LifeV User Manual

Revision: 1.7, , UTC Printed: November 12, 2010

G. Fourestey, S. Deparis

This manual is for LifeV (version 1.3.1, October 2010), a library for scientific computing using finite elements, specially aimed at fluid-structure interaction and blood flow simulation.

Contents

1 Generalities **4**
- 1.1 Scope of the document . . . 4
- 1.2 Language and nomenclature convection . . . 4
- 1.3 Software Management . . . 4
- 1.4 Compiling LifeV . . . 4
  - 1.4.1 Trilinos compilation . . . 6
  - 1.4.2 Compilation from git . . . 7
  - 1.4.3 Compilation from Official Distribution . . . 9
  - 1.4.4 Compiling Testsuite . . . 9

2 Learning by examples **11**
- 2.1 The Stokes Problem . . . 11
- 2.2 The Navier-Stokes Problem . . . 23
- 2.3 Fluid/Structure interactions . . . 25

List of Tables

2.1 Description of the discretization parameters. 17
2.2 Boundary Condition parameters description 18
2.3 Reference Finite Element parameters 19
2.4 Quadrature Rule description 19
2.5 Main parameters for the Trilinos solver 21
2.6 FSI problem data file parameters 30

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.

1.2 Language and nomenclature convection

`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.

A web site *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'd become a member, you'll 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.

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

1.4 Compiling LifeV

There are a few compilation tools and libraries we need to build and install before compiling *LifeV*.

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 and similarly for the header files using the compiler flag `-I` followed by the include path.

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.

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:

- `libtool 1.5` or newer (currently 2.2.6)
- `automake 1.7` or newer (currently 1.9.6)
- `autoconf 2.52` or newer (currently 2.65)
- `g++-4.0` or newer (currently 4.4.4)
- `CMake 2.8` or newer

In Mac OS X you get the autotools in Xcode and cmake can be installed using MacPorts `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 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:

- 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`. In Mac OS X using MacPorts install a fortran compiler typing `sudo port install gcc46` and openmpi with `sudo port install openmpi`.
- `BOOST`. Libraries to 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*` in devian systems or something similar in others. If you can't do something like that then download at http://www.boost.org. Make sure you

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

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. In Mac OS X using MacPorts type `sudo port install boost`.

- `HDF5` If you don't have the library hdf5 installed in your system, you could use the `sudo apt-get install libhdf5-openmpi-dev` instruction in devian linux systems or something similar for your particular system. There are detailed instructions on-line on how to build it for other systems and with other options, for example at 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`. For example get http://www.tacc.utexas.edu/tacc-projects/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. 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).

- `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 of last item. Download 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`.

- `PARMETIS`. Download 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`. Set of routines for solving unsymmetric sparse linear systems. Download from http://www.cise.ufl.edu/research/sparse/umfpack/. Requires the packages `UFconfig` and `AMD`. Place amd, ufconfig and umfpack in the same parent directory. Modify the file `UFconfig.mk` to specify the C compiler `CC = gcc`, `CPLUSPLUS=g++`, `CFLAGS=-O3 -fexceptions` and the blas, lapack and metis paths and libraries including the `-lpthread` flag. For Mac OS X you must uncheck the special options given for this system, you can use the blas and lapack from atlas or from the Accelerate framework `-framework Accelerate`. Type `make` in the umfpack directory. Move the files in amd `Lib` and `Include` to the umfpack `Lib` and `Include` directories. Finally move `UFconfig.h` to the corresponding `Include` directory of umfpack.

- `TRILINOS`.

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.sandia.gov/packages/.

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. The autotools build is no longer available after the 10.0 release. Trilinos now requires the CMake build system, version 2.8 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_TESTS:BOOL=OFF \
    -D Trilinos_ENABLE_Epetra:BOOL=ON \
    -D Trilinos_ENABLE_Tpetra:BOOL=ON \
    -D Trilinos_ENABLE_Kokkos:BOOL=ON \
    -D Trilinos_ENABLE_EpetraExt:BOOL=ON \
    -D Trilinos_ENABLE_AztecOO:BOOL=ON \
    -D Trilinos_ENABLE_Amesos:BOOL=ON \
    -D Trilinos_ENABLE_Teuchos:BOOL=ON \
    -D Trilinos_ENABLE_Ifpack:BOOL=ON \
    -D Trilinos_ENABLE_ThreadPool:BOOL=ON \
    -D Trilinos_ENABLE_ML:BOOL=ON \
    -D Trilinos_ENABLE_Triutils:BOOL=ON \
    -D Trilinos_ENABLE_Zoltan:BOOL=ON \
    -D Trilinos_ENABLE_Galeri:BOOL=OFF \
    -D Trilinos_ENABLE_Isorropia:BOOL=OFF \
    -D Trilinos_EXTRA_LINK_FLAGS:STRING="-lpthread" \
    -D TPL_ENABLE_UMFPACK:BOOL=ON \
    -D UMFPACK_LIBRARY_DIRS:PATH=/umfpack_library_path/ \
    -D UMFPACK_INCLUDE_DIRS:PATH=/umfpack_include_path/ \
    -D TPL_ENABLE_Pthread:BOOL=ON \
    -D TPL_ENABLE_BLAS:BOOL=ON \
    -D BLAS_LIBRARY_DIRS:PATH=/blas_library_path/ \
    -D BLAS_LIBRARY_NAMES:STRING="blas_library_name" \
    -D TPL_ENABLE_LAPACK:BOOL=ON \
    -D LAPACK_LIBRARY_DIRS:PATH=/lapack_library_path/ \
    -D LAPACK_LIBRARY_NAMES:STRING="lapack_library_name" \
    -D TPL_ENABLE_MPI:BOOL=ON \
    -D TPL_ENABLE_ParMETIS:BOOL=ON \
    -D ParMETIS_LIBRARY_DIRS:PATH=/parmetis_library_path/ \
    -D METIS_LIBRARY_DIRS:PATH=/metis_library_path/ \
    -D TPL_ENABLE_HDF5:BOOL=ON \
    -D MPI_BASE_DIR:PATH=/mpi_library_path/ \
    -D MPI_BIN_DIR:PATH=/mpi_binary_path/ \
    -D CMAKE_INSTALL_PREFIX:PATH=./ \
    $EXTRA_ARGS \
    /trilinos_source_path/
```

Simply modify the paths and names of libraries according to your particular configuration and run the shell script. 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. 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 configure, log into your cmcsforge account and go to the `Account Maintenance` tab. At the bottom of the page, in the `Shell Account Information`, you will see `Edit Keys`. click on that link, input your public key, then click on `update`. The server will take up to one hour to update your ssh account information, 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 developername@cmcsforge.epfl.ch:/gitroot/lifev/lifev
```

where developername is your account name on cmcsforge. Place yourself in the new directory

```
cd lifev
```

and type

```
git clone  developername@cmcsforge.epfl.ch:/gitroot/lifev/admin
```

Second, you have to generate the compilation environment by typing

```
make -f Makefile.git
```

Third, you have to execute the script `configure` it will automatically check the availability of the needed components for *LifeV* compilation. Type

```
./configure \
    --with-trilinos-lib=/your_trilinos_library_path/ \
    --with-trilinos-include=/your_trilinos_include_path/ \
    --enable-opt \
    --with-mpi=/usr/lib/openmpi \
    --with-mpi-lib=/usr/lib/openmpi/lib \
    --with-mpi-include=/usr/include/openmpi \
    --with-parmetis-include=/your_parmetis_include_path/ \
    --with-parmetis-lib=/your_parmetis_library_path/ \
    --with-umfpack-include=/your_umfpack_include_path/ \
    --with-umfpack-lib=/your_umfpack_library_path/ \
    --with-amd-include=/your_amd_include_path/ \
    --with-amd-lib=/your_amd_library_path/ \
    --with-hdf5=/usr \
    --prefix=/your_lifeV_library_install_path/
```

Change the path and name of mpi, mpi library and include if yours is different. You can check for additional flags for mpi typing `mpicc -show`.
Finally, you just have to use `make` to compile *LifeV* libraries and documentation. Enter

```
make
make install
```

Using `configure` allows you to maintain several concurrent build directories. Typically during development and testing, you need to have *LifeV* compiled with debugging flags, optimization flags or profiling flags.

With configure it is possible to compile a code in a directory which is different from the source directory. This is extremely useful to tacke our problem. Let's consider given the source directory to be store in the environment variable `$LIFEV_HOME` which could be different.

Here is what you can do to compile with debugging flags

```
cd $LIFEV_HOME
mkdir debug
cd debug
../configure CXXFLAGS="-g2 -O0"
make
```

Here we have `$(top_builddir) == $LIFEV_HOME/debug`.

