

FEEL MANUAL
A LIBRARY FOR
FINITE AND SPECTRAL ELEMENT METHODS IN
1D, 2D AND 3D

Version 0.91.1

Editor
Christophe PRUD'HOMME
`christophe.prudhomme@ujf-grenoble.fr`

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Contents

I	Tutorial	7
1	Building Feel++	9
	By Christophe Prud'homme, Baptiste Morin	
1.1	Building FEEL++	9
1.1.1	Getting the source via an archive	9
1.1.2	Getting the source via Subversion	9
1.1.3	Unix : dependencies	10
1.1.4	FEEL++ on Debian and Ubuntu	10
1.1.5	FEEL++ on Mac OS X	11
1.1.6	Compiling Feel	14
1.2	Programming environment	15
1.2.1	Boost C++ Libraries	15
1.2.2	FEEL++ Namepaces	16
2	Getting Started with Feel++	17
	By Christophe Prud'homme, Baptiste Morin	
2.1	Creating applications	17
2.1.1	Application and Options	17
2.1.2	Application, Logging, Archiving, Configuring	19
2.1.3	Initializing PETSc and Trilinos	21
2.2	Mesh Manipulation	21
2.2.1	Mesh definition	21
2.2.2	Mesh file format	22
2.2.3	Examples	22
2.2.4	Exporting meshes for post-processing	22
2.2.5	Iterating over the entities of a mesh	23
2.2.6	Load meshes	23
2.3	Computing integrals	24
2.3.1	Problem statement	24
2.3.2	Implementation	24
2.3.3	Quadrature	25
2.3.4	Complete example : application, mesh and integrals	26
2.3.5	Results	27
2.4	Function Spaces	28
2.4.1	Functions spaces definition	28
2.4.2	Using function space and functions	29

2.4.3	Results	30
2.5	Linear Algebra	30
2.5.1	Choosing a linear algebra backend	30
2.5.2	Solving	31
2.6	Variational Formulation	32
2.6.1	Principle	32
2.6.2	Standard formulation: the Laplacian case	32
2.6.3	Mixed formulation: the Stokes case	36
3	Feel++ Language Keywords	
	By Christophe Prud'homme	39
3.1	Keywords	39
3.2	Operators	42
3.2.1	Integrals	42
3.2.2	Projections	42
3.2.3	Meshes	43
II	Learning by Examples	45
4	Non-Linear examples	
	By Christophe Prud'homme	47
4.1	Solving nonlinear equations	47
4.1.1	A first nonlinear problem	48
4.1.2	Simplified combustion problem: Bratu	48
5	Heat sink	
	By Baptiste Morin, Christophe Prud'homme	49
5.1	Problem description	50
5.1.1	Domain	50
5.1.2	Inputs	50
5.2	Theory	52
5.2.1	Figure	52
5.2.2	Equations	54
5.2.3	Boundary conditions	54
5.2.4	Finite Element Method	55
5.3	Implementation	56
5.3.1	Application parameters	56
5.3.2	Surfaces	56
5.3.3	Equations	57
5.3.4	Outputs	58
5.4	Use cases	58
5.4.1	How to use it ?	58
5.4.2	Results	59
6	Natural convection in a heated tank	
	By Christophe Prud'homme	63
6.1	Description	63
6.2	Influence of parameters	64
6.3	Quantities of interest	64
6.3.1	Mean temperature	65
6.3.2	Flow rate	65
6.4	Implementation	67
6.5	Numerical Schemes	67

6.5.1	Stokes problem formulation and the pressure	67
6.5.2	The Stokes problem	67
6.5.3	Reformulation	67
6.5.4	Variational formulation	67
6.5.5	Implementation	68
6.5.6	Fix point iteration for Navier-Stokes	68
6.5.7	A Fix point coupling algorithm	69
6.5.8	A Newton coupling algorithm	70
7	2D Maxwell simulation in a diode	73
	By Thomas Strub, Philippe Helluy, Christophe Prud'homme	
7.1	Description	73
7.2	Variational formulation	74
7.3	Implementation	74
7.4	Numerical Results	74
8	Domain decomposition methods	75
	By Abdoulaye Samake, Vincent Chabannes, Christophe Prud'homme	
8.1	A Really Short Introduction	75
8.2	A 1D model	75
8.2.1	Schwartz algorithms	75
8.2.2	Variational formulations	76
8.3	A 2 domain overlapping Schwartz method in 2D and 3D	76
8.3.1	Schwartz algorithms	76
8.3.2	Variational formulations	76
8.3.3	Numerical results in 2D case	77
8.3.4	Numerical solutions in 2D case	78
8.4	Computing the eigenmodes of the Dirichlet to Neumann operator	78
8.4.1	Problem description and variational formulation	78
8.4.2	Numerical solutions	79
III	Programming with FEEL++	81
IV	Appendix	83
A	How to ?	85
A.1	Introduction	85
A.2	Meshes	85
A.2.1	What are the main execution options of a FEEL++ application ?	85
A.2.2	How to create a mesh?	85
A.2.3	What are the different parameters of the function domain() ?	86
A.2.4	How to loop on the degrees of freedom coordinates of a function ?	86
A.2.5	How to work with specific meshes ?	86
A.3	Language for Partial Differential Equations	87
A.3.1	What is the difference between using the "vf::project" function and solve a weak projection problem ?	87
A.3.2	How to do a quick L2 projection of an expression ?	87
A.3.3	How to compose FEEL++ operators ?	88

B	Random notes	89
B.1	Becoming a Feel++ developer	89
B.1.1	Interest	89
B.1.2	Creating RSA keys	89
B.1.3	Downloading the sources	89
B.2	Linear Algebra with PETSC	90
B.2.1	Using the Petsc Backend: recommended	90
B.2.2	List of solvers and preconditioners	90
B.2.3	What is going on in the solvers?	90
B.3	Weak Dirichlet boudary conditions	91
B.3.1	Basic idea	91
B.3.2	Laplacian	92
B.3.3	Convection-Diffusion	92
B.3.4	Stokes	93
B.4	Stabilisation techniques	94
B.4.1	Convection dominated flows	94
B.4.2	The CIP methods	94
B.5	Interpolation	95
C	GNU Free Documentation License	97
	Index	103

Part I

Tutorial

CHAPTER 1

Building Feel++

By Christophe Prud'homme, Baptiste Morin

Chapter ref: [**cha:tutorial-building**]

1.1 Building FEEL++

1.1.1 *Getting the source via an archive*

FEEL++ is distributed as a tarball once in a while. The tarballs are available at

<http://www.feelpp.org/files>

Download the latest tarball. Then follow the steps and replace x,y,z with the corresponding numbers

```
| tar xzf feel-x.y.z.tar.gz  
| cd feel-x.y.z
```

1.1.2 *Getting the source via Subversion*

In order to download the sources of FEEL++, you can download it directly from the source depository thanks to Subversion. To make it possible, you can download them anonymously or with an account in LJKForge that you have created. As an open-source project, we strongly suggest you to create an account and take part of the project with sharing your ideas, developments or suggests. If you're interested to participate and become a FEEL++ developer, please don't hesitate to see how it works in the appendix **B.1**. For now, if you want to get the sources without an account, open a command-line and type

```
| svn checkout svn://scm.forge.imag.fr/var/lib/gforge/chroot/scmrepos/svn/life/trunk/life/trunk feel
```

then you can go to the FEEL++ top directory with

```
| cd feel
```

You should obtain furthers directories such as :

```
applications/  # functional applications  
benchmarks/   # applications under test  
cmake/        # do not touch, used for compilation  
contrib/  
doc/         # tutorial and examples
```

```
examples/ # examples using Feel++
feel/     # Feel++ library
ports/    # used for Mac OS X installation
research/ # research projects using Feel++
testsuite/ # Feel++ unit tests testsuite
CMakeListe.txt # the file for cmake to build, do not modify
...
```

1.1.3 Unix : dependencies

In order to install FEEL++ on Unix systems (other than Mac OS X, in you have a Macintosh, please go to [1.1.5](#)), you have to install many dependencies before. Those libraries and programs are necessary for the compilation and installation of the FEEL++ librairies. This is the list of all the librairies you must have installed on your computer, and the *-dev packages for some of them.

Required packages:

- g++ (4.4 or 4.5, from now 4.6 is not yet completely working)
- MPI : openmpi (preferred) or mpich
- Boost (≥ 1.39)
- Petsc ($\geq 2.3.3$)
- Cmake (≥ 2.6)
- Gmsh¹
- Libxml2

Optional packages:

- Superlu
- Suitesparse(umfpack)
- Metis: scoth with the metis interface (preferred), metis (non-free)
- Trilinos ($\geq 8.0.8$)
- Google perftools
- Paraview², this is not stricly required to run FEEL++ programs but it is somehow necessary for visualisation
- Python (≥ 2.5) for the validation tools

Note that all these packages are available under Debian/GNU/Linux and Ubuntu. They should be available. Once you have installed those dependencies, you can jump to [1.1.6](#).

1.1.4 FEEL++ on Debian and Ubuntu

Debian

Debian is the platform of choice for FEEL++, it was developed mainly on it. The commands to install FEEL++ on Debian are

```
sudo apt-get update
sudo apt-get install feel++-apps libfeel-dev feel++-doc
```

¹Gmsh is a pre/post processing software for scientific computing available at <http://www.geuz.org/gmsh>

²Paraview is a few parallel scientific data visualisation platform, <http://www.paraview.org>

The interested user is encourage to follow the FEEL++ PTS page

- FEEL++ [Debian Packages Tracking System](#)

At the moment FEEL++ compiles and is available on the following Debian platforms:

- FEEL++ [Build results](#)

Ubuntu

FEEL++ was uploaded in the distribution Ubuntu-Natty (11.04) for the first time. The commands to install FEEL++ on Ubuntu are

```
| sudo apt-get update
| sudo apt-get install feel++-apps libfeel-dev feel++-doc
```

The interested user might want to follow the Ubuntu Launchpad FEEL++ page in order to know what is going on with FEEL++ on Ubuntu

- FEEL++ [Ubuntu Source Page for all Ubuntu versions](#)

1.1.5 FEEL++ on Mac OS X

FEEL++ is also working on Mac operating systems. The way to make it work is quite different.

Compilers

In order to FEEL++ and cmake work properly, you have to install differents compilers :

- Gcc

The first step is to install the latest version of Xcode. If your computer is recent, you can install it with your DVD that came with your machine (not the OS DVD, but the applications one). You don't have to install the complete Xcode (you can uncheck iOS SDK for example, it's not necessary here and requires a lot of memory). Xcode will provide your computer all basic tools to compile such as gcc 4.2. It's the first step, you'll see later how to easily install gcc 4.5 using MacPorts.

- Fortran

To build the Makefiles, cmake will need a Fortran compiler. To make it works, please go to [SourceForge.net](#) and download `gfortran-snwleo-intel-bin.tar.gz` which is the fortran compiler only (from now, don't download the complete install with gcc 4.6 because Feel needs gcc 4.5). To install it, go to the directory where you have downloaded the file and type in a command-line

```
| sudo tar -xvf gfortran-snwleo-intel-bin.tar -C /
```

MacPorts

Introduction MacPorts is an open-source community projet which aims to design an easy-to-use system for compiling, installing and upgrading open-source softwares on Mac OS X operating system. It is distributed under [BSD License](#) and facilitate the access to thousands of ports (softwares) without installing or compiling open-source softwares. MacPorts provides a single software tree which includes the latest stable releases of approximately 8050 ports targeting the current Mac OS X release (10.6 or 10.5). If you want more information, please visite their [website](#).

Installation To install the latest version of MacPorts, please go to [Installing MacPorts](#) page and follow the instructions. The simplest way is to download the *dmg* disk image corresponding to your version of Mac OS X. It is recommended that you install X11 (X Window System) which is normally used to display X11 applications.

If you have installed with the package installer (`MacPorts-1.x.x.dmg`) that means MacPorts will be installed in `/opt/local`. From now on we will suppose that macports has been installed in `/opt/local` which is the default MacPorts location. Note that from now on, all tools installed by MacPorts will be installed in `/opt/local/bin` or `/opt/local/sbin` for example (that's here you'll find `gcc4.5` once being installed).

Key commands In your command-line, the software MacPorts is called by the command `port`. Here is a list of key commands for using MacPorts, if you want more informations please go to [MacPorts Commands](#).

- `sudo port -v selfupdate` This action should be used regularly to update the local tree with the global MacPorts ports. The option `-v` enables verbose which generates verbose messages.
- `port info flowd` This action is used to get information about a port (description, license, maintainer, etc.)
- `sudo port install mypackage` This action install the port mypackage
- `sudo port uninstall mypackage` This action uninstall the port mypackage
- `port installed` This action displays all ports installed and their versions, variants and activation status. You can also use the `-v` option to also display the platform and CPU architecture(s) for which the ports were built, and any variants which were explicitly negated.
- `sudo port upgrade mypackage` This action upgrades installed ports and their dependencies when a Portfile in the repository has been updated. To avoid the upgrade of a port's dependencies, use the option `-n`.

Portfile A Portfile is a TCL script which usually contains simple keyword values and TCL expressions. Each package/port has a corresponding Portfile but it's only a part of a port description. FEEL++ provides some mandatory Portfiles for its compilation which are either not available in MacPorts or are buggy but FEEL++ also provides some Portfiles which are already available in MacPorts such as `gms` or `petsc`. They usually provide either some fixes to ensure FEEL++ works properly or new version not yet available in MacPorts. These Portfiles are installed in `ports/macosx/macports`.

MacPorts and FEEL++

To be able to install FEEL++, add the following line in `/opt/local/etc/macports/source.conf` at the top of the file before any other sources :

```
file:///<path to feel top directory>/ports/macosx/macports
```

Once it's done, type in a command-line :

```
cd <your path to feel top directory>/ports/macosx/macports
portindex -f
```

You should have an output like this :

```
Reading port index in <your path to feel top directory>/ports/macosx/macports
Adding port science/feel++
Adding port science/gmsh
Adding port science/petsc

Total number of ports parsed: 3
Ports successfully parsed: 3
Ports failed: 0
Up-to-date ports skipped: 0
```

You are now able to type

```
| sudo port install feel++
```

It might take some time (possibly an entire day) to compile all the requirements for FEEL++ to compile properly. If you have several cores on your MacBook Pro, iMac or MacBook we suggest that you configure macports to use all or some of them. To do that uncomment the following line in the file `/opt/local/etc/macports/macports.conf`

```
buildmakejobs    0 # all the cores
```

At the end of the `sudo port install feel++`, you have all dependencies installed which is a good point. To build all the Makefile, `cmake` is automatically launched but can have further libraries not found. Some are important, some are not. OpenMP is one of the necessary compiler to have and is not automatically installed on 64bits systems. We can install it now thanks to `gcc4.5` that has been installed because it takes part of the feel dependencies. To install it, please go to [Omni download](#) and download the latest stable release. Then type

```
| tar -xvf Omni-x.yz.tar.gz
| cd Omni-x.yz
| ./configure --with-cc=/opt/local/bin/gcc-mp-4.5
| make
```

Then, you have to become a super-user so type

```
| sudo su
```

Enter your password, then to complete the installation

```
| make install
```

After the install is complete, you can delete the sources in your download directory because `make install` has built it in an appropriate directory. To make it work properly, always check that `cmake` is using `gcc 4.5`. If `cmake` doesn't recognize openMP, enter the configuration mode using `ccmake` and enter `-fopenmp` in the box `OpenMP_CXX_FLAGS`

Snow Leopard & slepc/petsc

We have heard about issues with `petsc` and `slepc` with some new MacBook Pro with Snow Leopard while they are being installed with the `sudo port install feel++`. If it's the case, that probably means there is an issue with `atlas`. If `atlas` is already installed, you have to uninstall it (be careful with dependencies, they also have to be uninstalled). Once it's done, you should do

```
| cd <path to feel top directory>/ports/macosx/macports
| portindex -f
```

then type in the exact same order :

```
sudo port uninstall slepc
sudo port uninstall petsc
sudo port install -d petsc
sudo port install slepc
```

and type once again

```
sudo port install feel++
```

In that order, slepc and petsc will be installed before atlas, and feel will be properly installed.

Missing ports

cmake can build Makefiles even if some packages are missing (latex2html, VTK ...). It's not necessary to install them but you can complete the installation with MacPorts, cmake will find them by itself once they have been installed.

1.1.6 Compiling Feel

Feel build system uses cmake³ as its build system. Check that cmake is using gcc4.5 or 4.4 as C++ compiler (you can use the option `CMAKE_CXX_COMPILER=<path>/g++-4.5` where the path depends on your OS, it's probably `/usr/bin` or `/opt/local/bin` but you can also change it with the command `ccmake` and press `t` for advanced options). It's important, as cmake did not produce any Makefile, a CMakeCache.txt won't be created so you'll have to check each time that gcc 4.5 is the C++ compiler to be sure the build will be correct.

FEEL++, using cmake, can be built either in source and out of source and different build type:

- minsizerel : minimal size release
- release release
- debug : debug
- none(default)

CMake Out Source Build (preferred) The best way is to have a directory (FEEL for example) in which you have :

```
feel/
```

where feel is the top directory where the source have been downloaded. Placed in FEEL, you can create the build directory (feel.opt for example) and launch cmake with :

```
mkdir feel.opt
cd feel.opt
cmake <directory where the feel source are>
# e.g cmake ../feel if feel.opt is at the same
# directory level as feel
```

you can customize the build type:

```
# Choose g++ release
cmake -CMAKE_CXX_COMPILER=/usr/bin/g++-4.5
# Debug build type (-g...)
cmake -D CMAKE_BUILD_TYPE=Debug
# Release build type (-O3...)
cmake -D CMAKE_BUILD_TYPE=Release
...
```

³<http://www.cmake.org>

Once Cmake has made its work, you are now able to compile the library with

```
| make
```

Important : from now, all commands should be type in `feel.opt` or its subdirectories.

CMake In Source Build Be carefull, this is not advised and if you try this way, `cmake` won't let you do. If you really want to, you will have to modify the top `CMakeLists.txt`. You should consider out source builds by checking the next paragraph.

Enter the source tree and type

```
| cmake .
| make
```

To customize or change some build setting one can use the `cmake` curse interface `ccmake`

```
| ccmake . # configure and generate
| make
```

Compiling the Feel tutorial

The manual (which includes the tutorial) is edited with \LaTeX so you need to have installed the \LaTeX distribution on your computer. \LaTeX is a high-quality typesetting system, it includes features designed for the production of technical and scientific documentation. There are several ways to make it work, for example you can go on [MacTeX website](#) and follow the instructions to install the distribution. If the command `make check` in `feel.opt/` has been run before, the tutorial should be already compiled and ready. The steps are as follows to build the Feel tutorial

```
| cd feel.opt/doc/manual
| make pdf
```

The directory `doc/manual` contains all examples used in the tutorial. You will see how it works in the following parts.

1.2 Programming environment

We present here a quick list of all namespaces and librairies proposed by `FEEL++`, you'll see in the tutorial which starts at section ?? how you can use them.

1.2.1 Boost C++ Libraries

`FEEL++` depends on a number of libraries, some are required some are optional. Among the required libraries, The Boost C++ libraries play a very important role as they drive or shape the design of `FEEL++`. `FEEL++` uses in particular the following Boost libraries:

- `Boost.Parameter` : use to provide powerful interfaces to `FEEL++` and third party library such as
 - `PETSc` for the linear, nonlinear solvers
 - `SLEPc` for the eigenvalue solvers
 - `GMSH` for mesh generation
- `Boost.MPL` - meta programming library : use for type computations
- `Boost.Fusion` - linking meta-runtime programming: use for type computations used at runtime
- `Boost.Program_Options` - command-line options library : provides the command line options for the `FEEL++` applications as well as configuration files
- `Boost.Test` - Unit testing framework ; used by the `FEEL++` testsuite

1.2.2 FEEL++ *Namepaces*

- `Feel`
- `Feel::po`
- `Feel::mpl`
- `Feel::ublas`
- `Feel::math`
- `Feel::fem`
- `Feel::vf`

CHAPTER 2

Getting Started with Feel++

By Christophe Prud'homme, Baptiste Morin

Chapter ref: [cha:getting-started]

2.1 Creating applications

myapp.cpp

2.1.1 Application and Options

As a FEEL++ user, the first step in order to use FEEL++ is to create an application. Before writing anything, you have to include the `Application` header and the header which handles the internal FEEL++ options. Note that FEEL++ uses the `boost::program_options1` (po) library from Boost to handle its command line options.

```
#include <feel/options.hpp>
#include <feel/feelcore/feel.hpp>
#include <feel/feelcore/application.hpp>
#include <feel/feelalg/backend.hpp>
```

Next to ease the programming and reading, we use the `using` C++ directive to bring the namespace `Feel` to the current namespace

```
using namespace Feel;
```

Then we define the command line options that the applications will provide. Note that on the `return` line, we incorporate the options defined internally in FEEL++.

```
inline
po::options_description
makeOptions()
{
    po::options_description myappoptions("MyApp options");
    myappoptions.add_options()
        ("dt", po::value<double>()->default_value( 1 ), "time step value")
        ;

    // return the options myappoptions and the feel_options defined
    // internally by Feel
    return myappoptions.add( feel_options() ).add( backend_options( "myapp" ) );
}
```

¹http://www.boost.org/doc/html/program_options.html

In the example, we provide the options `dt` which takes an argument, a **double** and its default value is 1 if the options is not set by the command line. Then we describe the application by defining a class `AboutData` which will be typically used by the `help` command line options to describe the application

```
inline
AboutData
makeAbout ()
{
    AboutData about( "myapp" ,
                    "myapp" ,
                    "0.1",
                    "my first Feel application",
                    AboutData::License_GPL,
                    "Copyright (c) 2008 Universite Joseph Fourier");

    about.addAuthor("Christophe Prud'homme",
                  "developer",
                  "christophe.prudhomme@ujf-grenoble.fr", "");

    return about;
}
```

Now we turn to the class `MyApp` itself: it derives from `Feel::Application`. This class provides two constructors : one with only description and one with additionnal parameters which enables to add options `argc` and `argv`. This class `MyApp` has to redefine the `run()` method. It is defined as a pure virtual function in `Application`.

```
class MyApp: public Application
{
public:

    /**
     * constructor only about data and no options description
     */
    MyApp( int argc, char** argv, AboutData const& );

    /**
     * constructor about data and options description
     */
    MyApp( int argc, char** argv,
          AboutData const&,
          po::options_description const& );

    /**
     * This function is responsible for the actual work done by MyApp.
     */
    void run();
};
```

The implementation of the constructors is usually simple, we pass the arguments to the super class `Application` that will analyze them and subsequently provide them with a `Feel::po::variable_map` data structure which operates like a map. Have a look at the document `boost::program_options`² for further details. Here our two constructors do nothing (because `{}`).

```
MyApp::MyApp(int argc, char** argv,
             AboutData const& ad )
:
    Application( argc, argv, ad )
{}
MyApp::MyApp(int argc, char** argv,
             AboutData const& ad,
             po::options_description const& od )
:
    Application( argc, argv, ad, od )
{}
```

The `run()` member function holds the application commands/statements. Here we provide the smallest code unit: we print the description of the application if the `--help` command line options is set.

²http://www.boost.org/doc/html/program_options.html

```
void MyApp::run()
{
    /**
     * print the help if --help is passed as an argument
     */
    /** \code */
    if ( this->vm().count( "help" ) )
    {
        std::cout << this->optionsDescription() << "\n";
        return;
    }
    /** \endcode */
}
```

Finally the `main()` function can be implemented. We pass the results of the `makeAbout()` and `makeOptions()` to the constructor of `MyApp` as well as `argc` and `argv`. Then we call the `run()` member function to execute the application.

```
int main( int argc, char** argv )
{
    Feel::Environment env( argc, argv );

    /**
     * instantiate a MyApp class
     */
    /** \code */
    MyApp app( argc, argv, makeAbout(), makeOptions() );
    /** \endcode */

    /**
     * run the application
     */
    /** \code */
    app.run();
    /** \endcode */
}
```

Now, you can check if your first FEEL++ application is working. To compile `myapp`, type in a command-line :

```
cd feel.opt/doc/manual
make feel_doc_myapp
```

You are now able to execute it (obviously here `./feel_doc_myapp` won't produce anything but you can try it to check the execution is ok)

```
> ./feel_doc_myapp --help
myapp: my first Feel application
Allowed options:

MyApp options:
  --dt arg (=1)                                time step value

> ./feel_doc_myapp --authors
myapp: my first Feel application
      Author Name      Task      Email Address
-----
Christophe Prud'homme  developer  christophe.prudhomme@ujf-grenoble.fr
```

2.1.2 Application, Logging, Archiving, Configuring

FEEL++ provides some basic logging and archiving support: using the `changeRepository` member functions of the class `Application`, the logfile and results of the application will be stored in a subdirectory of `~/feel`. For example the following code

```
this->changeRepository( boost::format( "doc/tutorial/%1%" )
                       % this->about().appName() );
```

will create the directory `~/feel/doc/tutorial/` and will store the logfile and any files created after calling `changeRepository`. Refer to the documentation of `Boost::format` of further details about the

arguments to be passed to `changeRepository`. The logfile is named `~/feel/doc/tutorial/myapp-1.0`. The name of the logfile is built using the application name, here `myapp`, the number of processes, here `1` and the id of the current process, here `0`.

