LifeV User Manual

Revision: 1.8, , UTC Printed: January 10, 2013

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: `http://cmcsforge.epfl.ch/www/lifev/doxygen-release-2.2.0/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. `http://www.lifev.org/documentation/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 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:

- `g++-4.0` or newer (currently 4.5.4)
- `mpi`, with preference to `openmpi`
- `CMake 2.8` or newer.

In Mac OS X you get gcc 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`. Note however that MPI should be natively installed if you installed XCode.

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

- `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 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`. You can download it 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/cmake
```

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 `do-configure` shell script in the `opt` directory. It will automatically check the availability of the needed components for *LifeV* compilation :

```
cmake \
    -D CMAKE_BUILD_TYPE:STRING=RELEASE \
    \
    -D TPL_ENABLE_MPI:BOOL=ON \
    \
    -D ParMETIS_INCLUDE_DIRS:PATH=/parmetis_include_path/ \
    -D ParMETIS_LIBRARY_DIRS:PATH=/parmetis_library_path/ \
    \
    -D TPL_ENABLE_UMFPACK: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_Boost:BOOL=ON \
    -D TPL_Boost_INCLUDE_DIRS:STRING=/boost_include_path/ \
    \
    -D TPL_ENABLE_Trilinos:STRING=ON \
    -D Trilinos_INCLUDE_DIRS:PATH=/trilinos_include_path/ \
    -D Trilinos_LIBRARY_DIRS:PATH=/trilinos_library_path/ \
    \
    -D LifeV_VERBOSE_CONFIGURE:BOOL=OFF \
    -D CMAKE_VERBOSE_MAKEFILE:BOOL=ON \
    \
    -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=$MY_LIB_DIR \
    $* \
    $BASE_DIR/lifev
```

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

```
-D CMAKE_BUILD_TYPE:STRING=DEBUG\
```

Finally, you just have to use `make` to compile *LifeV* libraries and documentation. Enter

```
make -j 4
make install
```

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 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 following the instructions of the previous section,
4. compile and install it

   ```
   make -j 4
   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 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

$$\begin{cases} \alpha\,\boldsymbol{u} + \boldsymbol{\beta}\cdot\nabla\boldsymbol{u} - \nu\Delta\boldsymbol{u} + \nabla p & = \boldsymbol{f} \quad \text{in } \Omega \\ \nabla\cdot\boldsymbol{u} & = 0 \quad \text{in } \Omega \end{cases} \tag{2.1}$$

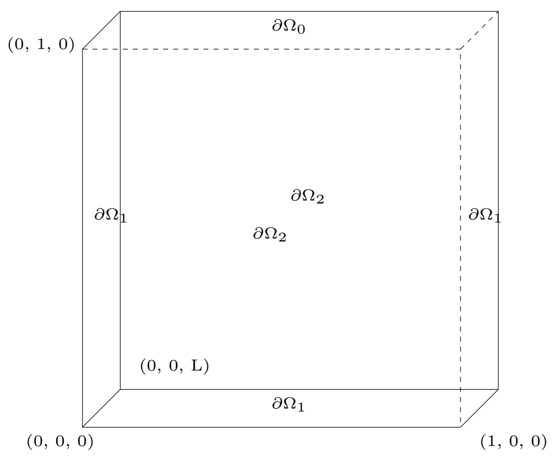
completed with suitable boundary conditions. Here $\Omega \subset \mathbb{R}^d$ $(d = 1, 2, 3)$ is the domain occupied by the fluid and ν is the kinematic viscosity of the fluid. If we set the convective acceleration $\boldsymbol{\beta}$ and α to zero, we get the Stokes equations

$$\begin{cases} -\nu\Delta\boldsymbol{u} + \nabla p & = \boldsymbol{f} \quad \text{in } \Omega \\ \nabla\cdot\boldsymbol{u} & = 0 \quad \text{in } \Omega\,. \end{cases} \tag{2.2}$$

We want to solve the following Stokes problem

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

where $\boldsymbol{n}$ is the normal unit vector to the domain boundary. The considered 3D domain Ω is represented by



Writing $\Gamma_D = \partial\Omega_0 \cup \partial\Omega_1$ and $H^1_{\Gamma_D}(\Omega) = \{\boldsymbol{v} \in H^1(\Omega) : \boldsymbol{v} = \boldsymbol{0} \text{ on } \Gamma_D\}$, we can introduce the bilinear forms

$$\begin{aligned} \forall\, \boldsymbol{u}, \boldsymbol{v} \in \left(H^1(\Omega)\right)^3 \quad &: \quad a(\boldsymbol{u}, \boldsymbol{v}) = \nu \int_\Omega \nabla \boldsymbol{u} \cdot \nabla \boldsymbol{v} dx \\ \forall\, \boldsymbol{v} \in \left(H^1(\Omega)\right)^3,\ q \in L^2(\Omega) \quad &: \quad b(\boldsymbol{v}, q) = \int_\Omega q \nabla \cdot \boldsymbol{v} dx\,. \end{aligned}$$

