LifeV User Manual

Revision: 1.9, , UTC Printed: February 20, 2015

G. Fourestey, S. Deparis

Contents

List of Tables

Chapter 1

Generalities

1.1 Scope of the document

This is an informal document dedicated to amateur or inexperienced users of the software library *LIFE V* (life 5).

The major objectives of this document are:

1. to compile the software library,

2. to provide examples of its use.

For a more detailed overview of *LifeV*'s main features (management of boundary conditions, time and space discretization, algebraic solvers and preconditioners, etc), see the doxygen webpage: `https://cmcsforge.epfl.ch/doxygen/lifev/`.

1.2 Language and nomenclature convention

`typesetting font style` is used to indicate parts of computer code, configure shell scripts, command-prompt instructions and webpages.

1.3 Software Management

The software source, its documentation and all related documents (this one included) are kept in a repository under revision control using `git`[1]. Its goal is to provide tools to manage software development in a concurrent environment. See `http://git-scm.com/documentation` for a tutorial.

As mentioned above, a website *à la* Sourceforge[2] `http://cmcsforge.epfl.ch` has been set up to host the source code and help the software management. It requires that you open an account[3] there and ask to join the project *LifeV* using the link at the bottom of the developers' list. Once you would become a member, you will gain access to all the facilities: tracker, task manager, git repository, forums, document manager and a few other tools which are very useful if not absolutely essential to such a project.

Finally, if you expect a frequent use of the `git` repository we recommend to costumize the `ssh` and `ssh-agent` in order to gain acces without the need to type your password everytime you issue a command. Please refer to `http://mah.everybody.org/docs/ssh` in order to configure your ssh agent.

[1]git is the fast version control system.
[2]`http://www.sourceforge.net`
[3]`https://cmcsforge.epfl.ch/account/register.php`

We advice every user to apply to the list lifev-users on `http://groups.google.com` where one can get in touch with other users and developers.

1.4 Compiling LifeV

There are a few compilation tools and libraries we need to build and install before compiling *LifeV*, here is a short presentation. Note that, in addition to the following description, the complete installation steps are available on the following webpages:

1. `http://www.lifev.org/documentation/installation-tutorial`,
2. `https://cmcsforge.epfl.ch/projects/lifev/wiki/LifeV_on_MacOSX`.

In computer science, a library is a set of subroutines or classes used to develop software. Usually they are downloaded as a so called "tarball" file compressed using the `tar` command. There are different ways to compress libraries but the most common is to use the command `tar -cvf` and further compress "zip" it with `gzip`. If your tarball has the suffix `.tar.gz` equivalent to `.tgz`, you can decompress "unzip" it with `gunzip` followed by the name of the `.tar.gz` file and extract its contents using `tar -xvf` followed by the name of the `.tar` file. If you find the libraries compressed with other formats please refer to the unix manuals `man` or the numerous on-line documents for further information.

Software libraries need to be extracted, compiled and installed. In unix-like systems, the libraries `.a` and `.so` files are installed usually in the directory `/usr/lib`, while header files `.h` are installed in the `/usr/include` directory. Compilers search for libraries there by default, but in principle they can be installed anywhere you want as long as you pass the path to the library using the compiler flag `-L` immediately followed by the library path (e.g. `-L/path/to/lib`) and similarly for the header files using the compiler flag `-I` followed by the include path. Libraries compiled from source are usually installed in `/usr/local/lib`, `/usr/local/include`.

Libraries are usually created with the prefix `lib` followed by the name of the library and linked with the compiler flag `-l` followed by its name (e.g. `-lblas`).

Compilation Environment

LifeV depends on a number of tools at compilation time that are part of the autotools from the GNU project[4] available in most Linux OS:

- `g++-4.0` or newer (currently `4.9.2`).
- `mpi`, with preference to `openmpi`.
- `CMake 2.8.11` or newer (currently `3.1.0`).

In Mac OS X you get gcc in Xcode and cmake can be installed using MacPorts with the command `sudo port install cmake`. You can check the version of a command typing the command followed by `--help`, for example type `cmake --help`.

LifeV depends on several optimized libraries, you can check if you have them installed using the `locate` command (after updating the search database with `sudo updatedb`) followed by the name of the library, for example `locate liblapack.a`, or go to the `/usr/lib` directory and search on the list with `ls`. It is important to notice that some libraries are linked to others and they should be compatible, therefore you should build them in the order of dependency and with compatible flags and compilers.

These are the optimized libraries you need to have installed:

[4] `http://www.gnu.org`

- A version of `MPI`. The message passing interface for C and Fortran compilers. For example http://www.open-mpi.org/. Once installed you can check the necessary flags for its use by typing `mpicc --show`.
  On a Debian system the command `sudo apt-get install libopenmpi*` should do the trick.
  In Mac OS X using MacPorts install a fortran compiler typing `sudo port install gcc46` and openmpi with `sudo port install openmpi`. Note however that MPI should be natively installed if you installed XCode.

- `BOOST`. Libraries which extend the functionality of C++. Check if they exist on your computer, they are many libraries with the prefix `libboost`.
  If you need to install them, try `sudo apt-get install libboost*` on Debian systems or something similar for other Linux distros.
  In Mac OS X using MacPorts type `sudo port install boost`.
  If you need to compile from source, download the libraries at http://www.boost.org.
  Make sure you include the line "`using mpi;`" in the configuration text file `project-config.jam`.
  You can specify the path to install using the flag `--prefix=/path/` when running `./bjam install`. But most of the time cross compilation of this library won't work completely.

