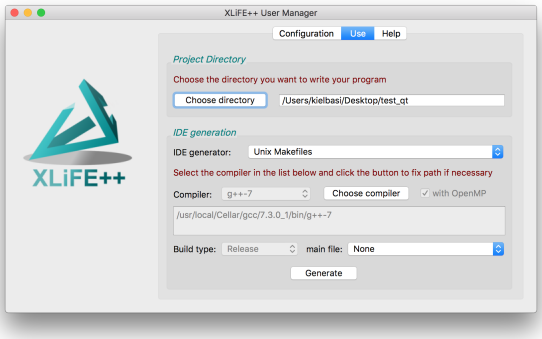
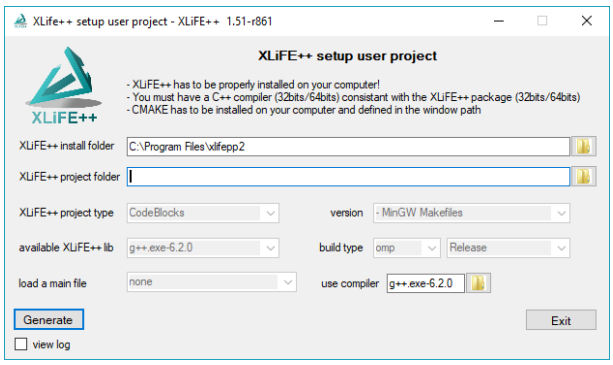


Figure 1.2: The "Use" tab of xlifepp-qt application

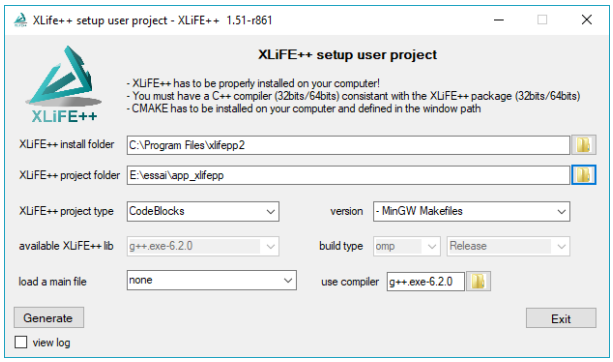
This application is a graphical user interface to the first 3 steps of the manual way.

The graphical way on WINDOWS

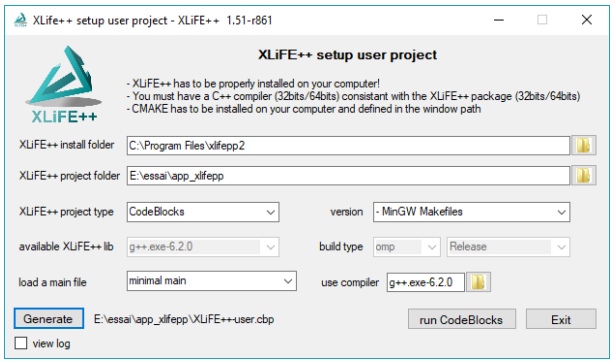
1. You run the generator `xlifepp_new_project.exe` located in the bin subdirectory of the XLiFE++ install directory. The XLiFE++ folder should be correct but you can fix it if necessary.



2. You select the folder in which you will write your program 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.

1.4.6 Example

Here follows an example showing how to install and use XLIFE++ with command line tools described in the previous sections. The character ~ denotes the home directory of the user. The commands are numbered starting from 1 ; each number is preceded by the name of the current working directory.