In variational form, the system (2.3) thus reads: find $\boldsymbol{u}_0 \in \left(H^1_{\Gamma_D}(\Omega)\right)^3$ et $p \in L^2(\Omega)$ such that

$$\left\{ \begin{array}{rll} a(\boldsymbol{u}_0, \boldsymbol{v}) + b(\boldsymbol{v}, p) & = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} dx \, + \, \langle F^0, \boldsymbol{v} \rangle & \forall\, \boldsymbol{v} \in \left(H^1_{\Gamma_D}(\Omega)\right)^3 \\ b(\boldsymbol{u}, q) & = \langle G^0, q \rangle & \forall\, q \in L^2(\Omega) \end{array} \right. \tag{2.4}$$

where F^0 et G^0 are terms due to non-homogeneous Dirichlet boundary conditions on $\partial\Omega_0$.

In order to solve (2.4) using *LifeV*, let us create a working directory `cavity_stokes` in the directory `<lifev directory>/lifev/navier_stokes/examples/` and get the following files:

- CMakeLists.txt
- cavity_stokes.cpp
- data_stokes

We first draw attention to the fact that the top-level directory `examples` contains also a CMakeLists.txt file:

```
INCLUDE(TribitsAddExecutableAndTest)
INCLUDE(AddSubdirectories)

ADD_SUBDIRECTORIES(
  cavity_ns
  cavity_stokes
  cylinder
  oseen_assembler
)
```

The last instructions recurse into the subdirectories. This does not actually cause another cmake executable to run.

Let's then have a look at the CMakeLists.txt file created in our working subdirectory `cavity_stokes`:

```
INCLUDE(TribitsAddExecutable)

# Make sure the compiler can find include files
INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR})

# Add executable called "cavity_stokes_test" that is built from the source files
# "cavity_stokes.cpp". (Note that the name of the executable has to be uniquely chosen.)
TRIBITS_ADD_EXECUTABLE(
  cavity_stokes_test
  SOURCES cavity_stokes.cpp
  COMM serial mpi
  )
```

```
# Access the data
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  data_cavity_stokes_test
  SOURCE_FILES data
  SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}
)

# Access the mesh
TRIBITS_COPY_FILES_TO_BINARY_DIR(
  meshes_cavity_stokes_test
  SOURCE_FILES cube4x4.mesh
  SOURCE_DIR ${CMAKE_SOURCE_DIR}/lifev/core/data/mesh/freefem/
)
```

More information about using cmake is available at http://www.cmake.org/cmake/help/help.html.

We can now look at the sources contained in the file cavity_stokes.cpp. We first include all the useful header files:

```
#include <lifev/core/LifeV.hpp>
#include <lifev/core/util/LifeChrono.hpp>
#include <lifev/core/array/MapEpetra.hpp>
#include <lifev/core/mesh/MeshData.hpp>
#include <lifev/core/mesh/MeshPartitioner.hpp>
#include <lifev/core/fem/FESpace.hpp>
#include <lifev/navier_stokes/solver/OseenData.hpp>
#include <lifev/navier_stokes/solver/OseenSolver.hpp>
#include <lifev/core/filter/ExporterEnsight.hpp>
#include <lifev/core/filter/ExporterHDF5.hpp>
#include <lifev/core/filter/ExporterEmpty.hpp>

// Trilinos-MPI communication definitions
#include <Epetra_ConfigDefs.h>
#ifdef HAVE_MPI
#include "Epetra_MpiComm.h"
#else
#include "Epetra_SerialComm.h"
#endif
```

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

The following instruction allows to use LifeV objects without refering to LifeV:: everytime. Without it, we have to write LifeV::RefFE instead of just RefFE for instance.

```
using namespace LifeV;
```

We introduce some typedef declarations to assign more pratical denominations to existing types and we label the different parts of our domain Ω to be able to apply later the boundary conditions.

```
typedef RegionMesh<LinearTetra>                 mesh_Type;
typedef OseenSolver< mesh_Type >                fluid_Type;
typedef fluid_Type::vector_Type                 vector_Type;
typedef boost::shared_ptr<vector_Type>          vectorPtr_Type;    // Pointer
typedef FESpace< mesh_Type, MapEpetra >         feSpace_Type;
typedef boost::shared_ptr<feSpace_Type>         feSpacePtr_Type;   // Pointer

const int BACK   = 1;
const int FRONT  = 2;
const int LEFT   = 3;
const int RIGHT  = 4;
const int BOTTOM = 5;
const int TOP    = 6;
```