Configuring Each application can be configured via the command line but also using a `.cfg` file. If they exist they may have been installed on your system along with FEEL++ or you may create your own configuration files. FEEL++ provides a way to look for them and parse them. The `.cfg` file is searched in the following order

1. look in the current directory
2. look in the directory `$HOME/feel/config/`
3. look in the directory `$INSTALL_PREFIX/share/feel/config/`, e.g. in Debian `/usr/share/feel/config/`

The name of the file can be constructed in two ways `<appname>.cfg` and `feel_<appname>.cfg` where `<appname>` is the string given in the `AboutData` data structure passed to the construction of the `Application` class. Here are two examples of the logfiles in the case that there was not `myapp.cfg` created.

```
> ./feel_doc_myapp
> cat ~/feel/doc/tutorial/myapp/myapp-1.0
myapp-1.0 is opened for debug
the value of dt is 1
the value of myapp-solver-type is gmres
the value of myapp-pc-type is lu

> ./feel_doc_myapp --dt=0.2
> cat ~/feel/doc/tutorial/myapp/myapp-1.0
myapp-1.0 is opened for debug
the value of dt is 0.2
the value of myapp-solver-type is gmres
the value of myapp-pc-type is lu
```

If you want to create a configured file, you have to create `myapp.cfg` such as, for example :

```
dt=1e-5
myapp-solver-type=cg
myapp-pc-type=ilu
```

This configured file will be parsed automatically before being executed. In that way you won't have to enter each time values you want to fix.

MPI Application

`mympiapp.cpp`

FEEL++ relies on MPI for parallel computations and the class `Application` initialises the MPI environment, for that, you should go to the appropriate repertory and call `ccmake ..`. Then turn on the `FEELPP_ENABLE_MPI_MODE`. To launch a parallel computation for your application, you have to call the application `mpirun` such as

```
mpirun -np 2 mympiapp
```

```
> cat ~/feel/mympiapp/mympiapp-2.0
mympiapp-2.0 is opened for debug
[Area 0] the value of dt is 1
[Area 0] we are on processor eta
[Area 0] this is process number 0 out of 2
> cat ~/feel/mympiapp/mympiapp-2.1
mympiapp-2.1 is opened for debug
[Area 0] the value of dt is 1
[Area 0] we are on processor eta
[Area 0] this is process number 1 out of 2

> mpirun -np 2 mympiapp --dt=0.01
> cat ~/feel/mympiapp/mympiapp-2.0
```

```

mympiapp-2.0 is opened for debug
[Area 0] the value of dt is 0.01
[Area 0] we are on processor eta
[Area 0] this is process number 0 out of 2
> cat ~/feel/mympiapp/mympiapp-2.1
mympiapp-2.1 is opened for debug
[Area 0] the value of dt is 0.01
[Area 0] we are on processor eta
[Area 0] this is process number 1 out of 2

```

2.1.3 Initializing PETSc and Trilinos

PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for parallelism.

The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems.

FEEL++ supports the PETSc and Trilinos framework, the class `Application` takes care of initialize the associated environments.

2.2 Mesh Manipulation

`mymesh.cpp`

In this section, we present some of the mesh definition and manipulation tools provided by FEEL++.

2.2.1 Mesh definition

We look at the definition of a mesh data structure. First, we define the type of geometric entities that we shall use to form our mesh. FEEL++ supports several geometric entities

- simplices: segment, triangle, tetrahedron
- tensorized entities: segment, quadrangle, hexahedron

To create a mesh, we use the following keywords `Simplex<Dim, Order, RealDim>` and `SimplexProduct <Dim, Order, RealDim>`. They have the same template arguments:

- `Dim`: the topological dimension of the entity (optional, default value = 1).
- `Order`: the order of the entity (usually 1, higher order in development).
- `RealDim`: the dimension of the real space (optional, default value = `Dim`).

```

typedef Simplex<Dim> convex_type;
//typedef Hypercube<Dim, 1, Dim> convex_type;

```

Now, you are able to define the mesh type, `Mesh<Entity>` by passing as argument the type of entity it is formed with (at the moment hybrid meshes are not supported).

```

typedef Mesh<convex_type > mesh_type;
typedef boost::shared_ptr<mesh_type> mesh_ptrtype;

```

`boost::shared_ptr` allows to manipulate a pointer on a mesh. It is customary, and usually a very good practice, to define the `boost::shared_ptr<>` counterpart which is used actually in practice.

2.2.2 Mesh file format

The next step is to read some mesh files. FEEL++ supports essentially the Gmsh mesh file format. It provides also some classes to manipulate Gmsh `.geo` files and generate `.msh` files. To begin, we use some helper classes and functions to generate a `.geo` file.

- `domain` is a function which allows to generate a `.geo` string description of simple domains such as simplex and hypercube.
- `createGMSHMesh` is a function which allows to generate a mesh file `.msh` automatically from a description file (`.geo` for example) by using the `_desc` parameter and store the generated mesh into the `_mesh` parameter allocated when calling the function.
- `GmshSimplexDomain` is a class which will enable you to create simplex domains (e.g segment, triangle or tetrahedron). It allows to modify
 - the characteristic size of the mesh (default $h = 0.1$)
 - the domain vertices (default is $(-1, -1, -1), (1, -1, -1), (-1, 1, -1), (-1, -1, 1)$)
- `GmshHypercubeDomain` is a class which will enable you to create hypercube domains (e.g cube) in 1,2 or 3D. It allows to modify
 - the characteristic size of the mesh (default $h = 0.1$)
 - the domain (default is the cube $[0; 1] \times [0; 1] \times [0; 1]$)

2.2.3 Examples

Here is an example of how to get a simplex or hypercube geometry :

```
auto mesh = createGMSHMesh( _mesh=new mesh_type,
                             _update=MESH_CHECK|MESH_UPDATE_FACES|MESH_UPDATE_EDGES|MESH_UPDATE_VERTICES,
                             _desc=domain( _name=(boost::format( "%1%-%2%" ) % shape % Dim),
                                             _shape=shape,
                                             _dim=Dim,
                                             _h=X[0] ),
                             _partitions=this->comm().size());
```

The parameter `_h` in the function `domain` allows to change the mesh characteristic size to `M_meshsize` which is given for example on the command-line using the Application framework, please refer to [2.1](#) for more details. You can see the meshes, by executing with the followings ordering :

```
| ./feel_doc_mymesh --shape=simplex
| ./feel_doc_mymesh --shape=hypercube
```

we generate some of the following graphics (which are located in `~/feel/doc/tutorial/mymesh/hypercube-x/h_y.z` or `~/feel/doc/tutorial/mymesh/simplex-x/h_y.z`)

To admire the meshes you've generated, you should go to `~/feel/doc/tutorial/mymesh/hypercube` (or `simplex`) and find out the `.msh` files. Last step is to launch gmsh with

```
| gmsh yourfile.msh
```

2.2.4 Exporting meshes for post-processing

We can export the mesh to two postprocessing formats supported at the moment :

- **Enight** (sos and case) which is supported by the software Enight and Paraview;
- **Gmsh** which is post-processing format of Gmsh.

.

Figure 2.1: Line in 1D

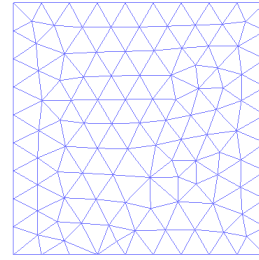


Figure 2.2: Cube in 2D

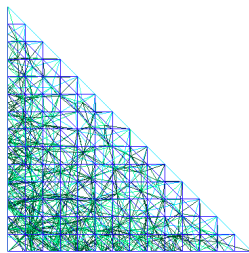


Figure 2.3: Tetrahedron

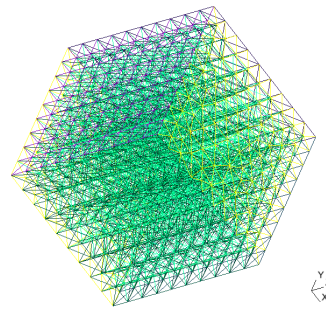


Figure 2.4: Unit cube in 3D

We define the exporter data structure type as follows :

```
/* export */
typedef Exporter<mesh_type> export_type;
typedef boost::shared_ptr<export_type> export_ptrtype;
```

By default, the export format is `ensight`. We can use the code `Exporter::setMesh()` and `Exporter::save()` to save the mesh to the format given to the command line `-exporter= <format>`

```
exporter->step(0)->setMesh( mesh );
exporter->step(0)->addRegions();
exporter->save();
```

To take into account the exporting format, you have to type thoses syntaxes

```
./feel_doc_mymesh --exporter= gmsh
./feel_doc_mymesh --exporter= paraview
```

2.2.5 Iterating over the entities of a mesh

FEEL++ mesh data structures provides powerful iterators that allows to walk though the mesh in various ways: iterate over element, faces , points, marked³ elements, marked faces, ...

2.2.6 Load meshes

FEEL++ supports gmsh meshes but not only, you can also work with medit (`.mesh`) or STL (`.stl`) files format. The full explanation is available in the HOW TO section you can find at [A.2.5](#). This is implemented in `loadmesh.cpp` which is a simple application which loads a mesh and calculate its surface and volume.

`loadmesh.cpp`

³associated to an integer flag denoting a region, material, processor

2.3 Computing integrals

myintegrals.cpp

2.3.1 Problem statement

If you have followed this tutorial in the order, you are now able to create a simple application and generate meshes. We are now interested in computing integrals on the meshes we have generated. Let's consider this mesh (no matter which one) on a domain Ω and parts of the domain, i.e. subregions and (parts of) boundary. In this tutorial the domain can be either

$$\Omega = [0, 1]^d \subset \mathbb{R}^d$$

or

$$\Omega = \{\mathbf{x} \in \mathbb{R}^d | x_i \geq 0, \sum_{i=1}^d x_i \leq 1\}$$

where $d = 1, 2$ or 3 and $\mathbf{x} = (x_1, \dots, x_d)$.

These domains are plotted in the section [2.2](#) (If it's not yet done, compile all examples which are used in this tutorial, go to `~/FEEL/feel.opt/doc/manual/` and type `make` and execute the application `feel_doc_mymesh`). The execution will result in several meshes as it is explained in [2.2.3](#). Here are the integrals we want to compute :

- $\int_{\Omega} 1$: the measure of the domain
- $\int_{\partial\Omega} 1$: the measure of the surface of the domain (Dim>1)
- $\int_{\Omega} x^2 + y^2 + z^2$: the integral over Ω of $(x, y, z) \mapsto x^2 + y^2 + z^2$ with the convention that $y = z = 0$ in 1D and $z = 0$ in 2D.
- $\int_{\Omega} \sin(x^2 + y^2 + z^2)$: the integral over Ω of $(x, y, z) \mapsto \sin(x^2 + y^2 + z^2)$ with the same convention as above.

2.3.2 Implementation

To compute an integral, we use the following function

```
integrate( <domain>, <expression under the integral> ).evaluate(<location>)
```

Domain

You have to indicate the domain on which we want to integrate. It consists in a pair of iterators over the elements owned by the current processor (the mesh is shared between the processors).

- To compute the integral over the region of Ω current processor, use

```
elements(mesh)
```

- To compute the integral on the boundary faces of the domain Ω , use

```
boundaryfaces(mesh)
```


Expression under the integrals

The language provided by FEEL++ (called the Finite Element Embedded Language) brings the keyword $Px()$, $Py()$ and $Pz()$ to denote the x , y and z coordinates.

The expression $x^2 + y^2 + z^2$ under the third integral should be written such as :

```
Px()*Px() + Py()*Py() + Pz()*Pz()
```

The constants are indicated by the function `constant()` (for example, we have to indicate `constant(1.0)` for the first integral).

Examples

First integral : domain area

To compute the first integral (the domain area) and to have the local contributions only, we have to pass 'false' to evaluate as follows :

```
double local_domain_area = integrate( _range=elements(mesh),
                                     _expr=constant(1.0)).evaluate(false)(0,0);
```

This compute the integral only on the region of the domain owned by the current processor (`local_domain_area`).

To obtain the total area, `integrate(...).evaluate()` computes the global integral in parallel which induces communications (`all_reduce`) to compute the sum of the local contributions

```
double global_domain_area= integrate( _range=elements(mesh),
                                     _expr=constant(1.0)).evaluate()(0,0);
```

Finally, we print to the log file the result of the local and global integral calculation.

```
Log() << "int_Omega 1 = " << global_domain_area
      << "[ " << local_domain_area << " ]\n";
```

Second integral : domain perimeter

The difference with the domain area computation resides in the elements with are iterating on: here we are iterating on the boundary faces of the domain to compute the integral using `boundaryfaces(mesh)` to provide the pairs of iterators. But the stages are the same as previously

```
double local_boundary_length = integrate( boundaryfaces(mesh),
                                         constant(1.0)).evaluate(false)(0,0);
double global_boundary_length = integrate( boundaryfaces(mesh),
                                         constant(1.0)).evaluate()(0,0);
Log() << "int_BoundaryOmega (1)= " << global_boundary_length
      << "[ " << local_boundary_length << " ]\n";
```

We can apply the same method to compute the third, and the last integral.

2.3.3 Quadrature

Feel computes automatically the quadrature associated to the expression under the integral. If the expression is polynomial then the quadrature is exact. If the expression is not polynomial, then each non-polynomial term in the expression is considered as a polynomial of degree 2 by default.

Here is how the 4-th integral can be computed letting Feel decide about the quadrature

```
double global_intsin = integrate( elements(mesh),
                                sin( Px()*Px() + Py()*Py() + Pz()*Pz() )
                                ).evaluate()(0,0);
double local_intsin = integrate( elements(mesh),
```

```
sin( Px()*Px() + Py()*Py() + Pz()*Pz() ) ).evaluate(false)
```

An alternative is to set yourself the quadrature by passing a new argument to the integrate function: `_Q<Order>()` where Order is the maximum polynomial order that the quadrature should integrate exactly.

```
double local_intsin2 = integrate( elements(mesh),
                                sin( Px()*Px() + Py()*Py() + Pz()*Pz() ),
                                _Q<2>()
                              ).evaluate(false)(0,0);
double global_intsin2 = integrate( elements(mesh),
                                  sin( Px()*Px() + Py()*Py() + Pz()*Pz() ),
                                  _Q<2>() ).evaluate() (0,0);
```

2.3.4 Complete example : application, mesh and integrals

Here, we'll see how to build an application which create a mesh, and compute some integrals on it.

Application building

As we can see in the creating applications section (2.1), we can add a description and some options to our new application. Here, we have created two functions `makeAbout()` and `makeOptions()` which create respectively the description and the list of options. `makeOptions()` creates a list of options (`myintegraloptions`), and adds two options :

- `hsize` which corresponds to the mesh size (default = 0.2)
- `shape` which corresponds to the shape of the domain (default = *hypercube*)

```
inline
po::options_description
makeOptions()
{
    po::options_description myintegraloptions("MyIntegrals options");
    myintegraloptions.add_options()
        ("hsize", po::value<double>()->default_value( 0.2 ), "mesh size")
        ("shape", Feel::po::value<std::string>()->default_value( "hypercube" ), "shape of the
    ;
    return myintegraloptions.add( Feel::feel_options() );
}
```

`makeAbout()` gives the description of the application :

```
inline
AboutData
makeAbout()
{
    AboutData about( "myintegrals" ,
                    "myintegrals" ,
                    "0.3",
                    "nD(n=1,2,3) MyIntegrals on simplices or simplex products",
                    Feel::AboutData::License_GPL,
                    "Copyright (c) 2008-2010 Universite Joseph Fourier");

    about.addAuthor("Christophe Prud'homme", "developer", "christophe.prudhomme@ujf-grenoble.fr");
    return about;
}
```

To get further information about the application, see creating applications section 2.1.

Class definition

Once the description and options have been defined, we create the class `Myintegrals`, with the constructor

```
MyIntegrals( po::variables_map const& vm, AboutData const& about )
```

which takes `vm` (a map which associates the available options with their default value), and `about` (containing our application's description).

The `run()` member function is redefined twice in this class :

```
void run();

void run( const double* X, unsigned long P, double* Y, unsigned long N );
```

These two versions are linked, because the first uses the second. Indeed, `run()` stores the parameters we have chosen with the options (mesh size, shape, ...), and uses them in the second version (`X` corresponds to these parameters : `X[0]=mesh_size` and `X[1]=mesh_shape`).

The `run()` member function

In this function, we create a subdirectory of `feel` in which there will be the results of our application, and logfiles containing :

- the application's name
- the mesh size
- the mesh shape

```
if ( !this->vm().count( "nochdir" ) )
Environment::changeRepository( boost::format( "doc/tutorial/%1%/h_%3%/" )
                               this->about().appName()
                               shape
                               meshSize );
```

It is also in `run()` that we create the mesh before to compute the integrals :

```
mesh_ptrtype mesh = createGMSHMesh( _mesh=new mesh_type,
                                     _update=MESH_CHECK|MESH_UPDATE_FACES|MESH_UPDATE_EDGES,
                                     _desc=domain( _name= (boost::format( "%1%-%2%" ) % sha
                                                         _shape=shape,
                                                         _order=1,
                                                         _dim=Dim,
                                                         _h=X[0] ),
                                                         _partitions=this->comm().size());
```

Now that we have a computational mesh, we can compute the integrals as we have seen before :

```
double local_domain_area = integrate( elements(mesh),
                                     constant(1.0)).evaluate()(0,0);

double global_domain_area=local_domain_area;
if ( this->comm().size() > 1 )
    mpi::all_reduce( this->comm(),
                    local_domain_area,
                    global_domain_area,
                    std::plus<double>() );

Log() << "int_Omega 1 = " << global_domain_area
      << "[ " << local_domain_area << " ]\n";
```

2.3.5 Results

After compiling `feel_doc_myintegrals` with the defaults options

```
--hsize arg (=0.20000000000000001)    mesh size
--shape arg (=hypercube)              shape of the domain
```

we obtain the values of each integrals, in each dimension (1, 2 and 3) :

```
-----
Execute MyIntegrals<1>
int_Omega 1 = 1[ 1 ]
int_Omega (x^2+y^2+z^2) = 0.333333[ 0.333333 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 4 max exact integration]= 0.310268[ 0.310268 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 2 max exact integration] = 0.310281[ 0.310281 ]
-----
Execute MyIntegrals<2>
int_Omega 1 = 1[ 1 ]
int_BoundaryOmega (1)= 4[ 4 ]
int_Omega (x^2+y^2+z^2) = 0.666667[ 0.666667 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 4 max exact integration]= 0.56129[ 0.56129 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 2 max exact integration] = 0.561299[ 0.561299 ]
-----
Execute MyIntegrals<3>
int_Omega 1 = 1[ 1 ]
int_BoundaryOmega (1)= 6[ 6 ]
int_Omega (x^2+y^2+z^2) = 1[ 1 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 4 max exact integration]= 0.731683[ 0.731683 ]
int_Omega (sin(x^2+y^2+z^2)) [with order 2 max exact integration] = 0.731693[ 0.731693 ]
-----
```

2.4 Function Spaces

myfunctionspace.cpp

2.4.1 Functions spaces definition

In order to resolve partial derivative equations, we have to define the function space on which we work. We can define a new type `space_type` which corresponds to our new functions space

```
typedef FunctionSpace<mesh_type, basis_type> space_type;
```

To define this function space, we have to define :

- `mesh_type` the mesh on which we work (see 2.2)
- `basis_type` the base of the function space (see 2.4.1)

Remark : We can use the librairie Boost.smartptr of Boost, which has a reference counter (in the `shared_ptr` class). It permits to manage the memory (if we have built an object and that there are no more pointers on it, it is automatically destructed).

Base of the function space

To define the base of functions we want to use, we have to indicate :

- The type of the polynomials (\mathbb{P}) we want to use
 - Lagrange (more used than the others because more customizable)
 - Legendre (only for hypercube)
 - Dubiner (only for simplex)
 - Crouzeix-Raviart
 - Raviart-Thomas
- The order of these polynomials (an integer in \mathbb{N}^*)
- The dimension of $\text{Im}(\mathbb{P})$ (`Scalar` if $\text{Im}(\mathbb{P}) = \mathbb{R}$, `Vectorial` if the dimension is up to 1)
- The continuity of polynomials at the interface (`Continuous` or `Discontinuous`)

After we have chosen this parameters, we can create the new function base (new type `basis_type`) :

```
typedef bases<Lagrange<0,Scalar,Discontinuous> > basis_type;
```

Here, we build a base with Lagrange polynomials, with order 0. The result of these polynomials is a scalar, and we authorized them to be discontinue at the interface between two elements. In our example, `p0_space_type` is the function space that holds piecewise constant \mathbb{P}_0 functions :

```
typedef FunctionSpace<mesh_type, bases<Lagrange<0, Scalar, Discontinuous> > >
p0_space_type;
```

Instantiation of the function space

We want now to instantiate the function space X_h of type `space_type` (that we created previously).

First, we build the mesh on which we want to work (of type `mesh_ptrtype`)

```
//! create the mesh
mesh_ptrtype mesh =
    createGMSHMesh( _mesh=new mesh_type,
                    //_update=MESH_CHECK|MESH_UPDATE_FACES|MESH_UPDATE_EDGES,
                    _update=MESH_CHECK|MESH_UPDATE_FACES|MESH_UPDATE_EDGES,
                    _desc=domain( _name= (boost::format( "%1%-%2%-%3%" ) % shape % Dim % 0
                    _shape=shape,
                    _dim=Dim,
                    _order=Order,
                    _h=X[0] ),
                    _partitions=this->comm().size());
```

Then we instantiate X_h using `space_type::New()` static member function and obtain a `space_ptrtype`.

```
space_ptrtype Xh = space_type::New( mesh );
```

We are now able to instantiate elements on X_h . There are two ways to proceed, using auto keyword allowing to infer automatically the type of X_h elements or knowing the actual type of the elements

```
// an element of the function space X_h
auto u = Xh->element( "u" );
// another element of the function space X_h
element_type v( Xh, "v" );
auto w = Xh->element( "w" );
```

2.4.2 Using function space and functions

Projection

First, we define some mathematical expression/functions

$$\begin{aligned} g(x, y, z) &= \sin(\pi x/2) \cos(\pi y/2) \cos(\pi * z/2) \\ f(x, y, z) &= (1 - x^2) (1 - y^2) (1 - z^2) (x^2 + y^2 + z^2)^{\alpha/2.0}, \quad \alpha = 3 \end{aligned}$$

In our example, these functions correspond to

```
auto g = sin(2*pi*Px())*cos(2*pi*Py())*cos(2*pi*Pz());
auto f = (1-Px()*Px())*(1-Py()*Py())*(1-Pz()*Pz())*pow(trans(vf::P())*vf::P(), (alpha/2.0));
```

(they are implemented using the new `auto` C++ keyword to infer the type of expression automatically).

Then we build the interpolant (Lagrange interpolant in this case since we chose Lagrange basis function), by calling the `vf::project()` function which can be applied on all or parts of the mesh thanks to the mesh iterators such as `elements(mesh)` or `markedelements(mesh, marker)`. The return object is the interpolant of the function in the space X_h given as an argument to `vf::project()`.

```
u = vf::project( Xh, elements(mesh), g );
v = vf::project( Xh, elements(mesh), f );
w = vf::project( Xh, elements(mesh), idv(u)-g );
```

Here, u is the projection of g on the nodes of the mesh. Note that u and g have not the same type. To evaluate the error of $(u - g)$ on the nodes, we have to evaluate the function u , with the function v .

Norm

To calculate norms, we use integrals. For example, the L_2 norm is defined by :

$$\|g\|_{L_2} = \sqrt{\int_{\Omega} (g)^2}$$

To compute this, we use the `integrate` function (See 2.3) with `elements(mesh)` which allows to integrate on the entire domain. Let's see an example of norms computation with log file printing :

```
double L2g2 = integrate( elements(mesh), g*g ).evaluate() (0,0);
double L2uerror2 = integrate( elements(mesh), (idv(u)-g)*(idv(u)-g) ).evaluate() (0,0);
Log() << "||u-g||_0=" << math::sqrt( L2uerror2/L2g2 ) << "\n";
double L2f2 = integrate( elements(mesh), f*f ).evaluate() (0,0);
double L2verror2 = integrate( elements(mesh), (idv(v)-f)*(idv(v)-f) ).evaluate() (0,0);
Log() << "||v-f||_0=" << math::sqrt( L2verror2/L2f2 ) << "\n";
```

2.4.3 Results

Execution without option

We have made a test code (it is still `myfunctionspace.cpp` which create a new application `myfunctionspace` with some options (added with a `makeOptions()` function), and a description (with a `makeAbout()` function). (See 2.1).