We first decompress the archive in a new directory (renamed xlifepp), create a build directory and launch the creation of the libraries:

```
~ (1) tar xf ~/Downloads/xlifepp-sources-v2.0.1-2018-05-09.tbz
~ (2) mv xlifepp-sources-v2.0.1-2018-05-09 xlifepp
~ (3) cd ~/xlifepp
~/xlifepp (4) mkdir build
~/xlifepp (5) cd build
~/xlifepp/build (6) cmake .. -DXLIFEPP_ENABLE_ARPACK=ON -DXLIFEPP_ENABLE_UMFPACK=ON
-- The CXX compiler identification is AppleClang 9.1.0.9020039
     .
     . (text intentionally removed)
     .
-- Build files have been written to: ~/xlifepp/build
~/xlifepp/build (7) make libs
Scanning dependencies of target xlifepp_form
[  0%] Building CXX object CMakeFiles/xlifepp_form.dir/src/form/BilinearForm.cpp.o
```

```
    .
    . (text intentionally removed)
    .
[100%] Built target libs
~/xlifepp/build (8)
```

Now, we can create an executable file. To do that, we choose to create a test directory and to compile one of the examples present in this documentation:

```
~/xlifepp/build (8) mkdir /tmp/test
~/xlifepp/build (9) cd /tmp/test
/tmp/test (10) ~/xlifepp/bin/xlifepp.sh
*********************************
*           xlifepp             *
*********************************
Project directory (default is current directory):

/tmp/test exists
The following generators are available on this platform:
1 -> Unix Makefiles
2 -> Ninja
3 -> Xcode
4 -> CodeBlocks - Ninja
5 -> CodeBlocks - Unix Makefiles
6 -> CodeLite - Ninja
7 -> CodeLite - Unix Makefiles
8 -> Sublime Text 2 - Ninja
9 -> Sublime Text 2 - Unix Makefiles
10 -> Kate - Ninja
11 -> Kate - Unix Makefiles
12 -> Eclipse CDT4 - Ninja
13 -> Eclipse CDT4 - Unix Makefiles
14 -> KDevelop3
15 -> KDevelop3 - Unix Makefiles
Your choice (default is 1): 1
The following compilers are available:
1 -> clang++-4.2.1
The following main files are available:
1 -> main.cpp
2 -> elasticity2dP1.cpp
3 -> helmholtz2d-Dirichlet_single_layer.cpp
4 -> helmholtz2dP1-DtN_scalar.cpp
5 -> helmholtz2dP1-cg.cpp
6 -> helmholtz2d_FEM_BEM.cpp
7 -> helmholtz2d_FE_IR.cpp
8 -> helmholtz3d-Dirichlet_single_layer.cpp
9 -> laplace1dGL60-eigen.cpp
10 -> laplace1dP1.cpp
11 -> laplace1dP10Robin.cpp
12 -> laplace2dP0_RT1.cpp
13 -> laplace2dP1-average.cpp
14 -> laplace2dP1-dirichlet.cpp
15 -> laplace2dP1-periodic.cpp
16 -> laplace2dP1_Neumann.cpp
17 -> laplace2dP2-eigen.cpp
18 -> laplace2dP2-transmission.cpp
19 -> maxwell2dN1.cpp
20 -> maxwell3D_EFIE.cpp
```

```
21 -> wave_2d_leap-frog.cpp
Your choice (default is 1): 14
Copying laplace2dP1-dirichlet.cpp
Cleaning CMake build files
You can use:
1 -> sequential
The following build types are available
1 -> Release
Copying CMakeLists.txt
-- The CXX compiler identification is AppleClang 9.1.0.9020039
-- Check for working CXX compiler: /usr/bin/clang++
-- Check for working CXX compiler: /usr/bin/clang++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- OpenMP is not used
-- XLiFE++ was compiled with clang++-4.2.1
-- XLiFE++ was compiled without OpenMP
-- XLiFE++ was compiled in Release mode
-- XLiFE++ libraries found !
-- Arpack is used
-- Umfpack is used
-- LAPACK is used
-- BLAS is used
-- AMOS is used
-- Magma is not used
-- Metis is not used
-- MPI is not used
-- Configuring done
-- Generating done
-- Build files have been written to: /tmp/test
/tmp/test (11) make
Scanning dependencies of target checklock
[  0%] Built target checklock
Scanning dependencies of target exec-x86_64-darwin-clang++-4.2.1-Release
[ 50%] Building CXX object CMakeFiles/exec-x86_64-darwin-clang++-4.2.1-Release.dir/laplace2dP1-
dirichlet.cpp.o
[100%] Linking CXX executable exec-x86_64-darwin-clang++-4.2.1-Release
[100%] Built target exec-x86_64-darwin-clang++-4.2.1-Release
/tmp/test (12) ./exec-x86_64-darwin-clang++-4.2.1-Release
__  __  __ _   ___  __
\ \/ / / /(_) / __\/__\_     _
 \  / / / | |/ _\ /_\_| |_ _| |_
 /  \/ /__| / /  //_|_   _|_   _|
/_/\_\____/_\/   \__/ |_|   |_|

 XLiFE++ v2.0.1-r79 (2018-07-20)
 running on july 25, 2018 at 16h08 on Darwin-i386 (MacBook-Pro)

computing FE term intg_Omega grad(u) | grad(v), using 1 threads :  done
reducing FE term A using pseudo reduction method
TermMatrix A computed, size 400 X 400 : SuTermMatrix A_u_v : block (v, u) -> matrix 400 X 400
of real scalar in symmetric_compressed sparse (csr,csc) (1521 coefficients)

solving linear system A * X = B (size 400) using umfpack
/tmp/test (13)  ls
CMakeCache.txt     cmake_install.cmake
```

```
CMakeFiles          exec-x86_64-darwin-clang++-4.2.1-Release
CMakeLists.txt      laplace2dP1-dirichlet.cpp
Makefile            log.txt
U_LD_Omega.vtu      print.txt
/tmp/test (14)
```