Here is what you can do to compile with optimization flags

```
cd $LIFEV_HOME
mkdir optimized
cd optimized
../configure CXXFLAGS="-O2"
make
```

Here we have `$(top_builddir) == $LIFEV_HOME/optimized`.
Note that you may have several concurrent `$(top_builddir)`
Be careful because `configure` will fail if you have already compiled *LifeV* in the source directory. Therefore is not a good idea to build in 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

   ```
   cd lifev-x.y.z
   configure
   ```

4. compile it

   ```
   make
   ```

Some comments in section1.4.2 apply also here.

1.4.4 Compiling Testsuite

LifeV comes with a testsuite covering a lot of features. It is located in the directory `testsuite`

```
|-- data
|-- test_bdf
|-- test_darcy
|-- test_essentialbc
|-- test_fe
|-- test_fsi_newton
|-- test_fsi_picard
|-- test_linearelasticity
```

```
|-- test_matrix
|-- test_mesh
|-- test_mixte
|-- test_naturalbc
|-- test_ns_bdf
|-- test_ns_cyl
|-- test_ns_sstress
|-- test_p2
|-- test_postproc
`-- test_q1
```

In order to compile the testsuite, you need the following steps

```
cd $(top_builddir)/testsuite
make check
```

If you just want to compile a specific test, say `test_darcy`

```
cd $(top_builddir)/testsuite/data
make check
cd $(top_builddir)/testsuite/test_darcy
make test_darcy _OR_ make check
```

Since most tests are using meshes that are located in `$(top_srcdir)/testsuite/data`, it is mandatory to execute `make check` in `$(builddir)/testsuite/data` in order to create the proper symlinks to the meshes when `$(top_builddir) != $(top_srcdir)`. It is also possible to use the command `make link` inside the repertory of a particular test in order to set the meshes available.

Chapter 2

Learning by examples

2.1 The Stokes Problem

Let us consider flow of a viscous and incompressible fluid described by its velocity u and pressure p. Its flow can be described, at low Reynolds number, by the Oseen Problem

$$\begin{cases} \alpha u + \beta \cdot \nabla u - \nu \Delta u + \nabla p & = f \\ \nabla \cdot u & = 0 \end{cases} \tag{2.1}$$

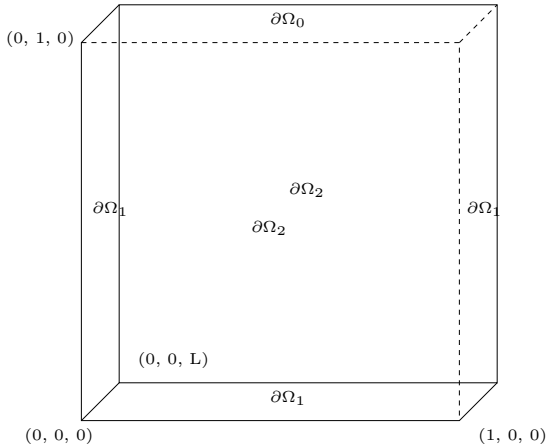
were ν is the kinematic viscosity of the fluid. If we set the convective acceleration β and α to zero, we get the Stokes equations

$$\begin{cases} -\nu \Delta u + \nabla p & = f \\ \nabla \cdot u & = 0 \end{cases} \tag{2.2}$$

We want to solve the following Stokes problem

$$\begin{cases} -\nu \Delta u + \nabla p & = f \\ \nabla \cdot u & = 0 \\ u = (1,0,0) & \text{on } \partial\Omega_0 \\ u = (0,0,0) & \text{on } \partial\Omega_1 \\ u \cdot n = 0 & \text{on } \partial\Omega_2 \end{cases} \tag{2.3}$$

on the 3D domain represented by



These equations can be written using the bilinear forms

$$\forall u, v \in H^1(\Omega) \quad : \quad a(u,v) = \nu \int_\Omega \nabla u \cdot \nabla v dx$$
$$\forall v \in H^1(\Omega),\ q \in L^2(\Omega) \quad : \quad b(v,q) = \int_\Omega q \nabla \cdot v dx$$

into the variational form: let $f \in L_0(\Omega)$, find $u \in H_0^1(\Omega)$ et $p \in L_0^2(\Omega)$ such that

$$\begin{cases} a(u,v) + b(v,p) &= (g,v) & \forall t \in (0,T), \forall v \in H_0^1(\Omega) \\ b(u,q) &= 0 & \forall t \in (0,T), \forall q \in L_0^2(\Omega) \end{cases} \tag{2.4}$$

In order to solve (2.4) using *LifeV*, let us create a working directory and get the following files:

- Makefile-cavity
- cavity.cpp
- data-cavity

from the `<lifev directory>/doc/manual/` directory. The library has evolved much during the last years and you will find a few differences between the instructions explained here and the `cavity.cpp` code updated by Gwenol Grandperrin in October of 2010.

Let's have a look a the makefile .

```
# path to the compiler
CC                = /usr/bin/g++

SOURCES           = cavity.cpp
OBJECTS           = $(SOURCES:.cpp=.o)
EXECUTABLE        = cavity

LIFELIBPATH       = -L<lifev lib directory path>
LIFELIBS          = -llifefilters -llifesolver -llifefem \
                    -llifealg -llifearray -llifecore -llifemesh
LIFEINCLUDEPATH   = -I<lifev include directory path>

TRILLIBPATH       = -L<trilinos lib directory path>
TRILLIBS          = -laztecoo -laztecoo -ltriutils -lml \
                    -lifpack -lamesos -lepetraext -lepetra \
                    -lteuchos  -llapack -lblas
TRILINCLUDEPATH   = -I<trilinos include directory path>

# type "mpicxx -show" to get an hint of the contents of
# the following variables
MPILIBPATH        = -L<mpi lib directory path>
MPILIBS           = -lmpi_cxx -lmpi -lopen-rte
MPIINCLUDEPATH    = -I<mpi include directory path>

METISLIBPATH  = -L<parmetis lib directory path>
METISLIBS         = -lparmetis -lmetis
METISINCLUDEPATH = -I<parmetis include directory path>

# uncomment this part for optimized compilation
```

```
LDFLAGS         = -g0 -O2 -DTHREEDIM -lm
# uncomment this part for debugging
#LDFLAGS         = -g2 -O0 -DTHREEDIM -lm


all: $(OBJECTS) $(EXECUTABLE)

$(OBJECTS): $(SOURCES)
$(CC) $(LDFLAGS) \
$(MPIINCLUDEPATH) $(TRILINCLUDEPATH) $(LIFEINCLUDEPATH) \
$(SOURCES) -o $@

$(EXECUTABLE): $(OBJECT)
echo "compiling the executable ... "
$(CC) $(CFLAGS) \
$(OBJECTS) $< -o $@ \
$(LIFELIBPATH) $(LIFELIBS) $(LIFEINCLUDEPATH) \
$(TRILLIBPATH) $(TRILLIBS) $(TRILINCLUDEPATH) \
$(METISPATH) $(METISLIBS) $(METISINCLUDE) \
$(MPILIBPATH) $(MPILIBS) $(MPIINCLUDEPATH) \


clean:
rm -rf *o cavity
```

You will need to fill the `<...>` with your local configuration paths. You can also change the `LDFLAGS` options in order to compile using the debug or the optimized mode in the g++ compiler. More information about using makefiles is available at http://www.gnu.org/software/make/manual/make.html.

Now that we have the makefile, we can look at the sources, contained in the file cavity.cpp

```
#include "Epetra_config.h"
#include "Epetra_MpiComm.h"
```

This part is mandatory in order to define the Epetra Communicators (that contain the MPI calls) and should be at the begining of each program.

```
#include <boost/program_options.hpp>
#include <life/lifecore/life.hpp>
#include <life/lifecore/application.hpp>
#include <life/lifearray/EpetraMatrix.hpp>
#include <life/lifealg/EpetraMap.hpp>
#include <life/lifemesh/partitionMesh.hpp>
#include <life/lifesolver/dataNavierStokes.hpp>
#include <life/lifefem/FESpace.hpp>
#include <life/lifefem/bdfNS_template.hpp>
#include <life/lifefilters/ensight.hpp>
#include <life/lifesolver/Oseen.hpp>
#include <iostream>

using namespace LifeV;
```

Use this to use LifeV objects without refering to LifeV:: everytime. Without it, we have to use LifeV::RefFE instead of just RefFE for instance.

```
typedef boost::function<Real ( Real const&,
                               Real const&,
                               Real const&,
                               Real const&,
                               ID const& )> fct_type;

typedef Oseen< RegionMesh3D<LinearTetra> >::vector_type  vector_type;
typedef boost::shared_ptr<vector_type>                   vector_ptrtype;

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

Real uLid(const Real& t, const Real& /*x*/, const Real& /*y*/, const Real& /*z*/, const ID& i)
{
  switch(i) {
  case 1:
    return 1.0;
    break;
  case 3:
      return 0.0;
      break;
  case 2:
      return 0.0;
    break;
  }
  return 0;
}
```

In this section, we have defined real functions that will be used in the boundary condition object. Boundary conditions functions must be defined using the following scheme

```
Real function_name ( const Real& time,
                     const Real& x, const Real& y, const Real& z,
                     const ID&   id )
```

where `time` is the simulation time, `x, y, z` are the space coodinates, and `ID` is the component of the variable we want to set. In our example, we want to set $(u_x, u_y, u_z) = (1, 0, 0)$ when we are in $\partial\Omega_1$ Therefore, when the ID is 1, i.e x, we return 1. For every other cases, i.e y and z, we return 0.