- `HDF5` If you don't have the library hdf5 installed in your system, you could use the `sudo apt-get install libhdf5-openmpi-dev` command on Debian systems or something similar for your particular distro. There are detailed instructions on-line on how to build it for other systems and with other options, see http://micro.stanford.edu/wiki/Install_HDF5#Build_and_Installation_from_Sources.
  In Mac OS X using MacPorts type `sudo port install hdf5` or build it from the sources to link it to the correct openmpi compilers.

- `BLAS`. On Debian systems run `sudo apt-get install libblas-dev`.
  In Mac OS X the system comes with blas and lapack as part of the Accelerate framework `-framework Accelerate`, and if using MacPorts type `sudo port install atlas` to install the atlas library (blas and lapack).
  To compile from source, get the libraries e.g. at https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2. To build just type `make`. To make use of the library remember to have the pthreads library and flag `-lpthread` while linking to the blas library `libgoto2_xxxxx_xx.xx.a`, whose exact name depends on the characteristics of your hardware.

- `LAPACK`. Fortran 90 Linear Algebra Routines for systems of simultaneous linear algebra equations, linear least-squares problems and matrix eigenvalue problems. You must pay attention to build the lapack using an optimized blas like the GotoBLAS (see above). Download it at http://www.netlib.org/lapack/. You need a fortran compiler (for example `gfortran`). Copy `make.inc.example` to `make.inc` and edit the path to the blas library followed by the flag `-lpthread` and type `make`.
  For a non-optimized version on a Debian system run `sudo apt-get install liblapack-dev`.

- `PARMETIS`. You can download ParMetis from http://glaros.dtc.umn.edu/gkhome/metis/parmetis/download. Set `CC=mpicc` in `Makefile.in`. and type `make`. In Mac OS X you need the include path flags `-I/usr/include` and `-I/usr/include/malloc`.

- `UMFPACK (now part of SuiteSparse)`. Set of routines for solving unsymmetric sparse linear systems.
  On a Debian system, install it with the command `sudo apt-get install libsuitesparse-dev`.
  To compile SuiteSparse from source, download it from http://faculty.cse.tamu.edu/davis/suitesparse.html and follow the instructions in the `README.txt` file (in particular, you'll want to edit the `SuiteSparse_config/SuiteSparse_config.mk` file according

to your configuration).
For Mac OS X you must uncomment the special options given for this system, so you can use the blas and lapack from atlas or from the Accelerate framework `-framework Accelerate`.

- `TRILINOS`. See the next section.

1.4.1 Trilinos compilation

LifeV depends on Trilinos, a set of object oriented C++ interfaces for packages like blas, lapack, parmetis, umfpack and many more. A copy of the source code is available for download at http://trilinos.org/download/.

After downloading, decompressing and extracting the tarball, you'll need to make a build directory anywhere you want to avoid build in the sources directory. (in the following script, we assume that the directories `trilinos` and `trilinos-build` are at the same level) Trilinos (latest version 11.12.1 at the time of writing) now requires the CMake build system 2.8.11 or newer. Go to the build directory and write a `do-configure` shell script like the following

```
#!/bin/bash

EXTRA_ARGS=$@

cmake \
    -D CMAKE_BUILD_TYPE:STRING=RELEASE \
    -D Trilinos_ENABLE_Amesos:BOOL=ON \
    -D Trilinos_ENABLE_Anasazi:BOOL=ON \
    -D Trilinos_ENABLE_AztecOO:BOOL=ON \
    -D Trilinos_ENABLE_Belos:BOOL=ON \
    -D Trilinos_ENABLE_Epetra:BOOL=ON \
    -D Trilinos_ENABLE_EpetraExt:BOOL=ON \
    -D Trilinos_ENABLE_Galeri:BOOL=OFF \
    -D Trilinos_ENABLE_Ifpack:BOOL=ON \
    -D Trilinos_ENABLE_Isorropia:BOOL=OFF \
    -D Trilinos_ENABLE_Kokkos:BOOL=ON \
    -D Trilinos_ENABLE_ML:BOOL=ON \
    -D Trilinos_ENABLE_TESTS:BOOL=OFF \
    -D Trilinos_ENABLE_Teuchos:BOOL=ON \
    -D Trilinos_ENABLE_ThreadPool:BOOL=ON \
    -D Trilinos_ENABLE_Tpetra:BOOL=ON \
    -D Trilinos_ENABLE_Triutils:BOOL=ON \
    -D Trilinos_ENABLE_Zoltan:BOOL=ON \
    \
    -D Trilinos_EXTRA_LINK_FLAGS:STRING="-lpthread" \
    -D TPL_ENABLE_Pthread:BOOL=ON \
    \
    -D TPL_ENABLE_BLAS:BOOL=ON \
    -D BLAS_INCLUDE_DIRS:PATH=/blas/include/dir/ \
    -D BLAS_LIBRARY_DIRS:PATH=/blas/lib/dir/ \
    -D BLAS_LIBRARY_NAMES:STRING="blas" \
    \
    -D TPL_ENABLE_LAPACK:BOOL=ON \
    -D LAPACK_INCLUDE_DIRS:PATH=/lapack/include/dir/ \
    -D LAPACK_LIBRARY_DIRS:PATH=/lapack/lib/dir/ \
    -D LAPACK_LIBRARY_NAMES:STRING="lapack" \
    \
    -D TPL_ENABLE_HDF5:BOOL=ON \
    -D HDF5_INCLUDE_DIRS:PATH/hdf5/include/dir/ \
    -D HDF5_LIBRARY_DIRS:PATH=/hdf5/lib/dir/ \
    \
    -D TPL_ENABLE_UMFPACK:BOOL=ON \
    -D UMFPACK_INCLUDE_DIRS:PATH=/umfpack/include/dir/ \
    -D UMFPACK_LIBRARY_DIRS:PATH=/umfpack/lib/dir/ \
    -D UMFPACK_LIBRARY_NAMES:STRING="umfpack;amd" \
    \
    -D TPL_ENABLE_MPI:BOOL=ON \
    -D MPI_BASE_DIR:PATH=/usr/lib/openmpi/ \
    -D MPI_BIN_DIR:PATH=/usr/bin \
    \
    -D TPL_ENABLE_ParMETIS:BOOL=ON \
    -D ParMETIS_LIBRARY_DIRS:PATH=/parmetis/lib/dir/ \
    \
    -D CMAKE_INSTALL_PREFIX:PATH=./ \
    $EXTRA_ARGS \
    ../trilinos/
```