This code applies the examples we have seen previously, and when we execute it without option, we obtain :

```
Execute MyFunctionSpace<2>
||u-g||_0=0.00463274
||v-f||_0=0.000483591
-----
Execute MyFunctionSpace<3>
||u-g||_0=0.00968019
||v-f||_0=0.000474499
```

As we have seen before, u is the projection of g in the function space. So on the mesh nodes, the error's norm has to be zero (the results we obtain here don't seems to be coherent).

2.5 Linear Algebra

FEEL++ supports three different linear algebra environments that we shall call *backends*.

- Gmm
- Petsc⁴
- Trilinos⁵

2.5.1 Choosing a linear algebra backend

To select a backend in order to solve a linear system, we instantiate the `Backend` class associated :

```
#include <feel/feelalg/backend.hpp>
boost::shared_ptr<Backend<double> > backend =
    Backend<double>::build( BACKEND_PETSC );
```

⁴Petsc is a suite of data structures and routines for the scalable solution of scientific applications modeled by PDE available at <http://www.mcs.anl.gov/petsc/petsc-as/>

⁵The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for scientific problems. <http://trilinos.sandia.gov/>

The backend provides an interface to solve

$$Ax = b \quad (2.1)$$

where A is a $n \times n$ sparse matrix and x, b vectors of size n . The backend defines the \mathbb{C}_+ types for each of these, e.g :

```
Backend<double>::sparse_matrix_type A;
Backend<double>::vector_type x, b;
```

In practice, we use the `boost::shared_ptr<>` shared pointer to ensure that we won't get memory leaks. The backends provide a corresponding **typedef**

```
Backend<double>::sparse_matrix_ptrtype A( backend->newMatrix( Xh, Yh ) );
Backend<double>::vector_ptrtype x( backend->newVector( Yh ) );
Backend<double>::vector_ptrtype b( backend->newVector( Xh ) );
```

where X_h and Y_h are function spaces providing the number of degrees of freedom that will define the size of the matrix and vectors thanks to the helpers functions `Backend::newMatrix()` and `Backend::newVector()`. In a parallel setting, the local/global processor mapping would be passed down by the function spaces.

2.5.2 Solving

To solve the linear problem $Ax = b$, the backend provides a function `solve` with three required parameters

```
solve(_matrix=A, _solution=x, _rhs=b)
```

where :

- the matrix A has a `sparse_matrix_ptrtype` type
- the solution x has a type `vector_type` or `vector_ptrtype`
- the second member vector b has a type `vector_ptrtype`

You can also add optional parameters like :

- a preconditioner : instead of solving $Ax = b$, we solve $P^{-1}Ax = P^{-1}b$. This method can be applied in iterative methods and permits to decrease the number of iterations in the resolution system
- a maximum number of iterations : this option is used with an iterative solving method
- a residual tolerance : the fraction $\frac{\|r^{(k)}\|}{\|r^{(0)}\|}$ is inferior to the residual tolerance with $r^{(k)} = b - Ax^{(k)}$ and $x^{(k)}$ the solution at the k^{th} iteration
- a absolute tolerance : $\|r^{(k)}\|$ is inferior to the absolute tolerance
- a different tolerance : sometimes, the residue doesn't decrease continuously during the iterations. The difference between two plots doesn't have to exceed the parameter choosen for the difference tolerance.
- a boolean to use transpose matrix : instead of solving $Ax = b$, we solve $A^t x = b$. If A is defined and positive, $A^t = A$.

To have a view of the values of the optional parameters, see the following code :

```
BOOST_PARAMETER_MEMBER_FUNCTION(
    (solve_return_type),
    solve,
    tag,
    (required
    (matrix, (sparse_matrix_ptrtype))
    (in_out(solution), *(mpl::or_<boost::is_convertible<mpl::_, vector_type>,
    boost::is_convertible<mpl::_, vector_ptrtype> >))
    (rhs, (vector_ptrtype)))
    (optional
    (prec, (sparse_matrix_ptrtype), matrix )
```

```

    (maxit, (size_type), 1000 )
    (rtolerance, (double), 1e-13)
    (atolerance, (double), 1e-50)
    (dtolerance, (double), 1e5)
    (reuse_prec, (bool), false )
    (transpose, (bool), false )
  )
}

```

The library `Boost::Parameters` allows you to enter parameters in the order you want. It supports deduced parameters, that is to say parameters whose identity can be deduced from their types.

2.6 Variational Formulation

2.6.1 Principle

A variational formulation of a problem is also called weak formulation. The key item is to bring a new function (called test function) and to integrate by parts. In that way we decrease the condition on our functions.

Let's considerate the equation to solve with boundary conditions where $u \in \Omega$ is the unknown

$$\begin{aligned} -\Delta u &= f \\ u &= u_D \quad \text{on } \Gamma_D \\ \nabla u \cdot n &= g \quad \text{on } \Gamma_N \end{aligned} \quad (2.2)$$

$\Gamma = \Gamma_D \cup \Gamma_N$ is the border of Ω . By integrating by parts with a function v (called test function) supposed piecewise regular, we obtain :

$$\int_{\Omega} \nabla u \cdot \nabla v - \int_{\Gamma} (\nabla u \cdot n) v = \int_{\Omega} f v$$

We have $u = u_D$ on Γ_D , we consequently take $v = 0$ on Γ_D and we got:

$$\int_{\Omega} \nabla u \cdot \nabla v - \int_{\Gamma_N} g v = \int_{\Omega} f v \quad u \in \Omega, \forall v \in V$$

where $V = \{v \in \Omega, v = 0 \text{ on } \Gamma_D\}$ with f and g which are known functions belonging to $C^0(\Omega)$. The test function v also has to be in \mathbb{H}^1 . The condition $v = 0$ on Γ_D is often used but we obviously can impose more binding boundaries conditions on the test function. More generally, V_h represents the function test's space.

2.6.2 Standard formulation: the Laplacian case

Mathematical formulation

laplacian.cpp

In this example, we would like to solve for the following problem in 2D

Problem 1 find u such that

$$-\Delta u = f \text{ in } \Omega = [-1; 1]^2 \quad (2.3)$$

with

$$f = 2\pi^2 g \quad (2.4)$$

and g is the exact solution

$$g = \sin(\pi x) \cos(\pi y) \quad (2.5)$$

The following boundary conditions apply

$$u = g|_{x=\pm 1}, \quad \frac{\partial u}{\partial n} = 0|_{y=\pm 1} \quad (2.6)$$

We propose here two possible variational formulations. The first one, handles the Dirichlet boundary conditions strongly, that is to say the condition is *incorporated* into the function space definitions. The second one handles the Dirichlet condition *weakly* and hence we have a uniform treatment for all types of boundary conditions.

First one : strong Dirichlet conditions The variational formulation reads as follows, we introduce the spaces

$$\mathcal{X} = \left\{ v \in H_1(\Omega) \text{ such that } v = g|_{x=-1, x=1} \right\} \quad (2.7)$$

and

$$\mathcal{V} = \left\{ v \in H_1(\Omega) \text{ such that } v = 0|_{x=-1, x=1} \right\} \quad (2.8)$$

We multiply (2.3) by $v \in \mathcal{V}$ then integrate over Ω and obtain

$$\int_{\Omega} -\Delta u v = \int_{\Omega} f v \quad (2.9)$$

We integrate by parts and reformulate the problem as follows:

Problem 2 we look for $u \in \mathcal{X}$ such as

$$\int_{\Omega} \nabla u \cdot \nabla v = \int_{\Omega} f v \quad \forall v \in \mathcal{V} \quad (2.10)$$

In the present space setting (2.8) and boundary conditions (7.7), we have the boundary term from the integration by parts which is dropped being equal to 0

$$\int_{\partial\Omega} \frac{\partial u}{\partial n} v = 0, \quad (2.11)$$

recalling that

$$\frac{\partial u}{\partial n} \stackrel{\text{def}}{=} \nabla u \cdot n \quad (2.12)$$

where n is the outward normal to $\partial\Omega$ by convention. We now discretize the problem, we create a mesh out of Ω , we have

$$\Omega = \bigcup_{e=1}^{N_{\text{el}}} \Omega^e \quad (2.13)$$

where Ω^e can be segments, triangles or tetrahedra depending on d and we have N_{el} of them. We introduce the finite dimensional spaces of continuous piecewise polynomial of degree N functions

$$X_h = \left\{ v_h \in C^0(\Omega), v_h|_{\Omega^e} \in \mathbb{P}_N(\Omega^e), v_h = g|_{x=-1, x=1} \right\} \quad (2.14)$$

and

$$V_h = \left\{ v_h \in C^0(\Omega), v_h|_{\Omega^e} \in \mathbb{P}_N(\Omega^e), v_h = 0|_{x=-1, x=1} \right\} \quad (2.15)$$

which are our trial and test function spaces respectively. We now have the problem we seek to solve which reads in our continuous Galerkin framework

Problem 3 we look for $u_h \in X_h \subset \mathcal{X}$ such that for all $v \in V_h \subset \mathcal{V}$

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h = \int_{\Omega} f v_h \quad (2.16)$$

Second one: weak Dirichlet conditions There is an alternative formulation which allows to treat weakly Dirichlet(Essential) boundary conditions similarly to Neumann(Natural) and Robin conditions. Following a similar development as in the previous section, the problem reads

Problem 4 we look for $u \in X_h \subset H_1(\Omega)$ such that for all $v \in X_h$

$$\int_{\Omega} \nabla u \cdot \nabla v + \int_{|x=-1, x=1} -\frac{\partial u}{\partial n} v - u \frac{\partial v}{\partial n} + \frac{\mu}{h} uv = \int_{\Omega} f v + \int_{|x=-1, x=1} -g \frac{\partial v}{\partial n} + \frac{\mu}{h} g v \quad (2.17)$$

where

$$X_h = \left\{ v_h \in C^0(\Omega), v_h|_{\Omega^e} \in \mathbb{P}_N(\Omega^e) \right\} \quad (2.18)$$

In (6.3), g is defined by (7.6). μ serves as a penalisation parameter which should be > 0 , e.g. between 2 and 10, and h is the size of the face. The inconvenient of this formulation is the introduction of the parameter μ , but the advantage is the *weak* treatment of the Dirichlet condition.

Feel formulation

First we define the f and g . To do that we use the `AUTO` keyword and associate to `f` and `g` their expressions

```
value_type pi = M_PI;
//! deduce from expression the type of g (thanks to keyword 'auto')
auto g = sin(pi*Px())*cos(pi*Py())*cos(pi*Pz());
gproj = vf::project( Xh, elements(mesh), g );

//! deduce from expression the type of f (thanks to keyword 'auto')
auto f = pi*pi*Dim*g;
```

where `M_PI` is defined in the header `cmath`. Using `AUTO` allows to defined `f` and `g` — which are moderately complex object — without having to know the actual type. `AUTO` determines automatically the type of the expression using the `__typeof__` keyword internally.

Then we form the right hand side by defining a linear form whose algebraic representation will be stored in a `vector_ptrtype` which is provided by the chosen linear algebra backend. The linear form is equated with an integral expression defining our right hand side.

```
auto F = backend(_vm=this->vm())->newVector( Xh );
form1( _test=Xh, _vector=F, _init=true ) =
    integrate( _range=elements(mesh), _expr=f*id(v) ) +
    integrate( _range=markedfaces( mesh, "Neumann" ),
        _expr=nu*gradv(gproj)*vf::N()*id(v) );
```

`form1` generates an instance of the object representing linear forms, that is to say it mimics the mathematical object ℓ such that

$$\begin{aligned} \ell : X_h &\mapsto \mathbb{R} \\ v_h &\rightarrow \ell(v_h) = \int_{\Omega} f v \end{aligned} \quad (2.19)$$

which is represented algebraically in the code by the vector `F` using the argument `_vector`. The last argument `_init`, if set to `true`⁶, will zero-out the entries of the vector `F`.

We now turn to the left hand side and define the bilinear form using the `form2` helper function which is passed (i) the trial function space using the `_trial` option, (ii) the test function space using the `_test` option, (iii) the algebraic representation using `_matrix`, i.e. a sparse matrix whose type is derived from one of the linear algebra backends and (iv) whether the associated matrix should be initialized using `_init`.

```
/** \code */
auto D = backend()->newMatrix( _test=Xh, _trial=Xh );
/** \endcode */

//! assemble  $\int_{\Omega} \nu \nabla u \cdot \nabla v$ 
```

⁶It is set to `false` by default.

```

/** \code */
form2( _test=Xh, _trial=Xh, _matrix=D ) =
    integrate( _range=elements(mesh), _expr=nu*gradt(u)*trans(grad(v)) );
/** \endcode */

```

Finally, we deal with the boundary condition, we implement both formulation described in appendix 7.2. For a *strong* treatment of the Dirichlet condition, we use the `on()` keyword of FEEL++ as follows

```

form2( _test=Xh, _trial=Xh, _matrix=D ) +=
    on( _range=markedfaces(mesh, "Dirichlet"),
        _element=u, _rhs=F, _expr=g );

```

Notice that we add, using `+=`, the Dirichlet contribution for the bilinear form. The first argument is the set of boundary faces to apply the condition: in gmsh the points satisfying $x = \pm 1$ are marked using the flags 1 and 3 ($x = -1$ and $x = 1$ respectively).

To implement the weak Dirichlet boundary condition, we add the following contributions to the left and right hand side:

```

form1( _test=Xh, _vector=F ) +=
    integrate( _range=markedfaces(mesh, "Dirichlet"),
        _expr=g*(-grad(v)*vf::N()+penaldir*id(v)/hFace()) );

```

```

form2( _test=Xh, _trial=Xh, _matrix=D ) +=
    integrate( _range=markedfaces(mesh, "Dirichlet"),
        _expr= ( -(gradt(u)*vf::N())*id(v)
                  - (grad(v)*vf::N())*idt(u)
                  +penaldir*id(v)*idt(u)/hFace() ) );

```

Note that we use the command line option `--weakdir` set to 1 by default to decide between weak/strong Dirichlet handling. Apart the uniform treatment of boundary conditions, the weak Dirichlet formulation has the advantage to work also in a parallel environment.

Next we solve the linear system

$$Du = F \quad (2.20)$$

where the `solve` function is implemented as follows

```

backend(_rebuild=true, _vm=this->vm())->solve( _matrix=D, _solution=u, _rhs=F );

```

Finally we check for the L_2 error in our approximation by computing

$$\|u - u_h\|_{L_2} = \sqrt{\int_{\Omega} (u - u_h)^2} = \sqrt{\int_{\Omega} (g - u_h)^2} \quad (2.21)$$

where u is the exact solution and is equal to g and u_h is the numerical solution of the problem (2.3) and the components of u_h in the P_2 Lagrange basis are given by solving (7.5).

The code reads

```

double L2error2 =integrate(_range=elements(mesh),
    _expr=(idv(u)-g)*(idv(u)-g)).evaluate()(0,0);
double L2error = math::sqrt( L2error2 );

Log() << "||error||_L2=" << L2error << "\n";

```

You can now verify that the L_2 error norm behaves like $h^{-(N+1)}$ where h is the mesh size and N the polynomial order. The H_1 error norm would be checked similarly in h^{-N} . The figure 2.6 displays the results using Paraview.

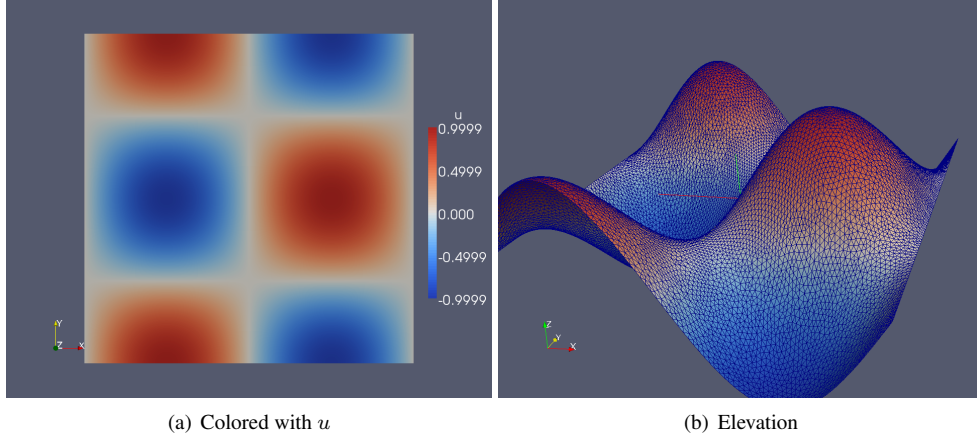


Figure 2.5: Solution of problem 4

2.6.3 Mixed formulation: the Stokes case

stokes.cpp

Mathematical formulation

We are now interested in solving the Stokes equations, we would like to solve for the following problem in 2D

Problem 5 find (\mathbf{u}, p) such that

$$-\mu \Delta \mathbf{u} + \nabla p = \mathbf{f} \quad \text{and} \quad \nabla \cdot \mathbf{u} = 0, \quad \text{in } \Omega = [-1; 1]^2 \quad (2.22)$$

with

$$\mathbf{f} = \mathbf{0} \quad (2.23)$$

where μ being the viscosity. The following boundary conditions apply

$$\mathbf{u} = \mathbf{1}_{|y=1}, \quad \mathbf{u} = \mathbf{0}_{|\partial\Omega \setminus \{(x,y) \in \Omega | y=1\}} \quad (2.24)$$

In problem (3), p is known up to a constant c , i.e. if p is a solution then $p + c$ is also solution. To ensure uniqueness we impose the constraint that p should have zero-mean, i.e.

$$\int_{\Omega} p = 0 \quad (2.25)$$

The problem 5 now reads

Problem 6 find (\mathbf{u}, p, λ) such that

$$-\mu \Delta \mathbf{u} + \nabla p = \mathbf{f}, \quad \nabla \cdot \mathbf{u} + \lambda = 0, \quad \text{and} \quad \int_{\Omega} p = 0, \quad \text{in } \Omega = [-1; 1]^2 \quad (2.26)$$

with

$$\mathbf{f} = \mathbf{0} \quad (2.27)$$

where μ being the viscosity. The following boundary conditions apply

$$\mathbf{u} = \mathbf{1}_{|y=1}, \quad \mathbf{u} = \mathbf{0}_{|\partial\Omega \setminus \{(x,y) \in \Omega | y=1\}} \quad (2.28)$$

The functional framework is as follows, we look for \mathbf{u} in $H_0^1(\Omega)$ and p in $L_0^2(\Omega)$. We shall not seek p in $L_0^2(\Omega)$ but rather in $L^2(\Omega)$ and use Lagrange multipliers which live in the constants whose space we denote $\mathbb{P}_0(\Omega)$, to enforce (2.25).

Denote $\mathcal{X} = H_0^1(\Omega) \times L^2(\Omega) \times \mathbb{P}_0(\Omega)$, the variational formulation reads we look for $(\mathbf{u}, p, \lambda) \in \mathcal{X}$ for all $(\mathbf{v}, q, \nu) \in \mathcal{X}$

$$\int_{\Omega} \mu \nabla \mathbf{u} : \nabla \mathbf{v} + \nabla \cdot \mathbf{v} p + \nabla \cdot \mathbf{u} q + q \lambda + p \nu = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \quad (2.29)$$

We build a triangulation Ω_h of Ω , we choose compatible (piecewise polynomial) discretisation spaces X_h and M_h , e.g. the Taylor Hood element ($\mathbb{P}_N/\mathbb{P}_{N-1}$) and we denote $\mathcal{X}_h = X_h \times M_h \times \mathbb{P}_0(\Omega)$. The discrete problem now reads, we look for $(\mathbf{u}_h, p_h, \lambda_h) \in \mathcal{X}_h$ such that for all $(\mathbf{v}_h, q_h, \nu_h) \in \mathcal{X}_h$

$$\int_{\Omega_h} \mu \nabla \mathbf{u}_h \cdot \nabla \mathbf{v}_h + \nabla \cdot \mathbf{v}_h p_h + \nabla \cdot \mathbf{u}_h q_h + p_h \nu_h + q_h \lambda_h = \int_{\Omega_h} \mathbf{f} \cdot \mathbf{v}_h \quad (2.30)$$

The formulation (2.30) leads to a linear system of the form

$$\underbrace{\begin{pmatrix} A & B & 0 \\ B^T & 0 & C \\ 0 & C^T & 0 \end{pmatrix}}_{\mathcal{A}} \underbrace{\begin{pmatrix} \mathbf{u}_h \\ p_h \\ \lambda_h \end{pmatrix}}_{\mathcal{U}} = \underbrace{\begin{pmatrix} F \\ 0 \\ 0 \end{pmatrix}}_{\mathcal{F}} \quad (2.31)$$

where A corresponds to the (\mathbf{u}, \mathbf{v}) block, B to the (\mathbf{u}, q) block and C to the (p, ν) block. \mathcal{A} is a symmetric positive definite matrix and thus the system $\mathcal{A}\mathcal{U} = \mathcal{F}$ enjoys a unique solution.

Feel formulation

Regarding the implementation of the Stokes problem 5, we can start from the laplacian case, from section 2.6.2. The implementation we choose to display here defines and builds \mathcal{X}_h , \mathcal{A} , \mathcal{U} and \mathcal{F} .

We start by defining and building \mathcal{X}_h : first we define the basis functions that will span each subspaces X_h , M_h and $\mathbb{P}_0(\Omega)$.

```
typedef Lagrange<2, Vectorial> basis_u_type;
typedef Lagrange<1, Scalar> basis_p_type;
typedef Lagrange<0, Scalar> basis_l_type;
typedef bases<basis_u_type, basis_p_type, basis_l_type> basis_type;
```

note that on the `typedef` we build a (MPL) vector of them. Now we are ready to define the function-space \mathcal{X}_h , much like in the Laplacian case:

```
typedef FunctionSpace<mesh_type, basis_type> space_type;
typedef boost::shared_ptr<space_type> space_ptrtype;
```

Next we define a few types which are associated with \mathcal{U} , u , p and λ respectively.

```
typedef space_type::element_type element_type;
```

Using these types we can instantiate elements of \mathcal{X}_h , X_h , M_h and $\mathbb{P}_0(\Omega_h)$ respectively:

They will serve in the definition of the variational formulation. We can now start assemble the various terms of the variational formulation (2.30). First we define some viscous stress tensor, $\tau(\mathbf{u}) = \nabla \mathbf{u}$, associated with the trial and test functions respectively

```
auto deft = gradt(u);
auto def = grad(v);
```

Then we define the total stress tensor times the normal, $\bar{\sigma}(\mathbf{u}, p)\mathbf{n} = -p\mathbf{n} + 2\mu\tau(\mathbf{u})\mathbf{n}$ where \mathbf{n} is the normal and $\bar{\sigma}(\mathbf{u}, p) = -p\mathbb{I} + 2\mu\tau(\mathbf{u})$:

```
// total stress tensor (trial)
auto SigmaNt = -idt(p)*N()+mu*deft*N();

// total stress tensor (test)
auto SigmaN = -id(p)*N()+mu*def*N();
```

We then form the matrix \mathcal{A} starting with block A , block B block C and finally the boundary conditions.

```

auto stokes = form2( _test=Xh, _trial=Xh, _matrix=D, _init=true );
boost::timer chrono;
stokes = integrate( elements(mesh), mu*inner(deft,def) );
std::cout << "mu*inner(deft,def): " << chrono.elapsed() << "\n"; chrono.restart();
stokes +=integrate( elements(mesh), - div(v)*idt(p) + divt(u)*id(q) );
std::cout << "(u,p): " << chrono.elapsed() << "\n"; chrono.restart();
stokes +=integrate( elements(mesh), id(q)*idt(lambda) + idt(p)*id(nu) );
std::cout << "(lambda,p): " << chrono.elapsed() << "\n"; chrono.restart();

stokes +=integrate( boundaryfaces(mesh), -inner(SigmaNt,id(v)) );
stokes +=integrate( boundaryfaces(mesh), -inner(SigmaN,idt(u)) );
stokes +=integrate( boundaryfaces(mesh), +penalbc*inner(idt(u),idt(v))/hFace() );

std::cout << "bc: " << chrono.elapsed() << "\n"; chrono.restart();

```

The figure 2.6 displays p and \mathbf{u} which are available in

```
ls ~/feel/doc/tutorial/stokes/Simplex_2_1_2/P2/h_0.05
```

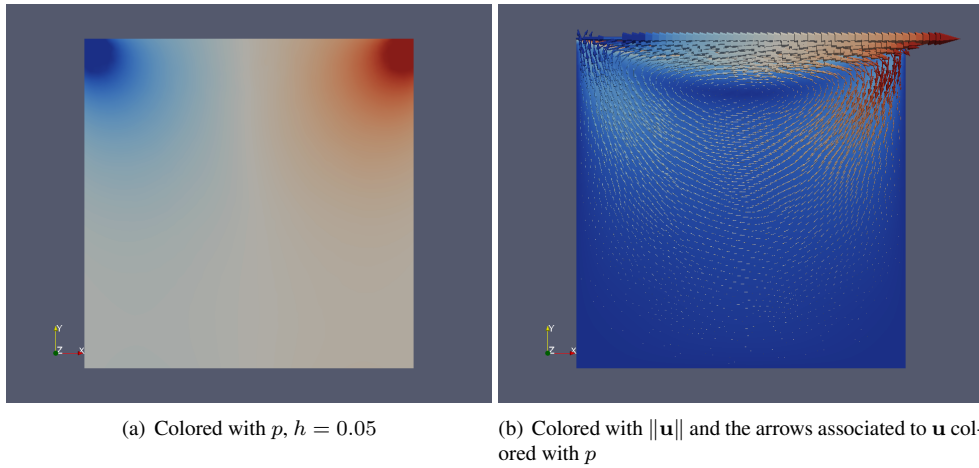


Figure 2.6: Solution of problem 5

CHAPTER 3

Feel++ Language Keywords

By Christophe Prud'homme

Chapter ref: [cha:appendix-feel]

3.1 Keywords

One of FEEL++ assets is its finite element embedded language. The language follows the C++ grammar, and provides keywords as well as operations between objects which are, mathematically, tensors of rank 0, 1 or 2.