We could have used another boundary condition, for instance

```
Real uLid(const Real& t, const Real& /*x*/, const Real& /*y*/, const Real& /*z*/, const ID& i)
{
  switch(i) {
  case 1:
      return x*(1 - x);
      break;
  case 3:
      return 0.0;
      break;
  case 2:
      return 0.0;
      break;
  }
  return 0;
}
```

The main difference is that, using this functions, the boundary condition on $\partial\Omega_1$ now becomes

$$u = (x(1-x), 0, 0) \text{ on } \partial\Omega_0$$

We can now proceed to the main block of the code.

```
int main( int argc, char** argv )
{
    MPI_Init(&argc, &argv);
    Epetra_MpiComm comm(MPI_COMM_WORLD);
```

These two lines will initialize the MPI process and create an Epetra communicator that will be used throughout the code for message passing. See

- http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Init.html
- http://trilinos.sandia.gov/packages/docs/r6.0/packages/epetra/doc/html/classEpetra_MpiComm.html

for more explanations.

```
    // a flag to see who's the master for output purposes
    bool verbose = comm.MyPID() == 0;

    if ( comm.MyPID() == 0 )
        {
            cout << "% using MPI" << endl;
            int ntasks;
            int err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
            std::cout << "My PID = " << comm.MyPID() << " out of "
                      << ntasks << " running." << std::endl;
        }
```

This block, although not necessarily in the comprehension of the FE resolution code, explains how to manage output from a parallel code. As we do not want every processor to output every piece of information, we set a master processor that will display relevant pieces of information on the console (0 in our case).

```
// Read from the data file. Its name can be given using the
// -f or --file argument after the name of launch program.
// By default, it's data.

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

In this part, a GetPot object (http://getpot.sourceforge.net/) is created and is linked to a data description file using the "-f" or "–file" parameters after the main program name. This GetPot object is used to store values like:

- the mesh name,
- the time step,
- the discretization order,
- the physics of the model,
- the solver information,
- ...

You can browse the default data file in every testsuite directory to see examples. In general the entries are filled with a default value if not specified, but some entries are obligatory, like the mesh name for instance.

A data object will be used to store this information. In our case, since we want to solve a Navier-Stokes problem, we need to use the DataNavierStokes object. Given the GetPot object we have just defined, it will parse the specified data file to retreive all the information necessary to run the simulation.

```
// everything ( mesh included ) will be stored in a class
boost::shared_ptr< DataTime > dataTime( new DataTime( dataFile ) );
boost::shared_ptr< DataMesh< RegionMesh3D<LinearTetra> > > dataMesh( new DataMesh< RegionMesh3D<

DataNavierStokes<RegionMesh3D<LinearTetra> > dataNavierStokes( dataFile, dataTime, dataMesh );
```

After this line, everything we need to know about our problem is stored in dataNavierStokes.

To build a FE solver for our cavity problem we need:

- a finite element space,
- the boundary conditions,
- a solver that will build and solve the linear system derived from our weak formulation.

```
// BCHandler is the class that stores the boundary conditions. Here we will
// set 3 boundary conditions:
// top               : (ux, uy, uz) = (1., 0., 0.) essential BC
// left, right, down : (ux, uy, uz) = (0., 0., 0.) essential BC
// front and rear    : uz = 0 essential BC

BCHandler bcH(3);
```

Name	Options	Description
mesh_dir		mesh directory path
mesh_name		mesh file name
timestep		problem time step
vel_order	P1 P1Bubble P2	velocity discretization order
press_order	P1 P2	pressure discretization order
order_bdf	1 2	time discretization order

Table 2.1: Description of the discretization parameters.

```
std::vector<ID> zComp(1);
zComp[0] = 3;

BCFunctionBase uIn  ( boost::bind(&uLid, _1, _2, _3, _4, _5) );
BCFunctionBase uZero( zero_scalar );

// boundary conditions definition.
// the first two are classical essential or dirichlet conditions
bcH.addBC( "Upwall",   UPWALL,   Essential, Full,      uIn,   3 );
bcH.addBC( "Wall",     WALL,     Essential, Full,      uZero, 3 );
```

Bondary conditions `Boundary Conditions` part. Here is the prototype of the `addBC` function

```
//! add new BC to the list (user defined function)
/*!
  \param name the name of the boundary condition
  \param flag the mesh flag identifying the part of the mesh where the boundary condtion applies
  \param type the boundary condition type: Natural, Essential, Mixte
  \param mode the boundary condition mode: Scalar, Full, Component, Normal, Tangential
  \param bcf the function holding the user defined function involved in this boundary condition
  \param std::vector<ID> storing the list of components involved in this boundary condition
*/
void addBC( const std::string&     name,
            const EntityFlag&      flag,
            const BCType&          type,
            const BCMode&          mode,
            BCFunctionBase&        bcf,
            const std::vector<ID>& comp );
```

`name` is a boundary condition description string, `flag` is the boundary condition number as defined in the mesh (in this example, the variables `UPWALL` and `WALL` are defined at the beginning of the file), `mode` is the mode, `bcf` is the function holding the user-defined function involved in the boundary condition. `type` and `mode` are respectively the boundary condition type and mode. Please refer to the table (**??**) for a description of their values.

`comp` is a vector storing the components in the boundary condition. The last boundary condition we want to impose is `SLIPWALL` (i.e 20 in the mesh file, or $\partial\Omega_2$ in (2.3)), will get a slipwall boundary condition, i.e

$$u \cdot n = 0 \tag{2.5}$$

Name	Options	Description
type	Natural	Neumann
	Essential	dirichlet
	Mixte	Robin
mode	Scalar	1 dimension BC
	Full	3 component BC
	Component	Sepate compenent BC
	Normal	Normal BC
	Tangential	Tangential BC

Table 2.2: Boundary Condition parameters description

This means, since the two concerned planes are defined by $z = 0$ and $z = L$, that

$$u_z = 0 \text{ for } z = 1, L$$

In order to set our third component (z), we define the vector

```
std::vector<ID> zComp(1);
zComp[0] = 3;
```

Then, we add an essential (Dirichlet) boundary condition on the z component by calling

```
bcH.addBC( "Slipwall", SLIPWALL, Essential, Component, uZero, zComp );
```

This will set a null function to the third component on the slipwall. We could have just easily defined another function for the others components using the same procedure.

Now we get the mesh `Mesh`. LifeV partitions meshes on the fly using the parMetis library, that means that you do not have to provide the partitioned mesh in order to have the simulation running.

```
// partitioning the mesh
partitionMesh< RegionMesh3D<LinearTetra> >
         meshPart(*dataNavierStokes.dataMesh()->mesh(), comm);
```

In our case, after the call to the `partitionMesh` constructor, `meshPart` will store the local part of the mesh. Using this local mesh, we can create our Finite Element spaces. In *LifeV*, a Finite Element Space is a class storing:

a mesh,

a reference Finite Element,

quadrature rules to integrate reference functions on the mesh elements or the boundaries.

A reference Finite Element in *LifeV* is a class containing the geometrical definition of the mesh elements and the polynomial approximation order we want to use.

```
// Now we proceed with the FESpace definition
// here we decided to use P2/P1 elements

const RefFE*    refFE_vel;
const QuadRule* qR_vel;
const QuadRule* bdQr_vel;

refFE_vel = &feTetraP2;
qR_vel    = &quadRuleTetra15pt; // DoE 5
bdQr_vel  = &quadRuleTria3pt;   // DoE 2
```

Name	Description
feTetraP1	P1 finite element on Tetrahedron
feTetraP1Bubble	P1-Bubble finite element on Tetrahedron
feTetraP2	P2 finite element on Tetrahedron

Table 2.3: Reference Finite Element parameters

Name	Description	exact p. order
quadRuleTetra1pt	1 point	1
quadRuleTetra3pt	3 point	2
quadRuleTetra5pt	5 point	
quadRuleTetra15pt	15 point	
quadRuleTetra64pt	64 point	

Table 2.4: Quadrature Rule description

After these lines, `refFE_vel` contains the desired reference finite element . See table 2.4 for the descrition of available parameters.

Quadrature rules `Quatrature Rules` are defined according the polynomial order we have defined.

Everything is ready to create our Finite Element space

```
// Everything is ready to build the FE space
// first the velocity FE space

if (verbose)
    std::cout << "Building the velocity FE space ... " << std::flush;