We next define real functions that will be used in the boundary condition object.

```
Real lidBC(const Real& /*t*/,
           const Real& /*x*/, const Real& /*y*/, const Real& /*z*/,
           const ID& i)
{
    switch (i)
    {
    case 1:
        return 1.0;
    default:
        return 0.0;
    }
}

Real fZero( const Real& /* t */,
            const Real& /* x */, const Real& /* y */, const Real& /* z */,
            const ID& /* i */ )
{
    return 0.0;
}
```

More generally speaking, 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 coordinates, 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 $\partial\Omega_0$ 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 can now proceed to the main block of the code.

```
int main(int argc, char** argv)
{
#ifdef HAVE_MPI
    MPI_Init(&argc, &argv);
#endif

    boost::shared_ptr<Epetra_Comm>   comm;
#ifdef EPETRA_MPI
    comm.reset( new Epetra_MpiComm( MPI_COMM_WORLD ) );
    int nproc;
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
#else
    comm.reset( new Epetra_SerialComm() );
#endif
```

These part 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` for more explanations.

The next 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).

```
    bool verbose(false);
    if (comm->MyPID() == 0)
    {
        verbose = true;
        std::cout << "[Initilization of MPI]" << std::endl;
#ifdef HAVE_MPI
        std::cout << "Using MPI (" << nproc << " proc.)" << std::endl;
#else
        std::cout << "Using serial version" << std::endl;
#endif
    }
```

Then, 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. By default (i.e. if no name is given) "data" is selected.

```
    if (verbose) std::cout << std::endl << "[Loading the data]" << std::endl;
    GetPot command_line(argc,argv);
    const std::string dataFileName = command_line.follow("data", 2, "-f","--file");
    GetPot dataFile(dataFileName);
```

This GetPot object is used to store values like:

- the mesh name,
- the time step (if any),
- 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).
We then create the mesh object and split it between processors

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

Table 2.1: Description of the discretization parameters

```
if (verbose) std::cout << std::endl << "[Loading the mesh]" << std::endl;
MeshData meshData;
meshData.setup(dataFile, "fluid/space_discretization");
if (verbose) std::cout << "Mesh file: " << meshData.meshDir()
                              << meshData.meshFile() << std::endl;
boost::shared_ptr< mesh_Type > fullMeshPtr(new mesh_Type);
readMesh(*fullMeshPtr, meshData);
MeshPartitioner< mesh_Type >   meshPart(fullMeshPtr, comm);
```

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. In our case, after the call to the `MeshPartitioner` constructor, `meshPart` will store the local part of the mesh. Using this local mesh, we can create our Finite Element spaces:

```
if (verbose) std::cout << std::endl << "[Creating the FE spaces]" << std::endl;
std::string uOrder =  dataFile( "fluid/space_discretization/vel_order",   "P2");
std::string pOrder =  dataFile( "fluid/space_discretization/press_order", "P1");

if (verbose) std::cout << "FE for the velocity: " << uOrder << std::endl
                           << "FE for the pressure: " << pOrder << std::endl;
if (verbose) std::cout << "Building the velocity FE space... " << std::flush;
feSpacePtr_Type uFESpacePtr( new feSpace_Type(meshPart, uOrder, 3, comm) );
if (verbose) std::cout << "ok." << std::endl;

if (verbose) std::cout << "Building the pressure FE space... " << std::flush;
feSpacePtr_Type pFESpacePtr( new feSpace_Type(meshPart,pOrder,1,comm) );
if (verbose) std::cout << "ok." << std::endl;

// Total degrees of freedom (elements of matrix)
UInt totalVelDof   = uFESpacePtr->map().map(Unique)->NumGlobalElements();
UInt totalPressDof = pFESpacePtr->map().map(Unique)->NumGlobalElements();
```

The constructor called for the instances of the FESpace class takes as arguments

- the mesh,
- the chosen elements type (here we decided to use P2/P1 elements),
- the fields dimension,

Name	Options	Description
type	Natural Essential Robin	Neumann Dirichlet
mode	Scalar Full Component Normal Tangential	1 dimension BC 3 component BC Separate component BC Normal BC Tangential BC

Table 2.2: Boundary Condition parameters description

- the MPI communicator.