Here are some notations :

- $f : \mathbb{R}^n \mapsto \mathbb{R}^{m \times p}$ with $n = 1, 2, 3$, $m = 1, 2, 3$, $p = 1, 2, 3$.
- Ω^e current mesh element

and here is the table which gathers all tools you may need:

Keyword	Math object	Description	Rank	$M \times N$
<code>P ()</code>	\vec{P}	current point coordinates $(P_x, P_y, P_z)^T$	1	$d \times 1$
<code>Px ()</code>	P_x	x coordinate of \vec{P}	0	1×1
<code>PY ()</code>	P_y	y coordinate of \vec{P} (value is 0 in 1D)	0	1×1
<code>Pz ()</code>	P_z	z coordinate of \vec{P} (value is 0 in 1D and 2D)	0	1×1
<code>C ()</code>	\vec{C}	element barycenter point coordinates $(C_x, C_y, C_z)^T$	1	$d \times 1$
<code>Cx ()</code>	C_x	x coordinate of \vec{C}	0	1×1
<code>CY ()</code>	C_y	y coordinate of \vec{C} (value is 0 in 1D)	0	1×1
<code>Cz ()</code>	C_z	z coordinate of \vec{C} (value is 0 in 1D and 2D)	0	1×1

Feel++ Language Keywords

Keyword	Math object	Description	Rank	$M \times N$
<code>N()</code>	\vec{N}	normal at current point $(N_x, N_y, N_z)^T$	1	$d \times 1$
<code>Nx()</code>	N_x	x coordinate of \vec{N} at current point	0	1×1
<code>Ny()</code>	N_y	y coordinate of \vec{N} at current point (value is 0 in 1D)	0	1×1
<code>Nz()</code>	N_z	z coordinate of \vec{N} at current point (value is 0 in 1D and 2D)	0	1×1
<code>eid()</code>	e	index of Ω^e	0	1×1
<code>emarker()</code>	$m(e)$	marker of Ω^e	0	1×1
<code>h()</code>	h^e	size of Ω^e	0	1×1
<code>hFace()</code>	h_Γ^e	size of face Γ of Ω^e	0	1×1
<code>mat<M,N>(m_11,</code> <code>m_12,...)</code>	$\begin{pmatrix} m_{11} & m_{12} & \dots \\ m_{21} & m_{22} & \dots \\ \vdots & & \end{pmatrix}$	$M \times N$ matrix	2	$M \times N$
<code>vec<M>(v_1,</code> <code>v_2,...)</code>	$(v_1, v_2, \dots)^T$	entries being expressions column vector with M rows	1	$M \times 1$
<code>trace(expr)</code>	$\text{tr}(f(\vec{x}))$	entries being expressions trace of $f(\vec{x})$	0	1×1
<code>abs(expr)</code>	$ f(\vec{x}) $	element wise absolute value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>cos(expr)</code>	$\cos(f(\vec{x}))$	element wise cosinus value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>sin(expr)</code>	$\sin(f(\vec{x}))$	element wise sinus value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>tan(expr)</code>	$\tan(f(\vec{x}))$	element wise tangent value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>acos(expr)</code>	$\text{acos}(f(\vec{x}))$	element wise acos value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>asin(expr)</code>	$\text{asin}(f(\vec{x}))$	element wise asin value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>atan(expr)</code>	$\text{atan}(f(\vec{x}))$	element wise atan value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>cosh(expr)</code>	$\cosh(f(\vec{x}))$	element wise cosh value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>sinh(expr)</code>	$\sinh(f(\vec{x}))$	element wise sinh value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>tanh(expr)</code>	$\tanh(f(\vec{x}))$	element wise tanh value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>exp(expr)</code>	$\exp(f(\vec{x}))$	element wise exp value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>log(expr)</code>	$\log(f(\vec{x}))$	element wise log value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>sqrt(expr)</code>	$\sqrt{f(\vec{x})}$	element wise sqrt value of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>sign(expr)</code>	$\begin{cases} 1 & \text{if } f(\vec{x}) \geq 0 \\ -1 & \text{if } f(\vec{x}) < 0 \end{cases}$	element wise sign of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>chi(expr)</code>	$\chi(f(\vec{x})) = \begin{cases} 0 & \text{if } f(\vec{x}) = 0 \\ 1 & \text{if } f(\vec{x}) \neq 0 \end{cases}$	element wise boolean test of f	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>id(f)</code>	f	test function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>idt(f)</code>	f	trial function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>idv(f)</code>	f	evaluation function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>grad(f)</code>	∇f	gradient of test function	$\text{rank}(f(\vec{x})) + 1$	$p = 1, m \times$
<code>gradt(f)</code>	∇f	gradient of trial function	$\text{rank}(f(\vec{x})) + 1$	$p = 1, m \times$
<code>gradv(f)</code>	∇f	evaluation function gradient	$\text{rank}(f(\vec{x})) + 1$	$p = 1, m \times$
<code>div(f)</code>	$\nabla \cdot \vec{f}$	divergence of test function	$\text{rank}(f(\vec{x})) - 1$	1×1^2

¹Gradient of matrix value functions is not implemented, hence $p = 1$
²Divergence of matrix value functions is not implemented, hence $p = 1$

Keyword	Math object	Description	Rank	$M \times N$
<code>divt(f)</code>	$\nabla \cdot \vec{f}$	divergence of trial function	$\text{rank}(f(\vec{x})) - 1$	1×1
<code>divv(f)</code>	$\nabla \cdot \vec{f}$	evaluation of function divergence	$\text{rank}(f(\vec{x})) - 1$	1×1
<code>curl(f)</code>	$\nabla \times \vec{f}$	curl of test function	1	$n = m, n \times$
<code>curlt(f)</code>	$\nabla \times \vec{f}$	curl of trial function	1	$m = n, n \times$
<code>curlv(f)</code>	$\nabla \times \vec{f}$	evaluation of function curl	1	$m = n, n \times$
<code>hess(f)</code>	$\nabla^2 f$	hessian of test function	2	$m = p = 1,$
<code>jump(f)</code>	$[f] = f_0 \vec{N}_0 + f_1 \vec{N}_1$	jump of test function	1	$m = 1, n \times$
<code>jump(f)</code>	$[\vec{f}] = \vec{f}_0 \cdot \vec{N}_0 + \vec{f}_1 \cdot \vec{N}_1$	jump of test function	0	$m = 2, 1 \times$
<code>jumpt(f)</code>	$[f] = f_0 \vec{N}_0 + f_1 \vec{N}_1$	jump of trial function	1	$m = 1, n \times$
<code>jumpt(f)</code>	$[\vec{f}] = \vec{f}_0 \cdot \vec{N}_0 + \vec{f}_1 \cdot \vec{N}_1$	jump of trial function	0	$m = 2, 1 \times$
<code>jumpv(f)</code>	$[f] = f_0 \vec{N}_0 + f_1 \vec{N}_1$	jump of function evaluation	1	$m = 1, n \times$
<code>jumpv(f)</code>	$[\vec{f}] = \vec{f}_0 \cdot \vec{N}_0 + \vec{f}_1 \cdot \vec{N}_1$	jump of function evaluation	0	$m = 2, 1 \times$
<code>average(f)</code>	$f = \frac{1}{2}(f_0 + f_1)$	average of test function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>averaget(f)</code>	$f = \frac{1}{2}(f_0 + f_1)$	average of trial function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>averagev(f)</code>	$f = \frac{1}{2}(f_0 + f_1)$	average of function evaluation	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>leftface(f)</code>	f_0	left test function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>leftfacet(f)</code>	f_0	left trial function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>leftfacev(f)</code>	f_0	left function evaluation	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>rightface(f)</code>	f_1	right test function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>rightfacet(f)</code>	f_1	right trial function	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>rightfacev(f)</code>	f_1	right function evaluation	$\text{rank}(f(\vec{x}))$	$m = n, n \times$
<code>maxface(f)</code>	$\max(f_0, f_1)$	maximum of right and left test function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>maxfacet(f)</code>	$\max(f_0, f_1)$	maximum of right and left trial function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>maxfacev(f)</code>	$\max(f_0, f_1)$	maximum of right and left function evaluation	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>minface(f)</code>	$\min(f_0, f_1)$	minimum of right and left test function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>minfacet(f)</code>	$\min(f_0, f_1)$	minimum of right and left trial function	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>minfacev(f)</code>	$\min(f_0, f_1)$	minimum of right and left function evaluation	$\text{rank}(f(\vec{x}))$	$m \times p$
<code>-</code>	$-g$	element wise unary minus		
<code>!</code>	$!g$	element wise logical not		
<code>+</code>	$f + g$	tensor sum		
<code>-</code>	$f - g$	tensor subtraction		
<code>*</code>	$f * g$	tensor product		
<code>/</code>	f / g	tensor division (g scalar field)		
<code><</code>	$f < g$	element wise less		
<code><=</code>	$f \leq g$	element wise less or equal		
<code>></code>	$f > g$	element wise greater		
<code>>=</code>	$f \geq g$	element wise greater or equal		
<code>==</code>	$f = g$	element wise equal		
<code>!=</code>	$f \neq g$	element wise not equal		
<code>&&</code>	$f \text{ and } g$	element wise logical and		

Keyword	Math object	Description	Rank	$M \times N$
	f or g	element wise logical or		

3.2 Operators

3.2.1 Integrals

Thank to its finite element embedded language, FEEL++ has its owned `integrate()` function, which can be written for example :

```
integrate( _range= elements(mesh), _expr= gradt(T)*trans(grad(v)) );
```

please notice that the order of the parameter is not important, these are `boost` parameters, so you can enter them in the order you want. To make it clear, there are two required parameters and 2 optional and they of course can be entered in any order provided you give the parameter name. If you don't provide the parameter name (that is to say `_range=` or the others) they must be entered in the order they are described below.

The required parameters are

- `_range` = domain of integration
- `_expr` = integrand expression

The optional parameters are

- `_quad` = quadrature to use instead of the default one, wich means `_Q<integer>()` where the integer is the polynomial order to integrate exactly
- `_geomap` = type of geometric mapping to use, that is to say :
 - `GEOMAP_HO` = high order approximation (same of the mesh)
 - `GEOMAP_OPT` = optimal approximation: high order on boundary elements, order 1 in the interior
 - `GEOMAP_O1` = order 1 approximation

3.2.2 Projections

It is also possible to make projections with the library, the interface is as follow :

```
project( _range, _space, _expr, _geomap );
```

where

- `_space` is the space in which lives the projected expression, it should be a nodal function space
- `_expr` the expression to project
- `_range` is the domain for the projection (optional, default: all elements from `space->mesh()`)
- `_geomap` is the type of geometric mapping approximation (optional, default = `GEOMAP_HO`)
- `_accumulate` (optional, default = false)

3.2.3 Meshes

FEEL++ enables full different ways to interact with the mesh on which you want to work. Mainly with the function `integrate`, the various keywords we have established will make your program's code easier. The interoperability between FEEL++ and GMSH is huge and provides various access to any point, item, domain or almost anything you want in a mesh. The access to different items of a mesh is possible thanks to the filters which enable the access of only a mesh's part. These helpful keywords are coded in `feel/feelmesh/filters.hpp`, we are here going to describe most of them.

To access one particular part of a mesh, you can use :

- `elements(mesh)` corresponds to all the elements of a mesh
- `markedelements(mesh, id)` corresponds to the precise element defined by the id. It can be any element (line, surface, domain, and so on).
- `faces(mesh)` corresponds to all the faces of the mesh.
- `markedfaces(mesh)` corresponds to all the faces of the mesh which are marked.
- `boundaryfaces(mesh)` corresponds to all elements that own a topological dimension one below the mesh. For example, if you mesh is a 2D one, `boundaryfaces(mesh)` will return all the lines (because of dimension $2 - 1 = 1$). These elements which have one dimension less, are corresponding to the boundary faces.
- `internalelements(mesh)` corresponds to all the elements of the mesh which are strictly within the domain that is to say they do not share a face with the boundary.
- `boundaryelements(mesh)` corresponds to all the elements of the mesh which share a face with the boundary of the mesh.
- `edges(mesh)` corresponds to all the edges of the mesh.
- `boundaryedges(mesh)` corresponds to all boundary edges of the mesh.

where id is the element's identifier : thanks to GMSH, this identifier can be an integer or a string, it depends on the identifier you have or you gave in the mesh .geo file.

Part II

Learning by Examples

CHAPTER 4

Non-Linear examples

By Christophe Prud'homme

Chapter ref: [cha:non-linear-ex]

4.1 Solving nonlinear equations

FEEL++ allows to solve nonlinear equations thanks to its interface to the interface to the PETSc nonlinear solver library. It requires the implementation of two extra functions in your application that will update the jacobian matrix associated to the tangent problem and the residual.

Consider that you have an application class `MyApp` with a backend as data member

```
#include <feel/feelcore/feel.hpp>
#include <feel/feelcore/application.hpp>
#include <feel/feelalg/backend.hpp>
namespace Feel {

class MyApp : public Application
{
public:

    typedef Backend<double> backend_type;
    typedef boost::shared_ptr<backend_type> backend_ptrtype;

    MyApp( int argc, char** argv,
    AboutData const& ad, po::options_description const& od )
    :
    // init the parent class
    Application( argc, argv, ad, od ),
    // init the backend
    M_backend( backend_type::build( this->vm() ) ),
    {
        // define the callback functions (works only for the PETSc backend)
        M_backend->nlSolver()->residual =
            boost::bind( &self_type::updateResidual, boost::ref( *this ), _1, _2 );
        M_backend->nlSolver()->jacobian =
            boost::bind( &self_type::updateJacobian, boost::ref( *this ), _1, _2 );
    }

    void updateResidual( const vector_ptrtype& X, vector_ptrtype& R )
    {
        // update the matrix J (Jacobian matrix) associated
        // with the tangent problem
    }
};
}
```

```

void updateJacobian( const vector_ptrtype& X, sparse_matrix_ptrtype& J)
{
    // update the vector R associated with the residual
}
void run()
{
    //define space
    Xh...
    element_type u(Xh);
    // initial guess is 0
    u = project( M_Xh, elements(mesh), constant(0.) );
    vector_ptrtype U( M_backend->newVector( u.functionSpace() ) );
    *U = u;

    // define R and J
    vector_ptrtype R( M_backend->newVector( u.functionSpace() ) );
    sparse_matrix_ptrtype J;

    // update R
    updateJacobian( U, R );
    // update J
    updateResidual( U, J );

    // solve using non linear methods (newton)
    // tolerance : 1e-10
    // max number of iterations : 10
    M_backend->nlSolve( J, U, R, 1e-10, 10 );

    // the solution was stored in U
    u = *U;
}
private:
    backend_ptrtype M_backend;
};
} // namespace Feel

```

The function `updateJacobian` and `updateResidual` implement the assembly of the matrix J (jacobian matrix) and the vector R (residual vector) respectively.

4.1.1 A first nonlinear problem

As a simple example, let Ω be a subset of \mathbb{R}^d , $d = 1, 2, 3$, (i.e. $\Omega = [-1, 1]^d$) with boundary $\partial\Omega$. Consider now the following equation and boundary condition

$$-\Delta u + u^\lambda = f, \quad u = 0 \text{ on } \partial\Omega. \quad (4.1)$$

where $\lambda \in \mathbb{R}_+$ is a given parameter and $f = 1$.

To be described in this section. For now see `doc/manual/nonlinearpow.cpp` for an implementation of this problem.

4.1.2 Simplified combustion problem: Bratu

As a simple example, let Ω be a subset of \mathbb{R}^d , $d = 1, 2, 3$, (i.e. $\Omega = [-1, 1]^d$) with boundary $\partial\Omega$. Consider now the following equation and boundary condition

$$-\Delta u + \lambda e^u = f, \quad u = 0 \text{ on } \partial\Omega \quad (4.2)$$

where λ is a given parameter. Ceci est généralement appelé le problème de Bratu et apparaît lors de la simplification de modèles de processus de diffusion non-linéaires par exemple dans le domaine de la combustion.

To be described in this section. For now see `doc/manual/bratu.cpp` for an implementation of this problem.

CHAPTER 5

Heat sink

By Baptiste Morin, Christophe Prud'homme

Chapter ref: **[cha:heatsink]**

This problem considers the performance of a heat sink designed for the thermal management of high-density electronic components. The heat sink is comprised of a base/spreader which in turn supports a number of plate fins exposed to flowing air. We model the flowing air through a simple convection heat transfer coefficient. From the engineering point of view, this problem illustrates the application of conduction analysis to an important class of cooling problems: electronic components and systems.

Our interest is in the conduction temperature distribution at the base of the spreader. The target is to study how the heat transfer occurs with different parameters on our heat sink. The heat generated by high-density electronic components is such that it's very expensive to cool large structures (data center). The cooling optimization is consequent in the run for decreasing operating costs.

A classical thermal CPU cooler looks like this

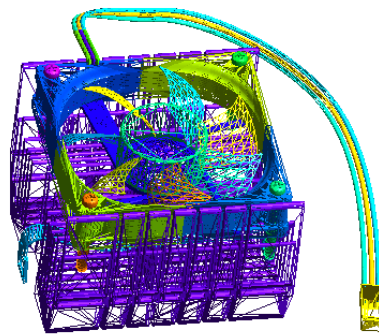


Figure 5.1: Mesh of a classical CPU cooler

We are here going to describe how it is theoretically working and how it is implected with FEEL++.

5.1 Problem description

5.1.1 Domain

We consider here a classical "radiator" which is a CPU heat sink. Those types of coolers are composed with a certain number of plate fins exposed to flowing air or exposed to a ventilator. Regarding the periodicity and geometry of our concern, we can make our study on a characteristic element of the problem : a half cell of the heat sink single thermal fin with its spreader at the basis. Let's take a look at the geometry of our problem :

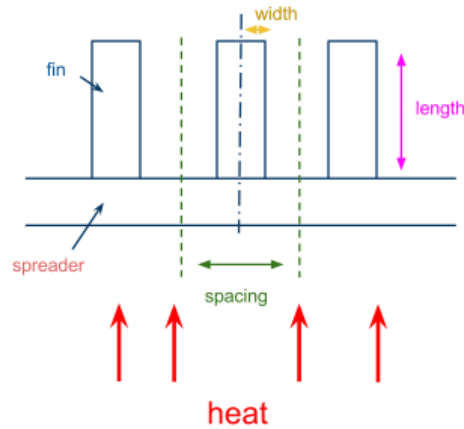


Figure 5.2: Geometry of heat sink

Our study is available in 2 or 3 dimensions, depending on the application's parameters. You'll see later how to work with it. Let's see on which meshes we are working on :



Figure 5.3: 2D mesh

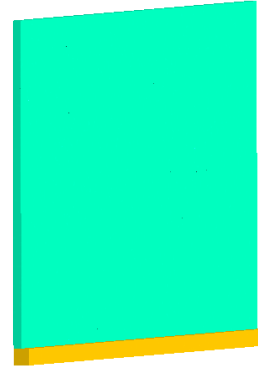


Figure 5.4: 3D mesh

5.1.2 Inputs

The implementation of those parameters is described in the section [5.3.1](#).

Material

Here the material parameter can be described with further parameters. We have, with $i = 1$ for the fin and $i = 2$ for the base :

- the thermal conductivity κ_i
- the material's density ρ_i
- the heat capacity of the material C_i

The term $\rho_i C_i$ corresponds to the heat volumetric capacity. In that way, we make possible the construction of a heat sink with 2 different materials. Here is a list of the well-known ones, ρ and C are gave at $298K$:

Material	Thermal conductivity (κ in $W.m^{-1}.K^{-1}$)	Density (ρ in $kg.m^{-3}$)	Heat Capacity (C in $J.kg^{-1}.K^{-1}$)
Aluminium	180 (alloys) or 290 (pure)	2700	897
Copper	386	8940	385
Gold	314	19320	129
Silver	406	10500	233

Physical

- Depth
This parameter is only to take into account for the 3D simulation. It represents the depth of the characteristical heat sink and is called `depth` in the application.
- Length
You can also parameterize the length of the fin. This one is called `L` in the application's parameters, its dimension is the meter.
- Width
Typically, this parameter is linked with constructor's standards. This parameter is called `width` in the application's implementation.

Thermal

- Heat flux
It represents the heat flux brought by the electronic component at the bottom of the base. Here it's typically the heat brought by the processor.
- Thermal coefficient
The thermal coefficient h named *therm_coeff* in the application is representative of the heat transfer between the fin and the air flow.
- Ambien temperature
This parameter called T_{amb} represents the temperature around the heat sink at the beginning. That means the ambient temperature before the computer is turned on.

Summary table

The following table displays the various fixed and variables parameters of this application.

Name	Description	Nominal Value	Range	Units
BDF parameters				
$time - initial$	begining	0		
$time - final$	end	50	$]0, 1500]$	
$time - step$	time step	0.1	$]0, 1[$	
$steady$	steady state	0	$\{0, 1\}$	
$order$	order	2	$[0, 4]$	
Physical parameters				
L	fin's length	$2 \cdot 10^{-2}$	$[0.02, 0.05]$	m
$width$	fin's width	$5 \cdot 10^{-4}$	$[10^{-5}, 10^{-4}]$	m
$deep$	heat sink depth	0	$[0, 7 \cdot 10^{-2}]$	m
Mesh parameter				
$hsize$	mesh's size	10^{-4}	$[10^{-5}, 10^{-3}]$	
Fin Parameters				
κ_f	thermal conductivity	386	$[100, 500]$	$W \cdot m^{-1} \cdot K^{-1}$
ρ_f	material density	8940	$[10^3, 12 \cdot 10^3]$	$kg \cdot m^{-3}$
C_f	heat capacity	385	$[10^2, 10^3]$	$J \cdot kg^{-1} \cdot K^{-1}$
Base/spreader Parameters				
κ_s	thermal conductivity	386	$[100, 500]$	$W \cdot m^{-1} \cdot K^{-1}$
ρ_s	material density	8940	$[10^3, 12 \cdot 10^3]$	$kg \cdot m^{-3}$
C_s	heat capacity	385	$[10^2, 10^3]$	$J \cdot kg^{-1} \cdot K^{-1}$
Heat Parameters				
T_{amb}	ambient temperature	300	$[300, 310]$	K
$heat_flux$	heat flux Q	10^6	$[0, 10^6]$	$W \cdot m^{-2}$
$therm_coeff$	thermal coefficient h	10^3	$[0, 10^3]$	$W \cdot m^{-2} \cdot K^{-1}$

Table 5.1: Table of fixed and variable parameters

5.2 Theory

5.2.1 Figure

The global domain is $\Omega = \Omega_1 \cup \Omega_2$ where Ω_1 is the fin's domain and Ω_2 the spreader's domain. We note $\partial\Omega$ the border of the domain Ω . The physical lines we are using will be noted as Γ_i such as described above. The following figure describes the parameters and the geometry we are using in the equations to solve our 3D issue : The following figures describe the parameters and the geometry we are using in the equations to solve our 2D or 3D issue :

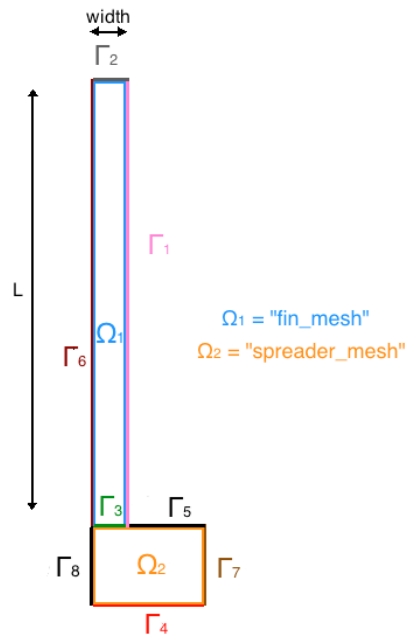


Figure 5.5: 2D geometry details

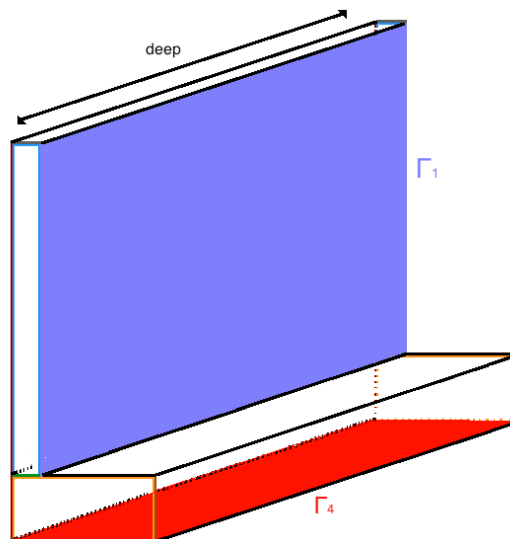


Figure 5.6: 3D geometry details

5.2.2 Equations

Our concern satisfies the heat equation which reads

$$\sum_{i=1}^2 \kappa_i \Delta T - \rho_i C_i \frac{\partial T}{\partial t} = 0 \quad (5.1)$$

$$\kappa_1 \frac{\partial T}{\partial n} = 0 \quad \text{on } \Gamma_2 \quad \text{and} \quad \Gamma_6 \quad (5.2)$$

$$\kappa_2 \frac{\partial T}{\partial n} = 0 \quad \text{on } \Gamma_5, \Gamma_7 \quad \text{and} \quad \Gamma_8 \quad (5.3)$$

$$\kappa_1 \frac{\partial T}{\partial n} = -h(T - T_{amb}) \quad \text{on } \Gamma_1 \quad (5.4)$$

$$\kappa_2 \frac{\partial T}{\partial n} = Q(1 - e^{-t}) \quad \text{on } \Gamma_4 \quad (5.5)$$

$$T|_{\Omega_1} = T|_{\Omega_2} \quad \text{on } \Gamma_3 \quad (5.6)$$

$$\kappa_1 \nabla T \cdot n = \kappa_2 \nabla T \cdot n \quad \text{on } \Gamma_3 \quad (5.7)$$

with $i = 1$ for the fin and $i = 2$ for the base and where κ_i is the thermal conductivity, ρ_i is the material's density ($kg.m^{-3}$ in the SI unit), C_i the heat capacity and T the temperature at a precise point (in 2D or 3D). To see how it has been coded, you can read [5.3.3](#).