Simply modify the paths of libraries according to your particular configuration and run the shell script (`chmod +x do-configure && ./do-configure`). For example, instead of `lapack_library_name` you should type the name of your lapack library without the `lib` prefix and the `.a` suffix. The prefix and suffix are automatically added by CMake.
If SuiteSparse was compiled from source the `UMFPACK_LIBRARY_NAMES` variable has to be modified so that it reads `"umfpack;suitesparseconfig;cholmod;colamd;amd"`
As an alternative to the above script, you can run

```
ccmake ../lifev
```

to get a graphical configuration menu (however `ccmake` needs `libncurses` to be installed), with many more options.
After the configuration is done, just type

```
make
```

that will compile the static files and further

```
make install
```

that will create and install the library files in two subdirectories `lib` and `include`, where it will respectively pack the objects files into library files (.a and .la files) and copy the include files (.h or .hpp files).
The Trilinos library is now installed in the build directory you created.

1.4.2 Compilation from git

You need first to have an account on `http://cmcsforge.epfl.ch` and be part of the *LifeV* project, see 1.3.
First, you need to checkout *LifeV*. `git` has been configured to use `ssh` and your `ssh` keys to access the repository via `ssh` without entering your password. When your ssh agent is properly configured, send your public key to the local administrator, such that it can be included in the gitolite configuration. Then you will be able to access the repositories without password.
It is now time to download and compile the code. Just type

```
git clone git@cmcsforge.epfl.ch:lifev.git lifev
```

and go to the newly created directory

```
cd lifev
```

Second, you must make a build directory apart from the lifev sources directory, for example in your home you can have a `lib` directory with a `lifev` subdirectory and further an optimized version subdirectory `opt` or the debugging mode subdirectory `debug`, or something similar according to your own taste.
Third, you have to execute the following `do-configure` shell script (again, modified to suite your configuration) in the `opt` directory. It will automatically check the availability of the needed components for *LifeV* compilation :

```
#!/bin/bash

```

```
EXTRA_ARGS=$@

TRILINOS_BUILD_DIR=/trilinos/build/dir/

cmake \
-D CMAKE_BUILD_TYPE:STRING=RELEASE \
\
-D TPL_ENABLE_MPI:BOOL=ON \
\
-D ParMETIS_INCLUDE_DIRS:PATH=/parmetis/include/dir/ \
-D ParMETIS_LIBRARY_DIRS:PATH=/parmetis/lib/dir/ \
\
-D TPL_ENABLE_BLAS:BOOL=ON \
-D BLAS_INCLUDE_DIRS:PATH=/blas/include/dir/ \
-D BLAS_LIBRARY_DIRS:PATH=/blas/lib/dir/ \
-D BLAS_LIBRARY_NAMES:STRING="blas" \
\
-D TPL_ENABLE_LAPACK:BOOL=ON \
-D LAPACK_INCLUDE_DIRS:PATH=/lapack/include/dir/ \
-D LAPACK_LIBRARY_DIRS:PATH=/lapack/lib/dir/ \
-D LAPACK_LIBRARY_NAMES:STRING="lapack" \
\
-D TPL_ENABLE_HDF5:BOOL=ON \
-D HDF5_INCLUDE_DIRS:PATH=/hdf5/include/dir/ \
-D HDF5_LIBRARY_DIRS:PATH=/hdf5/lib/dir/ \
\
-D TPL_ENABLE_Boost:BOOL=ON \
-D Boost_INCLUDE_DIRS:PATH=/boost/include/dir/ \
\
-D TPL_ENABLE_Trilinos:STRING=ON \
-D Trilinos_DIR:PATH=$TRILINOS_BUILD_DIR/lib/cmake/Trilinos \
-D Trilinos_INCLUDE_DIRS:PATH=$TRILINOS_BUILD_DIR/include/ \
-D Trilinos_LIBRARY_DIRS:PATH=$TRILINOS_BUILD_DIR/lib/ \
\
-D LifeV_VERBOSE_CONFIGURE:BOOL=OFF \
-D CMAKE_VERBOSE_MAKEFILE:BOOL=OFF \
\
-D LifeV_ENABLE_STRONG_CXX_COMPILE_WARNINGS:BOOL=OFF \
\
-D LifeV_ENABLE_ALL_PACKAGES:BOOL=ON \
-D LifeV_ENABLE_TESTS:BOOL=ON \
-D LifeV_ENABLE_EXAMPLES:BOOL=ON \
\
-D CMAKE_INSTALL_PREFIX:PATH=./ \
$EXTRA_ARGS \
../lifev
```

Do the same in the `debug` directory, replacing the first line by

```


\noindent Finally, you just have to use \ixv{make} to compile \lifev libraries and ↩
    documentation.
Enter
\begin{lstlisting}
make -j n
make install
```

where `n` is the number of parallel jobs.
Be careful because `do-configure` will fail if you have already compiled *LifeV* in the source directory. Therefore is not a good idea to build inside the sources.