Let's continue with the specification of the boundary conditions on all our boundary faces. BCHandler is the class that stores the boundary conditions. Here is the prototype of the `addBC` function that we use for the assignment.

```
//! 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 condition applies
\param type the boundary condition type: Natural, Essential, Robin
\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&     type,
            const bcMode_Type&     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, `mode` is the mode, `bcf` is the function holding the user-defined function involved in the boundary condition. Moreover, `type` and `mode` are respectively the boundary condition type and mode. Please refer to the table (2.2) for a description of their values. Finally, `comp` is a vector storing the components in the boundary condition.

In our example, we recall that we have to set essential boundary conditions on the top face, $\boldsymbol{u} = (1, 0, 0)$, and on the left, right and bottom faces, $\boldsymbol{u} = (0, 0, 0)$. We have:

```
    if (verbose) std::cout<< std::endl << "[Boundary conditions]" << std::endl;

    BCFunctionBase uZero(fZero);
    BCFunctionBase uLid(lidBC);

    BCHandler bcH;
    bcH.addBC( "Top"    , TOP    , Essential, Full, uLid , 3 );
    bcH.addBC( "Left"   , LEFT   , Essential, Full, uZero, 3 );
    bcH.addBC( "Right"  , RIGHT  , Essential, Full, uZero, 3 );
    bcH.addBC( "Bottom", BOTTOM, Essential, Full, uZero, 3 );
```

For the front and back faces, we have natural boundary conditions, $\boldsymbol{u}\cdot\boldsymbol{n} = 0$. Note however that this means

$$u_x = 0 \text{ for } x = 1, L$$

(since the two concerned planes are defined by $x = 0$ and $x = L$). So in order to set our first component, `x`, we define the vector

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

Then, we add an essential boundary condition on the x-component by calling

```
bcH.addBC( "Back"  , BACK  , Essential, Component, uZero, xComp );
bcH.addBC( "Front" , FRONT , Essential, Component, uZero, xComp );
```

This will set a null function to the first component on the front and back faces. We could have just easily defined another function for the others components using the same procedure.

We also mention that we have also the possibility to apply Flux boundary conditions. Here is the procedure that we can use:

```
// Get the number of Lagrange Multiplyers (LM) and set the offsets
std::vector<bcName_Type> fluxVector = bcH.findAllBCWithType( Flux );
UInt numLM = static_cast<UInt>( fluxVector.size() );

UInt offset = uFESpacePtr->map().map(Unique)->NumGlobalElements()
                     + pFESpacePtr->map().map(Unique)->NumGlobalElements();

for ( UInt i = 0; i < numLM; ++i )
    bcH.setOffset( fluxVector[i], offset + i );
```

In our case, we simply note that `numLM = 0` and so the loop is not entered.

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.

As our problem can be view as a special case of an Oseen system, we will first create and setup an object from a class, the `OseenData` class, designed to store and manage the related informations.

```
if (verbose) std::cout<< std::endl << "[Creating the problem]" << std::endl;
   boost::shared_ptr< OseenData > oseenData(new OseenData);
   oseenData->setup( dataFile );
```

Calling now the constructor of our solver 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

```
// The problem (matrix and rhs) is packed in an object called fluid
OseenSolver< mesh_Type > fluid (oseenData, *uFESpacePtr, *pFESpacePtr,
                                 comm, numLM);
```

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

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 2.3: Main parameters for the Trilinos solver

```
// Gets inputs from 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 (assemble the matrices):

```
fluid.buildSystem();
```

It 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.

```
 // Communication map:
MapEpetra fullMap(fluid.getMap());
```

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

It is useful to create vectors to store the solutions vector after the linear system is solved and before, to setup attributes of the `fluid` object: we set the right handside (`rhs`) to zero. As we deal with a Stokes problem, there is also no mass matrix (`alpha`) and no advection term (`beta`). These are the three arguments needed to update the system in the Oseen class:

```
    vector_Type beta( fullMap );
    vector_Type rhs ( fullMap );
    double alpha=0.; // Stokes problem
    beta *= 0.;  // Stokes problem
    rhs  *= 0.;
    fluid.updateSystem(alpha,beta,rhs);
    fluid.iterate(bcH);
```

The last line will result in solving the system (2.3) and we note that the member `iterate` requires the boundary conditions as parameters. More precisely, it will:

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

In *LifeV*, we mainly use Paraview in order to postprocess our problem solutions. Writing a Paraview solution is quite straightforward using the ExporterHDF5 class.

```
    boost::shared_ptr< ExporterHDF5<mesh_Type> > exporter;
    std::string const exporterType =  dataFile( "exporter/type", "ensight");
    exporter.reset( new ExporterHDF5<mesh_Type> ( dataFile, "cavity_stokes_example" ) );
    exporter->setPostDir( "./" );  // This is a test to see if M_post_dir is working
    exporter->setMeshProcId( meshPart.meshPartition(), comm->MyPID() );
```