5.2.3 Boundary conditions

The problem requires that the temperature and heat flux are continue on Γ_3 . Considering the problem's geometry, we also impose zero heat flux on the vertical surfaces of the spreader. Let's detail the conditions we have imposed :

- Homogeneous Neumann condition (5.2) and (5.3) : it represents the fact that the heat flux is only vertical (for Γ_6 and Γ_7) or the fact that the heat flux is only provided by Γ_4 (for Γ_2 and Γ_5).
- Homogeneous Neumann condition (5.4) : it imposes that the heat flux is brought by this surface (it mathematically represents that the heat sink is placed on the heat source).
- Non-homogeneous Neumann condition (5.5) : this boundary condition represents the transient state for the heat transfer calculation.
- Temperature continuity (5.6) : it imposes that the temperature is continue at the interface between the two materials (if there are two materials, we can also have the same one for the two pieces).
- Heat flux continuity (5.7) : it represents that the heat flux is continue at the interface between the two materials. Literally, it means that the two flows offset each other.

Theses conditions have been coded as explained in the section [5.3.3](#).

5.2.4 Finite Element Method

Let's apply the method to our concern, we introduce the test function v and we integrate the main equation, which reads now as :

$$\sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} - \kappa_i \int_{\Omega_i} v \Delta T = 0 \quad (5.8)$$

We integrate by parts, which leads to :

$$\sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T - \kappa_i \int_{\partial \Omega_i} (\nabla T \cdot n) v = 0 \quad (5.9)$$

then, by decomposing the borders $\partial \Omega_i$, we obtain :

$$\begin{aligned} & -\kappa_1 \int_{\Gamma_1} (\nabla T \cdot n) v - \kappa_2 \int_{\Gamma_4} (\nabla T \cdot n) v - \kappa_1 \int_{\Gamma_{2,6}} (\nabla T \cdot n) v - \kappa_2 \int_{\Gamma_{5,7,8}} (\nabla T \cdot n) v + \\ & \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T - \kappa_i \int_{\partial \Omega_i \cap \Gamma_3} (\nabla T \cdot n) v = 0 \end{aligned} \quad (5.10)$$

Now, we apply the conditions (5.2), (5.3), (5.4) and (5.5) which brings us to :

$$\int_{\Gamma_1} h v (T - T_{amb}) - \int_{\Gamma_4} v Q (1 - e^{-t}) + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T - \underbrace{\kappa_i \int_{\partial \Omega_i \cap \Gamma_3} (\nabla T \cdot n) v}_{=0 \text{ thanks to 5.7}} = 0 \quad (5.11)$$

Now we apply the boundary conditions (5.7) which results in :

$$h \int_{\Gamma_1} v (T - T_{amb}) - \int_{\Gamma_4} v Q (1 - e^{-t}) + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T = 0 \quad (5.12)$$

We can now start to transform the equation by putting in the right hand the known terms :

$$h \int_{\Gamma_1} v T + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{\partial T}{\partial t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T = \int_{\Gamma_4} v Q (1 - e^{-t}) + h T_{amb} \int_{\Gamma_1} v \quad (5.13)$$

We discretize $\frac{\partial T}{\partial t}$ where δt is the time step, such as:

$$h \int_{\Gamma_1} v T + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{T^{n+1} - T^n}{\delta t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T = \int_{\Gamma_4} v Q (1 - e^{-t}) + h T_{amb} \int_{\Gamma_1} v \quad (5.14)$$

Finally we obtain :

$$h \int_{\Gamma_1} v T + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{T^{n+1}}{\delta t} + \kappa_i \int_{\Omega_i} \nabla v \cdot \nabla T = \int_{\Gamma_4} v Q (1 - e^{-t}) + h T_{amb} \int_{\Gamma_1} v + \sum_{i=1}^2 \rho_i C_i \int_{\Omega_i} v \frac{T^n}{\delta t} \quad (5.15)$$

This is that equation which is implemented in the application `feel_heatsink`.

5.3 Implementation

5.3.1 Application parameters

The parameters of the application are implemented such as

```
inline
Feel::po::options_description
makeOptions()
{
    Feel::po::options_description heatsinkoptions("heatsink options");
    heatsinkoptions.add_options()
    // mesh parameters
    ("hsize", Feel::po::value<double>()->default_value( 0.1 ),
     "first h value to start convergence")
    ("L", Feel::po::value<double>()->default_value( 0.03 ),
     "dimensional length of the sink (in meters)")
    ("width", Feel::po::value<double>()->default_value( 0.0005 ),
     "dimensional width of the fin (in meters)")

    // 3D parameter
    ("deep", Feel::po::value<double>()->default_value( 0 ),
     "depth of the mesh (in meters) only in 3D simulation")

    // thermal conductivities parameters
    ("kappa_s", Feel::po::value<double>()->default_value( 386 ),
     "thermal conductivity of the base spreader in SI unit W.m^{-1}.K^{-1}")
    ("kappa_f", Feel::po::value<double>()->default_value( 386 ),
     "thermal conductivity of the fin in SI unit W.m^{-1}.K^{-1}")

    // density parameter
    ("rho_s", Feel::po::value<int>()->default_value( 8940 ),
     "density of the spreader's material in SI unit kg.m^{-3}")
    ("rho_f", Feel::po::value<int>()->default_value( 8940 ),
     "density of the fin's material in SI unit kg.m^{-3}")

    // heat capacities parameter
    ("c_s", Feel::po::value<double>()->default_value( 385 ),
     "heat capacity of the spreader's material in SI unit J.kg^{-1}.K^{-1}")
    ("c_f", Feel::po::value<double>()->default_value( 385 ),
     "heat capacity of the fin's material in SI unit J.kg^{-1}.K^{-1}")

    // physical coeff
    ("therm_coeff", Feel::po::value<double>()->default_value(50),
     "thermal coefficient")
    ("Tamb", Feel::po::value<double>()->default_value(300),
     "ambient temperature")
    ("heat_flux", Feel::po::value<double>()->default_value(1e6),
     "heat flux generated by CPU")

    ("steady", Feel::po::value<bool>()->default_value(false),
     "if true : steady else unsteady")

    // export
    ("export-matlab", "export matrix and vectors in matlab ");

    return heatsinkoptions.add( Feel::feel_options() );
}
```

5.3.2 Surfaces

To be able to calculate the surfaces in further dimension without changing the code, we have given the same names for the faces we were interested in. In 2D Γ_i represents a line whereas in 3D it represents a surface. The calculation of those surfaces which makes possible the calculation of averages temperature is as follow :

```
surface_base =
    integrate( _range= markedfaces(mesh, "gamma4"), _expr= cst(1.)).evaluate() (0,0);
```



```
surface_fin =
  integrate( _range= markedfaces(mesh, "gamma1"), _expr=cst(1.)).evaluate()(0,0);
```

5.3.3 Equations

First we start by calculate the non-steady state which means that we integrate all the time-independant terms, which is done with :

```
/*
 * Right hand side construction (steady state)
 */
form1( _test=Xh, _vector=F, _init=true ) =
  integrate( _range= markedfaces(mesh, "gamma1"), _expr= therm_coeff*Tamb*id(v) );

/*
 * Left hand side construction (steady state)
 */
form2( Xh, Xh, D, _init=true ) =
  integrate( _range= markedelements(mesh, "spreader_mesh"),
            _expr= kappa_s*gradt(T)*trans(grad(v)) );

form2( Xh, Xh, D) +=
  integrate( _range= markedelements(mesh, "fin_mesh"),
            _expr= kappa_f*gradt(T)*trans(grad(v)) );

form2( Xh, Xh, D) +=
  integrate( _range= markedfaces(mesh, "gamma1"),
            _expr= therm_coeff*idt(T)*id(v) );

form2(Xh, Xh, D) +=
  integrate( _range=markedelements(mesh, "spreader_mesh"),
            _expr=rho_s*c_s*idt(T)*id(v)*M_bdf->polyDerivCoefficient(0) )
+ integrate( _range=markedelements(mesh, "fin_mesh"),
            _expr=rho_f*c_f*idt(T)*id(v)*M_bdf->polyDerivCoefficient(0) );
```

Then, to compute the transient state, which means time dependant terms, you have to initialize the temperature (which is initialized as T_{amb} on X_h space) and create a new vector F_t which corresponds to the time dependent term. The code is as follow :

```
T = vf::project( _space=Xh, _expr=cst(Tamb) );
M_bdf->initialize(T);
auto Ft = M_backend->newVector( Xh );

for ( M_bdf->start(); M_bdf->isFinished()==false; M_bdf->next() )
{
  // update right hand side with time dependent terms
  auto bdf_poly = M_bdf->polyDeriv();
  form1( _test=Xh, _vector=Ft ) =
    integrate( _range=markedelements(mesh, "spreader_mesh"),
              _expr=rho_s*c_s*idv(bdf_poly)*id(v) ) +
    integrate( _range=markedelements(mesh, "fin_mesh"),
              _expr=rho_f*c_f*idv(bdf_poly)*id(v) );

  form1( _test=Xh, _vector=Ft ) +=
    integrate( _range= markedfaces(mesh, "gamma4"),
              _expr= heat_flux*(1-exp(-M_bdf->time()))*id(v) );

  // add contrib from time independent terms
  Ft->add( 1., F );

  // solve
  M_backend->solve( _matrix=D, _solution=T, _rhs=Ft );

  // both average temperatures
  Tav = integrate( _range=markedfaces(mesh, "gamma4"),
                  _expr=(1/surface_base)*idv(T) ).evaluate()(0,0);

  Tgamma1 = integrate( _range=markedfaces(mesh, "gamma1"),
                      _expr=(1/surface_fin)*idv(T) ).evaluate()(0,0);
```

```
// export results
out << M_bdf->time() << " " << Tavg << " " << Tgamma1 << "\n";

this->exportResults( M_bdf->time(), T );

}
```

5.3.4 Outputs

As you can see in the equation's implementation above, there are two outputs :

- GMSH format : this file contains the entire mesh and the temperatures associated to each degrees of freedom of the mesh. To open it, you just have to do as you always do with GMSH : `gmsheatsink-1_0.msh`. You will obtain the figure with the different temperatures, you are now able to click on "play" with its significative logo and admire the evolution
- averages file : this file is completed at each time step, each line contains the current time, the average temperature on Γ_4 (surface where is the contact between the heat sink and the heat source) and the average temperature on Γ_1 . To analyze this file, we recommend you to work with OCTAVE which is an open-source software similar to MATLAB. If it is installed, open a command line and go to `~/feel/heatsink/Simplex_*.*/0.000*/` and try :

```
> octave
octave:1> M=load('averages');
octave:2> plot(M(:,1),M(:,2))
octave:3> plot(M(:,1),M(:,3))
octave:4> plot(M(1:70,1),M(1:70,2))
octave:5> plot(M(1:70,1),M(1:70,3))
```

The 4th and 5th lines are here to observe the transient state.

5.4 Use cases

5.4.1 How to use it ?

To make easier the use of this application, we recommend you to use the configurations files. This is the fastest way : to do it, you just have to create the file `heatsink.cfg` and place it in the same directory that your application's executable.

We have created 3 typical `cfg` files such as :

```
# file heatsink_1.cfg
# spreader and fin in copper
# 2D simulation
hsize=1e-4

kappa_s=386 # W/m/K
c_s=385
rho_s=8940

kappa_f=386 # W/m/K
c_f=385 # J/kg/K
rho_f=8940

L=15e-3
width=5e-4

therm_coeff=1000 #W/ (m2K)
heat_flux=1e6

[bdf]
order=2
time-step=0.05
time-final=100
steady=0
```

```
[exporter]
format=gmsb

# file heatsink_3.cfg
# spreader in copper
# fin in aluminium
# 3D simulation
hsize=3e-4
kappa_s=386 # W/m/K
c_s=385
rho_s=8940

kappa_f=386 # W/m/K
c_f=385 # J/kg/K
rho_f=8940

L=15e-3
width=5e-4
deep=4e-2

therm_coeff=1000 #W/(m2K)
heat_flux=1e6

[bdf]
order=2
time-step=0.05
time-final=100
steady=0

[exporter]
format=gmsb
```

This file is the only modification you will have to bring to the application, in that way you won't have to compile each time the files (except for `heatsink.cpp` if you want to increase the order and/or the dimension, in that case you'll have to modify this parameter at then end of the file in the `main` method).

5.4.2 Results

2D cases

Here are some results of the 2D simulations considering different configurations files. The figures have been extracted thanks to GMSH and OCTAVE :

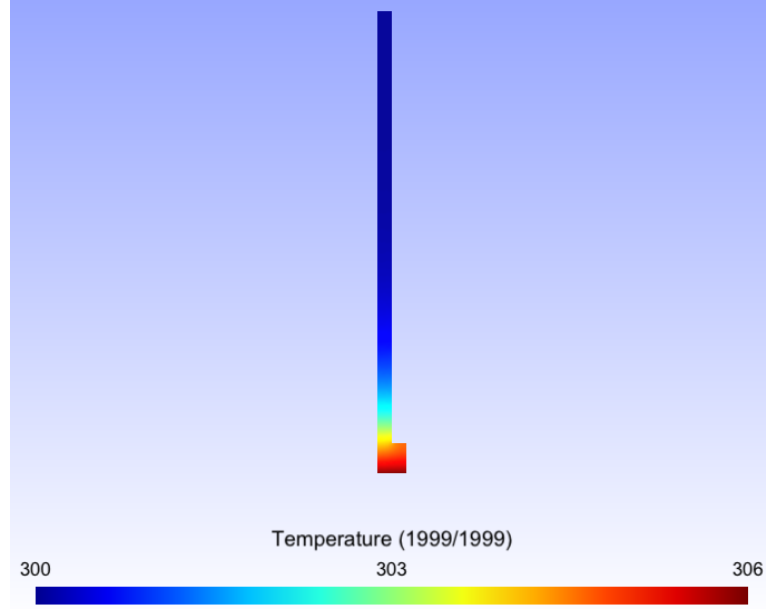


Figure 5.7: heatsink_1.cfg : steady state, spreader and fin in copper, $Q = 1e6$ and $h = 1e3$

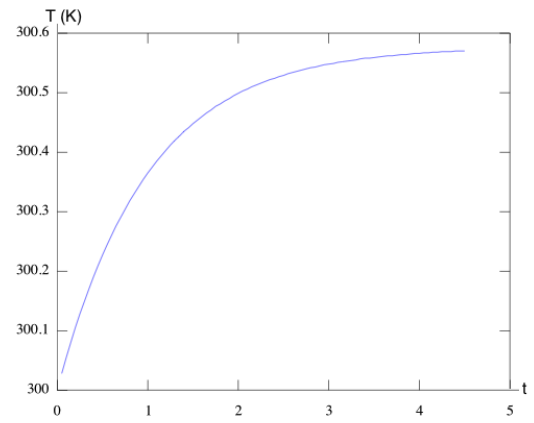
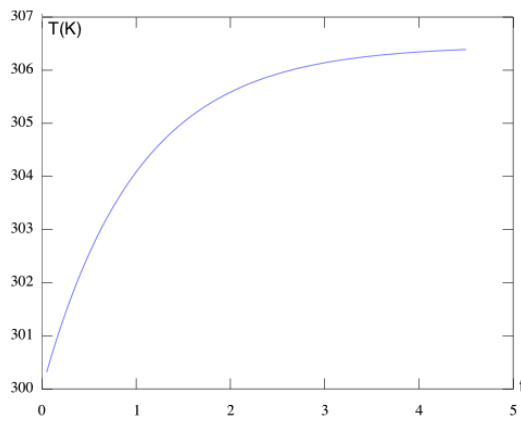


Figure 5.8: heatsink_1.cfg : transient state on Γ_4 Figure 5.9: heatsink_1.cfg : transient state on Γ_1

3D cases

Here is the result of 3D simulations considering the following configurations :

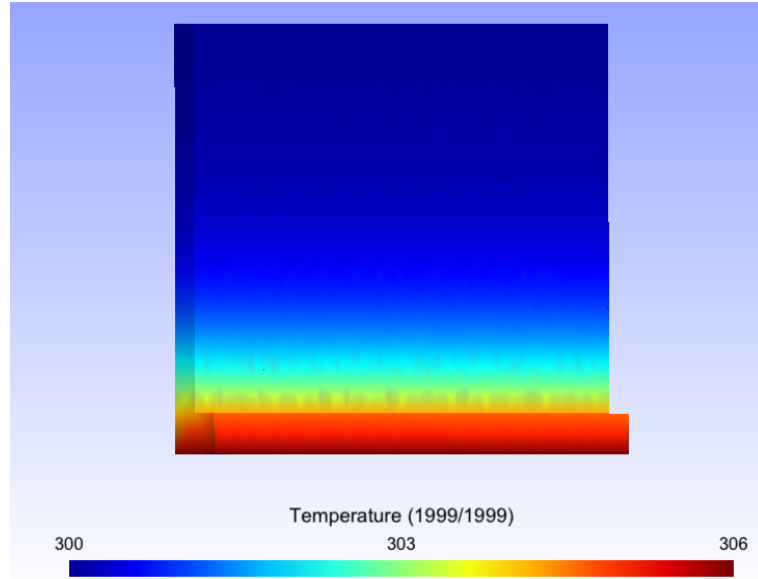


Figure 5.10: heatsink_3.cfg : spreader and fin in copper, $Q = 1e6$ and $h = 1e3$

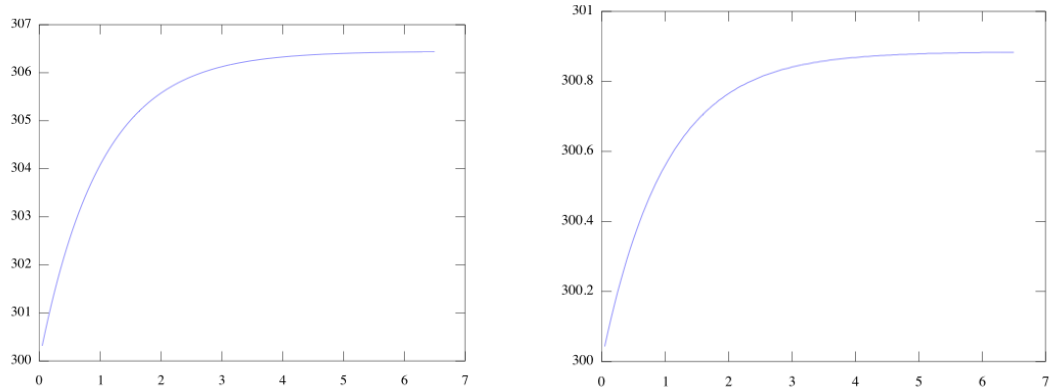


Figure 5.11: heatsink_3.cfg : transient state on Γ_4

Γ_1

CHAPTER 6

Natural convection in a heated tank

By Christophe Prud'homme

Chapter ref: [cha:natural-convection-2d]

6.1 Description

The goal of this project is to simulate the fluid flow under natural convection: the heated fluid circulates towards the low temperature under the action of density and gravity differences. This phenomenon is important in the sense it models evacuation of heat, generated by friction forces for example, with a cooling fluid.

We shall put in place a simple convection problem in order to study the phenomenon without having to handle the difficulties of more complex domains. We describe then some necessary transformations to the equations, then we define quantities of interest to be able to compare the simulations with different parameter values.

To study the convection, we use a model problem: it consists in a rectangular tank of height 1 and width W , in which the fluid is enclosed, see figure 6.1. We wish to know the fluid velocity \mathbf{u} , the fluid pressure p and fluid temperature θ .

We introduce the adimensionalized Navier-Stokes and heat equations parametrized by the Grashof and Prandtl numbers. These parameters allow to describe the various regimes of the fluid flow and heat transfer in the tank when varying them.

The adimensionalized steady incompressible Navier-Stokes equations reads:

$$\begin{aligned} \mathbf{u} \cdot \nabla \mathbf{u} + \nabla p - \frac{1}{\sqrt{\text{Gr}}} \Delta \mathbf{u} &= \theta \mathbf{e}_2 \\ \nabla \cdot \mathbf{u} &= 0 \text{ sur } \Omega \\ \mathbf{u} &= \mathbf{0} \text{ sur } \partial\Omega \end{aligned} \tag{6.1}$$

where Gr is the Grashof number, \mathbf{u} the adimensionalized velocity and p adimensionalized pressure and θ the adimensionalized temperature. The temperature is in fact the difference between the temperature in the tank and the temperature T_0 on boundary Γ_1 .

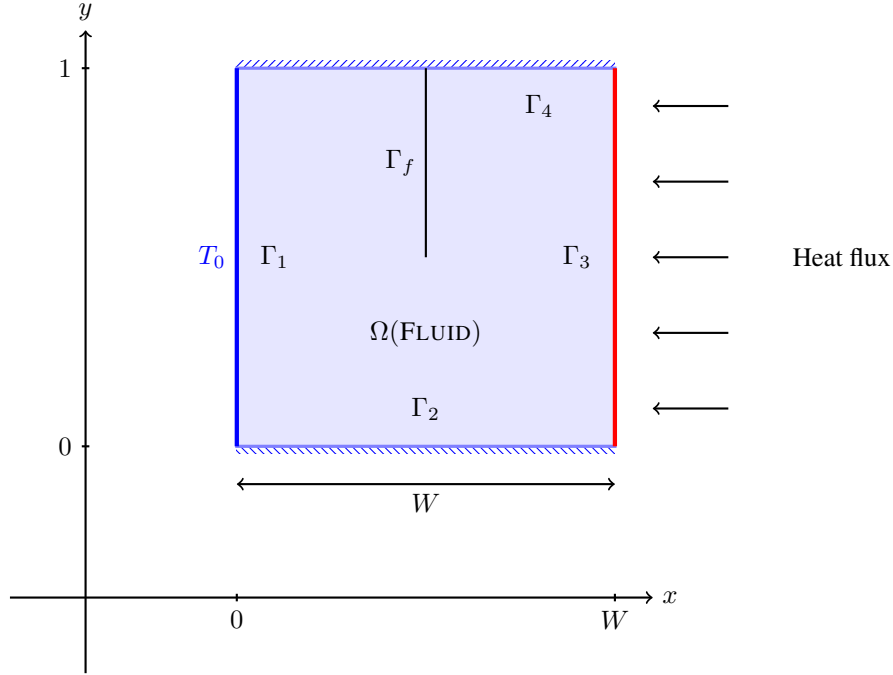


Figure 6.1: Geometry of the model

The heat equation reads:

$$\begin{aligned}
 \mathbf{u} \cdot \nabla \theta - \frac{1}{\sqrt{\text{GrPr}}} \Delta \theta &= 0 \\
 \theta &= 0 \text{ sur } \Gamma_1 \\
 \frac{\partial \theta}{\partial n} &= 0 \text{ sur } \Gamma_{2,4} \\
 \frac{\partial \theta}{\partial n} &= 1 \text{ sur } \Gamma_3
 \end{aligned} \tag{6.2}$$

where Pr is the Prandtl number.