FESpace< RegionMesh3D<LinearTetra>, EpetraMap > uFESpace(meshPart,
                                                         *refFE_vel,
                                                         *qR_vel,
                                                         *bdQr_vel,
                                                         3,
                                                         comm);
```

We define the reference finite element as P2 and two quadrature rules: one general and one for the boundary integration. Once these classes are defined, we call the FE space object constructor with the following input parameters:

- meshPart is the local partioned mesh,
- *refFE_vel is the reference finite element,
- *qR_vel and *bdQR_vel are the quadrature rules,
- 3 is the field dimension,
- comm is the MPI communicator.

Of course, we do the same with the pressure. This time, we use a P1 discretization

```
const RefFE*    refFE_press;
```

```
const QuadRule* qR_press;
const QuadRule* bdQr_press;

refFE_press = &feTetraP1;
qR_press    = &quadRuleTetra4pt;  // DoE 2
bdQr_press  = &quadRuleTria3pt;   // DoE 2

if (verbose)
    std::cout << "Building the pressure FE space ... " << std::flush;

FESpace< RegionMesh3D<LinearTetra>, EpetraMap > pFESpace(meshPart,
                                                         *refFE_press,
                                                         *qR_press,
                                                         *bdQr_press,
                                                         1,
                                                         comm);
```

Now we build the solver. In *LifeV*, a solver has the following properties:

- it builds and stores the linear FE matrices,
- it builds the preconditioners,
- it builds the linear solvers.

Calling the constructor will initialize the matrices, preconditioner and the linear solver but neither will be fully constructed. Instead, the matrices will be initialized using the velocity and pressure FE spaces

```
// now that the FE spaces are built, we proceed to the NS solver constrution
// we use the oseen class

if (verbose) std::cout << "Calling the fluid constructor ... ";

Oseen< RegionMesh3D<LinearTetra> > fluid (dataNavierStokes,
                                          uFESpace,
                                          pFESpace,
                                          comm);
```

Now that the class has been instantiated, we need to set it up with the data file parameters

```
// Now, the fluid solver is set up using the data file
fluid.setUp(dataFile);
```

Calling `setUp` will basically build the preconditioner and the linear solver using the AztecOO options [1] contained in the data file.

Then we can build the linear system

```
// then we build the constant matrices
fluid.buildSystem();
```

[1] see http://trilinos.sandia.gov/packages/docs/r9.0/packages/aztecoo/doc/html/classAztecOO.html for more information

name	options
solver	cg cg_condnum gmres (default) gmres_condnum cgs tfqmr bicgstab
conv	r0 rhs (default) Anorm noscaled sol
precond	none (default) none Jacobi Neumann ls sym_GS dom_decomp
scaling	none (default) Jacobi BJacobi row_sum sym_diag sym_row_sum equil sym_BJacobi
tol	default : 1e-6
kspace	default : 30
max_iter	default : 500
drop_tol	default : 0.

Table 2.5: Main parameters for the Trilinos solver

This will create the full finite element linear matrix. Note that, despite the fact that we passed both the velocity and the pressure FE spaces, the solver will consider only one finite element constructed by performing a direct sum of the two FE spaces. The associated "full" map can be retrieved using the `getMap` method

```
// this is the total map ( velocity + pressure ). it will be used to create
// vectors to store the solutions

EpetraMap fullMap(fluid.getMap());

if (verbose) std::cout << "ok." << std::endl;
```

Using this map is obligatory when we access the solution vector after the linear system is solved.

In *LifeV*, we mainly use paraview in order to postprocess our problem solutions. Writing a paraview solution is quite straightforward using the Ensight class. We call the Ensight constructor where we give the data file, the mesh and the filename of the solution file with references to the solution vector.

```
// finally, let's create an exporter in order to view the results
// here, we use the ensight exporter

Ensight<RegionMesh3D<LinearTetra> >
       ensight( dataFile, meshPart.mesh(), "cavity", comm.MyPID());

// we have to define a variable that will store the solution
vector_ptrtype velAndPressure ( new vector_type(fluid.solution(),
                                                Repeated ) );

// and we add the variables to be saved
// the velocity
ensight.addVariable( ExporterData::Vector, "velocity", velAndPressure,
                     UInt(0), uFESpace.dof().numTotalDof() );

// and the pressure
ensight.addVariable( ExporterData::Scalar, "pressure", velAndPressure,
                     UInt(3*uFESpace.dof().numTotalDof()),
                     UInt(3*uFESpace.dof().numTotalDof() +
                            pFESpace.dof().numTotalDof()) );

// a little barrier to synchronize the processes
MPI_Barrier(MPI_COMM_WORLD);
```

We are now set for the solution of the linear system.

```
vector_type beta( fullMap );
vector_type rhs ( fullMap );

beta        *= 0.;
rhs         *= 0.;

double alpha = 0.;
```

Using the full map defined above, we set the advection term and right handside to zero.

```
// updating the system with no mass matrix, advection and rhs set to zero,
// that is the stokes problem
fluid.updateSystem(alpha, beta, rhs );
```

In the Oseen class, `updateSystem` has 3 arguments:

- `alpha` is the coefficient in front of the mass term,
- `beta` is the advection term,
- `rhs` is the righthand side.

Setting these 3 terms to zero will result in solving the system (2.3). The linear system is solved by calling `iterate`, which requires the boundary conditions as parameters. The member `iterate` will:

- build the full matrix,
- apply the boundary conditions,
- solve the system.

```
// iterating the solver in order to produce the solution
fluid.iterate( bcH );

// a little postprocessing to see if everything goes according to plan
*velAndPressure = fluid.solution();
ensight.postProcess( 0 );
```

You must run the cavity executable named `cavity_example` using `mpirun -np procs cavity_example` where `procs` is the number of processors you want to use for your computation.

You may now visualize the result using Paraview.

2.2 The Navier-Stokes Problem

Now that we have decribed a simple stationary problem, let's have a look at the evolutionary case. In this example, we will consider the same domain, but this time we will solve the Navier-Stokes problem . Starting from the Oseen problem (2.1)

$$\begin{cases} \alpha u + \beta \cdot \nabla u - \nu \Delta u + \nabla p &= f \\ \nabla \cdot u &= 0 \end{cases}$$

We want to solve the non-stationary Navier-Stokes problem

$$\begin{cases} \dfrac{\partial u}{\partial t} + u \cdot \nabla u - \nu \Delta u + \nabla p &= f \\ \nabla \cdot u &= 0 \end{cases}$$

which can be written, using a semi-discretization of the time partial derivative as

$$\begin{cases} \alpha u^{n+1} + u^{n+1} \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} &= f^n \\ \nabla \cdot u &= 0 \end{cases}$$

Where α is a constant which depends on the time discretization order, Δt is the time step, u^{n+1} and p^{n+1} are the velocity and the pressure at the time $t^n = n\Delta t$. Note that f^n now contains terms resulting from the time discretization. Using a linearization β^n of (2.2) around u^{n+1}, we get the full semi-discrete linearized Navier-Stokes equations

$$\left\{ \begin{array}{rl} \alpha u^{n+1} + \beta^n \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} & = f^n \\ \nabla \cdot u & = 0 \end{array} \right.$$

Solving (2.2) using the framework we used for the stationary driven cavity is not difficult. Instead of setting α, β and f^n to zero, we give them their proper values. For instance, consider the first order discretization in time

$$\left\{ \begin{array}{rl} \dfrac{u^{n+1}}{\Delta t} + u^n \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} & = \dfrac{u^n}{\Delta t} \\ \nabla \cdot u & = 0. \end{array} \right.$$

Or the second order discretization

$$\left\{ \begin{array}{rl} \dfrac{3u^{n+1}}{2\Delta t} + u^n \cdot \nabla u^{n+1} - \nu \Delta u^{n+1} + \nabla p^{n+1} & = \dfrac{2u^n}{\Delta t} - \dfrac{u^{n-1}}{2\Delta t} \\ \nabla \cdot u & = 0. \end{array} \right.$$

Let's have a look at the code in `testsuite/test_cavity/main.cpp`. Until the first `fluid.iterate()`, the code is the same as the one used to compute the Stokes problem. Now, we want to use this Stokes solution to initialize our non-stationary Navier-Stokes problem, and be able to store a history of previous solutions in order to compute the time derivative. This can be done by using the following object

```
// bdf initialization with the stokes problem solution
BdfTNS<vector_type> bdf(dataNavierStokes.getBDF_order());
```

The backward differentiation formula, class `BdfTNS` , stores the previous solution u^n, u^{n-1} ... All we need to do is to construct this class with the correct time discretization order given in the data file (variable `fluid/discretization/bdf_order`) so that the stored vector will be resized correctly, and intialize it with our previously computed Stokes problem solution

```
bdf.bdf_u().initialize_unk( fluid.solution() );
```

We are now ready to enter the time loop