We have to define a variable that will store the solution:

```
    vectorPtr_Type velAndPressure;
    velAndPressure.reset( new vector_Type(*fluid.solution(), exporter->mapType() ) );
```

Finally, we add the variables to be saved:

```
    exporter->addVariable( ExporterData<mesh_Type>::VectorField, "velocity",
                           uFESpacePtr, velAndPressure, UInt(0) );
    exporter->addVariable( ExporterData<mesh_Type>::ScalarField, "pressure",
                           pFESpacePtr, velAndPressure,
                           UInt(3*uFESpacePtr->dof().numTotalDof()) );
    exporter->postProcess( 0 );
    exporter->closeFile();
```

The final statements stops the parallel part of the code and ends the main:

```
#ifdef HAVE_MPI
    MPI_Finalize();
#endif

    return 0;
}
```

You must run the cavity_stokes executable named `cavity_stokes_example` using the command `mpirun -np procs cavity_stokes_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 described 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)

$$\left\{ \begin{array}{rll} \alpha \boldsymbol{u} + \boldsymbol{\beta} \cdot \nabla \boldsymbol{u} - \nu \Delta \boldsymbol{u} + \nabla p & = \boldsymbol{f} & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} & = 0 & \text{in } \Omega \end{array} \right.$$

plus boundary conditions.

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

$$\left\{ \begin{array}{rll} \dfrac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} - \nu \Delta \boldsymbol{u} + \nabla p & = \boldsymbol{f} & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} & = 0 & \text{in } \Omega \end{array} \right.$$

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

$$\left\{ \begin{array}{rll} \alpha \boldsymbol{u}^{n+1} + \boldsymbol{u}^{n+1} \cdot \nabla \boldsymbol{u}^{n+1} - \nu \Delta \boldsymbol{u}^{n+1} + \nabla p^{n+1} & = \boldsymbol{f}^n & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} & = 0 & \text{in } \Omega \, . \end{array} \right.$$

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

$$\left\{ \begin{array}{rll} \alpha \boldsymbol{u}^{n+1} + \boldsymbol{\beta}^{n} \cdot \nabla \boldsymbol{u}^{n+1} - \nu \Delta \boldsymbol{u}^{n+1} + \nabla p^{n+1} & = \boldsymbol{f}^n & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} & = 0 & \text{in } \Omega \, . \end{array} \right.$$

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

$$\left\{ \begin{array}{rll} \dfrac{\boldsymbol{u}^{n+1}}{\Delta t} + \boldsymbol{u}^{n} \cdot \nabla \boldsymbol{u}^{n+1} - \nu \Delta \boldsymbol{u}^{n+1} + \nabla p^{n+1} & = \dfrac{\boldsymbol{u}^n}{\Delta t} & \text{in } \Omega \\ \nabla \cdot \boldsymbol{u} & = 0 & \text{in } \Omega \, . \end{array} \right.$$

Or the second order discretization

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

Let's have a look at the code in

`<lifev directory>/lifev/navier_stokes/examples/cavity_ns.cpp`.

Apart from the fact that we included another header file that allows to deal with the time discretization :

```
#include <lifev/core/fem/TimeAdvanceBDFNavierStokes.hpp>
```

the code is the same as the one used to compute the Stokes problem until the definition of the communication Epetra `fullMap`.

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 :

```
if (verbose) std::cout<< std::endl << "[Initialization of the simulation]"
                                                           << std::endl;
Real dt     = oseenData->dataTime()->timeStep();
Real t0     = oseenData->dataTime()->initialTime();
Real tFinal = oseenData->dataTime()->endTime ();

TimeAdvanceBDFNavierStokes<vector_Type> bdf;
bdf.setup(oseenData->dataTime()->orderBDF());

t0 -= dt * bdf.bdfVelocity().order();
(...)

oseenData->dataTime()->setTime(t0);
(...)
 // We get the initial solution using a steady Stokes problem
bdf.bdfVelocity().setInitialCondition( *fluid.solution() );

Real time = t0 + dt;
for (  ; time <=  oseenData->dataTime()->initialTime() + dt/2.; time += dt)
{
    oseenData->dataTime()->setTime(time);

    fluid.updateSystem(alpha,beta,rhs);
    fluid.iterate(bcH);
    bdf.bdfVelocity().shiftRight( *fluid.solution() );
}
```