1.4.3 Compilation from Official Distribution

The *LifeV* project provides releases, they are named using the following convention

`lifev-x.y.z.tar.gz`

Here is what you have to do:

1. download *LifeV* release `lifev-x.y.z.tar.gz`
2. unpack it

```
tar -xzf lifev-x.y.z.tar.gz
```

3. configure it following the instructions of the previous section,

4. compile and install it

```
make -j n
make install
```

1.4.4 Compiling Testsuites

LifeV comes with testsuites covering a lot of features. They are located in different directories, mainly depending on the physical or technical aspects they are concerned with. For example, you can find a number of tests in the `core` directory (`lifev/lifev/core/testsuite`) but `darcy, fsi, navier_stokes, structure` are other directories where you can find tests.

All these tests are automatically compiled once you have installed *LifeV*. To run them just type

```
make test
```

Chapter 2

Learning by examples

2.1 Reading data

In order to read input data, LifeV is integrated with the open-source library `GetPot` (http://getpot.sourceforge.net/). GetPot allows to easily handle the data regarding the different phases of the simulation, typically providing the mesh name, the discretization order, the physical parameters, the solver information and the time step (if any).
`GetPot` needs the name of the input file that can be linked through the flags "-f" or "–file" while launching the program, e.g.

```
$ ./myProgram.exe -f myData
```

The GetPot object allows to read the data from the file, and is constructed thanks to the name of the input file. If no name is provided, then LifeV uses the default input name "data".

```
GetPot command_line(argc,argv);
const std::string dataFileName = command_line.follow("data", 2, "-f","--file");
GetPot dataFile(dataFileName);
```

The input file must have a tree-structure, an example is as follows

```
# -*- getpot -*- (GetPot mode activation for emacs)
#----------------------------------------------------
#       Data file for the Laplacian example
#----------------------------------------------------

[finite_element]
    degree                          = P1

[mesh]
    nx                              = 40
    ny                              = 40
    nz                              = 40
    overlap                         = 0
    verbose                         = false

[prec]
    prectype                        = Ifpack # Ifpack or ML
    displayList                     = false

    [./ifpack]
        overlap                     = 2

        [./fact]
            ilut_level-of-fill      = 1
            drop_tolerance          = 1.e-5
            relax_value             = 100

    [../amesos]
        solvertype                  = Amesos_KLU #Amesos_KLU or Amesos_Umfpack

    [../partitioner]
        overlap                     = 0

    [../schwarz]
        reordering_type             = none #metis, rcm, none
        filter_singletons           = true

    [../]
[../]
```

and the data can be read as in folders. For instance, if we have the previous data file and we want to print the mesh sizes in the three directions, we only need to type

```
std::cout << "Number of elements in each direction"
          << "x: " dataFile( "mesh/nx", 15 )
          << "y: " dataFile( "mesh/ny", 15 )
          << "z: " dataFile( "mesh/nz", 15 )
          << std::endl;
```

where the values "15" are set in LifeV as default sizes. More generally, in every LifeV-based program, we need to access in the same manner the data through the GetPot object, providing also a safe default value in case some variables are not specifically set.
You can browse the default data file in every testsuite directory to see examples. Generally, some entries are compulsory (e.g. the mesh name for unstructured meshes), on the other hand others are filled with a default value if not specified in the input file.

2.2 The Poisson problem

In this section, we go through a first example dealing with the Poisson equation. At first we introduce the mathematical setting and well-posedness of the problem, presenting its finite dimensional formulation and the error estimates. Then we explain how to solve the Poisson problem using LifeV, going through the different stages that characterize the simulation. More in particular, we will cover the following topics:

- the preamble: including the headers and configuring MPI
- the construction of a structured mesh and the definition of the finite elements space
- the assembly of the stiffness matrix, the right-hand-side and the setting of boundary conditions
- the preconditioner and the solution of the linear system
- the exporting of data and the post-processing

2.2.1 Variational formulation and finite element discretization

Let $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ be a regular open bounded domain and let $\partial\Omega$ be its boundary such that $\partial\Omega = \Gamma_D \cup \Gamma_N$, $\mathring{\Gamma}_D \cap \mathring{\Gamma}_N = \emptyset$, our problem reads

$$\begin{cases} -\Delta u = f & \mathbf{x} \in \Omega, \\ u = g(\sigma) & \sigma \in \Gamma_D, \\ \partial_n u = h(\sigma) & \sigma \in \Gamma_N, \end{cases}$$

where $f = f(\mathbf{x})$ denotes the source term and $g(\sigma), h(\sigma)$ denote the Dirichlet and Neumann boundary conditions, respectively. Starting from the differential equation, we can derive the weak formulation of the problem. We introduce the spaces

$$V = H^1_d(\Omega) = \Big\{ v \in H^1(\Omega) : v|_{\Gamma_D} = 0 \Big\} \tag{2.1}$$

and

$$V_g = \Big\{ v \in H^1(\Omega) : v|_{\Gamma_D} = g(\sigma) \Big\}. \tag{2.2}$$

Finally, our problem reads: find $u \in V_g$, such that

$$a(u, v) = Fv \qquad \forall v \in V, \tag{2.3}$$

where

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v d\mathbf{x} \qquad \text{and} \qquad Fv = \int_\Omega f(\mathbf{x}) d\mathbf{x} + \int_{\Gamma_N} h(\sigma) d\sigma. \tag{2.4}$$