```
Real dt     = dataNavierStokes.getTimeStep();
Real t0     = dataNavierStokes.getInitialTime();
Real tFinal = dataNavierStokes.getEndTime ();

int iter = 1;

for ( Real time = t0 + dt ; time <= tFinal + dt/2.; time += dt, iter++)
{
    // inside the time loop, it's really like the initialization procedure,
    // exept that we now have an advection velocity, rhs and the mass matrix
    dataNavierStokes.setTime(time);

    if (verbose)
    {
        std::cout << std::endl;
        std::cout << "We are now at time " << dataNavierStokes.time()
                  << " s. " << std::endl;
        std::cout << std::endl;
    }

    chrono.start();
```

```
        // alpha coefficient for the mass matrix
        double alpha = bdf.bdf_u().coeff_der( 0 ) / dataNavierStokes.timestep();

        // extrapolation of the advection term
        beta = bdf.bdf_u().extrap();

        // rhs  part of the time-derivative
        rhs  = fluid.matrMass()*bdf.bdf_u().time_der( dataNavierStokes.timestep() );

        // update the Oseen system
        fluid.updateSystem( alpha, beta, rhs );

        // and we solve it
        fluid.iterate( bcH );

        // shifting the previous solutions
        bdf.bdf_u().shift_right( fluid.solution() );

        // postprocess
        *velAndPressure = fluid.solution();
        ensight.postProcess( time );

        // a barrier for sinchronization
        MPI_Barrier(MPI_COMM_WORLD);