The bdf object is designed to store the previous solutions, $\boldsymbol{u}^n$, $\boldsymbol{u}^{n-1}$, ... We had 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 initialize it with our previously computed Stokes problem solution.
Moreover, we erase the preconditioner build for the Stokes problem (a new one should be built for Navier-Stokes) :

```
fluid.resetPreconditioner();
```

We are now ready to enter the time loop to solve the problem. Inside this loop, it is really like the initialization procedure, except that we now have an advection velocity, right handside and the mass matrix.

```
int iter = 1;

for ( ; time <= tFinal + dt/2.; time += dt, iter++)
{
   oseenData->dataTime()->setTime(time);
   double alpha = bdf.bdfVelocity().coefficientFirstDerivative( 0 )
                                        / oseenData->dataTime()->timeStep();
   // extrapolates of the advection term
   beta = bdf.bdfVelocity().extrapolation();

   // rhs part of the time-derivative
   bdf.bdfVelocity().updateRHSContribution(oseenData->dataTime()->timeStep() );
   rhs  = fluid.matrixMass()*bdf.bdfVelocity().rhsContributionFirstDerivative();
   (...)
```

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

        // and solves it
        fluid.iterate( bcH );

        // shifts the previous solutions
        bdf.bdfVelocity().shiftRight( *fluid.solution() );
         (...)
        // exports the solution
        *velAndPressure = *fluid.solution();
        exporter->postProcess( time );
         (...)
         }
    (...)
    exporter->closeFile();
```

In addition to the data used for the Stokes problem, all needed informations about the time discretization can be found in the data file:

- the time step `dt`, named `fluid/time_discretization/timestep`,
- the initial time `t0`, named `fluid/time_discretization/initialtime`,
- the final simulation time `tFinal`, named `fluid/time_discretization/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 $\boldsymbol{\beta}$ 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 $\Omega^{\mathrm{s}}(t)$ (such as an arterial wall, a pipe-line, ...) surrounding a fluid under motion (blood, oil, ...) in the complement $\Omega^{\mathrm{f}}(t)$ of $\Omega^{\mathrm{s}}(t)$ in $\Omega(t)$ (see Fig. 2.1).

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.

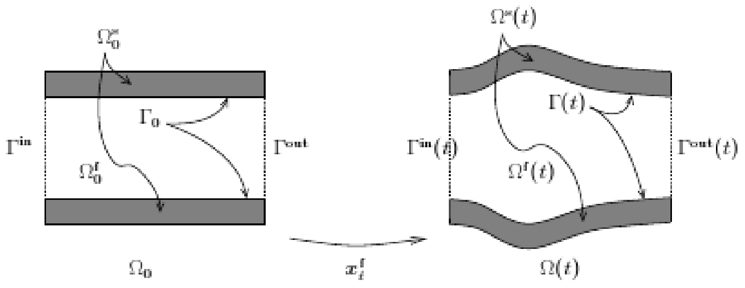


Figure 2.1: ALE mapping between the initial configuration and the configuration at time t.

We denote by $\Gamma^{\text{in}}(t)$ and $\Gamma^{\text{out}}(t)$ the inflow and outflow sections of the fluid domain, by $\boldsymbol{n}_{\text{f}}$ the fluid domain's outward normal on $\partial\Omega^{\text{f}}(t)$ and by $\boldsymbol{n}_{\text{s}}$ the one of the structure on the reference boundary $\partial\hat{\Omega}^{\text{s}}$. 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 $\Omega^{\text{f}}(t)$, 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

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

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

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