Under appropriate hypothesis, problem (2.3) is well-posed. In order to obtain a discrete solution of problem (2.3) based on the finite element method, we introduce at first a partition $\mathcal{T}_h$ of the domain Ω, and the finite-dimensional space

$$V_h = X^r_h = \Big\{ v_h \in C^0(\bar{\Omega}) : v_h|_K \in \mathbb{P}^r(), \forall K \in \mathcal{T}_h \Big\}, \tag{2.5}$$

where $\mathbb{P}^r$ denotes the polynomial functions of degree lower than or equal to r. For sake of simplicity we suppose homogeneous Dirichlet boundary conditions, i.e. $g = 0$, in case $g \neq 0$, it is possible to operate a lifting to ring back to the homogeneous case.
Finally, the finite dimensional problem reads: find $u_h \in V_h$ such that

$$a(u_h, v_h) = Fv_h \qquad v_h \in V_h. \tag{2.6}$$

Starting from equation (2.6), it is possible to write the problems in the form of a linear system, and it is possible to prove that under appropriate assumptions about the data and the regularity of the exact solution u, the discrete solution u_h satisfies the following error estimates

$$\|u - u_h\|_{L^2(\Omega)} \leq Ch^{r+1}|u|_{H^{r+1}}, \tag{2.7}$$

$$\|u - u_h\|_{H^1(\Omega)} \leq Ch^{r}|u|_{H^{r+1}}, \tag{2.8}$$

where h denotes the characteristic size of the mesh, r the polynomial degree employed and C is a constant independent of h and u, see e.g. [1].

2.2.2 LifeV simulation

To the aim of solving problem (2.6) using LifeV, we set $\Omega = (-1, 1)^3$, $\Gamma_D = \partial\Omega$ and the data f, g, h such that the exact solution of the Poisson problem is $u(\mathbf{x}) = sin(\pi x)sin(\pi y)sin(\pi z)$.

Preamble: headers and MPI configuration

We first include the different headers containing the data structures and the algorithms we are employing along the simulation: At first the Epetra data structures that allow the use of MPI

- the definition of the MPI environment, that is made exploiting the Epetra framework

```
#include <Epetra_ConfigDefs.h>
#ifdef EPETRA_MPI
#include <mpi.h>
#include <Epetra_MpiComm.h>
#else
#include <Epetra_SerialComm.h>
#endif
```

- the definition of the basis structures of LifeV, in particular meshes, finite elements spaces and expressions

```
#include <lifev/core/LifeV.hpp>
#include <lifev/core/util/LifeChronoManager.hpp>
#include <lifev/core/mesh/MeshPartitioner.hpp>
#include <lifev/core/mesh/RegionMesh3DStructured.hpp>
#include <lifev/core/mesh/RegionMesh.hpp>
#include <lifev/core/array/MatrixEpetra.hpp>
#include <lifev/core/fem/BCManage.hpp>
#include <lifev/eta/fem/ETFESpace.hpp>
#include <lifev/eta/expression/BuildGraph.hpp>
#include <lifev/eta/expression/Integrate.hpp>
#include <Epetra_FECrsGraph.h>
```

- the definition of the solver and the exporting routines

```
#include <Teuchos_ParameterList.hpp>
#include <Teuchos_XMLParameterListHelpers.hpp>
#include <Teuchos_RCP.hpp>
#include <lifev/core/algorithm/LinearSolver.hpp>
#include <lifev/core/algorithm/PreconditionerIfpack.hpp>
#include <lifev/core/filter/ExporterHDF5.hpp>
```

- some other useful classes

```
#include <boost/shared_ptr.hpp>
#include <lifev/eta/examples/laplacian/laplacianFunctor.hpp>
```

Next, we define the LifeV namespace

```
using namespace LifeV;
```

Structured mesh and finite element spaces

After having read the datafile as explained in Section 2.1, we build a cubic structured mesh and we divide it among the processors which are running the simulation

```
// Mesh
typedef RegionMesh< LinearTetra > mesh_Type;
boost::shared_ptr< mesh_Type > fullMeshPtr ( new mesh_Type ( Comm ) );

// Building structured mesh (in this case a cube)
regularMesh3D ( *fullMeshPtr, 0,
        dataFile( "mesh/nx", 15 ), dataFile( "mesh/ny", 15 ),
        dataFile( "mesh/nz", 15 ), dataFile ( "mesh/verbose", false ),
        2.0, 2.0, 2.0, -1.0,-1.0, -1.0 );

// Partitioning mesh, possibly with overlap
const UInt overlap ( dataFile( "mesh/overlap", 0 ) );
boost::shared_ptr< mesh_Type > localMeshPtr;

MeshPartitioner< mesh_Type > meshPart;
```

```

if ( overlap )
{
    meshPart.setPartitionOverlap ( overlap );
}

meshPart.doPartition ( fullMeshPtr, Comm );
localMeshPtr = meshPart.meshPartition();

// Clearing global mesh
fullMeshPtr.reset();
```