        chrono.stop();
        if (verbose) std::cout << "Total iteration time "
                               << chrono.diff()
                               << " s." << std::endl;
    }
```

The time step `dt`, the initial time `t0` and the final simulation time `tFinal` are found in the data file, their names are respectively `fluid/discretization/timestep`, `problem/Tstart` and `fluid/physics/endtime`. As we can see, a time step can be described as a follow up of several intuitive calls:

- computation of α (which should is constant in most cases),
- computation of β using the `Bdf` class,
- computation of the right hand side rhs,
- update of the system using these three variables,
- solution of the linear system.

After the system is solved, we simply update all time-dependent variables such as the storage of the previous solutions and we loop until all time steps are computed.

2.3 Fluid/Structure interactions

We now want to address the numerical solution of fluid-structure interaction problems. In order to address each problem in its natural setting, we choose to consider the fluid in an ALE (Arbitrary Lagrangian Eulerian) formulation and the structure in a pure Lagrangian framework.

The system under investigation occupies a moving domain $\Omega(t)$ in its actual configuration. It is made of a deformable structure $\Omega^{\rm s}(t)$ (such as an arterial wall, a pipe-line, ...) surrounding a fluid under motion (blood, oil, ...) in the complement $\Omega^{\rm f}(t)$ of $\Omega^{\rm s}(t)$ in $\Omega(t)$ (see Fig. 2.1).

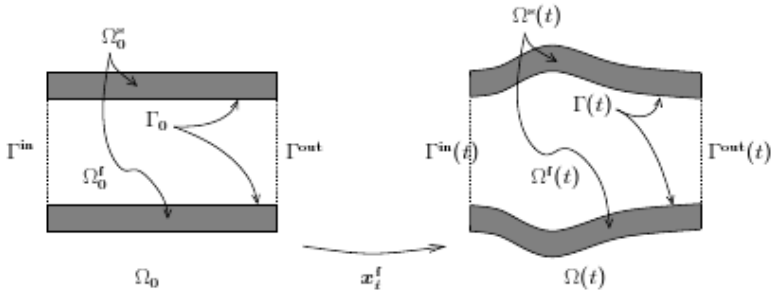


Figure 2.1: ALE mapping between the initial configuration and the configuration at time t.

We assume the fluid to be Newtonian, viscous, homogeneous and incompressible. Its behavior is described by its velocity and pressure. The elastic solid under large displacements is described by its velocity and its stress tensor. The classical conservation laws of the continuum mechanics govern the evolution of these unknowns.

We denote by $\Gamma^{\rm in}(t)$ and $\Gamma^{\rm out}(t)$ the inflow and outflow sections of the fluid domain, by $\boldsymbol{n}_{\rm f}$ the fluid domain's outward normal on $\partial\Omega^{\rm f}(t)$ and by $\boldsymbol{n}_{\rm s}$ the one of the structure on the reference boundary $\partial\hat{\Omega}^{\rm s}$. The boundary conditions on the fluid inlet and outlet can be either natural or essential (i.e., of Neumann or Dirichlet type, respectively), while on the interface we impose that the fluid and structure velocities match and so do the normal stresses. For simplicity, we assume zero body forces on both the structure and the fluid and that the boundary conditions on the remaining part of the structure boundary are of Dirichlet or of Neumann type.

The problem consists in finding the time evolution of the configuration $\Omega^{\rm f}(t)$, as well as the velocity $\boldsymbol{u}$ and pressure p for the fluid and the displacement $\boldsymbol{d}$ of the structure. We define the ALE mapping

$$\forall t\,,\ \boldsymbol{x}_t^{\rm f} : \hat{\Omega}^{\rm f} \to \Omega^{\rm f}(t),$$

i.e. a map that retrieves at each time the current configuration of the computational domain $\Omega^{\rm f}(t)$. Note in particular that on the reference interface Σ^s, $\boldsymbol{n}_{\rm f} \circ \boldsymbol{x}_t^{\rm f} = -\boldsymbol{n}_{\rm s}$. We denote by $\hat{\boldsymbol{x}}$ the coordinates on the reference configuration $\hat{\Omega}^{\rm f}$ and by $\boldsymbol{w} = \frac{d\boldsymbol{x}_t^{\rm f}}{dt}$ the domain velocity.

For simplicity, we denote in short by Fluid(...) and Str(...) the fluid and structure problems, respectively. Precisely, for given vector functions $\boldsymbol{u}_{\rm in}$, $\boldsymbol{g}_{\rm f}$ and $\boldsymbol{f}_{\rm f}$, Fluid$(\boldsymbol{u}, p, \boldsymbol{x}_t^{\rm f}; \boldsymbol{u}_{\rm in}, \boldsymbol{g}_{\rm f}, \boldsymbol{f}_{\rm f})$ means that we consider the following problem whose solution is $\boldsymbol{u}$, p and $\boldsymbol{x}_t^{\rm f}$:

$$\text{Fluid}(\boldsymbol{u}, p, \boldsymbol{x}_t^{\rm f}; \boldsymbol{u}_{\rm in}, \boldsymbol{g}_{\rm f}, \boldsymbol{f}_{\rm f}) : \begin{cases} \boldsymbol{\Delta}\boldsymbol{x}_t^{\rm f} = 0 \text{ in } \hat{\Omega}^{\rm f}, \\ \boldsymbol{x}_t^{\rm f} = 0 \text{ on } \partial\hat{\Omega}^{\rm f} \setminus \hat{\Sigma}, \\ \Omega_t^{\rm f} = \boldsymbol{x}_t^{\rm f}(\hat{\Omega}^{\rm f}), \\ \rho_{\rm f}\left(\left.\dfrac{\partial \boldsymbol{u}}{\partial t}\right|_{\boldsymbol{x}_0} + (\boldsymbol{u} - \boldsymbol{w}) \cdot \boldsymbol{\nabla}\boldsymbol{u}\right) \\ \qquad = \operatorname{div}(2\mu\boldsymbol{\epsilon}(\boldsymbol{u})) - \boldsymbol{\nabla} p + \boldsymbol{f}_{\rm f} \text{ in } \Omega_t^{\rm f}, \\ \operatorname{div}\boldsymbol{u} = 0 \text{ in } \Omega_t^{\rm f}, \\ \boldsymbol{u} = \boldsymbol{u}_{\rm in} \text{ on } \Gamma_t^{\rm in}, \\ \boldsymbol{\sigma}_{\rm f}(\boldsymbol{u}, p) \cdot \boldsymbol{n}_{\rm f} = \boldsymbol{g}_{\rm f} \text{ on } \Gamma_t^{\rm out}, \end{cases} \tag{2.6}$$

where $\rho_{\rm f}$ is the fluid density, μ its viscosity, $\boldsymbol{\epsilon}(\boldsymbol{u}) = (\boldsymbol{\nabla}\boldsymbol{u} + (\boldsymbol{\nabla}\boldsymbol{u})^T)/2$ is the strain rate tensor and $\boldsymbol{\sigma}_{\rm f}(\boldsymbol{u}, p) = -pId + 2\mu\boldsymbol{\epsilon}(\boldsymbol{u})$ the Cauchy stress tensor (Id is the identity matrix). Note that (2.6)

does not univocally define a solution $(\boldsymbol{u}, p, \boldsymbol{x}_t^{\mathrm{f}})$ as no boundary data are prescribed on the interface Σ_t.

Similarly, for given vector functions $\boldsymbol{g}_{\mathrm{s}}$, $\boldsymbol{f}_{\mathrm{s}}$, $\mathrm{Str}(\boldsymbol{d}; \boldsymbol{g}_{\mathrm{s}}, \boldsymbol{f}_{\mathrm{s}})$ means that we consider the following problem whose solution is $\boldsymbol{d}$:

$$\mathrm{Str}(\boldsymbol{d}; \boldsymbol{g}_{\mathrm{s}}, \boldsymbol{f}_{\mathrm{s}}) : \begin{cases} \rho_{\mathrm{s}} \dfrac{\partial^2 \boldsymbol{d}}{\partial t^2} = \mathrm{div}(\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d})) - \gamma \boldsymbol{d} + \boldsymbol{f}_{\mathrm{s}} \text{ in } \hat{\Omega}^{\mathrm{s}}, \\ \boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}) \cdot \boldsymbol{n}_{\mathrm{s}} = \boldsymbol{g}_{\mathrm{s}} \text{ on } \partial\hat{\Omega}^{\mathrm{s}} \setminus \Sigma_0, \end{cases} \tag{2.7}$$

where $\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d})$ is the first Piola–Kirchoff stress tensor, γ is a coefficient accounting for possible viscoelastic effects, while $\boldsymbol{g}_{\mathrm{s}}$ represents the normal traction on external boundaries. Appropriate models have to be chosen for the structure depending on the specific problem at hand.

Similarly to what we have noticed for (2.6), problem (2.7) can not define univocally the unknown $\boldsymbol{d}$ because a boundary value on Σ_0 is missing.

When coupling the two problems together, the "missing" boundary conditions are indeed supplemented by suitable matching conditions on the reference interface Σ^s. More precisely, if we denote by λ the interface variable corresponding to the displacement $\boldsymbol{d}$ on Σ^s, at any time the coupling conditions on the reference interface Σ^s are

$$\begin{aligned} \boldsymbol{x}_t^{\mathrm{f}} &= \lambda, \\ \boldsymbol{u} \circ \boldsymbol{x}_t^{\mathrm{f}} &= \dot{\boldsymbol{d}}_{\Sigma^s}, \\ (\boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u}, p) \cdot \boldsymbol{n}_{\mathrm{f}}) \circ \boldsymbol{x}_t^{\mathrm{f}} + \boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}) \cdot \boldsymbol{n}_{\mathrm{s}} &= 0, \end{aligned} \tag{2.8}$$

where $\dot{\boldsymbol{d}}_{\Sigma^s}$ denotes the temporal derivative of $\boldsymbol{d}_{|\Sigma^s}$. The system of equations (2.6)-(2.8) identifies our coupled fluid-structure problem. We suppose the problem to be discretized in time. When the solution is available at time t^n, we look for the solution at the new time level $t^{n+1} = t^n + \delta t$. If no ambiguity occurs, all the quantities will be referred to at time $t = t^{n+1}$. Without loss of generality we consider zero body forces, i.e., $\boldsymbol{f}_{\mathrm{f}} = 0$ and $\boldsymbol{f}_{\mathrm{s}} = 0$.

If we are given a displacement of the interface $\lambda(t^{n+1})$ at the time t^{n+1}, we can find its harmonic extension on the fluid domain by solving the following variational formulation of (**??**):

find $\boldsymbol{d}^{\mathrm{f}}{}_{t^{n+1}} \in H^1(\Omega_0^{\mathrm{f}})^3$ such that

$$\begin{cases} \displaystyle\int_{\Omega_0^{\mathrm{f}}} \boldsymbol{\nabla} \boldsymbol{d}^{\mathrm{f}}{}_{t^{n+1}} \cdot \boldsymbol{\nabla} \boldsymbol{\phi} = 0 & \forall \boldsymbol{\phi} \in H_0^1(\Omega_0^{\mathrm{f}})^3 \\ \boldsymbol{d}^{\mathrm{f}}{}_{t^{n+1}} = \lambda(t^{n+1}) & \text{on } \Gamma_0, \end{cases} \tag{2.9}$$

completed with appropriate boundary conditions on $\Gamma^{\mathrm{in}} \cup \Gamma^{\mathrm{out}}$.

Then we compute the velocity of the fluid domain as $\boldsymbol{w}^{\mathrm{f},n+1}_{|\Gamma(t^{n+1})} = 1/\delta t \, (\boldsymbol{d}^{\mathrm{f}}{}_{t^{n+1}|\Gamma_0} - \boldsymbol{d}^{\mathrm{f}}{}_{t^n|\Gamma_0}) \circ (\boldsymbol{x}^{\mathrm{f}}_{t^{n+1}})^{-1}$ and the velocity and pressure of the fluid at time t^{n+1} by solving:

find $(\boldsymbol{u}^{n+1}, p^{n+1}) = (\boldsymbol{u}(t^{n+1}), p(t^{n+1})) \in V^{\mathrm{f}}(t^{n+1}) \times Q^{\mathrm{f}}(t^{n+1})$ such that $\boldsymbol{u}^{n+1}_{|\Gamma(t^{n+1})} = \boldsymbol{w}^{\mathrm{f},n+1}_{|\Gamma(t^{n+1})}$, $\boldsymbol{u}^{n+1}_{|\Gamma^{\mathrm{in}}(t^{n+1})} = \boldsymbol{u}_{\mathrm{in}}(t^{n+1})$ and

$$\begin{cases} \dfrac{1}{\delta t} \displaystyle\int_{\Omega^{\mathrm{f}}(t^{n+1})} \rho_{\mathrm{f}} \boldsymbol{u}^{n+1} \boldsymbol{v}^{\mathrm{f}} + \int_{\Omega^{\mathrm{f}}(t^{n+1})} \rho_{\mathrm{f}} [(\boldsymbol{u}^{n+1} - \boldsymbol{w}^{\mathrm{f},n+1}) \cdot \boldsymbol{\nabla} \boldsymbol{u}^{n+1}] \boldsymbol{v}^{\mathrm{f}} \\ \quad + \mu \displaystyle\int_{\Omega^{\mathrm{f}}(t^{n+1})} \boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u}^{n+1}, p^{n+1}) \cdot \boldsymbol{\nabla} \boldsymbol{v}^{\mathrm{f}} = \dfrac{1}{\delta t} \displaystyle\int_{\Omega^{\mathrm{f}}(t^{n+1})} \rho_{\mathrm{f}} \boldsymbol{u}^{n} \boldsymbol{v}^{\mathrm{f}} + \int_{\Gamma^{\mathrm{out}}(t^{n+1})} \boldsymbol{g}_{\mathrm{f}} \boldsymbol{v}^{\mathrm{f}} \\ \displaystyle\int_{\Omega^{\mathrm{f}}(t^{n+1})} q^{\mathrm{f}} \, \mathrm{div} \, \boldsymbol{u}^{n+1} = 0 \end{cases} \tag{2.10}$$

for all $(\boldsymbol{v}^{\mathrm{f}}, q^{\mathrm{f}}) \in V_0^{\mathrm{f}}(t^{n+1}) \times Q^{\mathrm{f}}(t^{n+1})$, with

$$\begin{aligned} V^{\mathrm{f}}(t) &= \left\{ \boldsymbol{v}^{\mathrm{f}} \,|\, \boldsymbol{v}^{\mathrm{f}} \circ \boldsymbol{x}_t^{\mathrm{f}} \in H^1(\Omega_0^{\mathrm{f}})^3 \right\}, \\ V_0^{\mathrm{f}}(t) &= \left\{ \boldsymbol{v}^{\mathrm{f}} \in V^{\mathrm{f}}(t) \,|\, \boldsymbol{v}^{\mathrm{f}} \circ \boldsymbol{x}_t^{\mathrm{f}} = \boldsymbol{0} \text{ on } \Gamma_0 \cup \Gamma^{\mathrm{in}} \right\}, \\ Q^{\mathrm{f}}(t) &= \left\{ q^{\mathrm{f}} \,|\, q^{\mathrm{f}} \circ \boldsymbol{x}_t^{\mathrm{f}} \in L^2(\Omega_0^{\mathrm{f}}) \right\}, \end{aligned}$$

and where the fluid domain $\Omega^{\mathrm{f}}(t^{n+1})$ is given by

$$\Omega^{\mathrm{f}}(t^{n+1}) = \boldsymbol{x}^{\mathrm{f}}_{t^{n+1}}(\Omega^{\mathrm{f}}_0).$$

We can then compute $(\boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u}^{n+1}, p^{n+1}) \cdot \boldsymbol{n}_{\mathrm{f}}) \circ \boldsymbol{x}^{\mathrm{f}}_t$ on Γ_0, which by (**??**) has to be equal to the structure normal stresses.

On the structure side, given the same displacement $\lambda(t^{n+1})$, we can use the following scheme to approximate the arterial deformation and the domain velocity (see [**?**]):

find $(\boldsymbol{d}^{\mathrm{s},n+1}, \boldsymbol{w}^{\mathrm{s},n+1}) = (\boldsymbol{d}^{\mathrm{s}}(t^{n+1}), \boldsymbol{w}^{\mathrm{s}}(t^{n+1})) \in V^{\mathrm{s}} \times V^{\mathrm{s}}$ such that

$$\begin{cases} \dfrac{2}{\delta t^2}\displaystyle\int_{\Omega^{\mathrm{s}}_0} \rho_{\mathrm{s}} \boldsymbol{d}^{\mathrm{s},n+1}\boldsymbol{v}^{\mathrm{s}} & -\dfrac{2}{\delta t^2}\displaystyle\int_{\Omega^{\mathrm{s}}_0} \rho_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s},n} + \delta t \boldsymbol{w}^{\mathrm{s},n})\boldsymbol{v}^{\mathrm{s}} + \int_{\Omega^{\mathrm{s}}_0} \boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s},n+1}) \cdot \boldsymbol{\nabla}\boldsymbol{v}^{\mathrm{s}} \\ & = \displaystyle\int_{\partial\Omega^{\mathrm{s}}_0 \setminus \Gamma_0} \boldsymbol{g}_{\mathrm{s}} \cdot \boldsymbol{v}^{\mathrm{s}} \\ \boldsymbol{w}^{\mathrm{s},n+1} & = \dfrac{2}{\delta t}(\boldsymbol{d}^{\mathrm{s},n+1} - \boldsymbol{d}^{\mathrm{s},n}) - \boldsymbol{w}^{\mathrm{s},n} \\ \boldsymbol{d}^{\mathrm{s},n+1} & = \lambda(t^{n+1}) \quad \text{on } \Gamma_0, \end{cases} \tag{2.11}$$

for all $\boldsymbol{v}^{\mathrm{s}} \in V^{\mathrm{s}}$ such that $\boldsymbol{v}^{\mathrm{s}}_{|\Gamma_0} = 0$, with $V^{\mathrm{s}} = H^1(\Omega^{\mathrm{s}}_0)^3$. As for the fluid, we can then compute the structure normal stresses on the interface as $\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s},n+1}) \cdot \boldsymbol{n}_{\mathrm{s}}$ on Γ_0.

If for a given interface displacement $\lambda(t^{n+1})$ the fluid and structure normal stresses are at equilibrium, it means that the fluid-structure problem has been correctly solved. In general we impose the equilibrium in weak form, i.e.,

$$\int_{\Gamma(t^{n+1})} \boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u}, p) \cdot \boldsymbol{n}_{\mathrm{f}} \boldsymbol{v}^{\mathrm{f}} + \int_{\Gamma_0} \boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s}}) \cdot \boldsymbol{n}_{\mathrm{s}} \boldsymbol{v}^{s} = 0$$
$$\forall (\boldsymbol{v}^{\mathrm{f}}, \boldsymbol{v}^{\mathrm{s}}) \in V^{\mathrm{f}}(t^{n+1}) \times V^{\mathrm{s}} \text{ s.t. } \boldsymbol{v}^{\mathrm{f}} \circ \boldsymbol{x}^{\mathrm{f}}_t = \boldsymbol{v}^{\mathrm{s}} \text{ on } \Gamma_0.$$

Both integrals can be computed as residuals of the weak form of the equations. We consider the coupled problem at a particular time $t = t^{n+1}$. In order to write the interface equation associated to the global fluid-structure problem, we introduce a fluid and structure operator as follows.

Let S_{f} be the Dirichlet-to-Neumann (D-t-N) fluid map such that to any given interface displacement λ it associates the normal stress

$$S_{\mathrm{f}}(\lambda) = \sigma_{\mathrm{f}} := (\boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u}, p) \cdot \boldsymbol{n}_{\mathrm{f}}) \circ \boldsymbol{x}^{\mathrm{f}}_t \text{ on } \Gamma_0,$$

where $(\boldsymbol{u}, p)$ is the solution of the Navier-Stokes problem (2.10). On the other hand, we denote by S_{s} the D-t-N operator associated to the structure in Γ_0 such that to any given displacement λ of the interface Γ_0 it associates the normal stress exerted by the structure on Γ_0:

$$S_{\mathrm{s}}(\lambda) = \sigma_{\mathrm{s}} := (\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s}}) \cdot \boldsymbol{n}_{\mathrm{s}}) \text{ on } \Gamma_0,$$

where $\boldsymbol{d}^{\mathrm{s}}$ is the solution of (2.11).

Concerning the inverse of the solid operator, we can define S_{s}^{-1} as a Neumann-to-Dirichlet (N-t-D) map that at any given normal stress σ on Γ_0 associates the interface displacement $\lambda(t^{n+1}) = \boldsymbol{d}^{\mathrm{s},n+1}$ by solving a structure problem analogous to (2.11), but with the Neumann boundary condition

$$\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{s}) \cdot \boldsymbol{n}_{\mathrm{s}} = \sigma \text{ on } \Gamma_0$$

and then computing the restriction on Γ_0 of the displacement of the structure domain.

Moreover, we denote by S'_{s} the tangent operator associated to the structure problem and by $(S'_{\mathrm{s}})^{-1}$ its inverse. The latter is a N-t-D map that to any given normal stress σ on Γ_0 associates the corresponding displacement $\lambda(t^{n+1})$ of the interface by solving the linearized structure problem with boundary condition $\boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s}}) \cdot \boldsymbol{n}_{\mathrm{s}} = \sigma$ on Γ_0. Analogously, by $(S'_{\mathrm{f}})^{-1}$ we denote the inverse of the tangent operator S'_{f}. This is also a N-t-D map that for any given normal stress σ on Γ_0 computes

the corresponding displacement $\lambda(t^{n+1})$ of the interface through the solution of linearized Navier-Stokes equations with the boundary condition $(\boldsymbol{\sigma}_{\mathrm{f}}(\boldsymbol{u},p)\cdot\boldsymbol{n}_{\mathrm{f}})\circ\boldsymbol{x}^{\mathrm{f}} = \sigma$ on Γ_0. Using the definitions of the operators S_{f} and S_{s} and of their inverses, we can express the coupled fluid-structure problem in terms of the solution λ of a nonlinear equation defined only on Γ_0. More precisely, we can envisage three possible formulations for the interface equation which are all equivalent from a mathematical point of view, but give rise to different iterative methods.

First, we have the fixed-point formulation

$$\text{find } \lambda \text{ such that } S_{\mathrm{s}}^{-1}(-S_{\mathrm{f}}(\lambda)) = \lambda \text{ on } \Gamma_0. \tag{2.12}$$

This is a classical formulation in fluid-structure interaction problems, but it is worth pointing out that here the fixed point is the displacement of the sole interface, whereas in the literature the solution obtained via fixed-point algorithms usually represents the displacement of the whole solid domain.

The second possible approach is a slight modification of the previous equation (2.12)

$$\text{find } \lambda \text{ such that } S_{\mathrm{s}}^{-1}(-S_{\mathrm{f}}(\lambda)) - \lambda = 0 \text{ on } \Gamma_0, \tag{2.13}$$

which is more suitable for setting up a Newton iterative method. Again, this is applied solely to the interface displacement, instead of the whole solid displacement as proposed.

Let's have a look at the code located at `testsuite/test_fsi/main.cpp`. The first interesting part is the problem definition part, starting from these lines