$$\text{Fluid}(\boldsymbol{u}, p, \boldsymbol{x}_t^{\text{f}}; \boldsymbol{u}_{\text{in}}, \boldsymbol{g}_{\text{f}}, \boldsymbol{f}_{\text{f}}) : \begin{cases} \boldsymbol{\Delta}\boldsymbol{x}_t^{\text{f}} = 0 \text{ in } \hat{\Omega}^{\text{f}}, \\ \boldsymbol{x}_t^{\text{f}} = 0 \text{ on } \partial\hat{\Omega}^{\text{f}} \setminus \hat{\Sigma}, \\ \Omega_t^{\text{f}} = \boldsymbol{x}_t^{\text{f}}(\hat{\Omega}^{\text{f}}), \\ \rho_{\text{f}} \left(\left.\dfrac{\partial \boldsymbol{u}}{\partial t}\right|_{\boldsymbol{x}_0} + (\boldsymbol{u} - \boldsymbol{w}) \cdot \boldsymbol{\nabla}\boldsymbol{u} \right) \\ \qquad = \text{div}(2\mu\boldsymbol{\epsilon}(\boldsymbol{u})) - \boldsymbol{\nabla}p + \boldsymbol{f}_{\text{f}} \text{ in } \Omega_t^{\text{f}}, \\ \text{div}\,\boldsymbol{u} = 0 \text{ in } \Omega_t^{\text{f}}, \\ \boldsymbol{u} = \boldsymbol{u}_{\text{in}} \text{ on } \Gamma_t^{\text{in}}, \\ \boldsymbol{\sigma}_{\text{f}}(\boldsymbol{u}, p) \cdot \boldsymbol{n}_{\text{f}} = \boldsymbol{g}_{\text{f}} \text{ on } \Gamma_t^{\text{out}}, \end{cases} \tag{2.5}$$

where ρ_{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 $\boldsymbol{\sigma}_{\text{f}}(\boldsymbol{u}, p) = -pId + 2\mu\boldsymbol{\epsilon}(\boldsymbol{u})$ the Cauchy stress tensor (Id is the identity matrix). Note that (2.5) does not univocally define a solution $(\boldsymbol{u}, p, \boldsymbol{x}_t^{\text{f}})$ as no boundary data are prescribed on the interface Σ_t.

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

$$\operatorname{Str}(\boldsymbol{d};\boldsymbol{g}_\mathrm{s},\boldsymbol{f}_\mathrm{s}):\begin{cases}\rho_\mathrm{s}\dfrac{\partial^2\boldsymbol{d}}{\partial t^2}=\operatorname{div}(\boldsymbol{\sigma}_\mathrm{s}(\boldsymbol{d}))-\gamma\boldsymbol{d}+\boldsymbol{f}_\mathrm{s}\text{ in }\hat{\Omega}^\mathrm{s},\\ \boldsymbol{\sigma}_\mathrm{s}(\boldsymbol{d})\cdot\boldsymbol{n}_\mathrm{s}=\boldsymbol{g}_\mathrm{s}\text{ on }\partial\hat{\Omega}^\mathrm{s}\setminus\Sigma_0,\end{cases}\tag{2.6}$$

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 (2.5), problem (2.6) 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

$$\begin{gathered}\boldsymbol{x}^\mathrm{f}_t=\lambda,\\ \boldsymbol{u}\circ\boldsymbol{x}^\mathrm{f}_t=\dot{\boldsymbol{d}}_{\Sigma^s},\\ (\boldsymbol{\sigma}_\mathrm{f}(\boldsymbol{u},p)\cdot\boldsymbol{n}_\mathrm{f})\circ\boldsymbol{x}^\mathrm{f}_t+\boldsymbol{\sigma}_\mathrm{s}(\boldsymbol{d})\cdot\boldsymbol{n}_\mathrm{s}=0,\end{gathered}\tag{2.7}$$

where $\dot{\boldsymbol{d}}_{\Sigma^s}$ denotes the temporal derivative of $\boldsymbol{d}_{|\Sigma^s}$. The system of equations (2.5)-(2.7) 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 $\boldsymbol{d}^\mathrm{f}{}_{t^{n+1}}\in H^1(\Omega^\mathrm{f}_0)^3$ such that

$$\begin{cases}\displaystyle\int_{\Omega^\mathrm{f}_0}\boldsymbol{\nabla}\boldsymbol{d}^\mathrm{f}{}_{t^{n+1}}\cdot\boldsymbol{\nabla}\boldsymbol{\phi}=0 & \forall\boldsymbol{\phi}\in H^1_0(\Omega^\mathrm{f}_0)^3\\ \qquad\boldsymbol{d}^\mathrm{f}{}_{t^{n+1}}=\lambda(t^{n+1}) & \text{on }\Gamma_0,\end{cases}\tag{2.8}$$

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

$$\begin{cases}\displaystyle\frac{1}{\delta t}\int_{\Omega^\mathrm{f}(t^{n+1})}\rho_\mathrm{f}\boldsymbol{u}^{n+1}\boldsymbol{v}^\mathrm{f}+\int_{\Omega^\mathrm{f}(t^{n+1})}\rho_\mathrm{f}[(\boldsymbol{u}^{n+1}-\boldsymbol{w}^{\mathrm{f},n+1})\cdot\boldsymbol{\nabla}\boldsymbol{u}^{n+1}]\boldsymbol{v}^\mathrm{f}\\ \displaystyle\quad+\mu\int_{\Omega^\mathrm{f}(t^{n+1})}\boldsymbol{\sigma}_\mathrm{f}(\boldsymbol{u}^{n+1},p^{n+1})\cdot\boldsymbol{\nabla}\boldsymbol{v}^\mathrm{f}=\frac{1}{\delta t}\int_{\Omega^\mathrm{f}(t^{n+1})}\rho_\mathrm{f}\boldsymbol{u}^n\boldsymbol{v}^\mathrm{f}+\int_{\Gamma^\mathrm{out}(t^{n+1})}\boldsymbol{g}_\mathrm{f}\boldsymbol{v}^\mathrm{f}\\ \displaystyle\int_{\Omega^\mathrm{f}(t^{n+1})}q^\mathrm{f}\operatorname{div}\boldsymbol{u}^{n+1}=0\end{cases}\tag{2.9}$$

for all $(\boldsymbol{v}^\mathrm{f},q^\mathrm{f})\in V^\mathrm{f}_0(t^{n+1})\times Q^\mathrm{f}(t^{n+1})$, with