We notice that the domain origins from the point $(-1-1,-1)$ and has a length of 2 in each dimension. The `overlap` variable states the number of layers that are shared by two processors with contiguous subdomains.
We next define the finite element space, whose dimension is read by the input file. Then we build the corresponding Expression Template finite element space, whose use allows the user to adopt the templated expressions that refer to the weak formulation of the problem.

```
// Finite element space
typedef FESpace< mesh_Type, MapEpetra >                          uSpaceStd_Type;
typedef boost::shared_ptr< uSpaceStd_Type >                      uSpaceStdPtr_Type;
typedef ETFESpace< mesh_Type, MapEpetra, 3, 1 >                  uSpaceETA_Type;
typedef boost::shared_ptr< uSpaceETA_Type >                      uSpaceETAPtr_Type;
typedef FESpace<mesh_Type, MapEpetra >::function_Type            function_Type;

// Defining finite elements standard and Expression Template spaces
uSpaceStdPtr_Type uFESpace ( new uSpaceStd_Type ( localMeshPtr,
                    dataFile( "finite_element/degree", "P1" ), 1, Comm ) );
uSpaceETAPtr_Type ETuFESpace ( new uSpaceETA_Type ( localMeshPtr,
                    & ( uFESpace->refFE() ), & ( uFESpace->fe().geoMap() ), ↩
                        Comm ) );
```

Assembly of the stiffness matrix, the right-hand-side and the setting of boundary conditions

In order to assembly the system that corresponds to the finite element formulation, we need to build at first the graph that contains the topology of the matrix and then assemblying the matrix by constructing the elements $a(\phi_j, \phi_i)$

```
// Matrices and graphs
typedef Epetra_FECrsGraph                                        graph_Type;
typedef boost::shared_ptr<Epetra_FECrsGraph>                     graphPtr_Type;
typedef MatrixEpetra< Real >                                     matrix_Type;
typedef boost::shared_ptr< MatrixEpetra< Real > >                matrixPtr_Type;

graphPtr_Type systemGraph;
matrixPtr_Type systemMatrix;

if ( overlap )
{
    systemGraph.reset ( new graph_Type ( Copy, * ( uFESpace->map().map( ↩
        Unique ) ), 50, true ) );
}
else
{
    systemGraph.reset ( new graph_Type ( Copy, * ( uFESpace->map().map( ↩
        Unique ) ), 50 ) );
}

{
    using namespace ExpressionAssembly;

    buildGraph (
            elements ( localMeshPtr ),
            uFESpace->qr(),
                ETuFESpace ,
                ETuFESpace ,
                dot ( grad ( phi_i ) , grad ( phi_j ) )
            )
            >> systemGraph;
}

systemGraph->GlobalAssemble();

if ( overlap )
{
systemMatrix.reset( new matrix_Type ( ETuFESpace->map(), *systemGraph, true )↩
     );
}
else
{
systemMatrix.reset( new matrix_Type ( ETuFESpace->map(), *systemGraph ) );
}

// Clearing problem s matrix
systemMatrix->zero();

{
    using namespace ExpressionAssembly;

    integrate (
                elements ( localMeshPtr ),
                uFESpace->qr(),
                ETuFESpace,
                ETuFESpace,
                dot ( grad ( phi_i ) , grad ( phi_j ) )
            )
            >> systemMatrix;
}
```

Next, we build the solution and the right hand side vectors. For the latter, we employ the `laplacianFunctor` class, that is constructed by providing the function that describes the source term

```
// Vectors
typedef VectorEpetra                                    vector_Type;
typedef boost::shared_ptr<VectorEpetra>                 vectorPtr_Type;

vectorPtr_Type rhsLap;
vectorPtr_Type solutionLap;

if ( overlap )
{
    rhsLap.reset ( new vector_Type ( uFESpace->map(), Unique, Zero ) );
    solutionLap.reset ( new vector_Type ( uFESpace->map(), Unique, Zero ) );
}
else
{
    rhsLap.reset ( new vector_Type ( uFESpace->map(), Unique ) );
    solutionLap.reset ( new vector_Type ( uFESpace->map(), Unique ) );
}

rhsLap->zero();
solutionLap->zero();

Real sourceFunction ( const Real& /*t*/, const Real& x,
                      const Real& y, const Real& z,
                      const ID& /*i*/)
{
    return 3 * M_PI * M_PI
             * sin( M_PI * x )  * sin( M_PI * y ) * sin( M_PI * z );
}

boost::shared_ptr<laplacianFunctor< Real > >  laplacianSourceFunctor ( new ↩
    laplacianFunctor< Real >( sourceFunction ) );

{
    using namespace ExpressionAssembly;

    integrate (
                elements ( localMeshPtr ),
                uFESpace->qr(),
                ETuFESpace,
                eval(laplacianSourceFunctor, X) * phi_i
            )
            >> rhsLap;
}
```

We remark here that the use of the expression templates allows the user to define the equivalent of the weak formulation, in particular the elements of the matrix are built using the test and trial functions phi_i and phi_j, and setting the quadrature rule to adopt.