The result file `U_LD_Omega.vtu` can then be displayed using PARAVIEW.

1.5 Alternative installation and usage procedure, without cmake

1.5.1 Installation process

The procedure presented above requires CMAKE for both the installation and the usage of XLIFE++. 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:

- Download the archive (release or snapshot containing the sources) from

  http://uma.ensta-paris.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 a single library 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` (see details in the file `$XLDIR/etc/installLibs.README`). 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, the 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 (computation of Bessel functions),
`-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 Alternative installation and usage process, 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 subsection 1.5.2 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(_lang=en); // mandatory initialization of xlife++
  ...
}
```

Please see user documentation of XLiFE++ for a full description of command line options and of the `init` function and a full description of how to manage user options.

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

1.9 Credits

The XLiFE++ library has been mainly developped by E. Lunéville and N. Kielbasiewicz of POEMS lab (UMR 7231, CNRS-ENSTA Paris-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, unitary tests, PhD students' support
- 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
- P. Navaro (IRMAR lab), continuous integration
- E. Darrigrand-Lacarrieu (IRMAR lab), fast multipole methods
- E. Peillon (POEMS lab), evolution of (bi)linear forms

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
  SquareGeo sq(_origin=Point(0.,0.), _length=1, _nnodes=11);
  Mesh mesh2d(sq, triangle, 1, structured);
  Domain omega = mesh2d.domain("Omega");
  Space Vk(_domain=omega, _interpolation=P1, _name="Vk", _optimizeNumbering);
  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 but only 2 are relevant for users:

_verbose integer to set the verbose level. Default value is 1.

_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*.

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
```

See user documentation of XLiFE++ to learn how to define command line options or options files specific to your program and how to use them.

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 `SquareGeo`. Next, we precise the mesh element type (here triangle), the mesh element order (here 1), the mesh tool (here structured). The main mesh tool are 'structured' for simple geometries (rectangle, cube, ...) and 'gmsh' for general geometries. See user documentation of XLiFE++ for more examples of mesh definitions.

```
SquareGeo 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 (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 user documentation of XLiFE++ for more details).

```
Space Vk(_domain=omega, _interpolation=P1, _name="Vk", _optimizeNumbering);
```

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 :

$$-\int_\Omega v\Delta u dxdy + \int_\Omega vu dxdy = \int_\Omega vf dxdy$$

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

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

The 4 next lines in the program declare u and v and define the forms 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 dual to the unknown. Through the unknown, v is also defined on the same space.
- the right hand side needs the definition of the function f. Such function can be defined as a classical C++ function, but with a particular prototype. In this example, f (i.e. $cosxcosy$) 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` but it should be a `Complex` or a real/complex vector (`Reals`, `Complexes`).