6.2 Influence of parameters

what are the effects of the Grashof and Prandtl numbers ? We remark that both terms with these parameters appear in front of the Δ parameter, they thus act on the diffusive terms. If we increase the Grashof number or the Prandtl number the coefficients multiplying the diffusive terms decrease, and this the convection, that is to say the transport of the heat via the fluid, becomes dominant. This leads also to a more difficult and complex flows to simulate, see figure 6.2. The influence of the Grashof and Prandtl numbers are different but they generate similar difficulties and flow configurations. Thus we look only here at the influence of the Grashof number which shall vary in $[1, 1e7]$.

6.3 Quantities of interest

We would like to compare the results of many simulations with respect to the Grashof defined in the previous section. We introduce two quantities which will allow us to observe the behavior of the flow and heat transfer.

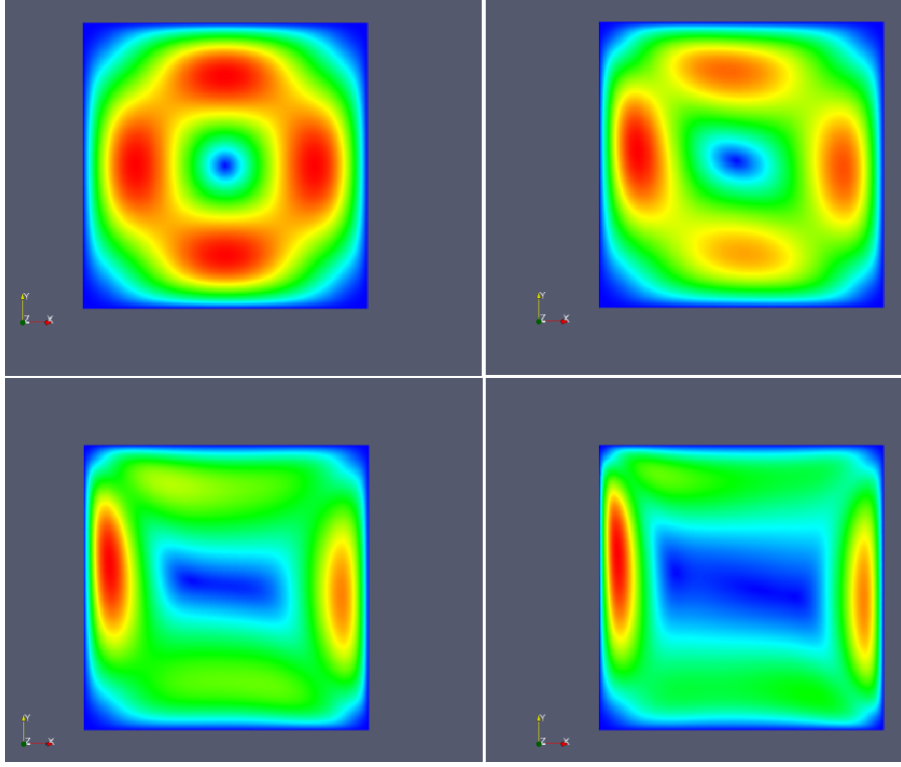


Figure 6.2: Velocity norm with respect to Grashof, $Gr = 100, 10000, 100000, 500000$. $h = 0.01$ and $Pr = 1$.

6.3.1 Mean temperature

We consider first the mean temperature on boundary Γ_3

$$T_3 = \int_{\Gamma_3} \theta \quad (6.3)$$

This quantity should decrease with increasing Grashof because the fluid flows faster and will transport more heat which will cool down the heated boundary Γ_3 . We observe this behavior on the figure 6.3.

6.3.2 Flow rate

Another quantity of interest is the flow rate through the middle of the tank. We define a segment Γ_f as being the vertical top semi-segment located at $W/2$ with height $1/2$, see figure 6.1. The flow rate, denoted D_f , reads

$$D_f = \int_{\Gamma_f} \mathbf{u} \cdot \mathbf{e}_1 \quad (6.4)$$

where $\mathbf{e}_1 = (1, 0)$. Note that the flow rate can be negative or positive depending on the direction in which the fluid flows.

As a function of the Grashof, we shall see a increase in the flow rate. This is true for small Grashof, but starting at $1e3$ the flow rate decreases. The fluid is contained in a boundary layer which is becoming smaller as the Grashof increases.

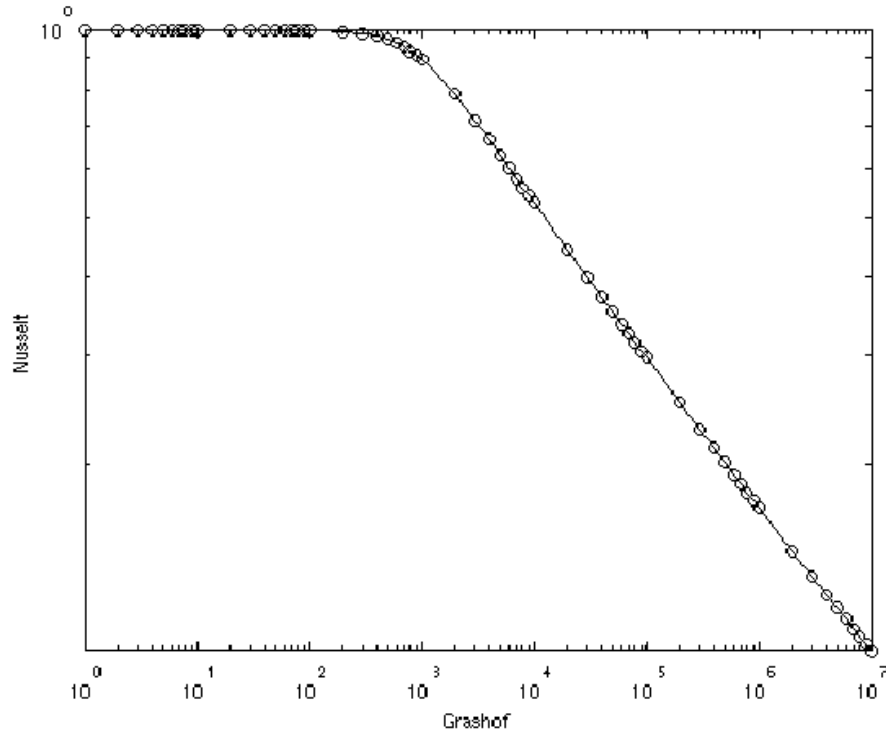


Figure 6.3: Mean temperature with respect to the Grashof number; $h = 0.02$ with \mathbb{P}_3 Lagrange element for the velocity, \mathbb{P}_2 Lagrange for the pressure and \mathbb{P}_1 Lagrange for the temperature.

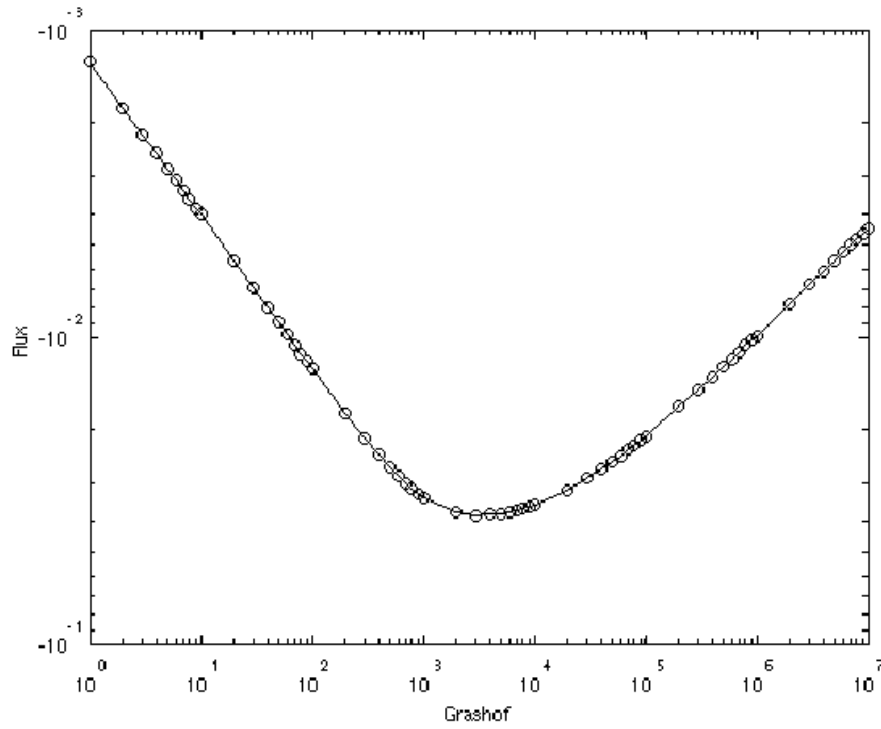


Figure 6.4: Behavior of the flow rate with respect to the Grashof number; $h = 0.02$, \mathbb{P}_3 for the velocity, \mathbb{P}_2 for the pressure and \mathbb{P}_1 for the temperature.

6.4 Implementation

This application is implemented in `feel/doc/manual/convection*.cpp`. The implementation solve the full nonlinear problem using the nonlinear solver framework.

6.5 Numerical Schemes

6.5.1 Stokes problem formulation and the pressure

6.5.2 The Stokes problem

Consider the following problem,

$$\text{Stokes: } \begin{cases} -\mu\Delta \mathbf{u} + \nabla p = \mathbf{f} \\ \nabla \cdot \mathbf{u} = 0 \\ \mathbf{u}|_{\partial\Omega} = 0 \end{cases} \quad (6.5)$$

where $\Omega \subset \mathbb{R}^d$. There are no boundary condition on the pressure. This problem is ill-posed, indeed we only control the pressure through its gradient ∇p . Thus if (\mathbf{u}, p) is a solution, then $(\mathbf{u}, p + c)$ is also a solution with c any constant. This comes from the way the problem is posed: the box is closed and it is not possible to determine the pressure inside. The remedy is to impose arbitrarily a constraint on the pressure, e.g. its mean value is zero. In other words, we add this new equation to the problem (6.5)

$$\int_{\Omega} p = 0 \quad (6.6)$$

Remark 1 (The Navier-Stokes case) *This is also true for the incompressible Navier-Stokes equations. We chose Stokes to simplify the exposure.*

6.5.3 Reformulation

In order to impose the condition (6.6), we introduce a new unknown, a Lagrange multiplier, $\lambda \in \mathbb{R}$ and modify the incompressibility equation. Our problem reads now, find (\mathbf{u}, p, λ) such that

$$\text{Stokes 2: } \begin{cases} -\mu\Delta \mathbf{u} + \nabla p &= \mathbf{f} \\ \nabla \cdot \mathbf{u} + \lambda &= 0 \\ \mathbf{u}|_{\partial\Omega} &= 0 \\ \int_{\Omega} p &= 0 \end{cases} \quad (6.7)$$

Remark 2 (The pressure as Lagrange multiplier) *The pressure field p can actually be seen as a Lagrange multiplier for the velocity \mathbf{u} in order to enforce the constraint $\nabla \cdot \mathbf{u} = 0$. λ will play the same role but for the pressure to enforce the condition (6.6). As $h \rightarrow 0$, $\lambda \rightarrow 0$ as well as the divergence of \mathbf{u} . Note also that $\int_{\Omega} \nabla \cdot \mathbf{u} \approx -\int_{\Omega} \lambda$ from the second equation.*

6.5.4 Variational formulation

The variational formulation now reads: find $(\mathbf{u}, p, \lambda) \in \mathbf{H}_0^1(\Omega) \times L_0^2(\Omega) \times \mathbb{R}$ such that for all $(\mathbf{v}, q, \eta) \in \mathbf{H}_0^1(\Omega) \times L_0^2(\Omega) \times \mathbb{R}$

$$\text{Stokes 3: } \begin{cases} \int_{\Omega} (\nabla \mathbf{u} : \nabla \mathbf{v} + \nabla \cdot \mathbf{v} p) &= \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \\ \int_{\Omega} (\nabla \cdot \mathbf{u} q + \lambda q) &= 0 \\ \int_{\Omega} p \eta &= 0 \end{cases} \quad (6.8)$$

Summing up all three equations we get the following condensed formulation:

$$\int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} + \nabla \cdot \mathbf{v} p + \nabla \cdot \mathbf{u} q + \lambda q + \eta p = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \quad (6.9)$$

where $\mathbf{H}_0^1(\Omega) = \left\{ \mathbf{v} \in \mathbf{L}^2(\Omega), \nabla \mathbf{v} \in [L^2(\Omega)]^{d \times d}, \mathbf{v} = 0 \text{ on } \partial\Omega \right\}$, $L_0^2(\Omega) = \left\{ v \in L^2(\Omega), \int_{\Omega} v = 0 \right\}$, and $\mathbf{L}^2(\Omega) = \left\{ \mathbf{v} \in [L^2(\Omega)]^d \right\}$ that is to say each component of a vector field of $\mathbf{L}^2(\Omega)$ are in $L^2(\Omega)$.

6.5.5 Implementation

```

/*basis*/
typedef Lagrange<Order, Vectorial> basis_u_type; // velocity
typedef Lagrange<Order-1, Scalar> basis_p_type; // pressure
typedef Lagrange<0, Scalar> basis_l_type; // multipliers
typedef bases<basis_u_type, basis_p_type, basis_l_type> basis_type;
/*space: product of the velocity, pressure and multiplier spaces*/
typedef FunctionSpace<mesh_type, basis_type, value_type> space_type;
// ...
space_ptrtype Xh = space_type::New( mesh );
element_type U( Xh, "u" );
element_type V( Xh, "v" );
element_0_type u = U.element<0>();
element_0_type v = V.element<0>();
element_1_type p = U.element<1>();
element_1_type q = V.element<1>();
element_2_type lambda = U.element<2>();
element_2_type nu = V.element<2>();
// ...
sparse_matrix_ptrtype D( M_backend->newMatrix( Xh, Xh ) );
form2( Xh, Xh, D, _init=true )=
    integrate( elements(mesh), im,
        // ∇u: ∇v
        mu*trace(deft*trans(def))
        // ∇ · vp + ∇ · uq
        - div(v)*idt(p) + divt(u)*id(q)
        // λq + ηp
        +id(q)*idt(lambda) + idt(p)*id(nu) );
// ...
    
```

6.5.6 Fix point iteration for Navier-Stokes

Steady incompressible Navier-Stokes equations

Consider the following steady incompressible Navier-Stokes equations, find (\mathbf{u}, p) such that

$$\underbrace{\rho \mathbf{u} \cdot \nabla \mathbf{u}}_{\text{convection}} - \underbrace{\nu \Delta \mathbf{u}}_{\text{diffusion}} + \nabla p = \mathbf{f} \text{ on } \Omega \quad (6.10)$$

$$\nabla \cdot \mathbf{u} = 0$$

$$\mathbf{u} = \mathbf{0} \text{ on } \partial\Omega$$

where ρ is the density of the fluid, ν is the dynamic viscosity of the fluid (la viscosité cinématique $\eta = \nu/\rho$) and \mathbf{f} is the external force density applied to the fluid, (e.g. $\mathbf{f} = -\rho g \mathbf{e}_2$ with $\mathbf{e}_2 = (0, 1)^T$). This equation system is nonlinear due to the $\mathbf{u} \cdot \nabla \mathbf{u}$ convection term. A simple approach to solve (6.10) is to use a fix point algorithm.

The fixpoint algorithm for NS reads as follows, find $(\mathbf{u}^{(k)}, p^{(k)})$ such that

$$\rho \mathbf{u}^{(k-1)} \cdot \nabla \mathbf{u}^{(k)} - \nu \Delta \mathbf{u}^{(k)} + \nabla p^{(k)} = \mathbf{f} \text{ on } \Omega$$

$$\nabla \cdot \mathbf{u}^{(k)} = 0$$

$$\mathbf{u}^{(k)} = \mathbf{0} \text{ on } \partial\Omega \quad (6.11)$$

$$(\mathbf{u}^{(0)}, p^{(0)}) = (\mathbf{0}, 0)$$

The system (6.11) is now linear at each iteration k and we can write the variational formulation accordingly. A stopping criterium is for example that $\|\mathbf{u}^k - \mathbf{u}^{(k-1)}\| + \|p^k - p^{(k-1)}\| < \epsilon$ where ϵ is a given tolerance (e.g. $1e-4$) and $\|\cdot\|$ is the L_2 norm.

Here is the implementation using FEEL++:

```
// define some tolerance  $\epsilon$ 
epsilon = 1e-4;
// set  $(\mathbf{u}^{(0)}, p^{(0)})$  to  $(0,0)$ 
velocity_element_type uk(Xh);
velocity_element_type uk1(Xh);
pressure_element_type pk(Ph);
pressure_element_type pk1(Ph);
// by default uk1, uk and pk, pk1 are initialized to 0

// assemble the linear form associated to  $\mathbf{f}$ 
// store in vector  $F$ , it does not change over the iterations

// iterations to find  $(\mathbf{u}^{(k)}, p^{(k)})$ 
do
{
    // save results of previous iterations
    uk1 = uk;
    pk1 = pk;

    //assemble for bilinear form associated to
    //  $\rho \mathbf{u}^{(k-1)} \cdot \nabla \mathbf{u}^{(k)} - \nu \Delta \mathbf{u}^{(k)} + \nabla p^{(k)}$ 
    // store in matrix  $A^{(k)}$ 

    // solve the system  $A^{(k)}X = F$  where  $X = (\mathbf{u}^{(k)}, p^{(k)})^T$ 

    // use uk, uk1 and pk, pk1 to compute the error estimation at each iteration
    error =  $\|\mathbf{u}^k - \mathbf{u}^{(k-1)}\| + \|p^k - p^{(k-1)}\|$ 
} while( error > epsilon );
```

6.5.7 A Fix point coupling algorithm

Coupling fluid flow and heat transfer: problem

Recall that we have to solve two coupled problems :

$$\text{Heat}(\mathbf{u}) \begin{cases} -\kappa \Delta T + \mathbf{u} \cdot \nabla T &= 0 \\ T|_{\Gamma_1} &= T_0 \\ \frac{\partial T}{\partial \mathbf{n}}|_{\Gamma_3} &= 1 \\ \frac{\partial T}{\partial \mathbf{n}}|_{\Gamma_2, \Gamma_4} &= 0 \end{cases}$$

and

$$\text{Stokes}(T) : \begin{cases} -\nu \Delta \mathbf{u} + \frac{1}{\rho} \nabla p = \mathbf{F} \\ \nabla \cdot \mathbf{u} = 0 \\ \mathbf{u}|_{\partial \Omega} = 0 \end{cases}$$

Where \mathbf{F} can be taken as $\begin{pmatrix} 0 \\ \beta(T - T_0) \end{pmatrix}$ for some $\beta > 0$. β is called the *dilatation coefficient*.

Coupling fluid flow and heat transfer: algorithm

Here is a simple algorithm fix point strategy in pseudo-code:

```
double tol = 1.e-6;
int maxIter = 50;
//Initial guess Un = 0
do
{
    // Find Tn solution of Heat(Un)
```

```
// Find Unpl solution of Stokes(Tn)
// compute stopTest = norme(Unpl - Un)
// Un = Unpl
}while((stopTest < tol) && (niter <= maxIter));
```

Remark 3 (The unsteady case) To solve the unsteady problems, one can insert the previous loop in the one dedicated to time discretization

6.5.8 A Newton coupling algorithm

A fully coupled scheme

Another possibility is to use a Newton method which allows us to solve the full nonlinear problem coupling velocity, pressure and temperature

$$\text{Find } X \text{ such that } F(X) = 0 \quad (6.12)$$

the method is iterative and reads, find $X^{(n+1)}$ such that

$$J_F(X^{(n)})(X^{(n+1)} - X^{(n)}) = -F(X^{(n)}) \quad (6.13)$$

starting with $X^{(0)} = \mathbf{0}$ or some other initial value and where J_F is the jacobian matrix of F evaluated at $X = ((u_i)_i, (p_i)_i, (\theta_i)_i)^T$. For any ϕ_k, ψ_l and ρ_m the *test* functions associated respectively to velocity, pressure and temperature, our full system reads, Find $X = ((u_i)_i, (p_i)_i, (\theta_i)_i)^T$ such that

$$\begin{aligned} F_1((u_i)_i, (p_i)_i, (\theta_i)_i) &= \sum_{i,j} u_i u_j a(\phi_i, \phi_k, \phi_j) - \sum_i p_i b(\phi_k, \psi_i) + \sum_i \theta_i c(\rho_i, \phi_k) + \sum_i u_i d(\phi_i, \phi_k) = 0 \\ F_2((u_i)_i, (p_i)_i, (\theta_i)_i) &= \sum_i u_i b(\phi_i, \psi_l) = 0 \\ F_3((u_i)_i, (p_i)_i, (\theta_i)_i) &= \sum_{i,j} u_i \theta_j e(\phi_i, \rho_j, \rho_m) + \sum_i \theta_i f(\rho_i, \rho_m) - g(\rho_m) = 0 \end{aligned} \quad (6.14)$$

where $F = (F_1, F_2, F_3)^T$ and

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}, \beta) &= \int_{\Omega} \mathbf{v}^T ((\nabla \mathbf{u}) \beta) \\ b(\mathbf{v}, p) &= \int_{\Omega} p (\nabla \cdot \mathbf{v}) - \int_{\partial\Omega} \mathbf{v} \cdot \mathbf{n} p \\ c(\theta, \mathbf{v}) &= \int_{\Omega} \theta \mathbf{e}_2 \cdot \mathbf{v} \\ d(\mathbf{u}, \mathbf{v}) &= \frac{1}{\sqrt{\text{Gr}}} \left(\int_{\Omega} \nabla \mathbf{u} : (\nabla \mathbf{v})^T - \int_{\partial\Omega} ((\nabla \mathbf{u}) \mathbf{n}) \cdot \mathbf{v} \right) \\ e(\mathbf{u}, \theta, \chi) &= \int_{\Omega} (\mathbf{u} \cdot \nabla \theta) \chi \\ f(\theta, \chi) &= \frac{1}{\sqrt{\text{GrPr}}} \left(\int_{\Omega} \nabla \theta \cdot \nabla \chi - \int_{\Gamma_1} (\nabla \theta \cdot \mathbf{n}) \chi \right) \\ g(\chi) &= \frac{1}{\sqrt{\text{GrPr}}} \int_{\Gamma_3} \chi \end{aligned} \quad (6.15)$$

Remark 4 Note that the boundary integrals are kept in order to apply the weak Dirichlet boundary condition trick, see next section B.3.

Jacobian matrix

In order to apply the newton scheme, we need to compute the jacobian matrix J_F by deriving each equation with respect to each unknowns, ie u_i, p_i and θ_i . Consider the first equation

- Deriving the first equation with respect to u_i we get

$$\frac{\partial F_1}{\partial u_i} = \sum_j u_j a(\phi_i, \phi_k, \phi_j) + \sum_i u_i a(\phi_i, \phi_k, \phi_j) + d(\phi_i, \phi_k) \quad (6.16)$$

- Deriving the first equation with respect to p_i we get

$$\frac{\partial F_1}{\partial p_i} = -b(\phi_k, \psi_l) \quad (6.17)$$

- Deriving the first equation with respect to θ_i we get

$$\frac{\partial F_1}{\partial \theta_i} = c(\rho_i, \rho_k) \quad (6.18)$$

Consider the second equation, only the derivative with respect to u_i is non zero.

$$\frac{\partial F_2}{\partial u_i} = b(\phi_i, \psi_l) \quad (6.19)$$

Finally the third component

- Deriving with respect to u_i

$$\frac{\partial F_3}{\partial u_i} = \sum_j \theta_j e(\phi_i, \rho_j, \rho_m) \quad (6.20)$$

- Deriving with respect to p_i ,

$$\frac{\partial F_3}{\partial p_i} = 0 \quad (6.21)$$

- Deriving with respect to θ_i ,

$$\frac{\partial F_3}{\partial \theta_i} = \sum_j u_j e(\phi_j, \rho_i, \rho_m) + f(\rho_i, \rho_m) \quad (6.22)$$

$$J_F = \begin{pmatrix} \frac{\partial F_1}{\partial u_i} & \frac{\partial F_1}{\partial p_i} & \frac{\partial F_1}{\partial \theta_i} \\ \frac{\partial F_2}{\partial u_i} & \frac{\partial F_2}{\partial p_i} (= 0) & \frac{\partial F_2}{\partial \theta_i} (= 0) \\ \frac{\partial F_3}{\partial u_i} & \frac{\partial F_3}{\partial p_i} (= 0) & \frac{\partial F_3}{\partial \theta_i} \end{pmatrix} \quad (6.23)$$

In order to implement J_F and solve (6.13), J_F can be expressed as the matrix associated with the discretisation of

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}, \beta_1) + a(\beta_1, \mathbf{v}, \mathbf{u}) + d(\mathbf{u}, \mathbf{v}) - b(\mathbf{v}, p) + c(\theta, \mathbf{v}) &= 0 \\ b(\mathbf{u}, q) &= 0 \\ e(\beta_1, \theta, \chi) + f(\theta, \chi) + e(\mathbf{u}, \beta_2, \chi) &= 0 \end{aligned} \quad (6.24)$$

where $\beta_1 = u^{(n)}$, $\beta_2 = \theta^{(n)}$ are known from the previous Newton iteration, indeed J_F is actually evaluated in $X^{(n)}$.