```
Problem( GetPot const& data_file, std::string _oper = "" )
    {
        using namespace LifeV;

        Debug( 10000 ) << "creating FSISolver with operator :  " << _oper << "\n";

        M_fsi = fsi_solver_ptr(  new FSISolver( data_file, _oper ) );
        Debug( 10000 ) << _oper << " set \n";

        MPI_Barrier(MPI_COMM_WORLD);
```

This will create a new fluid/structure interaction problem that will be solved using a non-linear Richardson algorithm on the following interface equation

$$\lambda^{k+1} = \lambda^k + \omega^k f(\lambda^k), \tag{2.14}$$

where f depends on the chosen FSI method. If we look at the FSI problem constructor the file `life/lifesolver/FSISolver.cpp`, we have

```
FSISolver::FSISolver( GetPot const& data_file,
                      std::string   __oper ):
    M_lambda      (),
    M_lambdaDot      (),
    M_firstIter (true),
    M_method    ( data_file("problem/method"     , "steklovPoincare") ),
    M_maxpf     ( data_file("problem/maxSubIter" , 300) ),
    M_defomega  ( data_file("problem/defOmega"   , 0.01) ),
    M_abstol    ( data_file("problem/abstol"     , 1.e-07) ),
    M_reltol    ( data_file("problem/reltol"     , 1.e-04) ),
    M_etamax    ( data_file("problem/etamax"     , 1.e-03) ),
    M_linesearch( data_file("problem/linesearch" , 0) ),
    M_epetraComm(),
    M_epetraWorldComm(),
```

```
M_localComm (new MPI_Comm),
M_interComm (new MPI_Comm),
out_iter    ("iter"),
out_res     ("res")
```

where

- `M_lambda` is the interface displacement as defined in (2.8),
- `M_lambdaDot` is the temporal derivative of `M_lambda`.

See table 2.6 for a complete of these options of options.

Name	Options	Description
method	fixedPoint exactJacobian	FSI resolution method name
maxSubIter	300	maximum nonlinear Richardson iterations
defOmega	0.01	default step in (2.14) (deprecated)
abstol reltol	1e-07 1e-04	abstol and reltol define the stoping criteria as abstol+reltol*norm(residual0) where residual0 is the first nonlinear Richardson FSI evaluation residual.
etamax	1e-03	Maximum error tolerance for residual in the linear solver.
linesearch monolithic	0 0	nonlinear Richardson algorithm linesearch (always use 0, i.e no line search, for now). monolithic description of the FSI problem (under development)

Table 2.6: FSI problem data file parameters

The `monolithic` description of the FSI problem is still under development and should not be used for now. Let's have the look at the rest of the FSI problem constructor code.