```
Real cosxcosy(const Point& P, Parameters& pa = defaultParameters)
{
  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 build 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 : Once Matrix and vector being assembled, you can now choose the solver you want. Here, a conjugate gradient solver is used with an initial guess vector equal to the constant vector 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 user documentation of XLiFE++ 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. 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)

4 Initialization

4.1 The init function

As said in chapter 2, 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 !!! These parameters can also be set from dedicated command line options:

```
Command line options for XLiFE++ executables:
  -h                              prints the current help
  -j <num>, -jobs <num>,          define the number of threads. Default is
  --jobs <num>                    -1, meaning automatical behavior.
  -l <string>, -lang <string>,    define the language used for messages.
  --lang <string>                 Default is en.
  -vl <num>, -verbose-level <num>, define the verbose level.
  --verbose-level <num>           Default is 0.
  -v                              set the verbose level to 1.
  -vv                             set the verbose level to 2.
  -vvv                            set the verbose level to 3.
```

To deal with these command line options, you just have to give standard arguments to the `init` function:

```
init(argc, argc, _lang=fr);
```

4.2 Managing your own options

XLiFE++ provides an `Options` object to manage user options, that can be read from the command line or from files.
What is an option ? An option has:

- A name, that will be used to get the value of an option, as for a `Parameter`.

  
  An option name cannot start with a sequence of "-" characters. First characters must be alphanumerical.

- One or several keys used to define an option (key in a file or command line option). A key cannot be used twice and some keys are forbidden, due to some keys dedicated to system options (see section 2.2 about the `init` function).

- A value, that can be scalar (`Int`, `Number`, `Real`, `Complex` or `String`) or vector (`Ints`, `Numbers`, `Reals`, `Complexes` or `Strings`). This is the defautl value of the option and determines its data type. For vector types, size is not relevant.

This class provides a default constructor. In order to define a user option, you can use the function `addOption`:

```
Options opts;
opts.addOption("t1", 1.2);
opts.addOption("t2", "tata");
opts.addOption("t3", "-t", Complex(0.,1.));
opts.addOption("t4", Strings("-tu","-tv"), Reals(1.1, -2.4, 0.7));
opts.addOption("t5", Strings("tata","titi"));
```

To parse options from file and or command line arguments, you may use one of the following line:

```
opts.parse(argc, argv);
String filename="param.txt";
opts.parse(filename);
opts.parse(filename, argc, argv);
```

where `argv` and `argv` are the arguments of the `main` function dedicated to command line arguments of the executable. When using both filename and command line arguments, the latter are given priority.

Let's now talk about how use options in a file or command-line.
First option is named "t1". So it can be used through the key "t1", "-t1" or "–t1". Fourth option is named "t4" and has 2 aliases: "-tu" et "-tv". So it can be used through the key "t4", "-t4", "–t4", "-tu" or "-tv".

```
./exec --t1 2.5 -t2 tutu -t3 2.5 -0.1 -t4 2.2 -4.8 1.4  -t5 "tutu" toto "ta ta" titi
```

When passing options from command line, you consider a `Complex` value as 2 `Real` values. Furthermore, quotes are not necessary for `String` values, unless the value contains special characters such as blank spaces, escape characters, ... Here is an example of ascii file used to define options:

```
t1 3.7
t2 "ti ti"
t3 (2.5,-0.2)
t4 2.5 -2 3.4 4.7
t5 "titi" tutu "ta ta" toto
```

When passing options from a file, a `Complex` value is now written as a complex, namely real part and imaginary part are delimited by a comma and inside parentheses. As for command-line case, quotes are not necessary for `String` values, unless the value contains special characters such as blank spaces, escape characters, ...

4.3 Using XLiFE++ with global parameters

4.3.1 Global constants and objects

Some global constants are available and may be useful to you:

pi_ the π constant with the `Real` precision

i_ the imaginary number with the `Complex` precision

theEulerConst the Euler-Mascheroni constant

theTolerance the precision used to convergence in norms

theEpsilon the machine epsilon

Some usefulw objets are also available:

thePrintStream_ the dedicated file stream to the print.txt file generated while executing a program using XLiFE++.

theCout Using this stream object means using either the standard output stream `std::cout` and the previous file stream `thePrintStream_`. This is the reason why we recommend you to always use this stream .

eol An alias to `std::endl`

4.3.2 Multi-threading

XLiFE++, when configured with OpenMP, uses automatically multi-threading. We saw that the `init` function enables you to define the number of threads that will be used, but you cal also change the number of threads everywhere you want in your program. To modify its value, use the `numberOfThreads` function:

```
numberOfThreads(24);
...
numberOfThreads(12);
...
```

When used without argument, the routine returns the number of threads currently defined.