$$\begin{aligned}V^\mathrm{f}(t)&=\left\{\boldsymbol{v}^\mathrm{f}|\,\boldsymbol{v}^\mathrm{f}\circ\boldsymbol{x}^\mathrm{f}_t\in H^1(\Omega^\mathrm{f}_0)^3\right\},\\ V^\mathrm{f}_0(t)&=\left\{\boldsymbol{v}^\mathrm{f}\in V^\mathrm{f}(t)|\,\boldsymbol{v}^\mathrm{f}\circ\boldsymbol{x}^\mathrm{f}_t=\boldsymbol{0}\text{ on }\Gamma_0\cup\Gamma^\mathrm{in}\right\},\\ Q^\mathrm{f}(t)&=\left\{q^\mathrm{f}|\,q^\mathrm{f}\circ\boldsymbol{x}^\mathrm{f}_t\in L^2(\Omega^\mathrm{f}_0)\right\},\end{aligned}$$

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

$$\left\{ \begin{aligned} \frac{2}{\delta t^2}\int_{\Omega^{\mathrm{s}}_0} \rho_{\mathrm{s}} \boldsymbol{d}^{\mathrm{s},n+1}\boldsymbol{v}^{\mathrm{s}} & - \frac{2}{\delta t^2}\int_{\Omega^{\mathrm{s}}_0} \rho_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s},n} + \delta t \boldsymbol{w}^{\mathrm{s},n})\boldsymbol{v}^{\mathrm{s}} + \int_{\Omega^{\mathrm{s}}_0} \boldsymbol{\sigma}_{\mathrm{s}}(\boldsymbol{d}^{\mathrm{s},n+1}) \cdot \boldsymbol{\nabla}\boldsymbol{v}^{\mathrm{s}} \\ & = \int_{\partial\Omega^{\mathrm{s}}_0 \backslash \Gamma_0} \boldsymbol{g}_{\mathrm{s}} \cdot \boldsymbol{v}^{\mathrm{s}} \\ \boldsymbol{w}^{\mathrm{s},n+1} & = \frac{2}{\delta t}(\boldsymbol{d}^{\mathrm{s},n+1} - \boldsymbol{d}^{\mathrm{s},n}) - \boldsymbol{w}^{\mathrm{s},n} \\ \boldsymbol{d}^{\mathrm{s},n+1} & = \lambda(t^{n+1}) \quad \text{on } \Gamma_0, \end{aligned} \right. \tag{2.10}$$

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 (2.9). 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 (2.10).

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 (2.10), 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

$$\text{find } \lambda \text{ such that } S_\mathrm{s}^{-1}(-S_\mathrm{f}(\lambda))=\lambda \text{ on } \Gamma_0. \tag{2.11}$$

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 (2.11)

$$\text{find } \lambda \text{ such that } S_\mathrm{s}^{-1}(-S_\mathrm{f}(\lambda))-\lambda=0 \text{ on } \Gamma_0, \tag{2.12}$$

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;

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

            M_fsi = fsi_solver_ptr(  new FSISolver( data_file, _oper ) );
            debugStream( 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

$$\lambda^{k+1}=\lambda^k+\omega^k f(\lambda^k), \tag{2.13}$$

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 (2.7),
- `M_lambdaDot` is the temporal derivative of `M_lambda`.

See table 2.4 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 (2.13) (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 2.4: 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 (2.13) will be solved on a lead structure processor (#0 processor within the structure intracommunicator).

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

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

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

debugStream( 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);

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

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

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

M_oper->setDataFromGetPot( data_file );


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

M_oper->setPreconditioner( precond );

M_oper->setup();

debugStream( 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 (2.13), 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 (2.13). 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;
debugStream( 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 (2.13).

Bibliography

Index