FEEL++ Implementation

Now we use the FEEL++ non linear framework in order to implement our Newton scheme (6.13). We need to define two new functions in our application

- `updateJacobian(X, J)` which takes as input $X = X^{(n)}$ and returns the matrix $J = J_F(X^{(n)})$
- `updateResidual(X, R)` which takes as input $X = X^{(n)}$ and returns the vector $R = F(X^{(n)})$

Remark 5 *Backend Only the PETSC backend supports the nonlinear solver framework. Use in the command line like in the first section*

```
--backend=petsc
```

Here is a snippet of code that implements the nonlinear framework.

```

class MyApp
{
public:
    void run();
    void updateResidual( const vector_ptrtype& X, vector_ptrtype& R );
    void updateJacobian( const vector_ptrtype& X, sparse_matrix_ptrtype& J);
    void solve( sparse_matrix_ptrtype& D, element_type& u, vector_ptrtype& F );
private:

    backend_ptrtype M_backend;
    sparse_matrix_ptrtype M_jac;
    vector_ptrtype M_residual;
};

void
MyApp::run()
{
    // ...

    // plug the updateResidual and updateJacobian functions
    // in the nonlinear framework
    M_backend->nlSolver()->residual = boost::bind( &self_type::updateResidual,
                                                    boost::ref( *this ), _1, _2 );
    M_backend->nlSolver()->jacobian = boost::bind( &self_type::updateJacobian,
                                                    boost::ref( *this ), _1, _2 );

    vector_ptrtype U( M_backend->newVector( u.functionSpace() ) );
    *U = u;
    vector_ptrtype R( M_backend->newVector( u.functionSpace() ) );
    this->updateResidual( U, R );
    sparse_matrix_ptrtype J;
    this->updateJacobian( U, J );
    solve( J, u, R );

    *U = u;
    this->updateResidual( U, R );
    // R(u) should be small
    std::cout << "R( u ) = " << M_backend->dot( U, R ) << "\n";

}

void
MyApp::solve( sparse_matrix_ptrtype& D, element_type& u, vector_ptrtype& F )
{
    vector_ptrtype U( M_backend->newVector( u.functionSpace() ) );
    *U = u;
    M_backend->nlSolve( D, U, F, 1e-10, 10 );
    u = *U;
}

void
MyApp::updateResidual( const vector_ptrtype& X, vector_ptrtype& R )
{
    // compute R(X)

    R=M_residual;
}

void
MyApp::updateJacobian( const vector_ptrtype& X, vector_ptrtype& R )
{
    // compute J(X)

    J=M_jac;
}

```

see bratu.cpp or nonlinearpow.cpp for example.

CHAPTER 7

2D Maxwell simulation in a diode

By Thomas Strub, Philippe Helluy, Christophe Prud'homme

Chapter ref: [cha:maxwell-2d]

7.1 Description

The Maxwell equations read:

$$\begin{aligned} \frac{-1}{c^2} \frac{\partial \mathbf{E}}{\partial t} + \nabla \times \mathbf{B} &= \mu_0 \mathbf{J} \\ \mathbf{B}_t + \nabla \times \mathbf{E} &= 0 \\ \nabla \cdot \mathbf{B} &= 0 \\ \nabla \cdot \mathbf{E} &= \frac{\rho}{\epsilon_o} \end{aligned}$$

where \mathbf{E} is the electric field, \mathbf{B} the magnetic field, \mathbf{J} the current density, c the speed of light, ρ density of electric charge, μ_0 the vacuum permeability and ϵ_o the vacuum permittivity.

In the midst industrial notament in aeronautics, systems Products must verify certain standards such as the receipt an electromagnetic wave emitted by a radar does not cause the inefficassité of part or all of the hardware in the system.

Thus, the simulation of such situations can develop when or during the certification of a new product to test its reaction to such attacks.

Also note that the last two equations are actually initial conditions, since if we assume they are true at the moment $t = 0$ then it can be deduced from the first two.

At $t = 0s$, we suppose that

$$\nabla \cdot \mathbf{B} = 0 \quad (7.1)$$

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_o} \quad (7.2)$$

Suppose that $\mathbf{B} = (B_x, B_y, B_z)^T$ and $\mathbf{E} = (E_x, E_y, E_z)^T$ i.e.

$$\frac{\partial B_x}{\partial x}(t=0) + \frac{\partial B_y}{\partial y}(t=0) + \frac{\partial B_z}{\partial z}(t=0) = 0 \quad (7.3)$$

$$\frac{\partial E_x}{\partial x}(t=0) + \frac{\partial E_y}{\partial y}(t=0) + \frac{\partial E_z}{\partial z}(t=0) = \frac{\rho}{\epsilon_o} \quad (7.4)$$

Differentiating the first of these two equations with respect to time, we get:

$$\begin{aligned} \frac{\partial}{\partial t} \frac{\partial}{\partial x} B_x + \frac{\partial}{\partial t} \frac{\partial}{\partial y} B_y + \frac{\partial}{\partial t} \frac{\partial}{\partial z} B_z = \\ \frac{\partial}{\partial x} \left(\frac{\partial}{\partial y} E_z - \frac{\partial}{\partial z} E_y \right) + \frac{\partial}{\partial y} \left(\frac{\partial}{\partial z} E_x - \frac{\partial}{\partial x} E_z \right) + \frac{\partial}{\partial z} \left(\frac{\partial}{\partial x} E_y - \frac{\partial}{\partial y} E_x \right) \end{aligned} = 0 \quad (7.5)$$

thanks to

$$\mathbf{B}_t + \nabla \times \mathbf{E} = 0 \quad (7.6)$$

So, for all $t \geq 0$,

$$\nabla \cdot \mathbf{B}(t) = \nabla \cdot \mathbf{B}(0) = 0 \quad (7.7)$$

We deduce the same way the second equation, using the charge conservation equation :

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{J}) = 0 \quad (7.8)$$

7.2 Variational formulation

7.3 Implementation

7.4 Numerical Results

CHAPTER 8

Domain decomposition methods

By Abdoulaye Samake, Vincent Chabannes, Christophe Prud'homme

Chapter ref: [cha:dd]

8.1 A Really Short Introduction

In mathematics, numerical analysis, and numerical partial differential equations, domain decomposition methods solve a boundary value problem by splitting it into smaller boundary value problems on subdomains and iterating to coordinate the solution between the adjacent subdomains. A coarse problem with one or few unknowns per subdomain is used to further coordinate the solution between the subdomains globally.

8.2 A 1D model

We consider the following laplacian boundary value problem

$$\begin{cases} -u''(x) = f(x) & \text{in }]0, 1[\\ u(0) = \alpha, u(1) = \beta \end{cases} \quad (8.1)$$

where $\alpha, \beta \in \mathbb{R}$.

8.2.1 Schwartz algorithms

The schwartz overlapping multiplicative algorithm with dirichlet interface conditions for this problem at n^{th} iteration is given by

$$\begin{cases} -u_1''(x) = f(x) & \text{in }]0, b[\\ u_1^n(0) = \alpha \\ u_1^n(b) = u_2^{n-1}(b) \end{cases} \quad \text{and} \quad \begin{cases} -u_2''(x) = f(x) & \text{in }]a, 1[\\ u_2^n(1) = \beta \\ u_2^n(a) = u_1^n(a) \end{cases} \quad (8.2)$$

where $n \in \mathbb{N}^*$, $a, b \in \mathbb{R}$ and $a < b$.

Let $e_i^n = u_i^n - u$ ($i = 1, 2$), the error at n^{th} iteration relative to the exact solution, the convergence rate is given by

$$\rho = \frac{|e_1^n|}{|e_1^{n-1}|} = \frac{a}{b} \frac{1-b}{1-a} = \frac{|e_2^n|}{|e_2^{n-1}|}. \quad (8.3)$$

8.2.2 Variational formulations

find u such that

$$\begin{aligned} \int_0^b u_1' v' &= \int_0^b f v \quad \forall v && \text{in the first subdomain } \Omega_1 =]0, b[\\ \int_a^1 u_2' v' &= \int_a^1 f v \quad \forall v && \text{in the second subdomain } \Omega_2 =]a, 1[\end{aligned}$$

8.3 A 2 domain overlapping Schwartz method in 2D and 3D

We consider the following laplacian boundary value problem

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = g & \text{on } \partial\Omega \end{cases} \quad (8.4)$$

where $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ and g is the dirichlet boundary value.

8.3.1 Schwartz algorithms

The schwartz overlapping multiplicative algorithm with dirichlet interface conditions for this problem on two subdomains Ω_1 and Ω_2 at n^{th} iteration is given by

$$\begin{cases} -\Delta u_1^n = f & \text{in } \Omega_1 \\ u_1^n = g & \text{on } \partial\Omega_1^{ext} \\ u_1^n = u_2^{n-1} & \text{on } \Gamma_1 \end{cases} \quad \text{and} \quad \begin{cases} -\Delta u_2^n = f & \text{in } \Omega_2 \\ u_2^n = g & \text{on } \partial\Omega_2^{ext} \\ u_2^n = u_1^n & \text{on } \Gamma_2 \end{cases} \quad (8.5)$$

8.3.2 Variational formulations

$$\int_{\Omega_i} \nabla u_i \cdot \nabla v = \int_{\Omega_i} f v \quad \forall v, i = 1, 2.$$

FEEL++ implementation

```
/*
  Implementation of the local problem
*/
template<Expr>
void
localProblem(element_type& u, Expr expr)
{
  // Assembly of the right hand side  $\int_{\Omega} f v$ 
  auto F = M_backend->newVector(Xh);
  form1( _test=Xh, _vector=F, _init=true ) =
    integrate( elements(mesh), f*id(v) );
  F->close();

  // Assembly of the left hand side  $\int_{\Omega} \nabla u \cdot \nabla v$ 
  auto A = M_backend->newMatrix( Xh, Xh );
  form2( _test=Xh, _trial=Xh, _matrix=A, _init=true ) =
    integrate( elements(mesh), gradt(u)*trans(grad(v)) );
  A->close();

  // Apply the dirichlet boundary conditions
  form2( Xh, Xh, A ) +=
    on( markedfaces(mesh, "Dirichlet") , u, F, g );

  // Apply the dirichlet interface conditions
```

```

    form2( Xh, Xh, A ) +=
        on( markedfaces(mesh, "Interface") ,u,F,expr);

// solve the linear system  $Au = F$ 
M_backend->solve(_matrix=A, _solution=u, _rhs=F );
}

unsigned int cpt = 0;
double tolerance = 1e-8;
double maxIterations = 20;
double l2erroru1 = 1.;
double l2erroru2 = 1;
/*
Iteration loop
*/
while( (l2erroru1 + l2erroru2) > tolerance && cpt <= maxIterations)
{
    // call the localProblem on the first subdomain  $\Omega_1$ 
    localProblem(u1, idv(u2));

    // call the localProblem on the first subdomain  $\Omega_2$ 
    localProblem(u2, idv(u1));

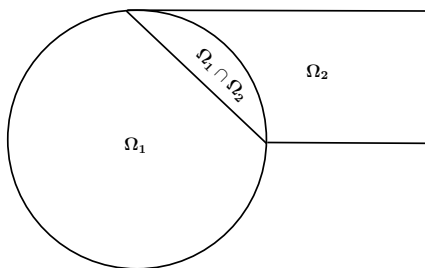
    // compute L2 errors on each subdomain
    L2erroru1 = l2Error(u1);
    L2erroru2 = l2Error(u2);

    // increment the counter
    ++cpt;
}
    
```

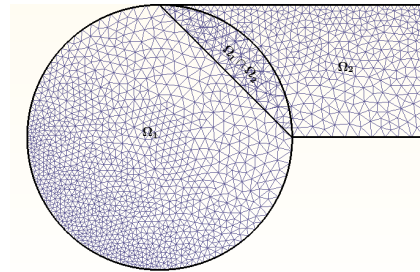
8.3.3 Numerical results in 2D case

The numerical results presented in the following table correspond to the partition of the global domain Ω in two subdomains Ω_1 and Ω_2 (see figure 8.2) and the following configuration:

1. $g(x, y) = \sin(\pi x) \cos(\pi y)$: the exact solution
2. $f(x, y) = 2\pi^2 g$: the right hand side of the equation
3. \mathbb{P}_2 approximation : the lagrange polynomial order
4. $hsize = 0.02$: the mesh size
5. $tol = 1e - 9$: the tolerance



(a) Two overlapping subdomains



(b) Two overlapping meshes

Figure 8.1: geometry

Number of iterations	$\ \mathbf{u}_1 - \mathbf{u}_{\text{ex}}\ _{L_2}$	$\ \mathbf{u}_2 - \mathbf{u}_{\text{ex}}\ _{L_2}$
11	2.52e-8	2.16e-8

8.3.4 Numerical solutions in 2D case

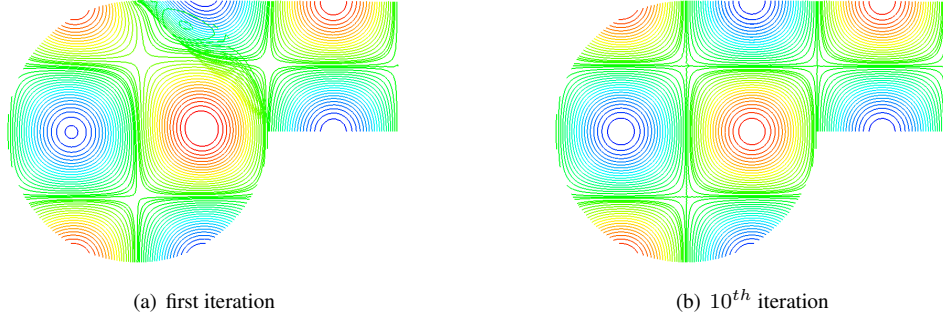


Figure 8.2: isovalues of solution in 2D

8.4 Computing the eigenmodes of the Dirichlet to Neumann operator

8.4.1 Problem description and variational formulation

We consider at the continuous level the Dirichlet-to-Neumann(DtN) map on Ω , denoted by DtN_Ω .

Let $u : \Gamma \mapsto \mathbb{R}$,

$$\text{DtN}_\Omega(u) = \kappa \frac{\partial v}{\partial n} \Big|_\Gamma$$

where v satisfies

$$\begin{cases} \mathcal{L}(v) := (\eta - \text{div}(\kappa \nabla))v = 0 & \text{dans } \Omega, \\ v = u & \text{sur } \Gamma \end{cases} \quad (8.6)$$

where Ω is a bounded domain of \mathbb{R}^d ($d=2$ or 3), and Γ its border, κ is a positive diffusion function which can be discontinuous, and $\eta \geq 0$. The eigenmodes of the Dirichlet-to-Neumann operator are solutions of the following eigenvalues problem

$$\text{DtN}_\Omega(u) = \lambda \kappa u \quad (8.7)$$

To obtain the discrete form of the DtN map, we consider the variational form of (8.6). let's define the bilinear form $a : H^1(\Omega) \times H^1(\Omega) \rightarrow \mathbb{R}$,

$$a(w, v) := \int_\Omega \eta w v + \kappa \nabla w \cdot \nabla v.$$

With a finite element basis $\{\phi_k\}$, the coefficient matrix of a Neumann boundary value problem in Ω is

$$A_{kl} := \int_\Omega \eta \phi_k \phi_l + \kappa \nabla \phi_k \cdot \nabla \phi_l.$$

A variational formulation of the flux reads

$$\int_\Gamma \kappa \frac{\partial v}{\partial n} \phi_k = \int_\Omega \eta v \phi_k + \kappa \nabla v \cdot \nabla \phi_k \quad \forall \phi_k.$$

So the variational formulation of the eigenvalue problem (8.7) reads

$$\int_{\Omega} \eta v \phi_k + \kappa \nabla v \cdot \nabla \phi_k = \lambda \int_{\Gamma} \kappa v \phi_k \quad \forall \phi_k. \quad (8.8)$$

Let B be the weighted mass matrix

$$(B)_{kl} = \int_{\Gamma} \kappa \phi_k \phi_l$$

The compact form of (8.8) is

$$Av = \lambda Bv \quad (8.9)$$

FEEL++ implementation

```
// Assembly of the right hand side  $B = \int_{\Gamma} \kappa v w$ 
auto B = M_backend->newMatrix( Xh, Xh );
form2( _test=Xh, _trial=Xh, _matrix=B, _init=true );
BOOST_FOREACH( int marker, flags )
{
    form2( Xh, Xh, B ) +=
        integrate( markedfaces(mesh,marker), kappa*idt(u)*id(v) );
}
B->close();

// Assembly of the left hand side  $A = \int_{\Omega} \eta v w + \kappa \nabla v \cdot \nabla w$ 
auto A = M_backend->newMatrix( Xh, Xh );
form2( _test=Xh, _trial=Xh, _matrix=A, _init=true ) =
    integrate( elements(mesh), kappa*gradt(u)*trans(grad(v)) + nu*idt(u)*id(v) );
A->close();

// eigenvalue solver options
int nev = this->vm()["solvereigen-nev"].template as<int>();
int ncv = this->vm()["solvereigen-ncv"].template as<int>();
// definition of the eigenmodes
SolverEigen<double>::eigenmodes_type modes;
// solve the eigenvalue problem  $Av = \lambda Bv$ 
modes=
    eigs( _matrixA=A,
          _matrixB=B,
          _nev=nev,
          _ncv=ncv,
          _transform=SINVERT,
          _spectrum=SMALLEST_MAGNITUDE,
          _verbose = true );
}
```

8.4.2 Numerical solutions

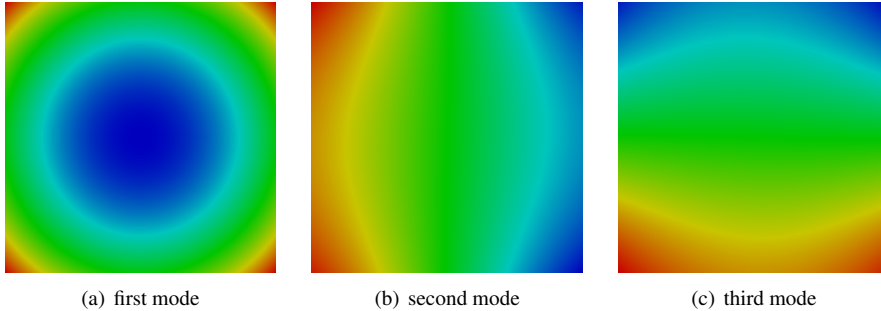


Figure 8.3: three eigenmodes

These numerical solutions correspond to the following configuration :

1. \mathbb{P}_2 approximation : the lagrange polynomial order
2. $hsize = 0.02$: the mesh size
3. $\mu = \kappa = 1$.

Part III

Programming with FEEL++

Part IV

Appendix

APPENDIX A

How to ?

A.1 Introduction

This section includes the FAQ available on [Feel web site](#), if you want to post a question, please visit it and follow the instruction to edit the FAQ.

A.2 Meshes

A.2.1 What are the main execution options of a FEEL++ application ?

Let's consider that your application is named `feelapp`, in that case you can modify the main execution options of your application with

```
! ./feelapp --shape="simplex" --nochdir --exporter-format=gmsht
```

These options are :

- `shape=["simplex", "hypercube"]` which is the shape of the generated mesh
- `nochdir` means that you want the result in the current directory (by default in `~/feel`)
- `exporter-format` enables you to choose the format of mesh results output

A.2.2 How to create a mesh?

Here is an example of how to create a mesh with GMSH generator :

```
mesh_ptrtype mesh =
    createGMSHMesh( _mesh=new mesh_type,
                    _update=MESH_CHECK!MESH_UPDATE_FACES!MESH_UPDATE_EDGES!MESH_RENUMBER,
                    _desc=domain( _name= (boost::format( "%1%-%2%-%3%" ) %"hypercube" %Dim %1).str(),
                    _shape="hypercube",
                    _dim=Dim,
                    _h=meshSize,
                    _xmin=-1.,
                    _xmax=1.,
                    _ymin=-1.,
                    _ymax=1. ) );
```

Here is an example of how to create a mesh with a `.geo` file :

```
mesh_ptrtype mesh =
    createGMSHMesh( _mesh=new mesh_type,
        _update=MESH_CHECK!MESH_UPDATE_FACES!MESH_UPDATE_EDGES!MESH_RENUMBER,
        _desc="???" );
```

A.2.3 What are the different parameters of the function domain() ?

The function `domain()` is located in `feel/feel/feefilters/gmsh.hpp` and enables to generate a simple geometrical domain from required and optional parameters. Its available options are :

- `_name = "string"` gives the prefix of the gmsh geo and mesh files,
- `_shape = "simplex", "hypercube", "ellipsoid"` gives the shape of the domain, it is one of these three possibilities
- `_dim = 1, 2 or 3` gives the topological dimension of the domain. For example if `_dim=2` and `_shape="simplex"` this will produce a triangle
- `_h = real value` gives the characteristic size of the mesh, e.g. `_h=0.1`
- `_xmin = real` gives the minimum x value of the domain for example `_xmin=-1`
- `_xmax = real` gives the maximum x value of the domain for example `_xmax=-1`
- `_ymin = real` gives the minimum x value of the domain for example `_ymin=-1.`
- `_ymax = real` gives the maximum y value of the domain for example `_ymax=1.`

A.2.4 How to loop on the degrees of freedom coordinates of a function ?

Take a look at the example which is in `feel/examples/snippets/dofpoints.cpp`

A.2.5 How to work with specific meshes ?

`loadmesh.cpp`

FEEL++ supports several meshes file formats. It supports essentially Gmsh mesh file format but other are acceptable, with some modifications :

- `medit (.mesh)`

There is a small difference between medit meshes and gmsh ones. The medit reader of Gmsh is able to read medit meshes, the issue comes from markers for areas of the edges where we want to apply different boundary conditions. Gmsh is currently using the Physical Entities (physical line, area, volume). Unfortunately, the medit reader of Gmsh considers the physical flag as null (to go deeper, you can check this part on [Gmsh web site](#)). This option is taken into account in FEEL++, the only modification is to put the optional parameter `physical_are_elementary_regions` as **true** in both functions `createGMSHMesh` and/or `loadGMSHMesh`. We have prepared a simple example which imports a medit mesh with a surface and volume calculation on it. You can find it in `feel/doc/manual/loadmesh.cpp`.

Please note that further medit meshes are presented in example in the directory `/feel/data/medit/`. The geo scripts are those which are produced by FEEL++ when reading those meshes.

- `Stl (.stl)`

You can also use `stl` files, those files are native to the stereolithography CAD software created by 3D Systems. These files describe only the surface geometry of a three dimensional object without any representation of color, texture or other common attributes. You have further examples of such files in `feel/data/stl`.

To use FEEL++ with `stl` files, you have to create a `geo` script to enable gmsh to remesh the file. The `stl` file you want to use has to be a volume mesh. The script is very small, you have all informations to make one at [Gmsh/stl section](#) on their web site. Once it's done, you just have to type

```
gmsht stl_file_name.geo -3
```

with `stl_file_name.stl` in the same directory. That command will produce you the correct `.msh` mesh that you could now use as usual without any modification in your FEEL++ application.

Take a look above how the remesh has produced a complete mesh with the file `pelvis.stl` and `pelvis.geo`:

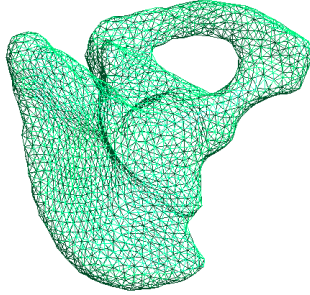


Figure A.1: Pelvis before remesh (stl)

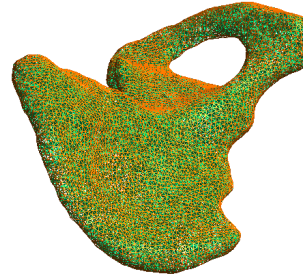


Figure A.2: Pelvis after remesh (msh)

A.3 Language for Partial Differential Equations

A.3.1 What is the difference between using the "vf::project" function and solve a weak projection problem ?