In multithreading environment, the stream `theCout` prints only on the thread 0.

4.3.3 The verbosity

We saw that the `init` function enables you to define the verbosity, but you cal also change the verbose level everywhere you want in your program. To modify its value, use the `verboseLevel(...)` function:

```
verboseLevel(10);
...
verboseLevel(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`) corresponding to outputs of each thread.

A External libraries

A.1 How to install 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.

On MAC OS, you may rely on the distribution provided by your favorite package manager (brew, port, ...)

On LINUX, you may rely on the distribution provided by your favorite package manager (apt-get, yum, rpm, ...)

On WINDOWS, It is very tricky to install BLAS and LAPACK by yourself, we highly recommend you to download binary files on the XLIFE++ website to avoid this annoying step

A.2 How to install UMFPACK library

UMFPACK is provided by SUITESPARSE (`http://faculty.cse.tamu.edu/davis/suitesparse.html`). 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 the same way, XLIFE++ offers you the ability to choose your UMFPACK installation:

- An old installation of UMFPACK alone (from FORTRAN sources);
- An installation of UMFPACK or SUITESPARSE from your favorite package manager;

On MAC OS, main package managers are brew and port

On LINUX, main package managers are apt-get, yum, rpm, ...

- An installation of SUITESPARSE sources by yourself. We highly recommend you to choose the release relying on CMAKE to be build.

On WINDOWS, we recommend you to download binary files on the XLiFE++ website to avoid this step.

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. See above.

A.3 How to install ARPACK library

ARPACK library can be obtained from the following URL: http://www.caam.rice.edu/software/ARPACK/. It requires BLAS and LAPACK routines. See above.

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.

On MAC OS, you may rely on the distribution provided by your favorite package manager (brew, port, ...)

If you use brew, and depending either on your OS or on the version of your compiler (g++ or clang++), installing ARPACK requires tu use it with BLAS and LAPACK libraries also provided by brew (OPENBLAS distribution), as default BLAS and LAPACK libraries found by CMAKE are the one provided by MAC OS. The same problem may occurs if you use the ARPACK library provided by XLiFE++

On LINUX, you may rely on the distribution provided by your favorite package manager (apt-get, yum, rpm, ...)

On WINDOWS, we recommend you to download binary files on the XLiFE++ website to avoid this step.

A.4 How to install MinGW 64 bits on WINDOWS

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.
The easiest and safest way to install MinGW-W64 is to avoid the use of an installer and to 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 CMAKE tutorial

CMAKE (http://cmake.org), as a cross-platform builder, only needs a configuration file named `CMakeLists.txt`, at the root directory of the software you develop, so-called *Source directory*. This configuration file is independent of the compiler and the platform. In order to generate native makefiles or workspaces of your favourite IDE (Visual Studio, Xcode, Eclipse, CodeBlocks, ...), CMAKE needs another directory, so-called *build directory*, in which files generated during the build process will be written.
The *build directory* is recommended not to be the *source directory*. For XLIFE++ compilation, we suggest you to set the *build directory* as a subdirectory of XLIFE++ install directory, with the name *build* for instance.

B.1 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.

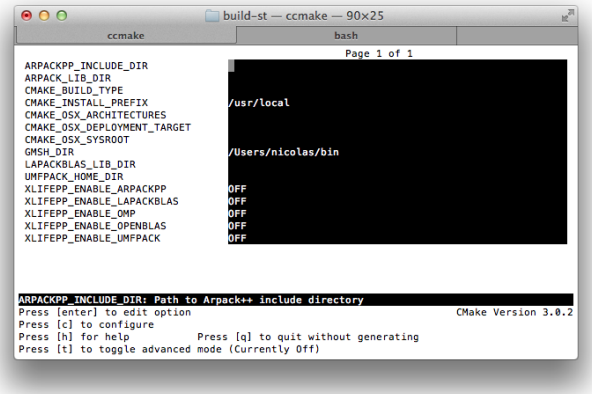


Figure B.1: `ccmake` (MacOS, Linux)

As it was said during introduction, a CMAKE call needs 2 directories : the *source directory* (command-line option -S) and the *build directory* (command-line option -B), but there is an easier way to call CMAKE: by calling it from the *build directory* and giving only the *source directory*, as in the following :