```
MPI_Group  originGroup, newGroup;
MPI_Comm   newComm;

MPI_Comm_group(MPI_COMM_WORLD, &originGroup);

if (numtasks == 1)
    {
        std::cout << "Serial Fluid/Structure computation" << std::endl;
        newComm = MPI_COMM_WORLD;
        fluid = true;
        solid = true;
        fluidLeader = 0;
        solidLeader = 0;

        M_epetraWorldComm.reset(new Epetra_MpiComm(MPI_COMM_WORLD));
        M_epetraComm = M_epetraWorldComm;

    }
else
    {
        int members[numtasks];
```

```
        solidLeader = 0;
        fluidLeader = 1 - solidLeader;

        if (rank == solidLeader)
            {
                members[0] = solidLeader;
                int ierr;
                ierr = MPI_Group_incl(originGroup, 1, members, &newGroup);
                solid = true;
            }
        else
            {
                for (int ii = 0; ii <= numtasks; ++ii)
                    {
                        if ( ii < solidLeader)
                            members[ii] = ii;
                        else if ( ii > solidLeader)
                            members[ii - 1] = ii;
                    }
                int ierr;
                ierr = MPI_Group_incl(originGroup,
                                      numtasks - 1,
                                      members,
                                      &newGroup);
                fluid = true;
            }

        MPI_Comm* localComm = new MPI_Comm;
        MPI_Comm_create(MPI_COMM_WORLD, newGroup, localComm);
        M_localComm.reset(localComm);

        M_epetraComm.reset(new Epetra_MpiComm(*M_localComm.get()));
        M_epetraWorldComm.reset(new Epetra_MpiComm(MPI_COMM_WORLD));
    }
```

This part is dedicated at assigning jobs to the different processors. Since we have to deal with a separate structure and fluid problems, we define two MPI groups where for each problem. For now, by convention, and since the structure problem is resolved more easily than the fluid problem by far, the structure group has only the #0 (leader) processor, whereas the fluid group is composed of all the other processors. At the end of these lines, two MPI intracommunicators (communicator within a single group of processes), one for the structure, one for the fuid, and one MPI intercommunicator (communicator within two or more groups of processes) are created. See http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html for more information on intra and intercommunicators.

From now on, each processor knows if it belongs to the structure or the fluid group. This is very important since each group will create its own problem, either stucture of fluid. Since the structure problem requires less ressources than the fluid problem, the FSI problem defined in (2.14) will be solved on a lead structure processor (#0 processor within the structure intracommunicator).

```
Preconditioner precond  = ( Preconditioner )
                          data_file("problem/precond"    ,
```

```
                                DIRICHLET_NEUMANN );

    Debug( 6220 ) << "FSISolver::preconditioner: " << precond << "\n";

    if ( !__oper.empty() )
    {
        M_method = __oper;
    }

    Debug( 6220 ) << "FSISolver::setFSIOperator " << M_method << "\n";

    this->setFSIOperator( M_method );

    M_oper->setFluid(fluid);
    M_oper->setSolid(solid);

    M_oper->setFluidLeader(fluidLeader);
    M_oper->setSolidLeader(solidLeader);

    Debug( 6220 ) << "FSISolver::setPreconditioner " << precond << "\n";

    std::cout << std::flush;
    M_oper->setComm(M_epetraComm, M_epetraWorldComm);

    Debug( 6220 ) << "FSISolver::setDataFromGetPot " << precond << "\n";
    std::cout << std::flush;

    M_oper->setDataFromGetPot( data_file );


    Debug( 6220 ) << "FSISolver::setPrecond " << precond << "\n";
    std::cout << std::flush;

    M_oper->setPreconditioner( precond );

    M_oper->setup();

    Debug( 6220 ) << "FSISolver:: variable setup " << precond << "\n";

    M_oper->setUpSystem(data_file);

    M_lambda.reset   (new vector_type(*M_oper->solidInterfaceMap()));
    M_lambdaDot.reset(new vector_type(*M_oper->solidInterfaceMap()));
    M_oper->buildSystem();
```

This part creates the proper numerical FSI Operator (fixed point, exact jacobian, ...) for solving (2.14), that is the fluid and the structure operators (S_f, S_s ...) defined previously, and set them up. You can have a look at the code which is in `life/lifesolver/FSIOpertator.hpp,cpp`, `life/lifesolver/exactJacobianBase.hpp,cpp` and `life/lifesolver/fixedPointBase.hpp,cpp`. The last two classes, exactJacobian and fixedPoint, derive from the class FSIOperator, which only deals with passing information (displacement or constrain) from the solid or the fluid to the fluid or solid via the interface. The the specialized classes exactJacobian or fixedPoint evalute the interface residual and solve $f(\lambda)$ as defined in (2.14). The last interesting part in the `FSISolver`

class, is the `iterate` member

```
M_oper->setTime(time);

fct_type fluidSource(zero_scalar);
fct_type solidSource(zero_scalar);

if(!M_monolithic)
    M_oper->updateSystem(fluidSource, solidSource);
else
    M_oper->updateSystem(*M_lambda);

// displacement prediction
MPI_Barrier(MPI_COMM_WORLD);

if (M_firstIter)
{
    M_firstIter = false;

    if(!M_monolithic)
        {
            *M_lambda      = M_oper->lambdaSolid();
            *M_lambda     += timeStep()*M_oper->lambdaDotSolid();
            *M_lambdaDot   = M_oper->lambdaDotSolid();
        }
}
else
{

    if(!M_monolithic)
        {
            *M_lambda      = M_oper->lambdaSolid();
            *M_lambda     += 1.5*timeStep()*M_oper->lambdaDotSolid(); // *1.5
            *M_lambda     -= timeStep()*0.5*(*M_lambdaDot);
            *M_lambdaDot   = M_oper->lambdaDotSolid();
        }
}


if (!M_monolithic)
    {
        M_oper->leaderPrint("norm( disp  ) init = ", M_lambda->NormInf() );
        M_oper->leaderPrint("norm( velo )  init = ", M_lambdaDot->NormInf());
    } else {
        M_oper->leaderPrint("norm( solution ) init = ", M_lambda->NormInf() );
    }

MPI_Barrier(MPI_COMM_WORLD);

int maxiter = M_maxpf;

// the newton solver
UInt status = 1;
Debug( 6220 ) << "Calling non-linear Richardson \n";
```

```
status = nonLinRichardson(*M_lambda,
                          *M_oper,
                          norm_inf_adaptor(),
                          M_abstol,
                          M_reltol,
                          maxiter,
                          M_etamax,
                          M_linesearch,
                          out_res,
                          time);

if(status == 1)
{
    std::ostringstream __ex;
    __ex << "FSISolver::iterate ( " << time << " )
             Inners iterations failed to converge\n";
    throw std::logic_error( __ex.str() );
}
else
{
    M_oper->leaderPrint("End of time " , time);
    M_oper->leaderPrint("Number of inner iterations       : ", maxiter );
    out_iter << time << " " << maxiter << " "
             << M_oper->nbEval() << std::endl;
}
if(!M_monolithic)
    M_oper->shiftSolution();
```

This code will perform one FSI time step. First, it "guesses" the interface displacement by interpolation of the previous displacement, then it will call the non-linear Richardson algorithm that will compute the new displacement by solving (2.14).

Bibliography

Index

algebra
    aztec, 6
autoconf, *see* autotools
automake, *see* autotools
autotools, 5
    autoconf, 5
    automake, 5
    configure, 8
    libtool, 5
aztec, *see* algebra

Backward Differentiation Formula, 25
Boundary Conditions, 17

compilers
    g++, 5
configure, *see* autotools

Finite Element
    Finite Element Space, 19
    Reference Finite Element, 19
Finite Element Space, *see* Finite Element
Fluid-Structure Interactions, 26

g++, *see* compilers
git, *see* versioning
GNU Makefile
    Makefile, 12

libtool, *see* autotools

make, 8
Makefile, *see* GNU Makefile
Mesh, 18
    Paritioning, 18

Navier-Stokes Problem, 24

Paritioning, *see* Mesh

Quatrature Rules, 19

Reference Finite Element, *see* Finite Element

ssh, 7
Stokes problem, 11

versioning
    git, 4