Boundary conditions

We explain here how to impose the boundary conditions. At first we give an identification number to the faces of our domain, and then define the function g that is assigned to the boundarues, in this case we have homogeneous Dirichlet conditions, so we construct the function `zeroFunction` that returns 0. for every value of the spatial domain. In LifeV, it is possible to impose the boundaries through a BCHandler object, which imposes on every boundary the corresponding Dirichlet datum by providing the keyword `Essential`.

```
// Cube s walls identifiers
const int BACK   = 1;
const int FRONT  = 2;
const int LEFT   = 3;
const int RIGHT  = 4;
const int BOTTOM = 5;
const int TOP    = 6;

Real zeroFunction (const Real& /*t*/, const Real& /*x*/, const Real& /*y*/, ↩
    const Real& /*z*/, const ID& /*i*/)
{
    return 0.;
}

BCHandler bcHandler;

BCFunctionBase ZeroBC ( zeroFunction );
BCFunctionBase OneBC ( nonZeroFunction );

bcHandler.addBC( "Back",   BACK,   Essential, Scalar, ZeroBC, 1 );
bcHandler.addBC( "Left",   LEFT,   Essential, Scalar, ZeroBC, 1 );
bcHandler.addBC( "Top",    TOP,    Essential, Scalar, ZeroBC, 1 );
bcHandler.addBC( "Front",  FRONT,  Essential, Scalar, ZeroBC, 1 );
bcHandler.addBC( "Right",  RIGHT,  Essential, Scalar, ZeroBC, 1 );
bcHandler.addBC( "Bottom", BOTTOM, Essential, Scalar, ZeroBC, 1 );

bcHandler.bcUpdate( *uFESpace->mesh(), uFESpace->feBd(), uFESpace->dof() );
bcManage ( *systemMatrix, *rhsLap, *uFESpace->mesh(), uFESpace->dof(),
            bcHandler, uFESpace->feBd(), 1.0, 0.0 );
```

Preconditioning and solving the system

Since our computation employs a parallel architecture. before solving the linear system we need to assemble both the matrix and the right hand side

```
systemMatrix->globalAssemble();
rhsLap->globalAssemble();
```

Next we set the solver parameters according to the input file `SolverParamList.xml` and the preconditioner that employs the Additive Schwarz method

```
// Solver and preconditioner
```

```
typedef LinearSolver::SolverType                        solver_Type;
typedef LifeV::Preconditioner                           basePrec_Type;
typedef boost::shared_ptr<basePrec_Type>                basePrecPtr_Type;
typedef PreconditionerIfpack                            prec_Type;
typedef boost::shared_ptr<prec_Type>                    precPtr_Type;

LinearSolver linearSolver ( Comm );
linearSolver.setOperator ( systemMatrix );

Teuchos::RCP< Teuchos::ParameterList > aztecList = Teuchos::rcp ( new Teuchos↩
    ::ParameterList );
aztecList = Teuchos::getParametersFromXmlFile ( "SolverParamList.xml" );

linearSolver.setParameters ( *aztecList );

prec_Type* precRawPtr;
basePrecPtr_Type precPtr;
precRawPtr = new prec_Type;
precRawPtr->setDataFromGetPot ( dataFile, "prec" );
precPtr.reset ( precRawPtr );

linearSolver.setPreconditioner ( precPtr );
```

And finally we solve the system

```
linearSolver.setRightHandSide( rhsLap );
linearSolver.solve( solutionLap );
```

Exporting and post processing

We set the exporter that employs the library HDF5, the result is then visible using a post processing tool, for instance Paraview (http://www.paraview.org/).

```
// Setting exporter
ExporterHDF5< mesh_Type > exporter ( dataFile, "exporter" );
exporter.setMeshProcId( localMeshPtr, Comm->MyPID() );
exporter.setPrefix( "laplace" );
exporter.setPostDir( "./" );
exporter.addVariable ( ExporterData< mesh_Type >::ScalarField, "temperature",↩
    uFESpace, solutionLap, UInt ( 0 ) );
exporter.postProcess( 0 );
exporter.closeFile();
```

Now, suppose we want to investigate the trend of the error by varying the mesh size, to check numerically the estimates (2.7), (2.8). At first, we need to define the function u and its gradient ∇u in our program, in the same way we defined the source term f

```
Real uExactFunction (const Real& /*t*/, const Real& x, const Real& y, const ↩
    Real& z, const ID& /*i*/)
{
    return sin( M_PI * y ) * sin( M_PI * z ) * sin ( M_PI * x );
}

function_Type uEx(uExactFunction);

VectorSmall< 3 > uGradExactFunction (const Real& /*t*/, const Real& x, const ↩
    Real& y, const Real& z, const ID& /*i*/)
{
    VectorSmall< 3 > v;

    v[0] = M_PI * cos( M_PI * x ) * sin( M_PI * y ) * sin( M_PI * z );
    v[1] = M_PI * sin( M_PI * x ) * cos( M_PI * y ) * sin( M_PI * z );
    v[2] = M_PI * sin( M_PI * x ) * sin( M_PI * y ) * cos( M_PI * z );

    return v;
}