To make it clear, let's considerate that we want to project a \mathbb{P}_1 scalar function σ on a \mathbb{P}_0 space. We have two alternatives to do it :

- Computing the \mathcal{L}_2 projection of σ onto the space

Here $kappa$ and v are \mathbb{P}_0 functions :

```
Matrix_M=integrate(elements(mesh), idt(kappa) * id(v));
Vector_F=integrate(elements(mesh), idv(sigma) * id(v));
```

- Use the project function `vf::project`

This function does a nodal projection : at the dof point the projection will be exactly equal to the projected function σ . It works as follow

```
kappa=vf::project(P0_space, elements(mesh), idv(sigma));
```

These two projections are in general different, if you compare the values in the vector, they will be (slightly) different. However as $h \rightarrow 0$ they should both converge to the σ function.

A.3.2 How to do a quick L2 projection of an expression ?

Let say that we have created two spaces, one scalar and one vectorial, we call them X_h and X_{hVec} and one wants to project some expressions on those spaces.

For example, we want to project $(x, y) \rightarrow \sqrt{x^2 + y^2} - 1$ on the scalar space and $(-2y, \cos x)$ on the vectorial space. First of all, one has to create projectors for the scalar and vectorial spaces, the code reads as follow :

```
#include <feel/feeldiscr/projector.hpp>
auto l2p = projector(Xh, Xh);
auto l2pVec = projector(XhVec, XhVec);
```

You can note that `projector(Space, Space)` returns a `boost::shared_ptr` on a `Projector` object which makes projecting functions on `Space` possible.

Then, one uses the function `Projector::project(Expression)` :

```
auto Circle = l2p->project( sqrt( pow((vf::Px()),2.0)+ pow((vf::Py()),2.0)) - 1 );
auto F = l2pVec->project( -2 * Py() * oneX() + cos(vf::Px()) * oneY() );
```

Here you can note that the types of `Circle` and `F` are respectively : $X_h\text{-type} :: \text{element_type}$ and $X_{hVec}\text{-type} :: \text{element_type}$

An equivalent way to write it is to use the `Projector::operator()` (`Expression`) :

```
auto Circle = (*l2p)( sqrt( pow((vf::Px()),2.0)+ pow((vf::Py()),2.0)) - 1 );
auto F = (*l2pVec)( -2 * Py() * oneX() + cos(vf::Px()) * oneY() );
```

`Projector::operator()` accepts many types of arguments, see `feel/feeldiscr/projector.hpp` for details.

A.3.3 How to compose FEEL++ operators ?

Let's considerate that we have created two spaces, one scalar X_h and one vectorial X_{hVec} . We also have two vectors a and b (of type $X_h\text{-type} :: \text{element_type}$).

One wants to do the following operation : $\text{div}(\text{grad}(a * b))$. The following expression is **not** yet implemented in FEEL++:

```
divv( gradv( idv(a) * idv(b) ) )
```

One has to do intermediate projections to compose the operators. Using the `Projector` class, the code reads :

```
#include <feel/feeldiscr/projector.hpp>

// create projectors on Xh and XhVec spaces
auto l2p = projector(Xh, Xh);
auto l2pVec = projector(XhVec, XhVec);

auto ab = l2p->project( idv(a)*idv(b) );
auto grad_ab = l2pVec->project( gradv(ab) );
auto div_grad_ab = l2p->project( divv(grad_ab) );
```

Here `div_grad_ab` has the type $X_h\text{-type} :: \text{element_type}$. There is an equivalent but verboseless way to write this composition : use the `Projector::operator()` which accepts has argument an expression or an `element_type`. So one could write :

```
#include <feel/feeldiscr/projector.hpp>

//create projectors on Xh and XhVec spaces
auto l2p = projector(Xh, Xh);
auto l2pVec = projector(XhVec, XhVec);

auto div_grad_ab = (*l2p)( divv( (*l2pVec)( gradv( (*l2p)( idv(a)*idv(b)) ) ) ) ) );
```

the `*` is needed before `l2p` or `l2pVec` since there are `boost::shared_ptr` objects. One could also create directly `Projector` objects :

```
Projector<Xh_type, Xh_type> l2p(Xh, Xh);

auto ab = l2p(idv(a)*idv(b));
```


APPENDIX B

Random notes

B.1 Becoming a Feel++ developer

B.1.1 Interest

Becoming a FEEL++ developer makes library improvements possible, you may have several proposals which may be usefull. Taking part of the project will enable you to commit some modifications or new applications, we will be glad to count you among us. As an open-source project under GNU licence, you will be able to commit and participate to the entire project and its various aspects. Our aim is that each user should be involved in the library's expansion. In the following part, you will see how you can become a FEEL++ collaborator.

B.1.2 Creating RSA keys

At the top of the manual, you have seen how to get the sources anonymously, if you want to checkout or commit properly, you will need an account on [Forge](#). After the administrator approval, you have to demand the rights to see the project tree.

Once it's done, you will have to create RSA keys to be able to connect to the server using ssh. To do that you have to type the commands : `ssh-keygen` and accept the 3 questions without typing anything. The generated key is placed in `~/.ssh/id_rsa.pub`, you just need to copy this file's content in your forge account. To make it, go on the Forge website and enter into your account's personnal page. At the bottom of the page, you'll have the possibility to edit your SSH keys, go into it and copy/paste the `id_rsa.pub` content. Once it's done, the number of your SSH keys in that page should have increased. Now, you will be able to connect to the server within an hour.

Important : If you don't have the same login on your computer as on Forge, you must add the commands in the `~/.ssh/config` file :

```
host forge.imag.fr
  user <your_login_forge>
```

B.1.3 Downloading the sources

To be able to download the FEEL++ sources, you need subversion and SSH > 1.xxx installed on your computer. In a command prompt, go where you want FEEL++ to be downloaded and type the following

command :

```
svn co svn+ssh://login@scm.forge.imag.fr/var/lib/gforge/chroot/scmrepos  
/svn/life/trunk/life/trunk feel
```

where login is your login name in the Forge platform.

You are now able to checkout, commit or add the file your judge usefull using svn, please don't forget to comment on your various actions. The first commit is subject to the approbation of one of the main developers.

B.2 Linear Algebra with PETSC

B.2.1 Using the Petsc Backend: recommended

Using the Petsc backend is recommended. To do that type in the command line

```
myprog --backend=petsc
```

then you can change the type of solvers and preconditioners by adding Petsc options at *the end of the command lines*, for example

```
-pc_type lu
```

will actually solve the problem in one iteration of an iterative solver (p.ex. gmres).

$$PAx = PB \tag{B.1}$$

where $P \approx A^{-1}$. Here A is decomposed in LU form and (B.1) is solved in one iteration.

B.2.2 List of solvers and preconditioners

List of some iterative solvers (Krylov subspace)

- cg, bicg
- gmres, fgmres, lgmres
- bcgs, bcgsl
- see `petsc/petscksp.h` for more

List of some preconditioners

- lu, choleski
- jacobi, sor
- ilu, icc
- see `petsc/petscpc.h` for more

B.2.3 What is going on in the solvers?

In order to monitor what is going on (iterations, residual...) Petsc provides some monitoring options

```
-ksp_monitor
```

For example

```
myprog --backend=petsc -ksp_monitor -pc_type lu
```

it should show only one iteration.

See <http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manualpages/KSP/KSPMonitorSet.html> for more details

B.3 Weak Dirichlet boudary conditions

B.3.1 Basic idea

Weak treatment

In order to treat the boundary conditions uniformly (i.e. the same way as Neumann and Robin Conditions), we wish to treat the Dirichlet BC (e.g. $u = g$) weakly.

Remark 6 *Initial Idea add the penalisation term $\int_{\partial\Omega} \mu(u - g)$ where μ is a constant. But this is not enough, this is not consistent with the initial formulation.*

One can use the Nitsche “trick” to implement weak Dirichlet conditions.

- write the equations in conservative form (i.e. identify the flux);
- add the terms to ensure consistency (i.e the flux on the boundary);
- symmetrize to ensure adjoint consistency;
- add a penalisation term with factor $\gamma(u - g)/h$ that ensures that the solution will be set to the proper value at the boundary;

Penalisation parameter

Remark 7 *Choosing γ γ must be chosen such that the coercivity(or inf-sup) property is satisfied. Difficult to do in general. Increase γ until the BC are properly satisfied, e.g. start with $\gamma = 1$, typical values are between 1 and 10.*

The choice of γ is a problem specially when h is small.

Advantages, disadvantages

Remark 8 *Weak treatment: Advantages*

- uniform(weak) treatment of all boundary conditions type
- if boundary condition is independant of time, the terms are assembled once for all
- the boundary condition is not enforced exactly but the convergence order remain optimal

Remark 9 *Weak treatment: Disadvantages*

- Introduction of the penalisation parameter γ that needs to be tweaked

Advantages, disadvantages

Remark 10 *Strong treatment: Advantages*

- Enforce exactly the boundary conditions

Remark 11 *Strong treatment : Disadvantages*

- Need to modify the matrix once assembled to reflect that the Dirichlet degree of freedom are actually known. Then even if the boundary condition is independant of time, at every time step if there are terms depending on time that need reassembly (e.g. convection) the strong treatment needs to be reapplied.
- it can be expensive to apply depending on the type of sparse matrix used, for example using CSR format setting rows to 0 except on the diagonal to 1 is not expensive but one must do that also for the columns associated with each Dirichlet degree of freedom and that is expensive.

B.3.2 Laplacian

Example: Laplacian

$$-\Delta u = f(\text{non conservative}), \quad -\nabla \cdot (\nabla u) = f(\text{conservative}), \quad u = g|_{\partial\Omega} \quad (\text{B.2})$$

the flux is vector ∇u

$$\int_{\Omega} \nabla u \cdot \nabla v + \int_{\partial\Omega} \underbrace{-\frac{\partial u}{\partial n} v}_{\text{integration by part}} \underbrace{-\frac{\partial v}{\partial n} u}_{\text{adjoint consistency: symetrisation}} + \underbrace{\frac{\gamma}{h} uv}_{\text{penalisation: enforce Dirichlet condition}} \quad (\text{B.3})$$

$$\int_{\Omega} f \nabla v + \int_{\partial\Omega} \left(\underbrace{-\frac{\partial v}{\partial n} g}_{\text{adjoint consistency}} + \underbrace{\frac{\gamma}{h} v g}_{\text{penalisation: enforce Dirichlet condition}} \right) \quad (\text{B.4})$$

Example: Laplacian

```
// bilinear form (left hand side)
form2( Xh, Xh, D ) +=
integrate( boundaryfaces(mesh), im_type(),
    -(gradt(u)*N())*id(v) // integration by part
    -(grad(v)*N())*idt(u) // adjoint consistency
    +gamma*id(v)*idt(u)/hFace(); // penalisation
// linear form (right hand side)
form1( Xh, F ) +=
integrate( boundaryfaces(mesh), im_type(),
    -(grad(v)*N())*g // adjoint consistency
    +gamma*id(v)*g/hFace(); // penalisation
```

B.3.3 Convection-Diffusion

Example: Convection-Diffusion

Remark 12 *Convection Diffusion* Consider now the following problem, find u such that

$$-\Delta u + \mathbf{c} \cdot \nabla u = f, \quad u = g|_{\partial\Omega}, \quad \nabla \cdot \mathbf{c} = 0 \quad (\text{B.5})$$

under conservative form the equation reads

$$\nabla \cdot (-\nabla u + \mathbf{c}u) = f, \quad u = g|_{\partial\Omega}, \quad \nabla \cdot \mathbf{c} = 0 \quad (\text{B.6})$$

the flux vector field is $\mathbf{F} = -\nabla u + \mathbf{c}u$. Note that here the condition, $\nabla \cdot \mathbf{c} = 0$ was crucial to expand $\nabla \cdot (\mathbf{c}u)$ into $\mathbf{c} \cdot \nabla u$ since

$$\nabla \cdot (\mathbf{c}u) = \mathbf{c} \cdot \nabla u + \underbrace{u \nabla \cdot \mathbf{c}}_{=0} \quad (\text{B.7})$$

Weak formulation for convection diffusion

Multiplying by any test function v and integration by part of (B.6) gives

$$\int_{\Omega} \nabla u \cdot \nabla v + (\mathbf{c} \cdot \nabla u)v + \int_{\partial\Omega} (\mathbf{F} \cdot \mathbf{n})v = \int_{\Omega} f v \quad (\text{B.8})$$

where \mathbf{n} is the outward unit normal to $\partial\Omega$. We now introduce the penalisation term that will ensure that $u \rightarrow g$ as $h \rightarrow 0$ on $\partial\Omega$. (B.8) reads now

$$\int_{\Omega} \nabla u \cdot \nabla v + (\mathbf{c} \cdot \nabla u)v + \int_{\partial\Omega} (\mathbf{F} \cdot \mathbf{n})v + \frac{\gamma}{h} \mathbf{u} \mathbf{v} = \int_{\Omega} f v + \int_{\partial\Omega} \frac{\gamma}{h} \mathbf{g} \mathbf{v} \quad (\text{B.9})$$

Finally we incorporate the symetrisation of the bilinear form to ensure adjoint consistency and hence proper convergence order

$$\int_{\Omega} \nabla u \cdot \nabla v + (\mathbf{c} \cdot \nabla u) v + \int_{\partial\Omega} ((-\nabla u + \mathbf{c}u) \cdot \mathbf{n}) v + ((-\nabla \mathbf{v} + \mathbf{c}\mathbf{v}) \cdot \mathbf{n}) \mathbf{u} + \frac{\gamma}{h} uv = \int_{\Omega} f v + \int_{\partial\Omega} ((-\nabla \mathbf{v} + \mathbf{c}\mathbf{v}) \cdot \mathbf{n}) \mathbf{g} + \frac{\gamma}{h} g v \quad (\text{B.10})$$

Example: Convection-Diffusion

```
// bilinear form (left hand side)
form2( Xh, Xh, D ) +=
integrate( boundaryfaces(mesh), im_type(),
// integration by part
-(gradt(u)*N())*idt(v) + (idt(u)*trans(idv(c))*N())*idt(v)
// adjoint consistency
-(grad(v)*N())*idt(u) + (idt(v)*trans(idv(c))*N())*idt(u)
// penalisation
+gamma*idt(v)*idt(u)/hFace());
// linear form (right hand side)
form1( Xh, F ) +=
integrate( boundaryfaces(mesh), im_type(),
// adjoint consistency
-(grad(v)*N())*g + (idt(v)*trans(idv(c))*N())*g
// penalisation
+gamma*idt(v)*g/hFace());
```

B.3.4 Stokes

Example: Stokes

Remark 13 *Stokes* Consider now the following problem, find (\mathbf{u}, p) such that

$$-\Delta \mathbf{u} + \nabla p = \mathbf{f}, \quad \mathbf{u} = \mathbf{g}|_{\partial\Omega}, \quad \nabla \cdot \mathbf{u} = 0 \quad (\text{B.11})$$

under conservative form the equation reads

$$\nabla \cdot (-\nabla \mathbf{u} + p\mathbb{I}) = \mathbf{f}, \quad (\text{B.12})$$

$$\nabla \cdot \mathbf{u} = 0, \quad (\text{B.13})$$

$$\mathbf{u} = \mathbf{g}|_{\partial\Omega} \quad (\text{B.14})$$

where $\mathbb{I}(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ (in 2D) $\forall \mathbf{x} \in \Omega$ is the identity tensor(matrix) field $\in \mathbb{R}^{d \times d}$. The flux tensor field is $\mathbf{F} = -\nabla \mathbf{u} + p\mathbb{I}$. Indeed we have the following relation, if \mathbb{M} is a tensor (rank 2) field and \mathbf{v} is a vector field

$$\nabla \cdot (\mathbb{M}\mathbf{v}) = (\nabla \cdot \mathbb{M}) \cdot \mathbf{v} + \mathbb{M} : (\nabla \mathbf{v}) \quad (\text{B.15})$$

where $\mathbb{M} : (\nabla \mathbf{v}) = \text{trace}(\mathbb{M} * \nabla \mathbf{v}^T)$, $*$ is the matrix-matrix multiplication and $\nabla \cdot \mathbb{M}$ is the vector field with components the divergence of each row of \mathbb{M} . For example $\nabla \cdot (p\mathbb{I}) = \nabla \cdot \begin{pmatrix} p & 0 \\ 0 & p \end{pmatrix}$ (in 2D) $= \nabla p$.

Weak formulation for Stokes

Taking the scalar product of (B.12) by any test function \mathbf{v} (associated to velocity) and multiplying (B.13) by any test function q (associated to pressure), the variational formulation of (B.12) reads, thanks to (B.15),

$$\int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} + p \nabla \cdot \mathbf{v} + \int_{\partial\Omega} ((-\nabla \mathbf{u} + p\mathbb{I})\mathbf{n}) \cdot \mathbf{v} = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} \quad (\text{B.16})$$

where \mathbf{n} is the outward unit normal to $\partial\Omega$. We now introduce the penalisation term that will ensure that $\mathbf{u} \rightarrow \mathbf{g}$ as $h \rightarrow 0$ on $\partial\Omega$. (B.16) reads now

$$\int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} + p \nabla \cdot \mathbf{v} + \int_{\partial\Omega} ((-\nabla \mathbf{u} + p\mathbb{I})\mathbf{n}) \cdot \mathbf{v} + \frac{\gamma}{h} \mathbf{u} \cdot \mathbf{v} = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} + \int_{\partial\Omega} \frac{\gamma}{h} \mathbf{g} \cdot \mathbf{v} \quad (\text{B.17})$$

Finally we incorporate the symetrisation of the bilinear form to ensure adjoint consistency and hence proper convergence order

$$\begin{aligned} \int_{\Omega} \nabla \mathbf{u} : \nabla \mathbf{v} + p \nabla \cdot \mathbf{v} + \int_{\partial\Omega} ((-\nabla \mathbf{u} + p\mathbb{I})\mathbf{n}) \cdot \mathbf{v} + ((-\nabla \mathbf{v} + q\mathbb{I})\mathbf{n}) \cdot \mathbf{u} + \frac{\gamma}{h} \mathbf{u} \cdot \mathbf{v} = \\ \int_{\Omega} \mathbf{f} \cdot \mathbf{v} + \int_{\partial\Omega} ((-\nabla \mathbf{v} + q\mathbb{I})\mathbf{n}) \cdot \mathbf{g} + \frac{\gamma}{h} \mathbf{g} \cdot \mathbf{v} \end{aligned} \quad (\text{B.18})$$

Example: Stokes

```
// total stress tensor (trial)
AUTO( SigmaNt, (-idt(p)*N()+mu*gradt(u)*N()) );
// total stress tensor (test)
AUTO( SigmaN, (-id(p)*N()+mu*grad(v)*N()) );
// linear form (right hand side)
form1( Xh, F ) +=
integrate( boundaryfaces(mesh), im,
trans(g)*(-SigmaN+gamma*id(v)/hFace()) );
// bilinear form (left hand side)
form2( Xh, Xh, D ) +=
integrate( boundaryfaces(mesh), im,
-trans(SigmaNt)*id(v)
-trans(SigmaN)*idt(u)
+gamma*trans(idt(u))*id(v)/hFace() );
```

B.4 Stabilisation techniques

B.4.1 Convection dominated flows

Consider this type of problem

$$-\epsilon \Delta u + \mathbf{c} \cdot \nabla u + \gamma u = f, \quad \nabla \cdot \mathbf{c} = 0 \quad (\text{B.19})$$

Introduce $\text{Pe} = \frac{|\mathbf{c}|h}{\epsilon}$ the *Péclet* number. The dominating convection occurs when, on at least some cells, $\text{Pe} \gg 1$. We talk about singularly (i.e. $\epsilon \ll h$) perturbed flows.

Without doing anything wiggles occur. There are remedies so called *Stabilisation Methods*, here some some examples:

- Artificial diffusion (streamline diffusion) (SDFEM)
- Galerkin Least Squares method (GaLS)
- Streamline Upwind Petrov Galerkin (SUPG)
- Continuous Interior Penalty methods (CIP)

B.4.2 The CIP methods

Add the term

$$\sum_{F \in \Gamma_{\text{int}}} \gamma h_F^2 |\mathbf{c} \cdot \mathbf{n}| [\nabla u][\nabla v] \quad (\text{B.20})$$

where Γ_{int} is the set of internal faces where the $\text{Pe} \gg 1$ (typically it is applied to all internal faces) and

$$[\nabla u] = \nabla u \cdot \mathbf{n}|_1 + \nabla u \cdot \mathbf{n}|_2 \quad (\text{B.21})$$

is the jump of ∇u (scalar valued) across the face. In the case of scalar valued functions

$$[u] = u\mathbf{n}|_1 + u\mathbf{n}|_2 \quad (\text{B.22})$$

Remark 14 (Choice for γ) γ can be taken in the range $[1e-2; 1e-1]$. A typical value is $2.5e-2$.

```
// define the stabilisation coefficient expression
AUTO( stab_coeff , (gamma abs(trans(N()))*idv(beta))*
      vf::pow(hFace(),2.0));

// assemble the stabilisation operator
form2( Xh, Xh, M ) +=
  integrate(
    // internal faces of the mesh
    internalfaces(Xh->mesh()),
    // integration method
    _Q<OrderOfPolynomialToBeIntegratedExactly>,
    // stabilisation term
    stab_coeff*(trans(jumpt(gradt(u)))*jump(grad(v))));
```

B.5 Interpolation

In order to interpolate a function defined on one domain to another domain, one can use the `interpolate` function. The basis function of the image space must be of `Lagrange` type.

```
typedef bases<Lagrange<Order, Vectorial> > basis_type; // velocity
typedef FunctionSpace<mesh_type, basis_type, value_type> space_type;
// ...
space_ptrtype Xh = space_type::New( mesh1 );
element_type u( Xh, "u" );
space_ptrtype Yh = space_type::New( mesh2 );
element_type v( Yh, "v" );

// interpolate u on mesh2 and store the result in v
interpolate( Yh, u, v );
```


APPENDIX C

GNU Free Documentation License

GNU Free Documentation License Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject

(or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.

9. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
15. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail

to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

Boost
 Parameter, 15
boost
 shared_ptr, 30

Class
 Application, 21
 Backend, 30
 Mesh, 21
cmake, 14

formulation
 variational, 32

laplacian, 32
 formulation
 feel, 34
 mathematical, 32

Libraries
 PETSc, 21
 Trilinos, 21

mesh, 21

PETSc, 21

Stokes, 36
 formulation
 feel, 37
 mathematical, 36

Trilinos, 21

Bibliography

- [1] Wikipedia. <http://fr.wikipedia.org>.
- [2] Vincent Chabannes, Gonalo Pena, and Christophe Prud'Homme. High order fluid structure interaction in 2D and 3D. Application to blood flow in arteries. Submitted to Elsevier Journal of Computational and Applied Mathematics (JCAM), February 2012.
- [3] Vincent Doyeux, Yann Guyot, Vincent Chabannes, Christophe Prud'Homme, and Mourad Ismail. Simulation of two-fluid flows using a finite element/level set method. Application to bubbles and vesicle dynamics. Submitted to Elsevier Journal of Computational and Applied Mathematics (JCAM), February 2012.
- [4] J.-L. GUERMOND. *Fluid Mechanics: Numerical Methods.*, pages 365–374. Oxford: Elsevier, Encyclopedia of Mathematical Physics, eds. J.-P. Franoise, G.L. Naber, and Tsou S.T. , (ISBN 978-0-1251-2666-3), volume 2 edition, 2006.
- [5] Incropera, DeWitt, Bergman, and Lavine. *Fundamentals of heat and mass transfer*. Wiley, 6 edition, 2007.
- [6] Gonalo Pena. *Spectral element approximation of the incompressible Navier-Stokes equations in a moving domain and applications*. PhD thesis, Lausanne, 2009.
- [7] Gonalo Pena and Christophe Prud'homme. Construction of a high order fluid-structure interaction solver. *Journal of Computational and Applied Mathematics*, In Press, Accepted Manuscript, 2009.
- [8] Christophe Prud'homme. A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming*, 14(2):81–110, 2006. <http://iospress.metapress.com/link.asp?id=8xwd8r59hg1hmlcl>.
- [9] Christophe Prud'homme. Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In *Workshop On State-Of-The-Art In Scientific And Parallel Computing*, Lecture Notes in Computer Science, page 10. Springer-Verlag, 2006. Accepted.
- [10] Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, and Gonalo Pena. Feel++: A Computational Framework for Galerkin Methods and Advanced Numerical Methods. Submitted to ESAIM Proc., January 2012.
- [11] Benjamin Stamm. *Stabilization strategies for discontinuous Galerkin methods*. PhD thesis, Lausanne, 2008.
- [12] Christoph Winkelmann. *Interior penalty finite element approximation of Navier-Stokes equations and application to free surface flows*. PhD thesis, Lausanne, 2007.