```
cmake relative/path/to/CMakeLists.txt [options]
ccmake [options] relative/path/to/CMakeLists.txt
```

B.2 Through GUI applications

When running CMAKE GUI application, you have to set the *source directory* and the *build directory*. 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`.

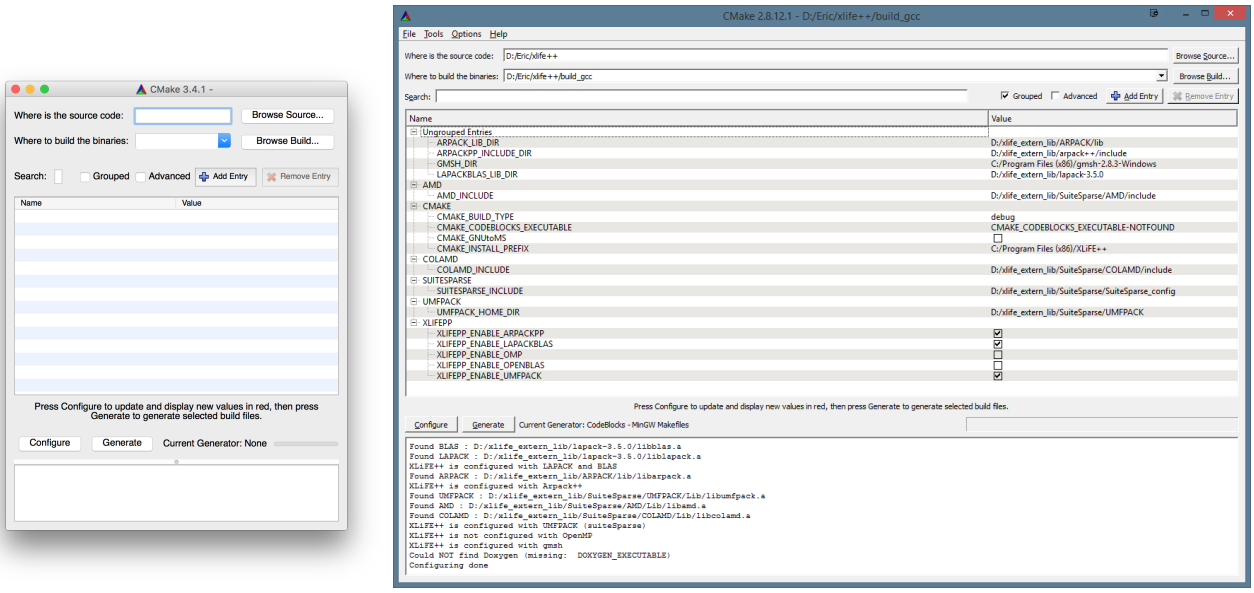


Figure B.2: CMAKE application (MacOS on the left and Windows on the right)

B.3 CMAKE options and cache entries

B.3.1 On the command line

The CMAKE help will give you the full list of command-line options. Only 2 are eventually useful:

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

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

```
cmake relative/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.

-D to define cache entries. Cache entries can be shown as dedicated configuration options for the software you want to build.

```
cmake relative/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

B.3.2 Through GUI applications

Available cache entries appears in the main frame after the first click on configure. You can organise them by grouping (cache entries beginning with the same prefix are in the same group whose name is the prefix in common) or by showing/hiding advanced cache entries.
Each time you change th evalue of one or several cache entries, you have to click on the *Configure* button again, before clicking on *Generate*