```

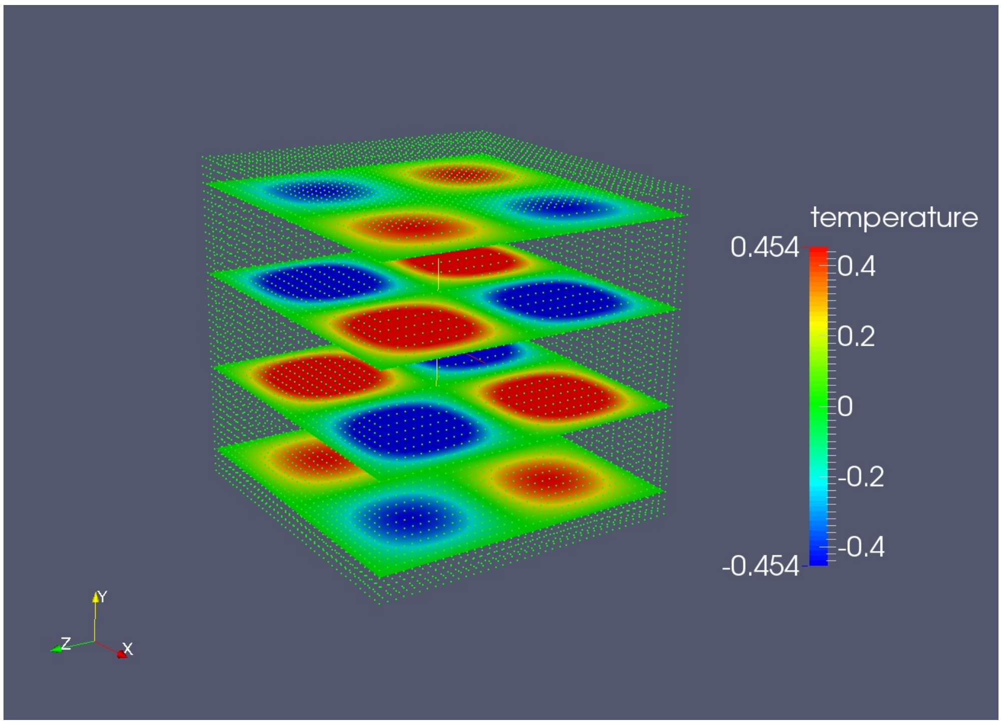

Figure 2.1: Result of Poisson equation with homogeneous Dirichlet boundary conditions

```
boost::shared_ptr<laplacianFunctor< Real > >  laplacianExactFunctor ( new ↩
    laplacianFunctor< Real >( uExactFunction ) );
boost::shared_ptr<laplacianFunctor< VectorSmall<3> > >  ↩
    laplacianExactGradientFunctor ( new laplacianFunctor< VectorSmall<3> >( ↩
    uGradExactFunction ) );
```

Next, we need to evaluate the norms following their definition, and to do that we still employ the Expression Templates provided in LifeV

```
Real L2ErrorLap = 0.0;
Real TotL2ErrorLap = 0.0;

Real H1SeminormLap = 0.0;
Real TotH1SeminormLap = 0.0;

{
    using namespace ExpressionAssembly;

    integrate (
            elements ( localMeshPtr ),
            uFESpace->qr(),
            ( eval(laplacianExactFunctor, X) - value (ETuFESpace, *↩
                solutionLap) )
            * ( eval(laplacianExactFunctor, X) - value (ETuFESpace, *↩
                solutionLap) )
        ) >> L2ErrorLap;

}

{
    using namespace ExpressionAssembly;

    integrate (
            elements ( localMeshPtr ),
            uFESpace->qr(),
```

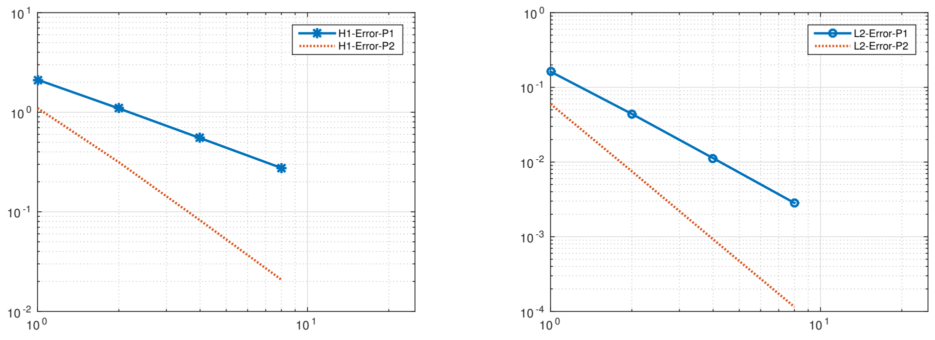

Figure 2.2: H^1 norm (left) and L^2 norm (right) vs mesh size for $r = 1, 2$.

```
                dot( eval(laplacianExactGradientFunctor, X) - grad( ETuFESpace, *↩
                     solutionLap ),
                eval(laplacianExactGradientFunctor, X) - grad( ETuFESpace, *↩
                     solutionLap ) )
           ) >> H1SeminormLap;

}
Comm->Barrier();
```

Finally, we gather the results of the different processors and print the norm. We collect the results for different mesh sizes and perform a convergence analysis, whose results are shown in Fig. 2.2.

```
Comm->SumAll (&L2ErrorLap, &TotL2ErrorLap, 1);
Comm->SumAll (&H1SeminormLap, &TotH1SeminormLap, 1);

if (verbose)
{
    std::cout << "TotError in L2 norm is "
              << sqrt( TotL2ErrorLap ) << std::endl;
    std::cout << "TotError in H1 norm is "
              << sqrt( TotL2ErrorLap + TotH1SeminormLap ) << std::endl;
}
```

Bibliography

[1] A. Quarteroni. *Numerical Models for Differential Problems.* Modeling, Simulation and Applications. Springer, Heidelberg, DE, 2009. Written for students of bachelor and master courses in scientific disciplines: engineering, mathematics, physics, computational sciences, and information science.